Concept Discovery from Un-Constrained Distributed Context

Vishal Goel(✉) and B.D. Chaudhary

School of Computing and Electrical Engineering, IIT Mandi, Mandi, India
sl3003@students.iitmandi.ac.in,
bdchaudhary@iitmandi.ac.in

Abstract. Formal Concept Analysis (FCA) is a method for analysing data set consisting of binary relation matrix between objects and their attributes to discover concepts that describe special kind of relationships between set of attributes and set of objects. These concepts are related to each other and are arranged in a hierarchy. FCA finds its application in several areas including data mining, machine learning and semantic web.

Few iterative MapReduce based algorithms have been proposed to mine concepts from a given data set. These algorithms either copy the entire data set (context) on each node or partition it in a specific manner. They assume that all attributes are known apriori and are ordered. These algorithms iterate based on the ordering of attributes. In some applications these assumptions will limit the scalability of algorithms.

In this paper, we present a concept mining algorithm which does not assume apriori knowledge of all attributes and permits the distribution of context on different nodes in an arbitrary manner. Our algorithm utilizes Apache Spark framework for discovering and eliminating redundant concepts in each iteration. When we aggregate data on attribute basis, we order the attributes based on the number of objects containing them. Our method relies on finding extents for combinations of attributes of particular size (say 'k'). An extent which is not regenerated in attribute combinations of size k + 1 corresponds to a valid concept. All concepts with particular intent size k are saved in one Resilient Distributed Data-set (RDD). We have tested our algorithms on two data sets and have compared its performance with earlier algorithm.

Keywords: Formal concept analysis · Distributed concept mining · Incremental mining

1 Introduction

FCA is a method for extracting all formal concepts from a given binary object-attribute relational database. A formal concept can be understood as a natural cluster of particular objects and particular attributes related to each other. For example if a query is made on an animal data-set to find all animals with certain attribute say '4 legs', the answer returned will contain a cluster or set of animals with '4 legs'. But a concept can give extended information about the same query such as every animal having '4 legs' also have 'tail' in the attributes. This is known as attribute implication. A data-set

N. Kumar and V. Bhatnagar (Eds.): BDA 2015, LNCS 9498, pp. 151–164, 2015.
DOI: 10.1007/978-3-319-27057-9_11

contains a number of formal concepts within it. These concepts mined have super-concept – sub-concept relationships among them. So if we find another concept containing '4 legs', 'tail' and 'long trunk' in attribute set, object set will probably contain only 'elephant'. As object set with single element 'elephant' is subset of object set of all animals with '4 legs', we say that the newly mined concept is a sub-concept of former concept. This type of super-concept – sub-concept relationship among all concepts can be visualized using a lattice structure, called concept lattice. Concept lattice provides an intuitive and powerful representation of data. For deeper understanding, related definitions and an example can be found in Sect. 2.

FCA finds its application in fields related to data mining and ontology engineering. FCA is proposed to be used in machine interpretable semantic web, association rule mining [9] and business application such as collaborative recommendation [10]. Several algorithms exist to mine formal concepts from a given data-set (also known as Context) on standalone systems. The applicability of such algorithms is restricted when the size of context is increased. These algorithms require scanning complete context to generate a new concept and thus when context is large, process of finding all concepts is time consuming. Each object or attribute can be present in a number of concepts, so the problem also demands high storage. This spurred interest in finding distributed solutions that would reduce the time for mining as well as enable distributed storage using low cost networked computing nodes.

The efforts in this direction exploited the MapReduce paradigm. Iterative MapReduce based solutions such as [3, 4, 12] were proposed to mine concepts. Two of the most popular algorithms are described in Sect. 3. These algorithms make certain assumptions that limit their ability to scale to large context size. One of them requires the entire context to be stored on every node whereas the other suffers from huge communication overhead at the end of each iteration. In practice we find large datasets that cannot be stored on a single machine. For example, dataset of emails sent by customers to a company might be used to find correlation between customers. Another example could be an e-commerce website that offers a number of products to its customers to categorize customers on specific product browsing habits.

In our approach context is neither stored on nodes statically nor is it scanned to compute concepts. Further, the context is aggregated on the lower dimension (either objects or attributes) as key-value pairs to create a distributed collection. In the processes of generating new key-value collection from an existing one and filtering concepts from parent collection, we efficiently utilize the MapReduce paradigm to scale well. Our approach is said to be unconstrained over the context because our algorithm does not use the context after generating the key-value pairs in the first iteration.

We have organized this paper in the following way. Basic definitions in formal concept analysis with an example can be found in Sect. 2. In Sect. 3 we briefly describe the two existing algorithms for distributed formal concept mining along with their drawbacks. In Sect. 4 we describe our concept mining algorithm in form of several steps. In Sect. 5 we describe our data processing pipeline and enlist the Spark functions we used for implementation. In Sect. 6 we demonstrate some experimental results. After discussing some facts about our algorithm in Sect. 7, we conclude our paper in Sect. 8.

2 Definitions and Properties in FCA

Formal Concept Analysis [1, 2] is a technique for knowledge discovery in databases (KDD). We adopt some notations as in [1]. Some of the basic definitions from [1] are reproduced below:

1. A formal context (G, M, I) consists of two sets G and M and of binary relation $I \subseteq G \times M$. The elements of G are called the objects, those of M the attributes of (G, M, I). If $g \in G$ and $m \in M$ are in relation I, we write $(g, m) \in I$ or g I m and read this as "object g has attribute M".
2. Let $A \subseteq G$ and $B \subseteq M$, then: $A^{\uparrow} = \{m \in M \mid \forall g \in A, (g\ I\ m)\}$ and $B^{\downarrow} = \{g \in G \mid \forall m \in B, (g\ I\ m)\}$.
   $A^{\uparrow\downarrow}$ (precisely $(A^{\uparrow})^{\downarrow}$) and $B^{\downarrow\uparrow}$ (precisely $(B^{\downarrow})^{\uparrow}$) are closure operators. From context Table 1, consider a given object set A = {1, 2}, then $A^{\uparrow} = \{c, g\}$, also $A^{\uparrow\downarrow} = \{c, g\}^{\downarrow} = \{1, 2, 5\}$ and we can notice that $A \subseteq A^{\uparrow\downarrow}$. Similarly $B^{\downarrow\uparrow}$ gives complete set of attributes present in objects containing all attributes in B and $B \subseteq B^{\downarrow\uparrow}$. Also $\varnothing^{\uparrow} = M$, set of all attributes and $\varnothing^{\downarrow} = G$, set of all objects. The symbol '$\varnothing$' represents a null set.
3. (A, B) is formal concept of (G, M, I) iff, $A \subseteq G$, $B \subseteq M$, $A^{\uparrow} = B$, and $A = B^{\downarrow}$. The set A is called the extent and the set B is called the intent of the formal concept (A, B).
4. For a context (G, M, I), a concept (A1, B1) is sub-concept of a concept (A2, B2) (and equivalently (A2, B2) is super-concept of (A1, B1)) iff $A1 \subseteq A2$ or equivalently, $B2 \subseteq B1$. We denote $\leq$-sign to express this relation and thus we have (A1, B1) $\leq$ (A2, B2): $\Leftrightarrow$ $A1 \subseteq A2$ or $B2 \subseteq B1$. We say (A1, B1) is proper sub-concept of (A2, B2) if (A1, B1) $\neq$ (A2, B2) holds.

An example of formal context is shown in Table 1. Rows represent the objects with their unique object IDs while column represent the attributes of objects. Presence of an attribute in an object is shown by 'x' symbol in the context. Let object set $A_I = \{1, 2, 4\}$ and attribute set $B_I = \{c, d\}$, then (A_I, B_I) is a formal concept since $A_I^{\uparrow} = B_I$ and $B_I^{\downarrow} = A_I$. Let $A_J = \{1, 2\}$, then (A_J, B_I) is not a formal concept since $B_I^{\downarrow} \neq A_J$. Consider concepts Y = ({1, 2, 4}, {c, d}) and X = ({1, 4}, {c, d, f}), and we can say X is proper sub-concept of Y (or C2 < C1). This ordering can be expressed in the form of a lattice (see Fig. 1) where each node represents a concept derived from the context and it is connected to all related concepts. The concepts reachable from a particular concept in strictly upward direction in the lattice are super-concepts while the concepts reachable in strictly downward direction are called sub-concepts. The top concept in the lattice is computed using the null attribute set $\varnothing$ as $<\varnothing^{\downarrow}, \varnothing^{\downarrow\uparrow}>$ whereas the bottom concept is computed using the null object set $\varnothing$ as $<\varnothing^{\uparrow\downarrow}, \varnothing^{\uparrow}>$. Generally the top concept is equal to $<G, \varnothing>$ as no attribute is common to all objects in G while the bottom concept is equal to $<\varnothing, M>$ as no object contains all the attributes in M. The top and the bottom concepts are labeled with id C0 and C16 in the lattice.

The concept finding problem can also be viewed as a problem of discovering maximal rectangles of crosses 'x' in the context table. The transpose of a context table

Table 1. An example Formal Context, Objects 'G' in first column, Attributes 'M' in first row and attribute presence in an object is marked by 'x' mark.

	a	b	c	d	e	f	g
1			x	x	x	x	
2	x	x	x	x			x
3	x	x					
4	x		x	x		x	
5	x		x		x	x	x

also contains the same number of maximal rectangles. Therefore if attributes are treated as objects and objects are treated as attributes, newly mined concepts will correspond to concepts from original context, but with interchanged extents-intent pair. So if the complexity of any concept mining algorithm depends un-evenly on context dimension, finding concepts on context transpose might reduce runtime. We utilize this property in our implementation.

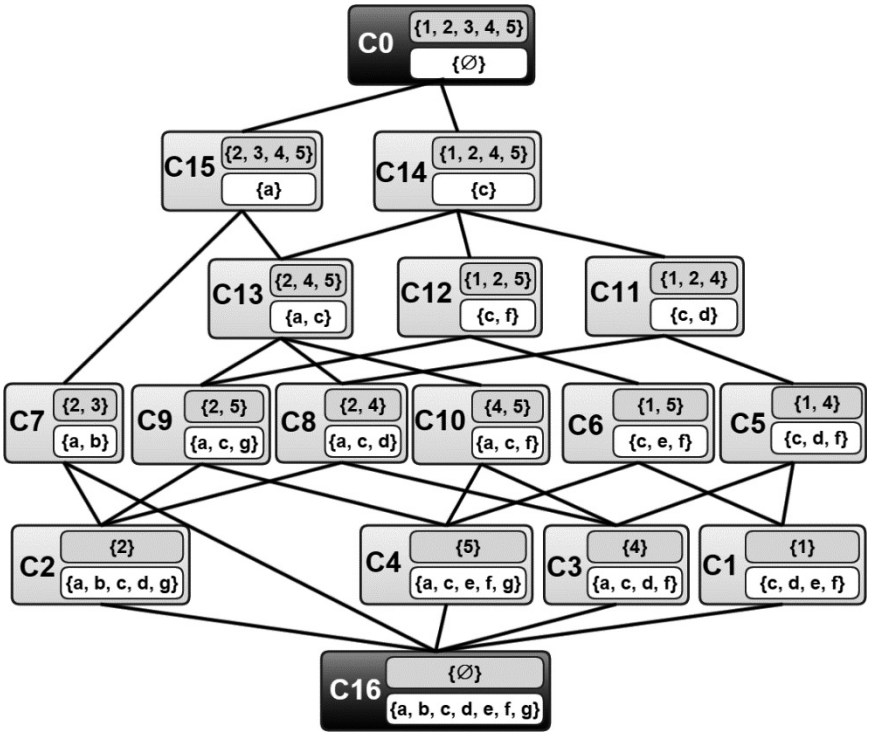

Fig. 1. Concept lattice corresponding to context in Table 1, each node representing concept with Id (left), Extent (top-right) and Intent (bottom-right)

3 Related Work

There are several stand-alone algorithms for concept mining [13–15]. As we are mainly interested in distributed mining algorithms, we compare our approach with two representative algorithms in this area, namely, Close-by-One [3] (referred as MR-CbO) and MR-Ganter [4]. Both of them are implemented using MapReduce paradigm. MR-CbO was the first attempt in parallelizing concept mining process by using several networked nodes to mine concepts in parallel. This algorithm mines concepts using a top-down approach. The top concept is evaluated in the driver code of MapReduce job. Then the iterations are launched with single key-value pair created from the top concept. In the Map phase of each iteration the mapper processes the input pair using intent of the concept and adds non-preexisting attributes to generate several concepts. These concepts are validated by the reducer using a canonicity test. The concepts which pass canonicity test in reduce phase are saved as valid concepts and are used in Map phase of next iteration for further concept generation. By using attribute ordering and canonicity test, the algorithm ensures that each concept is validated only once in a distributed setting. This allows it to run in continuous iterations without any synchronization with the driver. The only drawback with this algorithm is that it requires full context to be present on every node during the Map computation. This confines the algorithm to contexts which can fit in the memory of a single machine. However, as discussed earlier, several data-sets in practice have a large context.

The second algorithm MR-Ganter addresses the issue associated with large context storage. It allows context to be partitioned in several disjoint parts and stores them statically on different nodes. It takes advantage of the fact that concept intents are closed under intersection. Thus for an input intent and every non-preexisting attribute in it, the mappers generate new intermediate local intent(s) which are merged during reduce phase and checked for canonicity. However, this approach has some limitations. In this algorithm each intent can be generated multiple times within iteration as well as across different iterations. As only the new intents are required to be used as input for the next iteration, a repository needs to be maintained to keep track of al-ready generated intents and needs to be checked every time a concept is generated. MR-Ganter+, an enhanced version of MR-Ganter uses a distributed hash table to speedup this check. It is also able to mine concepts in fewer iterations. However, both MR-Ganter and MR-Ganter+ require the newly generated concepts to be broadcasted to all the mappers at the end of each iteration. As the number of concepts generated is substantial, this approach imposes huge network overhead. Further, both the above algorithms do not store the extent along with the intent, which if required needs re-computation.

Our approach neither requires scanning the context for concept generation nor does it require storing the entire context on a single node. Thus, we can process larger data sets simply by adding more machines.

4 Methodology

4.1 Data Format and Input

Conventional algorithms assume input context in tabular form with row-wise storage, i.e. object along with its attributes. But in practice, many data storage systems support append-only writes to avoid locking issues that arise from fine-grained up-dates to existing data objects. The incoming data in such cases is <object id, attribute id> pair. Thus, new objects and new attributes get added to the dataset over time. As big context tables are often very sparse, this representation (key-value pair) has immense storage benefits compared with the tabular representation. An object-attribute pair is added only for the existing attributes of the object whereas the tabular representation would have to store null values for every non-existing attribute in the table. We use the key-value representation in our system. In this paper, we restrict ourselves to mining formal concepts from a snapshot of the dataset taken at a particular time. Issues regarding the merging of formal concepts mined over different snapshots are beyond the scope of this paper.

4.2 Data Aggregation

Recall that finding concepts on the context table or its transpose yields the same result. Thus, we can aggregate data by objects or by attributes. As the total number of concepts is bounded by the number of different possible combinations of objects or attributes (whichever is lower), we aggregate on the dimension with lower count to reduce the search space for valid concepts. Spark has functions that can approximate the distinct number of objects/attributes within a specified timeout. We use that function to limit the maximum number of iterations required by the algorithm to mine all concepts. In the dataset corresponding to our example context, it can be found that the number of objects is less than the number of attributes. So we aggregate on the basis of object ID's. The aggregated key-value pairs are listed as candidate concepts of key size one (referred as CC_1) in Table 2.

4.3 Concept Finding Approach

The candidate concept set of key size 2 (CC_2, Table 3) is generated using CC_1 table as follows: we take all combinations of the keys from CC_1 to produce the candidate concept keys of CC_2; the candidate concept values are generated by intersecting the corresponding values of the keys in CC_1. Those keys in CC_2 which have their value-part as null are discarded from the candidate set. In general, candidate concept set of higher order e.g. CC_{N+1} is generated from candidate concept set CC_N by merging key-value pairs in CC_N where the keys differ only in a single element.

If the value corresponding to any candidate concept in CC_1 is repeated in CC_2, then that candidate concept in CC_1 is not considered to be a valid concept. The rest of the candidate concepts from CC_1 (whose values are not regenerated in CC_2) are copied to Valid Concepts of key size 1 (referred as VC_1, Table 4). Thus, 4 out of the 5 candidate

key-value pairs of size 1 are declared as valid concepts. Continuing in this manner, CC_2 is used to generate CC_3 (Table 5). Filtering out the candidate concepts in CC_2 whose values are regenerated in CC_3, we are left with 6 out of 10 pairs that are valid (VC_2 - Table 6). It may also be noted that CC_3 has only 7 elements out of the 10 possible combinations. The idea of generating 'N + 1'th order candidate set with reduced number of elements using filtered candidates from 'N'th order candidate set is borrowed from Chap. 6 in [11].

We now generalize the process of concept mining. Let CC_N be the set of candidate key-value pairs $<K_N, V>$ of key size 'N' and CC_{N+1} be the candidate set of pairs $<K_{N+1}, V>$ with key size 'N + 1'. Let $V_X = \{V | < K_{N+1}, V > \in CC_{N+1}\}$, be the set of values in CC_{N+1}. Then valid concept set of key size N (VC_N) is given by:

$$VC_N = \{ < K_N, V > | < K_N, V > \in CC_N \text{ and } V \notin V_X \} \quad (1)$$

4.4 Additional Important Insights

It may be noticed that key-value pairs in CC_1 (number of elements = 'N') are listed in lexicographic order of the keys. To generate elements in CC_2 (number of elements = ${}^{N}C_2$) each key in CC_1 is combined with all the keys that succeed in the order. So from N keys in CC_1, first key will be used to generate first N − 1 elements of CC_2, 2nd key will generate next N − 2 elements, and so on. In general, the K'th key in CC_1 is paired with the next 'N-K' elements to generate elements in CC_2 (highlighted in Tables 2 and 3). By enforcing this ordering on the keys, we ensure that the same combination is not regenerated. The value (Set) in each pair of CC_2 is evaluated by intersecting the value from parent key pairs in CC_1. In a distributed implementation, we could partition the computation such that the 'N-K' combinations of the K'th key reach a single node. This way of splitting will however cause uneven number of key-value pairs across nodes causing communication (shuffling) bottlenecks. To minimize the communication overhead, we reorder the keys in CC_1 in ascending (non-descending) order of the number of elements in the value. After reordering, the first key in CC_1 will have least number of elements in the value. The number of combinations of this key in CC_2 will be largest ('N − 1') but the number of elements in each one of them will be small (the cardinality of the intersection operation cannot be greater than the cardinality of the smallest set). Thus by reordering strategy, even when we have large number of keys reaching a node, the number of elements associated to them will be small. Thus, we can reduce the communication overhead.

We now illustrate the combination of two ordered keys of size N to generate a new concept key of size N + 1. Let us assume two ordered keys of size N, $P_N = \{I_1, I_2 \ldots I_{N-1}, A\}$ and $Q_N = \{I_1, I_2 \ldots I_{N-1}, B\}$ which have the first n − 1 elements common. We combine them to give a new ordered key of size 'N + 1' as:

$$K_{N+1} = \{I_1, I_2 \ldots I_{N-1}, A, B\} \text{ only if } A < B \quad (2)$$

In this way candidate keys are combined to create next order candidate keys. The effect can be observed in highlighted keys in Tables 5, 7 and 8.

Table 2. Candidate concepts of key-size = 1 (CC_1)

Key	Value
{1}	{c, d, e, f}
{2}	{a, b, c, d, g}
{3}	{a, b}
{4}	{a, c, d, f}
{5}	{a, c, e, f, g}

Table 3. Candidate concepts of key size = 2 (CC_2)

Key	Value
{1, 2}	{c, d}
~~{1, 3}~~	~~{∅}~~
{1, 4}	{c, d, f}
{1, 5}	{c, e, f}
{2, 3}	{a, b}
{2, 4}	{a, c, d}
{2, 5}	{a, c, g}
{3, 4}	{a}
{3, 5}	{a}
{4, 5}	{a, c, f}

Table 4. Valid concepts of key size = 1 (VC_1) using CC_1 and CC_2

Id	Extent	Intent
C1	{1}	{c, d, e, f}
C2	{2}	{a, b, c, d, g}
C3	{4}	{a, c, d, f}
C4	{5}	{a, c, e, f, g}

Table 5. Candidate concepts of key-size = 3 (CC_3)

Key	Value
{1, 2, 4}	{c, d}
{1, 2, 5}	{c}
{1, 4, 5}	{c, f}
{2, 3, 4}	{a}
{2, 3, 5}	{a}
{2, 4, 5}	{a, c}
{3, 4, 5}	{a}

Table 7. Candidate and valid concepts of key size = 4 (VC_4), since no element for CC_5

Id	Key / Extent	Value / Intent
C14	{1, 2, 4, 5}	{c}
C15	{2, 3, 4, 5}	{a}

Table 6. Valid concepts of key size = 2 (VC_2) using CC_2 and CC_3

Id	Extent	Intent
C5	{1, 4}	{c, d, f}
C6	{1, 5}	{c, e, f}
C7	{2, 3}	{a, b}
C8	{2, 4}	{a, c, d}
C9	{2, 5}	{a, c, g}
C10	{4, 5}	{a, c, f}

Table 8. Valid concepts of key size = 3 (VC_3) using CC_3 and CC_4

Id	Extent	Intent
C11	{1, 2, 4}	{c, d}
C12	{1, 4, 5}	{c, f}
C13	{2, 4, 5}	{a, c}

5 Implementation Using Spark

Apache Spark [5] is a popular open-source distributed data processing framework, which is optimized to best utilize main memory of network nodes and hence is faster than previous technologies. Spark defines its distributed storage abstraction as Resilient Distributed Datasets (RDDs) [6]. It provides two sets of operations – transformations and actions. Transformation allows coarse-grained manipulation of the data in RDD. These transformations are carried out when an action is specified. For more details, refer to Scala API documentation [7].

Data can be inputted from local file or any distributed file system to create a temporary RDD (say R1). This RDD contains object-attribute pairs. Recall our discussion from Sect. 4.2 that we need to aggregate on the dimension with lower count to reduce the search space. As the size of the dimensions is not known apriori, we approximate the count with the help of "map" followed by "countApproxDistinct" transformation. This gives us an approximate number of distinct objects and attributes in the dataset. We aggregate on the dimension with the lower count. For instance, if the number of objects is lower than the number of attributes, we aggregate attributes treating object as key using "combineByKey" transformation to create new RDD, (say R2). Thus object is paired with attributes <key=object Id, value={attribute Id, attribute Id …}>. If the attributes are less, then we aggregate objects based on attributes: <key=attribute Id, {value=object Id, object Id …}>. Next we order the keys within the chosen dimension according to the number of elements. We count the number of elements in the values of each key by using "countByKey" action. We re-label keys in ascending numeric value according to increasing number of elements in their values. This RDD corresponds to CC_1. At this point we no longer require R1 and R2 RDDs, so we remove them from memory.

We create CC_2 RDD using "cartesian" transformation of CC_1 with itself followed by "filter" transformation as specified in Eq. 2. Then using CC_1 and CC_2, we generate VC_1 using "reduceByKey" followed by "filter". As soon as VC_1 is generated, CC_1 is not required anymore and thus can be purged from memory. For the subsequent candidate set generation (e.g. CC_3), we use "map" on the previous candidate set (e.g. CC_2) followed by self-join (using "join" followed by "filter"). The data pipeline is shown in Fig. 2. The top and the bottom concepts are evaluated using reduce operations on CC_1.

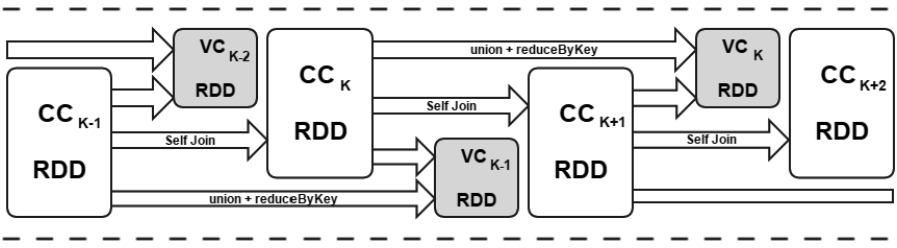

Fig. 2. Spark data pipeline for our implementation

We used Scala interactive shell for our experiments. Scala shell is a single threaded application and thus allows us to run a single job at a time on the cluster. However, Spark also supports running several jobs launched by different threads in parallel. If we use this Spark feature in our application then after generating CC_{K+1} from CC_K, we can immediately proceed to generate CC_{K+2} without waiting for the generation of VC_K. VC_K can be generated by a separate thread and thus execution gets speeded up. The new pipeline in this case will look similar to the one in Fig. 3.

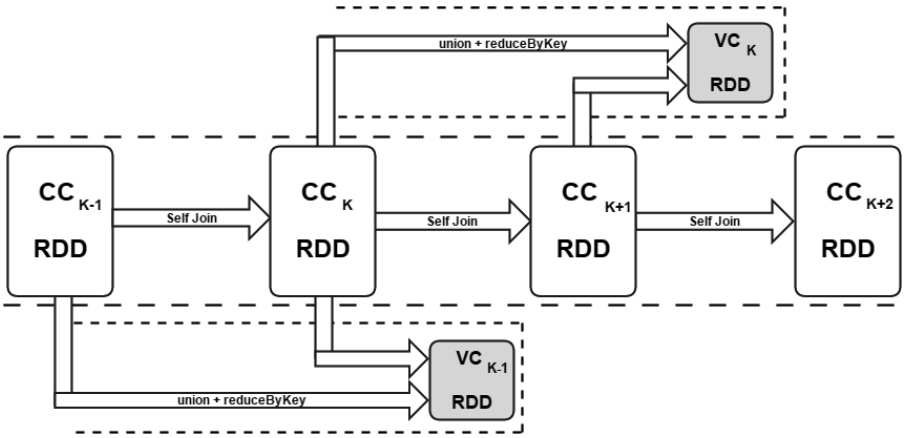

Fig. 3. A parallel (Multithreaded) pipeline for implementation

6 Evaluation and Results

For experiments we used two standard datasets from [8]. The multivalued attributes in data were translated to binary attributes. Then the binary attributed context(s) were converted into text file of randomly ordered object-attribute pairs (one per line) before giving them as input to our algorithm. The details about the converted datasets are shown in Table 9. We used 5 lab machines with core i5-2310 (2.9 GHz) processor, 4 GB Memory and running Ubuntu 12.04 LTS operating system. We ran Spark master on one node and used others as slaves to run Spark standalone cluster. Each slave was configured to use a single CPU core and 512 MB of memory. In figures, when we show results with 'K' nodes, it means 'K' slaves. In this paper, we experimentally compared our approach with MR-CbO algorithm as porting MR-Ganter to Spark was inefficient.

Table 9. Dataset Specifications

Dataset	SPECT	Mushroom
Object-attribute pairs	2042	186852
Distinct attributes	23	119
Distinct objects	267	8124
Concepts	21550	238710

For the smaller dataset (SPECT), MR-CbO performs much faster than our algorithm as the storage of such context takes very less space in memory and scanning small context is not computationally expensive (Figs. 4 and 5).

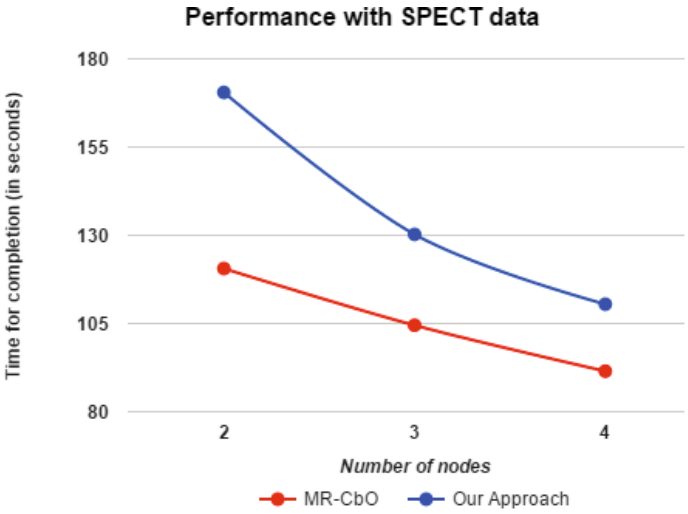

Fig. 4. Result comparison on SPECT dataset

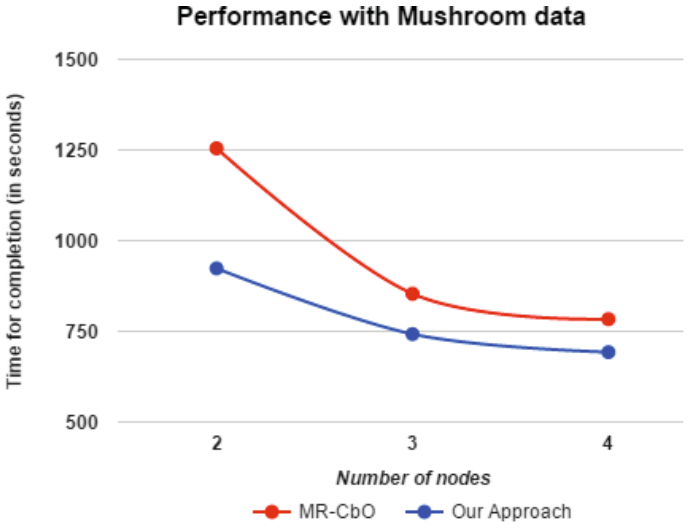

Fig. 5. Result comparison on Mushroom dataset

For the larger dataset (Mushroom), our algorithm performs faster than MR-CbO. Recall that MR-CbO stores the entire context on every node. As the dataset increases in size, the performance is limited by the memory on the machine. Storing large context with limited memory and scanning it several times to generate every new concept is computationally expensive. Thus, MR-CbO algorithm did not perform well for large data set.

7 Discussion

7.1 Scalability with Respect to Context

In this paper, we experimentally compared our approach with MR-CbO algorithm. As MR-CbO requires the entire context to be stored in memory, we used datasets that fit in the memory of a single node in our experiments. But our algorithm can scale with larger context size as it does not require the entire context to be stored on a single node at any step of the algorithm. It should be noted that the first step (generating CC_2) uses "cartesian" transformation followed by "filter" whereas the later steps use "map-join-filter" transformation sequence. We do so because generating CC_2 using "map-join-filter" transformation sequence would result in entire context to be generated on a single node as intermediate data. The "cartesian" transformation although not very efficient is a distributed operation, and we employ it only in the first step. We can improve performance of the other steps by using a different transformation sequence (like "map-aggregateByKey-flatMap") thereby avoiding "join" transformation for generating next level candidate concept set. Testing the performance of this optimization and others like custom Spark partitioner to minimize network shuffling of keys during new candidate concept generation is part of our future work.

7.2 Comparison with Distributed Frequent Item-Set Mining

Although our approach resembles frequent item-set mining, there are several notable differences between concept mining and frequent item-set mining. In frequent item-set mining, item-sets which pass the support threshold are used to generate larger candidate item-sets whereas in concept mining all possible combinations of item-sets are considered to generate larger candidate concepts. Further, these generated concepts have to be validated as described in Eq. 1 and the concept lattice is formed. By applying the threshold criteria, we can mine frequent item-sets from the concept lattice.

7.3 Concept Explosion

Even with a small number of objects (8194 in the Mushroom data-set), the number of concepts mined are 2,38,710. When the concepts mined are large, it is hard to visualize them. This raises several interesting questions regarding concept generation methodology - "should all concepts be generated? Can we rank concepts?" We are currently exploring methods to handle the concept explosion problem.

7.4 Mining Links Between Concepts

Currently, no distributed algorithm mines links between concepts. This would be useful in visualization. In our algorithm, the concepts of different sizes are present in different RDDs. Hence, a supplementary algorithm needs to be designed for mining links between concepts. Concept lattice could be represented using Spark's graph processing component namely "GraphX" or any other graph framework for visualization or further analytics.

8 Conclusion

In this paper, we presented an incremental approach to mine formal concepts from a context which is assumed to be present in the form of object-attribute pairs. We validated our approach by comparing the concepts generated with that in MR-CbO. Our approach is more scalable than the earlier approaches as it neither requires scanning the context for concept generation nor does it require storing the entire context on a single node. Thus, we can process larger data sets simply by adding more machines. Experimental results show that our algorithm outperforms the previously proposed algorithms as the data-set size grows. Thus, our algorithm is suitable for practical concept mining applications.

Acknowledgments. Authors would like to thank Dr. Sriram Kailasam, Assistant Professor at IIT Mandi for his help in improving the manuscript.

References

1. Ganter, B., Wille, R.: Formal Concept Analysis. Springer, Berlin (1999)
2. Wille, R.: Restructuring lattice theory: an approach based on hierarchies of concepts. In: Ferré, S., Rudolph, S. (eds.) ICFCA 2009. LNCS, vol. 5548, pp. 314–339. Springer, Heidelberg (2009)
3. Krajca, P., Vychodil, V.: Distributed algorithm for computing formal concepts using map-reduce framework. In: Adams, N.M., Robardet, C., Siebes, A., Boulicaut, J.-F. (eds.) IDA 2009. LNCS, vol. 5772, pp. 333–344. Springer, Heidelberg (2009)
4. Xu, B., de Fréin, R., Robson, E., Ó Foghlú, M.: Distributed formal concept analysis algorithms based on an iterative mapreduce framework. In: Domenach, F., Ignatov, D.I., Poelmans, J. (eds.) ICFCA 2012. LNCS, vol. 7278, pp. 292–308. Springer, Heidelberg (2012)
5. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, pp. 10–10 (2010)
6. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, pp. 2–2. USENIX Association (2012)

7. Spark programming guide. http://Spark.apache.org/docs/latest/programming-guide.html. Accessed 01 July 2015
8. UCI Machine Learning Repository: Data Sets. http://archive.ics.uci.edu/ml/datasets.html. Accessed: 01 July 2015
9. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Efficient mining of association rules using closed itemset lattices. Inf. Syst. **24**, 25–46 (1999)
10. du Boucher-Ryan, P., Bridge, D.: Collaborative recommending using formal concept analysis. Knowl.-Based Syst. **19**(5), 309–315 (2006)
11. Rajaraman, A., Ullman, J.: Mining of Massive Datasets. Cambridge University Press, New York (2012)
12. Ying., W., Mingqing, X.: Diagnosis rule mining of airborne avionics using formal concept analysis. In: International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC). IEEE (2013)
13. Ganter, B., Reuter, K.: Finding all closed sets: a general approach. Order **8**(3), 283–290 (1991)
14. van der Merwe, D., Obiedkov, S., Kourie, D.G.: AddIntent: a new incremental algorithm for constructing concept lattices. In: Eklund, P. (ed.) ICFCA 2004. LNCS (LNAI), vol. 2961, pp. 372–385. Springer, Heidelberg (2004)
15. Kuznetsov, S.O.: Learning of simple conceptual graphs from positive and negative examples. In: Żytkow, J.M., Rauch, J. (eds.) PKDD 1999. LNCS (LNAI), vol. 1704, pp. 384–391. Springer, Heidelberg (1999)