Scalable formal concept analysis algorithms for large datasets using Spark

Raghavendra K Chunduri1 · Aswani Kumar Cherukuri1

Received: 18 June 2018 / Accepted: 18 October 2018 / Published online: 8 November 2018
© Springer-Verlag GmbH Germany, part of Springer Nature 2018

Abstract
In the process of knowledge discovery and representation in large datasets using formal concept analysis, complexity plays a major role in identifying the formal concepts and constructing the concept lattice (digraph of the concepts). For identifying the formal concepts and constructing the digraph from the identified concepts in large datasets, various distributed algorithms are available. However, the existing distributed algorithms are not well suited for concept generation, because the generation of concepts is an iterative process. Existing algorithms are implemented using distributed frameworks like MapReduce and Open MP. These frameworks are not appropriate for iterative applications. Hence, there is a need for efficient distributed algorithms for both formal concept generation and concept lattice digraph construction in large formal contexts. In this paper, we present efficient algorithms using Apache Spark. The various performance metrics used in evaluation prove that the proposed algorithms are more efficient for concept generation and lattice graph construction than existing algorithms.

Keywords Apache Spark · Concept lattice · Formal concept analysis · Hadoop · MapReduce · Resilient distributed dataset

1 Introduction

Formal concept analysis (FCA), invented by Wille in the early 80’s, is mainly used for analysis of object-attribute relationships and knowledge representation based on two notions: formal context (input) and formal concepts (output). The formal context is the input to FCA, which consists of sets of objects, sets of attributes and a binary relation that specifies which objects have which attributes (Ganter 1984). Objects, attributes and the relation between objects and attributes are represented using a cross table, with rows representing the objects, columns representing the attributes and structure describing the relation between objects and attributes. Formal concepts are derived from the formal context (input). Derived concepts are sorted in the order of inclusion and organized hierarchically to form a complete lattice called a concept lattice (Ganter and Wille 1994). Concept lattice is the core structure of FCA that formulates the knowledge. Users can easily find the incidence relationship among objects and attributes (Belohlavek 2008). The concepts in the concept lattice constitute a partial order relation that reflects the generalization and specialization within the concepts. There are various algorithms available in the literature for identifying concepts and constructing the concept lattice. The existing algorithms are divided into batch and incremental algorithms. The batch algorithms identify the concepts more quickly than the incremental algorithms because of a canonicity test that helps to avoid listing the same concept again (Xin and Ming-Wen 2017). For the same reason, in batch algorithms it is hard to build the incidence relationship among the concepts during concept generation. Hence, the concept lattice structure is not available explicitly in batch algorithms. Incremental algorithms maintain the incidence relationship among concepts; hence, they can obtain the concept lattice structure after concept generation. But the incremental algorithms do not have a canonicity test, which results in listing the same concept more than once. This increases the concept generation time. The problem of determining all concepts in the given formal context is #P-Complete (Kuznetsov 1999). The existing algorithms perform well for a smaller context, but they are computationally weak when they need to find the concepts in a large
formal context and construct the lattice graph from the listed concepts.

In the Big Data era, FCA is widely used as an efficient data analysis technique. Several applications are using FCA for knowledge discovery and representation. For example, in supervised learning some of the classification methods proposed are based on FCA (Olga et al. 2013). Also, in natural language processing FCA is used for learning concept hierarchies from text corpas (Philipp et al. 2005). With the extensive use of FCA in diverse fields, the complexity issues became a bottleneck for the effective use of FCA in all the domains. Thus, efficient FCA algorithms are needed for knowledge discovery(concept generation) and knowledge representation(concept lattice digraph construction) when dealing with large datasets. In this paper efficient distributed algorithms for concept generation and concept lattice construction using Apache Spark are discussed. Apache Spark is a distributed in-memory processing framework used for iterative and interactive data analysis in large datasets. The proposed concept generation algorithm is a distributed batch processing algorithm that works iteratively for identifying concepts in the given large formal context. Then the lattice construction algorithm implemented using the Spark Graphx module treats every concept as a node and constructs the digraph. The experimental analysis of the proposed work proves that the algorithm for concept generation is performing better than other existing distributed approaches while finding concepts. The algorithms use all the features of Apache Spark; the new canonicity test introduced in the proposed work is based on Spark storage level features. This test improved the performance of the algorithm significantly by eliminating the processing of duplicate concepts. Maintaining the parent_index value for every concept helps in finding the incidence relationship among the generated concepts for the construction of the lattice graph.

The remainder of the paper is organized as follows. Section 2 recalls the basic definitions of FCA; Sect. 3 briefly introduces Apache Spark and its advantages over other existing Big Data processing frameworks. Section 4 gives a detailed note on the related work. In Sect. 5 the proposed work is discussed in detail. The experiments are demonstrated using the proposed method and the analysis of the results are presented in Sect. 6. Finally the paper concludes with recommended directions for future work.

2 Formal concept analysis

In this section, the basic definitions of FCA from Ganter and Wille are reproduced verbatim. The basics of FCA according to Ganter (1984) and Belohlavek (2008) given as

Definition 2.1 A formal context is a triplet \( K = (G, M, I) \) where \( G \) is a non-empty set of objects, \( M \) represents a non-empty set of attributes, and \( I \) is a relation between \( G \) and \( M \), a subset of \( G \times M \) represents the cartesian product. For the pair of elements \( g \in G \text{ and } m \in M \text{ if } (g, m) \in I \) then this relation is expressed as object \( g \) has attribute \( m \) and writes as \( gIm \). The derivation operators for a set \( A \subseteq G \) and \( B \subseteq M \) defined as

\[
A^\uparrow = \{ m \in M \mid \text{for each } g \in A : (x, y) \in I \},
\]

\( A^\uparrow \) is the set of attributes common to the objects

\[
B^\downarrow = \{ g \in G \mid \text{for each } m \in B : (x, y) \in I \},
\]

\( B^\downarrow \) is the set of objects common to the attributes

Definition 2.2 A formal concept of a context \( K (G, M, I) \) is a pair \((A, B)\) defined as \( A \subseteq G \) and \( B \subseteq M \), such that

\[
A^\uparrow = B \text{ and } B^\downarrow = A
\]

\( A \) is called the extent and \( B \) is called the intent of the concept \((A, B)\)

Definition 2.3 The collection of all formal concepts in a given context \( K (G, M, I) \) order by \( \subseteq \) is called concept lattice. A particular node in a lattice can be reached in various paths while hierarchical structures restrict each node to possess a single parent. The meet-mutual-sub-concept relation and the join-mutual-super-concept relation in a concept lattice is transitive that facilitates, a sub-concept of any given concept can be reached by traveling upwards from it. The top node in a concept lattice describes the generalization capability with all the objects of context in its extent. Similarly, the bottom node represents the specialization by exhibiting all the attributes of the context in its intent. From both generalization and specialization if an attribute-m(object-g) attached to a node in the lattice, then all the nodes below(above) must also contain the attribute-m(object-g). The lattice structure also gives both probabilistic and deterministic rules. The probabilistic rules are called as classification rules whereas the deterministic rules are known as functional dependencies.

Definition 2.4 An attribute implication represents an expression \( P \rightarrow Q \), where \( P, Q \subseteq M \), is true in context \( K \) if each object which has all attributes from \( P \) also has all attributes from \( Q \).

The number of possible implications that can be derived from a context can be exponential. For e.g. if there are \( m \) number of elements in the attribute set \( M \) then \( 2^m \) implications are possible from the context.
Definition 2.5 A set $X$ of attribute implications are called complete and sound with respect to a formal context $K$, if $X$ is true in $K$ and each implication is true in $K$ follows from $X$.

Definition 2.6 A set $X$ on non-redundant attribute implications which is complete and sound with respect to a formal context $K$ is a base with respect to a context $K$.

The basics of FCA models, applications and techniques are nicely presented in the literature. FCA is widely used in diversified fields. Numerous applications use FCA efficiently for effective results. Priss (2005, 2007) has widely used FCA in natural language processing applications. Dmitrif (2017) has given a detailed overview of FCA and its applications in information retrieval. Kuznetsov and Poelmans (2013), have given a comprehensive literature review of FCA in various domains. Aswani Kumar et al. (2010, 2014) has proposed various methods for knowledge reduction in FCA. Aswani Kumar (2012) also proposed how fuzzy $k$-means clustering method is used on reduced contexts for efficient association rule mining. FCA is also widely used in the field of Machine Learning. Kuznetsov et al. (2016) has given a detailed overview on how FCA can be used in machine learning. Ferrandin et al. (2013), proposed a method for hierarchical classification using FCA. du Patrick and Bridge (2006), have proposed a collaborative filtering method using formal concept analysis. Singh and Kumar (2014), have given a detailed overview on applications of FCA and its research trends. Andrews and Orphanides (2010), extended the FCA for analysing large datasets. Zhao et al. (2018), proposed methods for matching bio-medical ontologies using formal concept analysis. Wassim et al. (2014), used FCA for analysing smart environments by discovering interesting patterns in the sensor data. Stumme (2002) proposed methods for efficient data mining based on FCA. Loia et al. (2017) extended the FCA to fuzzy data, online data streams and proposed solutions to various applications that use Big Data. Examples of his works include, methods based on fuzzy FCA to automatically build semantic web ontology (Loia et al. 2012). Václav et al. (2009), used FCA for analysing social networks. Qi et al. (2014), introduced three-way formal concept analysis for real-world decision making. Sumangali and Kumar (2017, 2018), has given a comprehensive overview on the basics of FCA in recent works and concept lattice simplication using techniques of attribute clustering.

3 Apache Spark

Distributed frameworks like MapReduce and its variants are popular and highly successful in implementing very large-scale data intensive applications on commodity hardware. However, these frameworks are not very suitable for iterative applications and for applications that handle real-time data streaming. Hence, Apache Spark intended to overcome the drawbacks of MapReduce, using its underlying architecture. Spark is well suited for iterative applications that require access to the same data multiple times. The in-memory computations in Apache Spark are ten times faster than the computations in Hadoop MapReduce. Spark is a well established distributed framework that has been widely covered in the literature. So we only provide an overview of Spark in this article.

Apache Spark is an open-source cluster computing framework that supports flexible in-memory data processing that enables batch and real-time data processing. The idea behind the implementation of Spark is to develop a computing framework for distributed machine learning algorithms. Spark provides an interface for programming, entire clusters with inherent data-parallelism and fault tolerance (Zaharia et al. 2010). Spark is integrated closely with other Big Data processing frameworks like Hadoop and accesses any Hadoop data sources while running. Spark abstracts the tasks of job submission, resource scheduling, tracking and communication between nodes, execution, and the low-level operations that are inherent in parallel data processing. Spark is used for a wide range of large-scale data processing tasks in machine learning and iterative analytics. Spark Core, Spark SQL, Spark MLlib and Spark Graphx, and Spark Streaming real-time are the main components of Apache Spark. In the proposed work Spark Core, Graphx (Joseph et al. 2014) components are used extensively for concept generation and lattice construction. A general idea is given for processing of streaming data using Spark Streaming.

3.1 Spark resilient distributed datasets (RDD)

RDDs are Spark’s core components. RDDs are the collections of data that are distributed and partitioned across all the nodes in a cluster. RDDs in Spark are fault-tolerant. This means that if a task on a given node fails, the RDD can be reconstructed automatically on the remaining nodes to complete the job. RDDs in Spark operate in parallel on data (Zaharia et al. 2012). RDDs in Spark can be created using snippets.

```
val inputRDD = sc.textFile(inputFile)
val inputRDD = sc.parallelize(list)
```

RDDs in Spark support two types of operations:—, transformations and actions. Examples of transformations and actions are given as

```
Transformations: Map, FlatMap, Reduce and others
Actions: count, reduce, saveAsTextFile and others
```

Once RDDs in spark are created, various operations can be performed on the distributed dataset. These operations
are split into transformations and actions. A transformation operation in Spark creates a new dataset from an existing one, where as an action operation returns a value by performing computations on the dataset. An action operation returns its results to the driver program. Transformations in Spark are lazy, they do not compute results right away. Transformations are computed when an action requires a result that needs to be sent to a driver program. Each RDD maintains a pointer to one or more parents along with the metadata about it; that is, the type of relationship it has with its parent. When a RDD transformation is called, the RDD keeps a reference to its parent, which is called lineage. Spark creates a lineage graph with all the series of transformations that are applied. When the driver submits the job, the lineage graph is serialized to the worker nodes. Each of these worker nodes applies transformations on different nodes and allows effective fault-tolerance by logging all the transformations that are used to build the dataset. If a partition of a RDD is lost, the RDD has enough information from the lineage graph to recompute the failed partition; thus, lost data can be recovered quickly, without requiring the costly computation again.

3.2 Caching RDDs

The most important feature of the Spark is its ability to cache the data in memory across the cluster. The cache method is used to cache the data, and the method tells Spark that a RDD should be kept in memory. The first time an action is called on a RDD, that action initiates a computation, and the data is read from the disk and stored into memory. Hence, on the first operation, the time taken to run the task is longer because the input is read from disk. Thereafter the data can be read directly from cache memory, which avoids expensive I/O operations and speeds up the processing and significantly improves the performance of the algorithm.

3.3 Applications of Apache Spark

Several applications are extended using Spark for processing large datasets. Zadeh et al. (2016), implemented methods for matrix computations and optimizations using Apache Spark. Alsheikh et al. (2016), used Apache Spark for mobile data analytics using deep learning. Shanahan and Dai (2015), proposed large scale distributed data science using Apache Spark. Yu et al. (2016), introduced cluster computing framework using apache spark for analysing geo spatial data. Domoney et al. (2015), solved smart city problems using Apache Spark. Gupta and Manish (2016), introduced a framework for efficient cyber security network intrusion detection using Apache Spark. Yu et al. (2018), has discussed the optimization of Spark’s fault tolerance by using check point mechanism. The optimization process proposed by the authors are useful for the long running spark applications. Because of the wide applicability of FCA in many fields, efficient algorithms for processing large datasets are required. Hence, efficient algorithms are proposed for concept generation and concept graph construction using Apache Spark and tested against large datasets.

4 Related work

Several algorithms have already been proposed for concept generation and concept lattice construction (Nourine and Raynand 1999; Kuznetsov 1993; Bordat 1986; Lindig 2000; Norris 1978). All these algorithms are best suited for execution on a single node cluster and are not efficient enough to handle large datasets. There are a few concept generation algorithms implemented using MapReduce and its iterative variants like HaLoop and Twister. However these algorithms are only feasible for smaller datasets, and are not efficient enough for dealing with large datasets because of bottle-necks in MapReduce. All the existing algorithms that have adopted the MapReduce framework have not addressed the bottlenecks of MapReduce like efficient use of CPU and memory, in-memory computations and disk I/O after each phase in MapReduce.

Xu et al. (2012), proposed an iterative concept generation algorithm using Twister. Twister is a lightweight iterative runtime environment for iterative MapReduce applications. The algorithms are implementations of Ganter next concept algorithm using iterative MapReduce framework. The algorithms are called MRGanter and MRGanter+. However the iterative approach in Twister is not designed well enough to handle fault tolerance and failures effectively. A single failure will result in executing the current iteration again from the beginning, irrespective of the level of completion of the iteration before the failure occurs. This increases the execution time in case of failures. Also in Twister, it is hard to eliminate the shuffling, sorting and grouping of redundant data. In the Ganter’s algorithm the concept lattice structure is not available immediately because the lattice is an implicit property of the generated concepts.

Nilander et al. (2016), proposed a parallel algorithm implemented using OpenMP based on the Ganter’s next closure algorithm. OpenMP is an API that uses multithreading and executes the algorithm using threads and shared memory. OpenMP is not feasible for larger datasets because of its architectural complexity. In shared memory systems all the data needs to be loaded into shared memory. This is a problem when working on datasets with high dimensionality. Authors also did not discuss the processing of duplicate
concepts, or how the CPU and memory are used efficiently while running threads on the CPU cores. There are no fault tolerance techniques in OpenMP architectures, which means it is hard to return to a stable state after recovery from a failure.

Bhatnagar and Lalit (2015) proposed a MapReduce implementation of concept generation. This algorithm performs computations at reduce phase. This algorithm is not able to find all the formal concepts for the given formal concept. Only a sufficient set of concepts are identified during the single iteration. The authors did not mention how the generated concepts are sufficient and how the existing concepts can generate the remaining concepts when needed.

Chunduri et al. (2017, 2018) proposed an approach using HaLoop, another iterative MapReduce framework. HaLoop takes a large number of iterations to process the larger datasets. HaLoop uses a cache memory concept, but does not support in-memory computations. Since HaLoop adopts most of the architectural model of Hadoop and uses MapReduce, the bottlenecks in the MapReduce system still exist in HaLoop based concept generation.

Loia et al. (2017), proposed a distributed fuzzy concept analysis method for online stream processing in smart cities using Apache Storm. Under this proposal the online streaming data is processed by an incremental algorithm for generating concepts. Based on the work proposed by the authors, we tried executing the SparkConceptGeneration and SparkLatticeConstruction algorithms on the real time streaming data using Spark Streaming.

The Table 1 shows the various properties of various distributed algorithms including the proposed work.

All the algorithms shown in Table 1 are batch processing algorithms that do not have an incidence relationship among the generated concepts. Without an incidence relationship among the generated concepts it is hard to construct the lattice after concept generation. Except the HaLoop implementation, all the other algorithms discussed in Table 1 uses fault tolerance of Hadoop framework. In HaLoop extra care is taken to handle fault tolerance, to avoid iterations to start from scratch after a failure. The OpenMP architecture does not possess any fault tolerance. To overcome the drawbacks in the existing work, the proposed algorithms efficiently use Spark to improve the concept generation process. The proposed algorithm identifies the concepts on the static datasets.

There are a few concept lattice construction algorithms in the literature, but they are not distributed in nature. Muangprathub (2014) proposed a novel algorithm for building a concept lattice, which depends on the size of the extent for lattice construction. The algorithm purely depend on the size of the extent for calculating the concept level; there are chances that different sizes of the concept extent may sometimes fit in the same level, which is not addressed in this approach. This algorithm takes too long to construct the lattice with a large number of formal concepts because of its sequential approach.

5 Proposed algorithms for concept generation and lattice construction

In this section, the proposed work is described in two stages. The first stage identifies the formal concepts for the input context. The proposed algorithm is named SparkConceptGeneration algorithm. The digraph construction takes place in the second stage and the algorithm is named as SparkLatticeConstruction algorithm. The pseudocode of the two algorithms is given in this section with a detailed explanation.

5.1 Explanation of the SparkConceptGeneration algorithm

The SparkConceptGeneration algorithm is a distributed algorithm formalized by a recursive function NeighborConcept(), which lists all the formal concepts starting from the least formal concept. The recursive function NeighborConcept() takes a tuple called concept as an input that has five parameters. The five parameters are the extent and intent of the concept, isValidateNeighbor a boolean flag, size of the intent and parent_index. The parameter isValidateNeighbor is used to determine whether a particular input concept can
generate the neighbor concept. This parameter is useful for the first step in the canonicity test. The parameter parent_index determines the level of the concept in the lattice graph. The value of the parent_index is set to ‘1’ for the least formal concept and the value gets incremented for the neighbor concepts that are generated from every concept. The NeighborConcept() function generates all the upper neighbor concepts in different iterations and stops after the greatest formal concept is found. Every iteration of the recursive process undergoes a two step canonicity test to make sure that the generated concepts are not considered for processing again. The concepts generated in each iteration are saved into Spark’s RDD and persist in the cache. The second step of the canonicity checks the cache before processing the concept and decides whether or not to process the concept. The pseudocode for the two step canonicity test is given as isCanonical method in algorithm 4. The SparkConceptGeneration algorithm starts by creating a Spark context and takes a formal context file as an input. The objects and attributes in the input file are separated by a run time parameter determined based on the type of data. In this paper “,” is considered as the run time parameter. The pseudocode is formulated using “,”. The pseudocode below returns a tuple that has objects and attributes of the context. They are saved into RDDs called contextObjectsRDD and contextAttributesRDD using RDD actions.

The flowchart in Fig. 1 represents the execution flow of NeighborConcept() recursive function is presented in algorithm 1.

The overview of the steps in SparkConceptGeneration algorithm are presented here.

**Step 1:** Initially, the least formal concept is calculated, and the function NeighborConcept() is called with the least formal concept. For the least formal concept the isValidNeighbor flag is set to true and the parent_index is set to 1.

**Step 2:** The recursive function NeighborConcept() lists all the neighbor formal concepts for the least formal concepts and increments the parent_index by 1 for the generated neighbor concepts.

**Step 3:** Two step canonicity tests are performed for each of the concept generated and then the upper neighbor concepts for every concept are listed.

**Step 4:** The isValidNeighbor flag for each concept is checked during the first step of the canonicity test. If the value of this flag is true, then the cache will be checked to see whether the concept is present. If it is present in the cache then the next concept in the RDD will be picked otherwise the current concept is processed.

**Step 5:** The steps 1 through 4 are repeated until the greatest formal concept is reached. The pseudocode for the SparkConceptGeneration algorithm, works as follows.

---

```
1 val sc = new SparkContext(new 
   SparkConf().setAppName("appName"));
2 val context = sc.textFile(inputFile).map ( line => val data = 
   line.split(",");
3 val contextObjectsRDD = data.head ;
4 val attributesRDD = data.tail; }
```

---

Fig. 1 Execution flow of recursive function NeighborConcept
The steps 1, 2, 3 and 4 in algorithm 1 are the initialization steps, followed by a two step canonical test in steps 5 and 6. Step 5 checks for the isValidNeighbor element of each concept to find whether the given concept is able to generate any neighbors. If isValidNeighbor is not able to generate any concepts, then the next concept in the RDD will be picked. If a concept is able to generate neighbors then the intent of the concept is checked to make sure that the algorithm is not processing an already listed concept. The isCanonical () method in algorithm 4 checks the cached concepts and returns true if the intent of the concept does not match any of the concepts intent in the cache. If it matches it will return false. If the second step in the canonical test fails, the SparkConceptGeneration algorithm proceeds with the next concept in the list. After the greatest formal concept is reached, the algorithms stop listing the concepts and writes all the generated concepts to a file. The programming model of SparkConceptGeneration is shown in Fig. 2. For listing all the formal concepts, the context forming operators $\uparrow$ and $\downarrow$ are required. The pseudocode for calculating the context forming operators is discussed in algorithm 2 and 3. For calculating the $\uparrow$ and $\downarrow$ the context and its inverse has to be converted into map. The below pseudocode construct the map and its inverse from the given formal context.

### Algorithm 1: SparkConceptGeneration

**Input**: Concept concept  
**Output**: neighbor

1. $\text{ValidConcept}(\text{concept})$
2. $\text{minRDD} = \text{contextObjectsRDD - context}$.context
3. $\text{val RDD} = \text{contextObjectsRDD - context}$.context
4. $\text{val neighbors} = \text{new}$
5. if $\text{conceptisValidNeighbor}$ then
6. $\text{if isCanonical}($concept.intent$) \text{then}$
7. $\text{parent_index} = \text{parent_index + 1}$
8. for $n$ in $\text{contextObjectsRDD - context}$.context$\text{do}$
9. $\text{B}_1 = \text{context}.\text{context}(n) \text{.objects}(\text{concept})$
10. $\text{if isValidNeighbor} = \text{false}$ then
11. $\text{neighbors} = \text{neighbor}(\text{Context}($A_1$, $B_1$), true,$B_1$).object$\text{.parent}$
12. $\text{neighbors}.\text{cache}$
13. $\text{end}$
14. $\text{end}$
15. $\text{end}$
16. $\text{for neighbor in neighbors do}$
17. $\text{Neighbor} = \text{Context}(\text{concept})$
18. $\text{end}$
19. $\text{end}$

Second step in two step Canonical test

### Algorithm 2: AttributeConceptFormingOperator

1. $\text{Function}: \text{attributeConceptFormingOperator}($concept.attribute$)$
2. $\text{Input}: \text{attribute}$
3. $\text{Output}: \text{all concepts sharing the attribute}$
4. $\text{val } \text{objectRDD} = \text{contextInverseAsMap}.\text{get}(\text{concept}.\text{attribute})$.toSeq
5. $\text{return attributeRDD}$

### Algorithm 3: ObjectConceptFormingOperator

1. $\text{Function}: \text{objectConceptFormingOperator}($concept.object$)$
2. $\text{Input}: \text{object}$
3. $\text{Output}: \text{all attributes sharing the object}$
4. $\text{val attributeRDD} = \text{contextInverseAsMap}.\text{get}(\text{concept}.\text{attribute}).\text{toSet}$
5. $\text{return attributeRDD}$

### Algorithm 4: IsCanonical

1. $\text{Function}: \text{isCanonical}($concept.intent$)$
2. $\text{Input}: \text{intent of the concept, neighbors RDD}$
3. $\text{Output}: \text{True/False}$
4. $\text{val neighbors = getPersistentRDD()}$
5. $\text{val count = neighbors.filter(2.contexts(\text{concept}.\text{intent})).count()}$
6. if count $>$ 0 then
7. $\text{true}$
8. $\text{false}$
9. $\text{true}$
10. $\text{end}$

5.2 Extension of SparkConceptGeneration algorithm to process streaming data

The SparkConceptGeneration algorithm is slightly modified to process the streaming data. The pseudocode discussed in algorithm 5 (named ProcessStreamingDataForFCA) takes the streaming data as input and then divides it into batches. Each batch of data is converted into a RDD and the continuous stream of RDDs creates a DStream. Since algorithm 1 works on static data, the streaming data is collected for certain time interval and then the streaming is stopped. Now transformations are applied on the created data streams to convert them into the desired format. Once the data is ready, the SparkConceptGeneration algorithm continues its execution and generates concepts. Then SparkLatticeConstruction algorithm is executed to construct the concept lattice. Because of the resource limitations, the streaming environment is simulated by reading one of the datasets from the Hadoop Distributed File System (HDFS).
A new Spark streaming context is created in step 2 of algorithm 5, then the data is read from HDFS in batches, a data stream is created and finally written into a file which is the input for the SparkConceptGeneration algorithm. The purpose of adding Spark Streaming is to prove that the proposed works are suitable for streaming datasets.

5.3 Explanation of the SparkLatticeConstruction algorithm

The SparkLatticeConstruction algorithm takes the concept file as an input and constructs the concept digraph using the Spark Graphx module. The pseudocode for the SparkLattice method is discussed in algorithm 6.

```scala
val sc = new SparkContext(new SparkConf().setAppName("graph generation"));
val lines = sc.textFile("NeighborConceptsFile").map {
    line =>
        (Concept(line))
    }
```

The above pseudocode takes the concepts file as an input and converts each line of the file to a concept tuple and stores it into a RDD. Now the concepts are sorted based on the parent index and zipped with a unique index using the zipWithIndex method. A vertex table is constructed by taking the zipped index as vertex id and the concept tuple as its property. Now the difference between the parent_index for each concept will be calculated, and if the difference is equal to 1, then the intent of one concept is checked with the intent of other concept. If the intents have common attributes then an entry into the edge table is added. This process identifies all the edges in the graph and constructs a digraph of concepts using the vertex table and edge table.

Figure 3 explains the programming model of a Spark lattice construction algorithm. The concepts generated by

Algorithm 5: ProcessStreamingDataForFCA

```scala
val sc = new SparkContext(new SparkConf().setAppName("coreName"));
val ssc = new StreamingContext(sparkConf, Seconds(10));
val lines = ssc.textFileStream("hdfs://input/data.txt");
lines.saveAsTextFile("context.txt");
```
the Spark concept generation algorithm are taken from the file and a RDD of concepts is created. The concept RDD is then distributed among workers to determine vertices and edges of the graph. Finally the digraph of the concepts is constructed, which is the desired output.

The SparkConceptGeneration algorithm for generating the formal concepts has the time complexity $G^2 \times M$. The canonical test to verify the intent of the concepts in the cache is $G \times M$ which can be ignored. The SparkLatticeConstruction algorithm for building the graph has the time complexity $v^2 \times e^2$ where $v$ is the total number of vertices and $e$ is the total number of edges.

The execution flows of the algorithms discussed in Sect. 5 is illustrated with a small dataset that has 4 objects and 4 attributes.

**Step 1:** Initially the input context is read from the file and the least formal concept is calculated. The least formal concept for the input context is,

$$(A, B) = (\phi_1, \phi_1) = (\{\phi\}, \{\phi\})$$

**Step 2:** For the least formal concept the flag isValidNeighbor is set to true, size of the intent is 3 and parent_index is set to 1.

**Step 3:** Now the function NeighborConcept in algorithm 1 will be called with the least formal concept as input.

**Step 4:** According to line 2 of algorithm 1

objectSet = \{x_1, x_2, x_3, x_4\}

concept.extent = A = \{\phi\}

minSet = objectSet - concept.extent = \{x_1, x_2, x_3, x_4\} - \{\phi\} = \{\phi\}

**Step 5:** Now the minSet is converted to RDD in line 3 of algorithm 1.

**Step 6:** In line 5 of algorithm 1, the condition concept isValidNeighbor is checked. The default value is set to true for the least formal concept and this condition is passed.

**Step 7:** Now the function isCanonical is called with the intent of the least formal concept as initial parameter in line 6 of algorithm 1. The isCanonical function searches the cache to check that the concept is not generated earlier. The least formal concept is the first concept that is getting processed, and the cache will be empty. In this case isCanonical returns false.

**Step 8:** The parent_index is incremented and the value is set to 2.

**Step 9:** Loop from lines 8 to 18 will run for $x = \{x_1, x_2, x_3, x_4\}$

LoopIteration 1: for $x = x_1$, $B_1 = \{\phi \cup x_1\} = \{a, c\}$, $A_1 = B_1 = \{x_1\}$

$$\min ((A_1 - A) - \{x\}) = \{x_1, x_2, x_3, x_4\} \cap ((\{x_1\} - \{\phi\}) - \{x_1\}) = \emptyset$$

therefore neighbors := ((\{x_1\}, \{a, c\}))

Since the condition in line 11 is passed, the value of isNeighbor is set to true and the neighbor is cached.

LoopIteration 2: Similarly lines 9 to 18 will be repeated for $x = x_2$: $B_1 = \{\phi \cup x_2\} = \{a\}$, $A_1 = B_1 = \{x_1, x_2, x_4\}$

$$\min ((A_1 - A) - \{x\}) = \{x_1, x_2, x_3, x_4\} \cap ((\{x_1, x_2, x_4\} - \{\phi\}) - \{x_1\}) = \emptyset$$

therefore neighbors do not change, the object $x_2$ is removed from min.

the new value of min = \{x_1, x_3, x_4\}

LoopIteration 3: for $x = x_3$: $B_1 = \{\phi \cup x_3\} = \{b, c, d\}$

A_1 = B_1 = \{x_3\}

now $\min ((A_1 - A) - \{x\}) = \{x_1, x_3, x_4\} \cap ((x_3 - \{\phi\}) - \{x_3\}) = \emptyset$

therefore neighbors := ((\{x_1\}, \{a, c\}), (\{x_3\}, \{b, c, d\}))

Since the condition in line 11 is passed, the value of isNeighbor is set to true and the neighbor is cached.

LoopIteration 4: for $x = x_4$: $B_1 = \{\phi \cup x_4\} = \{a, b, d\}$, $A_1 = B_1 = \{x_4\}$

$$\min ((A_1 - A) - \{x\}) = \{x_1, x_3, x_4\} \cap ((x_4 - \{\phi\}) - \{x_4\}) = \emptyset$$

therefore neighbors := ((\{x_1\}, \{a, c\}), (\{x_3\}, \{b, c, d\}), (\{x_4\}, \{a, b, d\}))

© Springer
Since the condition in line 11 is passed, the value of isNeighbor is set to true and the neighbors are cached.

**Step 10:** The lines 8 to 18 are distributed across the cluster and processed parallelly, the generated concepts will be processed again based on the results of conditions in line 5 and 6 of algorithm 1. If the conditions are passed, neighbor concepts will be calculated, otherwise the next concept in the list will be processed.

**Step 11:** This process will be repeated until the highest formal concept is calculated. After the highest formal concept is calculated the output will be written to a file.

The following are the concepts identified by the SparkConceptGeneration algorithm for the formal context in Table 2.

Now the SparkLatticeConstruction algorithm takes Table 3 as input. The vertex table then consists of all the concepts and the difference between each concept is calculated. If the difference is equal to 1, then the intent of the first concept will be compared with the intent of the second concept. For example, from Table 3, the difference between concept number 3 and concept number 6 is one, but there is no common attribute in their intent. This results in failing of the condition in line 8 of SparkLatticeConstruction algorithm; hence, no edge is added between concept 3 and concept 6, which can be seen in the graph represented in Fig. 4.

### 6 Experimental analysis

The algorithms SparkConceptGeneration and SparkLatticeConstruction are implemented using Apache Spark Scala API. The experiments were run on the Google cloud cluster with various configurations of worker nodes ranging from 1 to 16 and tested on the various datasets from the UCI machine learning repository. The three datasets considered for the experiments, ranging from a small Car Evaluation dataset that has 1728 objects to a large Poker-Hand dataset that has half a million objects. The experiments are conducted for five times on the clusters using datasets with worker nodes ranging from 1 to 16 and average of the experiments is documented. The single execution of the experiment produces different results each time the algorithms are executed, hence we considered the average values. The best results were observed when the algorithm executed on 1, 4, 10, and 16 node clusters. In formal concept analysis, generating concepts from the given formal context is the important step, so most of the evaluations are based on the SparkConceptGeneration algorithm and compared with the other distributed works implemented using MapReduce. The following parameters are evaluated as part of the experimental analysis.

#### 6.1 Execution time

Execution time is one of the performance metrics that measures the time required for the generation of concepts and construction of the concept lattice. The Tables 4, 5, 6 and 7 represents the datasets, the number of objects, attributes in the dataset, the number of nodes and the execution time to process each dataset for generation of concepts. The results are depicted for each experiment and the average of the results is given in the last row of Tables 4, 5, 6 and 7. For all the datasets, with an increase in the number of worker nodes, the execution time significantly decreases. A detailed comparison of the proposed work with other distributed implementations is discussed in the sequel.

Tables 4, 5, 6 and 7 represents the execution times of the SparkConceptGeneration algorithm on a cluster that has 1, 4, 10, 16 nodes. For each of the cluster instances, the algorithm is executed on 3 input datasets and the results are observed.

Figure 5 shows the average values(presented in last row of Tables 4, 5, 6 and 7) of the experiments when the algorithm

---

**Table 2** Example formal context

| I   | a | b | c | d |
|-----|---|---|---|---|
| x₁  | X |   |   |   |
| x₂  | X |   |   |   |
| x₃  |   | X | X |   |
| x₄  |   | X |   | X |

**Table 3** The Concepts for the given formal context along with their parent index

| Concept no | Concept extent | Concept intent | parent_index |
|------------|----------------|----------------|--------------|
| 1          | x₁, x₂, x₃, x₄ | φ              | 4            |
| 2          | x₁, x₂, x₄     | a              | 3            |
| 3          | x₃, x₄         | b, d           | 3            |
| 4          | x₁, x₃         | c              | 3            |
| 5          | x₄             | a, b, d        | 2            |
| 6          | x₁             | a, c           | 2            |
| 7          | x₃             | b, c, d        | 2            |
| 8          | φ              | a, b, c, d     | 1            |

**Fig. 4** Lattice Digraph generated for context in Table 2
is executed on different datasets. From the graph in Fig. 5 one can observe that the algorithm is generating concepts quickly with an increase in the number of nodes, i.e., as the number of executors that work in parallel increases, the execution time is decreases. The execution time also depend on the size of the dataset. For smaller datasets, the concepts are generated in a short span. For the Car Evaluation dataset on a 16 node cluster the algorithm has generated 35 concepts in 16 seconds, and for the Poker hand dataset it took 657
Table 8 shows the number of concepts generated for each dataset and the time it took for the construction of the lattice graph. With the increase in the number of nodes, the time for constructing the graph is decreasing. The vertex table and edge table constructed in the SparkLatticeConstruction algorithm are RDDs that are distributed across the worker nodes for distributive construction of the graph.

From the graph in Fig. 6 one can understand that the time taken for construction of lattice graph from the Car Evaluation dataset concepts on a 16 node cluster is 86 seconds and for the Poker-Hand dataset is 578 seconds.

6.2 CPU utilization

CPU utilization is another performance metrics that is evaluated. The results show the average of maximum CPU usage for generating all the concepts. Then the results are compared with the MapReduce approach for concept generation.

The graph in Fig. 7 shows that the Spark is efficiently using CPUs for application processing. With the increase in the number of nodes, the number of CPU cores increase. For the SparkConceptGeneration algorithm, on a single node cluster, the CPU is completely occupied with the application tasks. With increasing of nodes, more CPU is available. The two MapReduce algorithms have not used more than 60% of the CPUs capacity. In the MapReduce implementation, every TaskTracker has the map and reduce slots which are not generic slots. When a MapReduce application starts it may spend hours in map phase. During this time the reducer slots are idle and the CPU% is not very high because of the empty reducer slots. Spark has the concept of tasks which are generic and always tries to use the maximum amount of CPU usage, and thus CPU use is almost 80% in all cases in the SparkConceptGeneration algorithm. This proves that the Spark is efficiently using CPU resources in the cluster to improve the performance of the algorithm.

Figure 8 represents the CPU usage of the Spark Concept generation algorithm while executing on a Google cloud cluster that has 4 worker nodes. The maximum CPU% used on this cluster is 96.67%.

6.3 Memory usage

Memory usage is a vital performance metrics. The results in Fig. 9 shows that the Spark is using maximum memory in every case. The cluster is configured with 64 GB of memory on each node and Spark is using 90% of the memory. Spark is an in-memory processing engine that keeps all of the data and intermediate results in memory. In-memory storage is the main reason for the faster speed of Spark while

| Dataset             | Number of concepts generated | 1 Node cluster | 4 Node cluster | 10 Node cluster | 16 Node cluster |
|---------------------|------------------------------|----------------|----------------|-----------------|-----------------|
| Car evaluation      | 35                           | 746            | 383            | 132             | 86              |
| Adult               | 12678                        | 2976           | 1672           | 763             | 345             |
| Poker hand          | 148726                       | 5634           | 2996           | 1532            | 578             |
processing. In Map Reduce, the intermediate results are written to the HDFS (every time) and the data is read back again from the HDFS. The memory usage in the MapReduce algorithms is less because of the lack of in-memory computations. The HaLoop based MapReduce approach stores data in the cache, but will not do any computations. HaLoop uses more memory than the Twister environment because of the various caches it is supporting. Figure 10 represents the memory usage of the SparkConceptGeneration algorithm while executing on a Google cloud cluster that has 4 worker nodes and 30 GB of memory. The maximum memory used for in-memory processing is around 22 GB and the cache memory used for storing the generated concepts is around 350 MB.

6.4 Comparison of execution times

The execution times of the proposed algorithms are compared with existing related works. There are no works in the literature that used Apache Spark for concept generation, so we have considered the MapReduce implementations for concept generation for the analysis and observed that in all cases the Spark implementation perform better than the existing approaches. The underlying architecture
of Spark significantly reduced the number of iterations to generate concepts when compared with iterative MapReduce approaches like HaLoop and Twister. The in-memory computations greatly reduced the concept generation time by generating a large number of concepts within a short span. For the datasets represented in Tables 9, 10, 11, 12 and 13 represent the execution times of all the three approaches (Spark, Twister and HaLoop) when executed on different ranges of the cluster using the original datasets that are considered in the MapReduce approach.

All the algorithms for concept generation considered for evaluation were implemented using HaLoop and Twister based MapReduce environments. These algorithms are executed on 1 node, 4 node, 10 node and 16 node clusters.
and the graphs in Figs. 11, 12, 13 and 14 compare three approaches. In every case the Spark implementation performed better than the other two approaches. The main reason for the better performance of Spark are its data-level parallelism and in-memory computations. Spark runs its job by spawning different threads running inside the executor. A thread is a lightweight process that runs part of the task, whereas in MapReduce the map and reduce processes are heavyweight. Spark extensively uses the CPU and memory, where as MapReduce implementations fail to do so because of the architectural complexity.

The graph in Fig. 15 (values presented in Table 14) shows the number of iterations that all three algorithms underwent when executed on a 16 node cluster. The numbers of iterations in Spark are determined by RDD partitions and the size of the memory. Spark takes fewer iterations compared with HaLoop and Twister because of data-level parallelism in RDDs. In the HaLoop and Twister environments the number of iterations is determined by the size of the data block, which is 64 MB. The main reason for more iterations in MapReduce is due to its complex architecture. Each iteration runs as an independent MapReduce job. HaLoop and Twister still use the underlying architecture of Hadoop which is not a good design for iterative applications.

The graph in Fig. 16 shows the running time of the algorithm for each iteration. During the first iterations, both

| Table 9 | Datasets for comparison of execution times |
|---------|------------------------------------------|
| Dataset | Objects | Attributes |
| Mushroom | 8124 | 125 |
| Movie | 10,000 | 150 |
| Adult | 45,222 | 337 |
| Census-income | 103,950 | 133 |
| Anon-web | 32,711 | 294 |

| Table 10 | Execution times of Spark, HaLoop and Twister approaches in seconds on the cluster that has one nodes |
|---------|-----------------------------------------------------------------------------------------------|
| Algorithm | Mushroom (Objects: 8124, Attributes: 125) | Movie (Objects: 10,000, Attributes: 150) | Adult (Objects: 45,222, Attributes: 337) | Census-income (Objects: 103,950, Attributes: 133) | Anon-web (Objects: 32,711, Attributes: 294) |
| HaLoop MR concept generation algorithm | 6432 | 7987 | 8792 | 14321 | 5791 |
| Twister MR concept generation algorithm | 9851 | 11,054 | 11,787 | 20,765 | 8619 |
| Spark concept generation algorithm | 1782 | 2198 | 2663 | 3876 | 1943 |

Fig. 11 Comparison of Spark and MapReduce implementations on single node cluster.
Spark and MapReduce approaches took almost the same time. Spark completed later iterations in a very short time by storing all the data in the cache memory and reusing the cached data. The graph in Fig. 16 shows the time taken for the first iterations on the Poker–Hand dataset. With the MapReduce approach, every iteration involves disk access.

Table 11 Execution times of Spark, HaLoop and Twister approaches in seconds on the cluster that has four nodes

| Algorithm                                      | Mushroom | Movie | Adult | Census-income | Anon-web |
|-----------------------------------------------|----------|-------|-------|---------------|----------|
|                                               | Objects: 8124 | Objects: 10,000 | Objects: 45,222 | Objects: 103,950 | Objects: 32,711 |
| HaLoop MR concept generation algorithm        | 764      | 908   | 1286  | 2758          | 1043     |
| Twister MR concept generation algorithm       | 933      | 1178  | 1489  | 3175          | 1361     |
| Spark concept generation algorithm            | 421      | 543   | 745   | 822           | 441      |

Table 12 Execution times of Spark, HaLoop and Twister approaches in seconds on the cluster that has ten nodes

| Algorithm                                      | Mushroom | Movie | Adult | Census-income | Anon-web |
|-----------------------------------------------|----------|-------|-------|---------------|----------|
|                                               | Objects: 8124 | Objects: 10,000 | Objects: 45,222 | Objects: 103,950 | Objects: 32,711 |
| HaLoop MR concept generation algorithm        | 96       | 145   | 246   | 305           | 217      |
| Twister MR concept generation algorithm       | 166      | 237   | 386   | 439           | 323      |
| Spark concept generation algorithm            | 68       | 121   | 197   | 293           | 165      |

Table 13 Execution times of Spark, HaLoop and Twister approaches in seconds for datasets on the cluster that has 16 nodes

| Algorithm                                      | Mushroom | Movie | Adult | Census-income | Anon-web |
|-----------------------------------------------|----------|-------|-------|---------------|----------|
|                                               | Objects: 8124 | Objects: 10,000 | Objects: 45,222 | Objects: 103,950 | Objects: 32,711 |
| HaLoop MR concept generation algorithm        | 76       | 147   | 189   | 273           | 184      |
| Twister MR concept generation algorithm       | 129      | 174   | 264   | 321           | 238      |
| Spark concept generation algorithm            | 36       | 98    | 119   | 235           | 109      |
for reading and writing output to disk after the map and reduce phases, and shuffling and sorting operations. Thus, the time it takes to complete a particular iteration is high. This eventually increases the running time of the application.

In Spark only first iteration takes time for reading input data from disk. In later iterations Spark always tries to take data from cache. This is helpful for low processing times during iteration while executing data on large datasets.

Fig. 13 Comparison of Spark and MapReduce approaches on 10 node cluster

Fig. 14 Comparison of Spark and MapReduce approaches on 16 Node cluster
Fig. 15  Number of iterations that Spark, HaLoop and Twister based environments taken for execution of each dataset on 16 node cluster

Table 14  Number of iterations taken by Spark and Hadoop approaches

| Algorithm                        | Mushroom | Movie | Adult | Census-income | Anon-web |
|----------------------------------|----------|-------|-------|---------------|----------|
| HaLoop MR concept generation     | 66       | 93    | 167   | 225           | 79       |
| Twister MR concept generation    | 97       | 122   | 207   | 295           | 84       |
| Spark concept generation         | 23       | 29    | 43    | 54            | 24       |

Fig. 16  Running time of algorithms under different iteration values
6.5 Evaluation of streaming algorithm

The Car Evaluation dataset is loaded into HDFS and read as datastreams using Spark. It took almost 10 min on a 16 node cluster for reading data from HDFS and creating datastreams. Now the datastreams are written to a file and the SparkConceptGeneration algorithm is executed on the 16 node cluster by taking the generated file as input. The execution time for generating concepts is the same as the execution time shown in Table 7. In future works, the limitations of the streaming algorithm in handling online data streams will be addressed.

6.6 Fault tolerance test

For fault tolerance testing, a node was crashed while the job was running on Car Evaluation dataset after ten iterations on a 4 node cluster; This slows the job by 44 seconds (20% on average). The data partitions on the lost node are recomputed and cached parallel to other nodes quickly with
the help of a lineage graph. The part of the lineage graph of the SparkConceptGeneration algorithm is shown in Fig. 17. During stage 0 the file was read from disk and converted to a RDD. Various transformations and actions are performed on the transformed RDD as part of the concept generation process can be seen in stage 4 and stage 5.

7 Conclusion

There are several works proposed for the generation of concepts and lattice graph construction. The work proposed in this paper can be seen as evidence of computing formal concepts and constructing a lattice graph by isolated nodes. The main drawback of the existing distributed algorithms is acquiring hardware with several processor cores, more efficiently using system resources like memory and CPU, and handling fault tolerance effectively. Also existing batch processing applications do not provide the lattice structure after concept generation. The proposed model overcomes all the drawbacks and effectively uses system resources and builds the digraph of the concepts efficiently. The in-memory computations in Apache Spark helped in generating concepts more quickly and the two step canonicity eliminated the processing of duplicate concepts. With all these benefits, the experimental analysis conducted on the proposed model also proves that it works better than other existing distributed approaches. Furthermore, the core idea behind the implementation of these algorithms is to employ FCA in various domains to get effective results. This paper has introduced the core idea of the distributed concept generation and concept lattice construction algorithms using Apache Spark, but it should be clear that certain extensions are possible. For example, the proposed algorithm can be extended to process dynamically growing contexts, and processing of fuzzy valued and many valued contexts. To handle the dynamic contexts, an incremental variant of concept generation and concept lattice construction algorithms should be proposed. Another possible extension is to identify concepts by taking online streaming data. So future work will focus on developing incremental variants of concept generation and, concept lattice construction algorithms using Spark streaming to process online data streams. Also efficient concept generation in many valued and fuzzy valued contexts. Finding attribute implications in large contexts and extending the scope of FCA to all prominent domains that needs to process large datasets.

References

Alsheikh M, Niyato D, Lin S, Tan H, Han Z (2016) Mobile big data analytics using deep learning and Apache Spark. IEEE Netw 30(3):22–29

Andrews S, Orphanides C (2010) Analysis of large data sets using formal concept lattices. In: Proceedings of the 7th international conference on concept lattices and their applications, Sevilla, Spain, October 19–21

Aswani Kumar Ch (2012) Fuzzy clustering-based formal concept analysis for association rules mining. Appl Artif Intell 26(3):274–301

Aswani Kumar C, Srinivas S (2010) Concept lattice reduction using fuzzy K-Means clustering. Expert Syst Appl 37(3):2696–2704

Aswani Kumar C, Singh Prem, Li J (2014) Concepts reduction in formal concept analysis with fuzzy setting using Shannon entropy. Int J Mach Learn Cybern 8(1):179–189

Belohlavek R (2008) Introduction to formal concept analysis. Internal report. Dept of Computer science. Palack‘y University, Olomouk, Czech Republic

Bhatnagar R, Lalit K (2015) An efficient map-reduce algorithm for computing formal concepts from binary data. In: IEEE international conference on big data

Bordat JP (1986) Calcul pratique du treillis de Galois d’une correspondance. Mathematiques et Sciences Humaines 96(1):31–47

Chunduri R, Kumar CA, Tamir M (2017). Concept generation in formal concept analysis using MapReduce framework. In: International conference on big data analytics and computational intelligence (ICBDAC 2017)

Chunduri R, Kumar CA (2018) Hadoop MapReduce approach for concept generation in formal concept analysis. J Inf Knowl Manag 17(3):1850029

Dmitry I (2017) Introduction to formal concept analysis and its applications in information retrieval and related fields. In: 2014 Russian summer school in information retrieval, Nizhniy Novgorod, Russia. Part of the communications in computer and information science. CCIS vol 505. Springer, pp 42–141

Domoney W F, Ramli N, Alareh S, Stuart DW (2015) Smart city solutions to water management using self-powered, low-cost, water sensors and Apache Spark data aggregation. In: Proceedings of the 3rd international renewable and sustainable energy conference (IRSEC)-2015

du Patrick BR, Bridge D (2006) Collaborative recommending using formal concept analysis. Knowl Based Syst 19(1):309–315

Ferrandin M, Nievola JC, Enembreck F, Scalabrin EE, Kredens KV, Avila B (2013) Hierarchical classification using FCA and cosine similarity function. In: Proceedings of the 2013 international conference on artificial intelligence at LasVegas

Ganter B (1984) Two basic algorithms for formal concept analysis (preprint). Technical Report 831, Technische Hochschule, Darmstadt

Ganter B, Wille R (1994) Formal concept analysis, Mathematical foundations. Springer, Berlin

Gupta G, Manish K (2016) A framework for fast and efficient cyber security network intrusion detection using Apache Spark. Proceed Comput Sci 93:824–831

Joseph EG, Reynolds SX, Ankur D, Crankshaw Daniel, Michael JF, Stoica I (2014) GraphX: graph processing in a distributed dataflow frame. In: OSDI’14 Proceedings of the 11th USENIX conference on operating systems design and implementation, pp 599–613

Kuznetsov S (1993) A fast algorithm for computing all intersections on operating systems design and implementation. Automat Doc Math Linguist 27(5):11–21

Kuznetsov S (1999) Learning simple conceptual graphs from positive and negative examples. Eur Conf Princ Data Min Knowl Discov 1704(1):384–391

Kuznetsov S (2016) Knowledge representation and processing with formal concept analysis. WIREs Data Min Knowl Discov 3(3):200–215
Lindig C (2000) Fast concept analysis. In: International conferences on conceptual structures (ICCS 2000)
Loia V, De Maio C, Fenza G, Senatore S (2012) Hierarchical web resources retrieval by exploiting fuzzy formal concept analysis. Inf Process Manag 48(3):399–418
Loia V, De Maio C, Fenza G, Francesco O (2017) Distributed online temporal fuzzy concept analysis for stream processing in smart cities. J Parall Distrib Comput 110:31–41
Muangprathub J (2014) A novel algorithm for building concept lattice. Appl Math Sci 8(11):507–515
Nilander RM, Sérgio MD, Henrique CF, Luis EZ (2016) Parallelization of the next Closure algorithm for generating the minimum set of implication rules. Artif Intell Res 5(2):40
Norris E (1978) An algorithm for computing the maximal rectangles in a binary relation. Revue Roumaine des Mathematiques Pure ser Appliquees 23(2):243–250
Nourine L, Raynad O (1999) A fast algorithm for building lattices. Inf Process Lett 71(1):199–204
Olga P, Onishchenko A, Gurov S (2013) Classification methods based on formal concept analysis. In: 35th European conference on information retrieval (ECIR 2013) March 24, Moscow
Philipp C, Andreas H, Steffen S (2005) Learning concept hierarchies from text corpora using formal concept analysis. J Artif Intell Res 24:305–339
Priss U (2005) Linguistic applications of formal concept analysis. Formal concept analysis. Springer, Berlin, pp 149–160
Priss U (2007) Formal concept analysis in information science. Annu Rev Inf Sci Technol 40(1):521–543
Qi J, Wei L, Yao Y (2014) Three-way formal concept analysis. In: Proceedings of the international conference on rough sets and knowledge technology, pp 732–741
Shanahan J, Dai L (2015) Large Scale distributed data science using Apache Spark. In: Proceeding KDD ’15 proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining, pp 2323-2324
Singh PK, Kumar CA (2014) Knowledge representation using formal concept analysis: a study on concept generation. In: Global trends in intelligent computing research and development, pp 306-336
Stumme G (2002) Efficient data mining based on formal concept analysis. In: Proceedings of the international conference on database and expert systems applications, pp 534–546
Sumangali K, Kumar CA (2017) A comprehensive overview on the foundations of formal concept analysis. Knowl Manag E-Learn 9(4):512–538
Sumangali K, Kumar CA (2018) Concept lattice simplification in formal concept analysis using attribute clustering. J Ambient Intell Hum Comput 2018:1–17
Václav S, Zdenek H, Kocibova J, Ajith A (2009) Analyzing social networks using FCA: complexity aspects. In: Proceeding of IEEE/WIC/ACM international joint conference on web intelligence and intelligent agent technology
Wassim D, Sami B, Souleiman H, Edward C (2014) Using formal concept analysis for organizing and discovering sensor capabilities. Comput J 58(3):356–367
Xin L, Ming-Wen S (2017) Constructing Lattice based on irreducible concepts. Int J Mach Learn Cybern 8(1):109–122
Xu B, Frézin R, Robson E, Foghli M (2012) Distributed formal concept analysis algorithms based on MapReduce framework. In: International conference on formal concept analysis
Yu J, Wu J, Sarwat M (2016) A demonstration of GeoSpark: A cluster computing framework for processing big spatial data. In: 32nd IEEE international conference on data engineering, ICDE 2016—Helsinki, Finland, pp 1410–1413
Yu J, Ying C, He J (2018) Towards fault tolerance optimization based on checkpoints of in-memory framework spark. J Ambient Intell Hum Comput 2018:1–7
Zadeh RB, Meng X, Staple A, Yavuz B, Pu L, Venkataraman S, Sparks E, Ulanov A, Zaharia M (2016) Matrix computations and optimization in Apache Spark. In: Proceeding KDD ’16 proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, pp 31–38
Zaharia M, Chowdhury M, Franklin M, Shenker S, Stoica I (2010) Spark: cluster computing with working sets. In: HotCloud’10 proceedings of the 2nd USENIX conference on Hot topics in cloud computing
Zaharia M, Chowdhury M, McCauley M, Franklin M, Shenker S, Stoica I, Das T, Dave A, Ma J (2012) Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI’12 Proceedings of the 9th USENIX conference on networked systems design and implementation
Zhao M, Zhang S, Li W, Chen G (2018) Matching biomedical ontologies based on formal concept analysis. J Biomed Semant 9:11

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.