Journal of Big Data

CASE STUDY Open Access

On using MapReduce to scale algorithms for Big Data analytics: a case study

Phongphun Kijsanayothin[1], Gantaphon Chalumporn[2*] and Rattikorn Hewett[2]

*Correspondence:
g.chalumporn@ttu.edu
[2] Department of Computer Science, Texas Tech University, TTU, Lubbock, USA
Full list of author information is available at the end of the article

Abstract

Introduction: Many data analytics algorithms are originally designed for in-memory data. Parallel and distributed computing is a natural first remedy to scale these algorithms to "Big algorithms" for large-scale data. Advances in many Big Data analytics algorithms are contributed by MapReduce, a programming paradigm that enables parallel and distributed execution of massive data processing on large clusters of machines. Much research has focused on building efficient naive MapReduce-based algorithms or extending MapReduce mechanisms to enhance performance. However, we argue that these should not be the only research directions to pursue. We conjecture that when naive MapReduce-based solutions do not perform well, it could be because certain classes of algorithms are not amendable to MapReduce model and one should find a fundamentally different approach to a new MapReduce-based solution.

Case description: This paper investigates a case study of a scaling problem of "Big algorithms" for a popular association rule-mining algorithm, particularly the development of Apriori algorithm in MapReduce model.

Discussion and evaluation: Formal and empirical illustrations are explored to compare our proposed MapReduce-based Apriori algorithm with previous solutions. The findings support our conjecture and our study shows promising results compared to the state-of-the-art performer with 7% increase in performance on the average of transactions ranging from 10,000 to 120,000.

Conclusions: The results confirm that effective MapReduce implementation should avoid dependent iterations, such as that of the original sequential Apriori algorithm. These findings could lead to many more alternative non-naive MapReduce-based "Big algorithms".

Keywords: Big Data analytics algorithms, Association rules mining, MapReduce, Parallel computing

Introduction

Scale adds cost. It also adds complexity and can make even the simplest computing infeasible. Many data analytics algorithms are originally designed for in-memory data. When facing with huge volume of data, these algorithms fail to scale due to limitation of processing capacity, storage capacity and operations on a single machine. Thus, to improve scalability and efficiency, parallel and distributed algorithms are developed to

scale existing algorithms to "Big algorithms" to perform large-scale computing on multiple processors.

While these parallel and distributed algorithms improve the data analytic performance, they also inherit some overheads from input data partition, workloads balancing, increases in communication costs and aggregation of information at local nodes to global information [35]. Because memory consumption is of concern in a single machine environment, most data analytic algorithms aim to optimize in-memory requirements [11]. A recent programming paradigm called *MapReduce* [12] has been developed to address and alleviate these issues. MapReduce provides horizontal scaling to petabytes of data on thousands of commodity servers, an easy-to-understand programming model, and a high degree of reliability when failed nodes occur. It was introduced by Google and has been implemented in many open source platforms including the most widely used, the Apache Hadoop [5]. MapReduce is one of the key enablers to revolutionize the development of many "Big algorithms" by offering a simplified programming model for massive scale distributed data processing on multiple machines. Its popularity is due to its high scalability, fault-tolerance, simplicity and independence from the programming language or the data storage system [16]. Nevertheless, like any approach/paradigm, MapReduce has limitations and there has been researched to understand its limitations, both theoretically (e.g., [1, 18]) and empirically (e.g., [7, 9, 10, 41]).

In scaling existing data analytic algorithms using MapReduce, much research has focused on either building efficient naïve MapReduce-based algorithms that tend to mimic the original analytic solutions on a sequential machine or extending MapReduce framework to enhance performance [16, 33]. However, we argue that these should not be the only research directions to pursue. We conjecture that when naive MapReduce-based solutions do not perform well, it could be because certain classes of algorithms do not have a natural fit to MapReduce model and one should find a fundamentally different approach to a new MapReduce-based solution instead of trying to patch existing solutions.

This paper investigates a case study of a scaling problem of "Big algorithms" for a popular association rule-mining algorithm, particularly the development of the Apriori algorithm in MapReduce model. To illustrate, we explore formal and empirical studies, propose a simple approach to an alternative MapReduce-based solution to the Apriori algorithm and compare its performance with previous solutions in order to gain understanding of this conjecture. Note that, the paper does not address the limitations of MapReduce framework but rather shows that MapReduce-based algorithms do not necessarily yield optimal performance. Our study shows promising results that support the conjecture.

The rest of the paper will transition to the case study that will focus on MapReduce-based solutions for Apriori algorithm in Association Rule Mining. "Apriori algorithms: background and remarks" section discusses related work relevant to this case study, namely parallel and distributed Apriori algorithms especially in MapReduce framework. "Preliminaries" section provides a brief overview of association rule mining, Apriori algorithm and MapReduce. "Case description" section describes naive MapReduce-based Apriori along with *AprioriPMR* [28], a non-naive MapReduce-based solution for Apriori algorithm in more details, and introduces *AprioriS*, our proposed non-naive

solution. *AprioriPMR* has published results showing superior performance to naive MapReduce-based solution (referred to as traditional parallel Apriori algorithm) [28]. We compare our approach and that of *AprioriPMR* with complexity analysis in "Case description" section and shows experimental results on the performance of the two solutions in "Experiments and results" section. "Discussion and evaluation" section provides discussion and conclusion.

Apriori algorithms: background and remarks

Apriori algorithm [3] is a key to association rule mining [2], an algorithm that generates rules that relate set of items based on their frequencies of occurrences in a market basket transaction database. Apriori finds frequent itemsets of incremental set sizes (or levels) up to a maximum number of items in the transactions. The frequent itemsets of size n (n-itemsets) are calculated from those of previous level of size n-1. Many sequential Apriori algorithms and their variations have been proposed to improve scalability by minimizing the number of database scanning operations or using the structure to facilitate efficient storage and counting (e.g., hash-based technique, Partitioning, Transaction reduction, Sampling and Dynamic itemset counting) [8, 32, 34, 40]. However, these sequential algorithms have high I/O cost because of large memory usage and multiple scans [35]. This has led to the development of parallel and distributed Apriori algorithms.

Early parallel Apriori algorithms include Count Distribution, Data Distribution, and Candidate Distribution [4]. The first parallelizes data for counting exchanges, the second parallelizes tasks/processes for data set exchanges, and finally the last is a hybrid of the first two, respectively. While these helps to improve performance, parallel/distributed approaches often add overheads (e.g., from partitioning data, workload balancing, communication and information aggregation). Many parallel Apriori algorithms have been developed to address these issues [39]. These algorithms differ by memory management, load balancing techniques, data decomposition and layout implementations [35].

High overheads of distributed system management, and the lack of fault tolerance, and high-level parallel programming languages make development of parallel algorithms difficult [35]. Recently, the MapReduce framework introduced by Google [12] has been developed to overcome these issues. MapReduce has quickly become popular and wildly get adopted. There are various field of researches that use MapReduce to enhance their performance, for example survey research in health care [14, 29], government [6], sentiment analysis [19], set operations [15], or real-time data analytic [37]. There are researches that focus on state-of-the-art of MapReduce and its applications [20, 24].

Various implementations of Apriori algorithms on MapReduce framework are discussed in [35]. Majority of MapReduce-based Apriori algorithms [21, 23, 26, 27, 31, 38] follow the level-wise process of traditional sequential Apriori where overall execution requires multiple MapReduce phases, each of which finds frequent itemsets of each set size. The next phase will not be able to start until all frequent itemsets of current level is completed. These algorithms vary by implementations inside MapReduce. Some use "Combiner" to aggregate intermediate local results from "Mapper" and some are slightly different by merging first two levels into one and the rest follows the same way [21]. Scheduling invocations and waiting time overheads are another major performance

issue of such algorithms. Much research addresses these issues by either improving the implementations of MapReduce-based Apriori algorithms or extending MapReduce mechanisms. Lin et al. [26] proposed methods to improve execution control in MapReduce to reduce waiting time. However, this somewhat seems to oppose the purpose and benefit of MapReduce framework whose automated process control aims to make it easy for users and developers. More details are described in [35].

There is a small class of MapReduce-based Apriori algorithms [17, 22, 28, 36] that are distinct from all of the above. Each aims to improve the performance over the traditional level-wise sequential or parallel Apriori but because they are focused in different aspects of the development (e.g., cloud storage, intelligent search), they have never been compared to realize their common property. In fact, these algorithms have an important characteristic that their computation of frequent itemsets uses a small number of MapReduce phases and thus are free or almost free of level-wise process. We will discuss more on this later.

Li and Zhang's 1-phase MapReduce-based Apriori [22] is the first such algorithm that requires a single iteration of MapReduce to find all frequent itemsets. Its Map step generates all attainable itemsets in one step (via powerset) instead of a traditional level-wise approach, to obtain frequent itemsets of one set size at a time. The *AprioriPMR* (Powerset MapReduce) algorithm by Moa and Guo [28] employs the same principle using two phases of MapReduce. The first phase produces frequent itemset of size one that are used to generate the rest of frequent itemsets in the second phase. In addition, unlike Li and Zhang's approach that uses a Combiner, an extended Step of MapReduce, to count support in a local machine, *AprioriPMR* does not.

MapReduce-based Apriori by Imran and Ranjan [16] uses powerset and thus, inherits the same characteristic. However, their approach also employs vertical layout and set interaction in additional MapReduce phases to reduce number of scans and overhead. The vertical layout associates each item with a set of transaction containing it whereas a horizontal layout associates each transaction with items in it. The paper focuses on improving its performance over an existing similar approach [13] that uses only vertical layout and set interaction.

In this paper, we propose a simple MapReduce-based Apriori algorithm that requires only one Mapreduce phase in a general MapReduce environment without using a Combiner or other features. Table 1 summarizes this last group of MapReduce-based Apriori algorithms. The core MapReduce refers to MapReduce phases that are used to compute frequent itemsets as opposed to pre-processing like data partitioning. As shown in Table 1, *AprioriPMR* is the closest to our implementation in that it does not require additional features by focusing on using MapReduce environment in the most general sense. In addition, in [28], *AprioriPMR* has empirically shown to outperform traditional (i.e., level-wise) parallel MapReduce-based algorithms. For these reasons, we will focus our comparison studies with *AprioriPMR* in the following sections.

Preliminaries

Terms and problem statement of association rule mining

The terms and notations introduced in this section will be used throughout the rest of the paper. Let $I = \{i_1, i_2, i_3,\ldots,i_m\}$ be a set of items and $D = \{T_1, T_2, T_3,\ldots,T_n\}$ be a database

Table 1 Toward level-wise free MapReduce-based Apriori algorithms

Approach	Core MapReduce	Extended/additional MapReduce	Description
Li and Zhang [22]	One-phase	Yes	Hadoop combiner
Moa and Guo [28]	Two-phase	No	No
Imran and Ranjan [16]	One-phase	Yes	Vertical layout and set intersection
Ours	One-phase	No	No

Summarization of MapReduce-based Apriori algorithms

of customer transactions, each of which is a set of items bought by customers. Thus, $T_i \subseteq I$ for all $i = 1,\ldots,n$. Given two sets of items (or *itemsets*) X and Y with no overlapping items, i.e., $X, Y \subseteq I$ and $X \cap Y = \emptyset$, an *association rule* between X and Y is denoted by $X \rightarrow Y$. We define *support* of X, $sup(X)$, to be a ratio of a number of database transactions that contain X over a total number of transactions in the database D. *Support of rule* $X \rightarrow Y$, denoted by $sup(X \rightarrow Y)$, is defined to be a percentage of transactions in D containing both X and Y.

Confidence of rule $X \rightarrow Y$, denoted by $conf(X \rightarrow Y)$ is defined to be a percentage of transactions in D containing X that also contains Y. That is, $sup(X \rightarrow Y) = |X \cap Y|/|\mathrm{D}|$ and $conf(X \rightarrow Y) = |X \cap Y|/|X|$ in percentages. In other words, $sup(X \rightarrow Y)$ measures a probability that a transaction contains both X and Y while $conf(X \rightarrow Y)$ measures a probability that a transaction that contains X will also contain Y. Given the above contexts, the problem statement of association rule mining is to identify "interesting" association rules, which are typically defined by those whose support and confidence are acceptably high (i.e., at least as high as the user specified thresholds) [2, 3].

Specifically, algorithms for association rule mining has two processes: (1) Finding, from the database D, all possible frequent itemsets (i.e., itemsets whose support is greater than or equal to the minimum support threshold specified by the user), and (2) Generating, from frequent itemsets in previous process, all possible association rules and keep only "strong" rules (i.e., rules whose confidence is greater than or equal to the minimum confidence threshold set by the user).

Apriori algorithm

Apriori algorithm [3] is an important core process of association rule mining. It finds frequent itemsets from a given database of transactions of items. Apriori uses a level-wise approach to find frequent itemsets of different sizes, from itemsets of size one (level one) up to itemsets of size m, a maximum itemset size (level m). From this point on, we will use the term *n-itemset* to refer to an itemset of size n. Frequent itemsets of current level are derived from those of previous level using the Apriori property, which states that all non-empty subsets of a frequent itemset must be frequent Since if $X \subseteq Y$ then $sup(X) \geq sup(Y)$ and thus frequent Y implies frequent X. Similarly, a superset of infrequent itemset must be infrequent.

The algorithm has two main steps: (1) generating candidate itemsets, and (2) testing for frequent itemsets. Specifically, let C_k be a set of "candidate" k-itemsets, and and L_k be a set of (large) "frequent" k-itemsets. In Step 1, to improve efficiency, C_k applies a

self-joining operation, of relational database, to $L_{k\text{-}1}$, i.e., $L_{k-1} \otimes L_{k-1}$ to obtain eligible k-itemsets by pruning away k-itemsets that are supersets of infrequent itemsets using the Apriori property as a pruning principle, see Step 2 tests which of the k-itemsets satisfies the support threshold to be assigned in L_k.

Apriori is an iterative process that continues a level-wise generate-and-test until there is no frequent itemsets in the previous level to be processed further. In the worst case, it reaches level m, the maximum itemset size possible. In fact, m is in other word the maximum size of all transactions in the database (see complexity analysis in "Case description" section). Apriori has been extended to many variations to minimize the number of database scans and numerous techniques to enhance performance for scalability [35, 39].

MapReduce

MapReduce [12] is a highly scalable programming paradigm that enables massive volume of data processing by means of parallel execution on large clusters. Today MapReduce paradigm has been implemented in open source frameworks such as Apache Hadoop [5] and NoSQL MongoDB [30]. MapReduce is a simplified programming model since all the parallelization, communication, load balancing and fault tolerance are automatically handled by framework operations in MapReduce system [12, 35].

Inspired by primitives of functional programming languages such as Lisp, MapReduce uses two main user-defined functions: *Map* and *Reduce*. Both input and output of these functions must be of the form (*key*, *value*) pairs. The reason for this restriction will become clear later. Figure 1 shows a typical form of Map and Reduce functions.

At an initial setup, MapReduce system [12] splits the data into pieces of manageable size, starts up copies of programs on cluster nodes and assigns each idle node a Map or Reduce task. We will refer to a node assigned to a Map task, as a *Map node* where the Map function is executed. A *Reduce node* is defined similarly.

As shown in Fig. 1, the Map function takes an input (*key*, *value*) pair and produces a set of *intermediate* (*key*, *value*) pairs that are buffered in the Map node's local disk. Reduce nodes receive the locations and use remote procedure calls to read buffered data. All intermediate values are sorted and those associated with the same intermediate key are grouped together as (*key, list of values*). The Reduce function takes an intermediate key and its corresponding list of values, merges these values to form a smaller set of values. This continues iteratively until no intermediate pairs to be processed. An *iterator* supplies the intermediate values to Reduce. This enables our ability to handle lists of values that are too large to fit in memory. The output of the Reduce function is appended to the output file. Parallelizing recursive algorithms may require multiple MapReduce

Map: input $\longrightarrow$ (*key*, *value*)

Reduce: (*key*, {$value_1$, $value_2$, ... $value_s$}) $\longrightarrow$ (key, *value-result*)

Fig. 1 A format of basic Map and Reduce functions. Shown a basic data format of Map and Reduce functions

phases. In each MapReduce phase, a computing node may be assigned to do different multiple Map or Reduce tasks before the main Reduce step is completely done. This is why a restricted format of (*key*, *value*) pair needs to be maintained for outputs of both Map and Reduce functions.

Like many distributed systems, MapReduce was designed to reduce the amount of data sent across different computing nodes. To save network bandwidth, the intermediate pair data are read from local disks, and only a single copy of the data is written to local disks. Data processing at the node location helps reduce overhead in moving data. Additionally, reliability is achieved by re-assigning the task of a failed node to another available node.

Understanding MapReduce process can help develop effective MapReduce-based algorithms for Big Data analytics. MapReduce enables Map and Reduce tasks to be executed on different machines in parallel, but we obtain the final result only after all Reduce tasks are completed [35]. Thus, nodes that have finished their Reduce tasks have to wait for other Reduce nodes to complete before they can be released to take other tasks. This also prevents invocation of next MapReduce phase and can result in poor performance.

Case description

This section contrasts two types of MapReduce-based Apriori algorithms: naive and non-naive. The naive type refers to those traditional parallel Apriori solutions that apply MapReduce programming paradigm in a straightforward manner without modifying problem-solving concepts. Instead, it mimics the sequential solution. This type of naive solutions will be described in "Naive MapReduce-based Apriori" section. The non-naive type refers to alternative solutions that do not follow the original sequential process. In particular, we present *AprioriPMR* (PMR for Powerset MapReduce) [28] and our proposed solution called *AprioriS* (S for Simplified) in "Non-naive MapReduce-based Apriori" section followed by "Illustrations" section that provides complexity comparisons between the last two systems.

Naive MapReduce-based Apriori

Naive MapReduce-based Apriori algorithms (e.g., [21, 23, 26, 27, 31, 38]) follow the level-wise process of traditional sequential Apriori. These algorithms typically require multiple MapReduce phases, each of which finds frequent itemsets of each set size. The next phase will not be able to start until all frequent itemsets of current level is completed. To improve performance of Naïve MapReduce-based Apriori some merge first two levels into one and the rest follows incremental level-wise process [21]. Some discards infrequent items in each level [11]. Figure 2 shows a pseudocode of traditional naive MapReduce-based Apriori. There are other variations of the implementations. However, they all share the core characteristics of iterations of solution findings by levels. Each of the Map functions takes each of the input transactions in the corresponding splits (or blocks) as input and executes the Map function opportunistically.

As shown in Fig. 2, the native algorithm starts by finding all frequent 1-itemsets in the first MapReduce phase where the Map function takes input transaction and produces all possible (*key, value*) pairs where *key* is a 1-itemset of the transaction (thus, *value* represents one frequency count for each occurrence). An intermediate

MapReduce Phase 1

Map(*Transaction*)	***Reduce***(*key*, *value*)

// *Transaction* is a set of items
for each item *a* in *Transaction* **do**
output({*a*},1)
end for
	// *key* is a 1-itemset of *Transaction*
	// *value* is a list of numbers, each of which represents a frequency counts of corresponding *key*
	sum = 0
	for each *v* in *value* **do**
	sum = *sum* + *v*
	end for
	output(*key*, *sum*)

Output: A set of frequent 1-itemsets, F_1	

MapReduce Phase 2

Map(*Transaction*)	***Reduce***(*key*, *value*)

// *Transaction T* is a set of items
T_2 = set of all 2-itemsets $\subseteq T$
for each set $S \in T_2$ **do**
$C_S = \{A \mid 1\text{-itemset } A \subseteq S\}$
if $C_S \cap F_1 = C_S$ **then**
output(*S*,1)
end for
	// *key* is a subset of *Transaction*
	// *value* is a list of numbers, each of which represents a frequency counts of corresponding *key*
	sum = 0
	for each *v* in *value* **do**
	sum = *sum* + *v*
	end for
	output(*key*, *sum*)

Output: A set of frequent 2-itemsets, F_2	

…

MapReduce Phase *k*

Map(*Transaction*)	***Reduce***(*key*, *value*)

// *Transaction T* is a set of items
T_k = set of all *k*-itemsets $\subseteq T$
for each set $S \in T_k$ **do**
$C_S = \{A \mid k\text{-1-itemset } A \subseteq S\}$
if $C_S \cap F_{k\text{-}1} = C_S$ **then**
output(*S*,1)
end for
	// *key* is a subset of *Transaction*
	// *value* is a list of numbers, each of which represents a frequency counts of *key*
	sum = 0
	for each *v* in *value* **do**
	sum = *sum* + *v*
	end for
	output(*key*, *sum*)

Output: A set of frequent *k*-itemsets, F_k	

Fig. 2 Basic naive MapReduce-based Apriori. Shown a basic concept of naïve MapReduce-based Apriori of phase 1, phase 2 and phase k, in form of pseudocode

step sorts and collects, from the same *key*, the corresponding values as a list of *values*. The Reduce function sums the values till no more (*key*, *value*) pairs to be processed. Another intermediate step (not shown) prunes the infrequent (*key, value*) pairs that do not meet the support criterion and produces the Frequent 1-itemsets F_1. This completes the Reduce step of MapReduce phase 1. MapReduce phase 2 starts only after the Reduce step of phase 1 finishes.

In phase 2, the Map step takes input transaction and produces all possible (*key, value*) pairs where *key* is a 2-itemset of the transaction where all proper non-empty subsets of the *key* must be frequent. This is to reduce the number of candidate itemsets in current level by using the pruning principle that all subsets of frequent itemsets must be frequent. One way to do this is by determining if all 1-itemsets of each

MapReduce Phase 1

Map(*Transaction*)	***Reduce***(*key*, *value*)

// *Transaction* is a set of items
for each item *a* in *Transaction* **do**
output(*a*,1)
end for
	// *key* is an item of *Transaction*
	// *value* is a list of numbers, each of which represents a frequency counts of corresponding *key*
	sum = 0
	for each *v* in *value* **do**
	sum = *sum* + *v*
	end for
	output(*key*, *sum*)

Output: $F = \{a \mid a \in Transaction \text{ and } \{a\} \text{ is frequent itemset}\}$	

MapReduce Phase 2

Map(*Transaction*)	***Reduce***(*key*, *value*)

// *Transaction* is a set of items
$T = Transaction \cap F$
for each subset *S* of *T* **do**
output(*S*, 1)
end for
// *S* is non-empty subset
	// *key* is a subset of *Transaction*
	// *value* is a list of numbers, each of which represents a frequency counts of corresponding *key*
	sum = 0
	for each *v* in *value* **do**
	sum = *sum* + *v*
	end for
	output(*key*, *sum*)

Output: All frequent itemsets	

Fig. 3 Basic idea of *AprioriPMR*. Shown a basic concept of *AprioriPMR* of Phase 1 and Phase 2, in form of pseudocode

potential candidate 2-itemset S of the transaction are frequent (i.e., $C_s \cap F_1 = C_S$). The rest of this MapReduce phase follows similarly to those of phase 1. Note that the sequential Apriori in [3] generates a candidate set by using self-joining to generate a potential candidate that is a superset of a frequent itemset (since by pruning principle, a superset of infrequent itemset is infrequent and thus, generating such a candidate set will not be fruitful). However, our focus is not to improve naive Apriori by another naive Apriori.

The bottom part of Fig. 2 generalizes MapReduce phase 2 to phase k. The concepts are the same. The Map function checks if all subsets of a potential candidate of current level are frequent using results of frequent itemsets from a previous level. The Reduce functions of all levels are coded the same while the *key* (itemset) in each level has different size implicitly controlled by the Map in each phase.

Non-naive MapReduce-based Apriori

This section describes, in more details, the two non-naive MapReduce-based Apriori Algorithms: *AprioriPMR* (Power-set MapReduce) [28] and our proposed *AprioriS* (Simple). They are non-naive because they do not follow the original traditional Apriori's level-wise process on the sequential machine as illustrated in Section *A*.

AprioriPMR

The *AprioriPMR* applies two phases of MapReduce. The first phase produces a set of items whose frequency of occurrences satisfies the support criterion to be frequent. The

results of the first phase are used to generate the rest of frequent itemsets in the second phase. Figure 3 illustrates our pseudocode of the *AprioriPMR*'s concept.

As shown in Fig. 3, the first phase focuses on identifying frequent 1-itemsets. The concept is the same as that of the first phase of the traditional Apriori but the format of the *key* is slightly different for simplicity. The major difference between the naive MapReduce-based Apriori and the *AprioriPMR* is in *AprioriPMR*'s MapReduce phase 2 where the Map function generates *all* possible subsets of frequent items in each transaction (by means of Powerset). As shown in the Map function of phase 2 in Fig. 3, by using results from phase 1, set T excludes items that do not occur frequently enough. This is crucial to make sure that a subset of T would not be a superset of infrequent itemsets. For example, suppose $T=\{a, b, c\}1$ where only $\{a\}$ and $\{b\}$ are frequent 1-itemsets. If this is the case, Map could generate $S \subseteq T$, say $S=\{b, c\}$, which is useless because S is infrequent (recall, by Apriori principle, every subset of a frequent itemset must be frequent. Therefore, a superset of infrequent itemset is infrequent). The Reduce function works the same way as in other algorithms. Experiments on performance of *AprioriPMR* compared with naive MapReduce-based solution (referred to as traditional parallel Apriori algorithm) have shown significant improvement [28]. However, the authors did not provide explanation or insight of the characteristic that contributes to this improvement other than their proposed mechanism.

AprioriS

Our hypothesis is that the number of iterations or MapReduce phases may impact the performance because each phase has to wait for the previous phase to finish before it can start. This has motivated us to propose *AprioriS*, a simple MapReduce-based Apriori Algorithm that only requires a single phase of MapReduce. Later sections will show that *AprioriS* is not just simple but also turns out to be powerful in its performance on large clusters. Figure 4 shows pseudocode of simple our non-naive *AprioriS*.

As shown in Fig. 4, the proposed *AprioriS* has a single MapReduce phase. The Map step generates (*key*, *value*) pairs where *key* is a subset of the transaction of any size. The Reduce sums the frequency count of each subset. An intermediate step can be added to identify frequent itemsets for any given threshold. The fact that there is no iterative control helps the parallel execution to be more effective.

Map(*Transaction*)	***Reduce***(*key*, *value*)

// *Transaction T* is a set of items
for each subset *S* of *T* **do**
output(*S*, 1)
end for
// *S* is non-empty subset
	// *key* is a subset of *Transaction*
	// *value* is a list of numbers, each of which represents a frequency counts of corresponding *key*
	sum = 0
	for each *v* in *value* **do**
	sum = *sum* + *v*
	end for
	output(*key*, *sum*)

Output: All frequent itemsets	

Fig. 4 Our Proposed *AprioriS*. Shown a basic concept of our proposed non-naïve Apriori, in form of pseudocode

Table 2 Database of transactions

Transaction IDs	Transactions of items
T_1	{a, b, c}
T_2	{a, b}
T_3	{b, c}
T_4	{a, b, c, f}
T_5	{c, d, e}

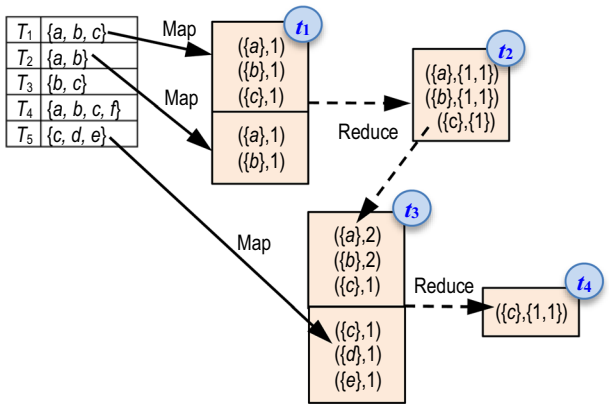

Fig. 5 Partial results of MapReduce executions for 1-itemsets. Illustration of MapReduce execution of 1-itemsets of naïve MapReduce-based Apriori

AprioriPMR and *AprioriS* are closely related. They produce the same identical result, in the other words, have same accuracy. Both are level-free or close to level-free and apply MapReduce environment in the most general sense without additional mechanisms. *AprioriPMR* has empirically shown to outperform traditional (i.e., level-wise) parallel MapReduce-based algorithm [28]. We will now compare these two algorithms to see how a slight difference in the number of iterations and MapReduce phases can impact performance.

Illustrations

This section gives a small example to illustrate how MapReduce-based *AprioriPMR* and *AprioriS* work.

Given a database of transactions shown in Table 2. To illustrate opportunistic characteristic of MapReduce, Fig. 5 shows a partial result of the execution of Map and Reduce functions applied to the database in Table 2 in order to obtain frequent 1-itemsets. To simplify our illustration, we omit showing an explicit data partitioning to each machine.

As shown in Fig. 5, at time t_1, two map functions are executed in parallel to obtain results of form (1-*itemset*, 1). As soon as there is an idle machine ready to work on the corresponding resulting data partition, execution of reduce function starts and produces intermediate results of form (1-*itemset*, *value*) where *value* is an aggregated set of frequency counts as shown at time t_2. The Reduce function continues to finish reducing and obtains the result as shown in the top part of the data shown

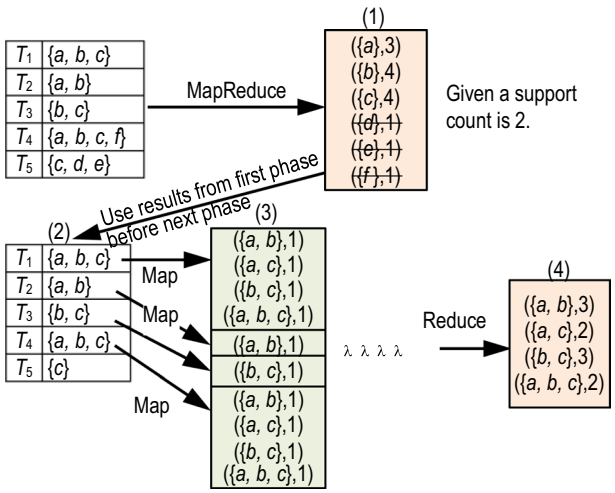

Fig. 6 Partial executions of *AprioriPMR*. Illustration of MapReduce execution of *AprioriPMR*

at time t_3. At the same time, an idle machine executes Map function and obtains the results shown at the bottom part in parallel. As soon as these results at t_3 are obtained and there is an idle machine to work on this data partition, the machine starts working on reducing and obtains the immediate result as shown at time t_4. Note that MapReduce framework gains efficiency by executing Reduce process as soon as all conditions are readily applicable. It has no control to wait till all the mappings to finish before starting to reduce. However, MapReduce in a current phase cannot start until all MapReduce, particularly all reduces, in the previous phase is finished.

The above executions occur in any MapReduce-based level-wise Apriori to find frequent 1-itemsets in the first level. Next we compare how *AprioriPMR* and *AprioriS* work. Figure 6 shows partial executions of *AprioriPMR*.

AprioriPMR has two MapReduce phases. The top part of Fig. 6 shows the results obtained after the first MapReduce phase. Frequent 1-itemsets as shown in box (1) of Fig. 6. These are the same as those of the native level-wise Apriori. The bottom part of Fig. 6 executes the last MapReduce phase after using the results in the first phase to prune infrequent items (box (2)). The Map and Reduce steps continue opportunistically in the same fashion as described in Fig. 5 but for itemsets of all possible sizes. The intermediate results are shown in box (3) and final results for all subsets of size two and beyond in box (4).

Figure 7 illustrates results of *AprioriS* that requires only one single MapReduce phase. Apriori is straightforward. Next we analyze its performance theoretically and empirically.

Complexity analysis

Let n be the number of transactions in the database. Each transaction contains at most m distinct items. For the *AprioriS*, the Map step scans the database and maps each transaction to all possible (*key, value*) pairs, where *key* represents an itemset of the transaction (i.e., a subset of items in the transaction) and *value* represents a list

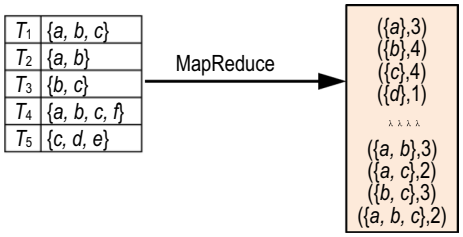

Fig. 7 Execution of *AprioriS* and results. Illustration of MapReduce execution and results of our *AprioriS*

of frequency of occurrences of *key* (which is one when the order pair is first created). Thus, the Map step generates at most $n2^m$ keys. The Reduce step sums the frequencies in a corresponding *value* of the same *key*. Each sum takes constant time and thus, the Reduce step takes $O(n2^m)$. Combining the two steps, *AprioriS* takes $O(n2^m)$.

Comparing with the *AprioriPMR* that consists of a two-phase MapReduce. The first phase produces frequent 1-itemsets (i.e., itemsets of size one). In each transaction, the Map step generates at most m (*key, value*) pairs. Thus, it generates at most nm keys. Similarly, to the Reduce step in our approach, the Reduce step of *AprioriPMR* sums the frequencies in *value* of the pairs with the same key. Thus, it takes at most $O(nm)$ giving a total first phase time complexity of $O(nm)$. In the second phase, *AprioriPMR* takes the results from the first phase, and then uses them as a basis to generate and find the rest of frequent itemsets of sizes two to m at most. Note that the pruning from the first phase does not change the worst case of the remaining number of 1-itemsets to be processed. Thus, similar argument to the MapReduce phase in *AprioriS*, the *AprioriPMR* takes $O(n2^m)$. Adding the two phases, the total time of *AprioriPMR* is $O(nm) + O(n2^m) = O(n2^m)$.

Even though both have the same upper bound, in practice, *AprioriPMR* needs twice as much time for setting up each MapReduce phase compared to that of *AprioriS*. Also, the time *AprioriPMR* spent in the first phase (of $O(nm)$) cannot be absorbed by the second phase as in a theoretical bound. Next will explore empirical results to see if the actual performances match with our observations.

Experiments and results

This section compares the performance of our proposed *AprioriS* with *AprioriPMR* [28]. We implemented both algorithms in MapReduce framework of open source MongoDB [30] version 3.4.9. We ran experiments on 3.40 GHz 4-cores with 6 MB L3 cache CPU, 8 GB DDR3 1600 MHz RAM, and 1 TB SATA hard disk storage.

Table 3 shows a comparison of execution times of *AprioriPMR* and *AprioriS* on a single machine with varying number of transactions from 20,000 to 120,000. Here we set the *sup*, a support threshold criterion to be 5% and m, maximum number of items in each transaction to be 10.

As shown in Table 3, execution times of *AprioriS* outperform those of *AprioriPMR* in all cases. The percentages of the decreased execution times are shown in the last column of the table. On the average, the execution time of *AprioriS* is less than that of *AprioriPMR* by $7.6 \pm 0.05\%$. This is quite different considering how similar they are. We can see

Table 3 Varying number of transactions

Number of transactions	Execution time (s)		Decreased time (%)
	AprioriPMR	Our AprioriS	
10,000	11.17	8.88	20.6
20,000	17.42	15.42	11.5
30,000	25.00	22.54	9.8
40,000	31.68	28.83	9.0
50,000	38.52	35.84	7.0
60,000	43.26	40.62	6.1
70,000	49.50	46.62	5.8
80,000	57.29	54.07	5.6
90,000	63.08	60.87	3.5
100,000	70.06	67.02	4.3
110,000	78.06	74.08	5.1
120,000	81.58	78.85	3.3

Table 4 Varying maximum number items in transactions

m (Max. # of items in transactions)	Execution time (secs)		Decreased time (%)
	AprioriPMR	Our AprioriS	
10	11.17	8.88	20.6
11	17.31	15.31	11.6
12	31.15	28.70	7.9
13	56.99	55.09	3.3
14	146.51	143.40	2.1
15	291.42	288.20	1.1
16	608.08	591.46	2.7
17	1222.37	1215.84	0.5
18	2683.42	2685.14	−0.1
19	Out of memory	Out of memory	NA

from the last column of the table that as the number of transaction increases, the gap differences of the execution times between the two algorithms are smaller. This is consistent with our complexity analysis that the two algorithms have the same asymptotic upper bound where m and n, the number of transactions, are contributing factors. Since m is fixed and so as n grows the execution times of the two are asymptotically the same.

Table 4 compares execution times of the two algorithms on a single machine environment with m, the maximum number of items in transactions, ranging from 10 to 18. Here we fixed n to be 10,000 and support threshold to be 5%.

As shown in Table 4, except for when $m = 18$, *AprioriS* outperforms *AprioriPMR*. Observing the last column, as m increases, the gap differences of the execution times between the two algorithms decrease. This exhibits the same behavior as in the previous experiments and confirms with the theoretical complexity analysis. When $m = 18$, the execution time of *AprioriS* is 2685.14 while that of *AprioriPMR* is 2683.42. Thus, *AprioriPMR* does slightly better. The difference is only about 0.1%. In fact, when m is 19 and beyond, the execution times of both cases ran out of memory.

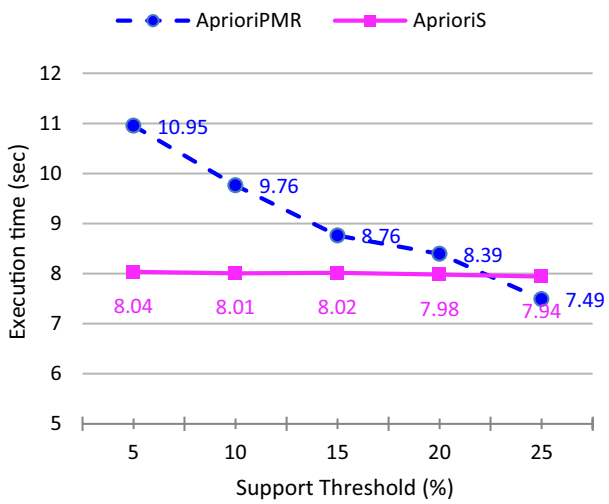

Fig. 8 Varying support thresholds. Result of execution time comparison between *AprioriPMR* and proposed *AprioriS* when vary support thresholds from 5 to 25%

Another important factor that can influence the performance of Apriori is a support threshold *sup*. Figure 7 shows the results obtained when we varied the *sup* with 10,000 transactions and *m*, the maximum number of items in transactions was set to 10.

Figure 8 shows results of the execution times obtained when support threshold percentages change from five to 25. We ran the experiments with 50,000 transactions (*n*) and a maximum of 10 items in each transaction (m) on a single machine environment. As shown in Fig. 8, *AprioriS* outperforms *AprioriPMR* in all cases except for support 25%. As support increases, *AprioriPMR*'s execution time decreases whereas *AprioriS* stays more or less the same. This shows that support threshold impacts the number of pruning elements, which in turn impacts the performance of *AprioriPMR*. In particular, pruning of the 1-itemsets in the first MapReduce phase significantly impacts the execution time of *AprioriPMR*. On the other hand, as shown in Fig. 8, *AprioriS*'s performance does not appear to change much with varying support thresholds. In practice, this can be beneficial since a support threshold is either subjectively specified by users or empirically determined by experiments, which can be time consuming. *AprioriS* frees the users from this burden and enable them to focus on results.

All the results obtained from a single machine environment so far have one common behavior in that as each varying individual variable: *n, m* and *sup* grows larger while the rest are fixed (as in Tables 3, 4 and Fig. 8, respectively), the gap differences that *AprioriS* outperforms *AprioriPMR* decreases and in some case *AprioriPMR* ends up performing slightly better towards the end of the variable's range. This may raise a question whether *AprioriS* will scale well in comparison with *AprioriPMR*. In a large-scale environment, these MapReduce-based Apriori algorithms are expected to use large clusters. Thus, to answer the above question, we present experiments to compare execution times of the two algorithms on multiple machines.

The results of the execution times of *AprioriPMR* and *AprioriS* with support threshold 25%, and a maximum of 10 items in transactions are shown in Fig. 9. Parts (a) and (b) give results obtained from 50,000 to 100,000 transactions, respectively. As shown

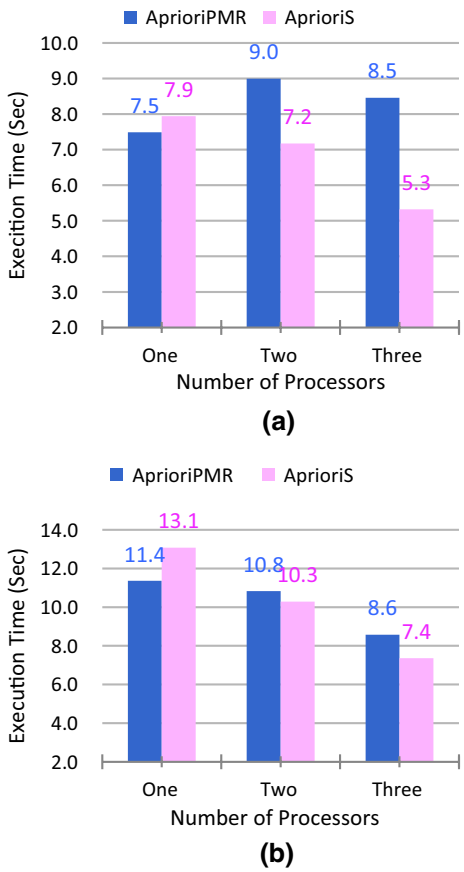

Fig. 9 Comparisons of execution times with multiple machines. Result of execution time comparison between *AprioriPMR* and proposed *AprioriS* on one, two and three machines environments. In part **a**, number of transactions is set to 50,000 while in part **b** number of transactions is set to 100,000

in Fig. 9, for a single machine environment, *AprioriPMR* outperforms *AprioriS* but as number of processors increases, *AprioriS* outperforms *AprioriPMR* by 20.3% for two processors and 37.1% for three processors in case (a), and by 5% for two processors and 14.2% for three processors in case (b). The performance differences are higher when dealing with lower number of transactions [i.e., case (a)]. Another observation is that unlike *AprioriPMR*, the execution time of *AprioriS* decreases as the number of processors increases as expected.

For *AprioriPMR*, however as shown in (a), its execution time for 50,000 transactions increases with two processors. This unexpected increase could be due to the effect of overhead during this particular run. However, as the number of processors changes from two to three, the execution time decreases as expected. On the other hand, when the number of transactions increases to 10,000, as shown in part (b), the effect of the overhead does not outweigh the *AprioriPMR*'s execution time, as it decreases when the number of processors increases as expected. As mentioned above, the results obtained from *AprioriS* do not share the same phenomenon. Thus, to compare the effect of multiple processors, we focus on the rate of improvement from using two processors to three.

In part (a), with 50,000 transactions, *AprioriPMR*'s execution time reduces by about 6% while that of *AprioriS* is by 25.8%. *AprioriS* improves at a much higher percentage rate than AprioriMPR. In part (b), with 100,000 transactions, the execution time of *AprioriPMR* is reduced by 20.9% while that of *AprioriS* is reduced by 28.5%. Although *AprioriS* again improves at a higher percentage rate than AprioriMPR, the gap differences between the two algorithms become narrower (i.e., the difference of 19.8% in case of 50,000 transactions to the difference of 7.6% in case of 100,000 transactions).

AprioriPMR has already been shown to outperform the traditional level-wise MapReduce-based Apriori [28]. Our experiments give evidence that *AprioriS* outperforms *AprioriPMR* in several aspects of scalability and practices although they have only a minor implementation difference. In particular, we reduce from one-level solution in *AprioriPMR* to zero-level solution in *AprioriS*. Thus, even one level of MapReduce-based algorithm can make such a difference on the performance. The experiments also illustrate that the ranking of the performance results obtained in a single machine environment may not carry over in a multi-processor environment. This is due to the design and implementation of the MapReduce-based solution whether it fully utilizes the parallel operating environment.

Discussion and evaluation

The proposed non-naive *AprioriS* algorithm has several advantages. First, it has a simple concept that is easy to understand and implement. The Map and Reduce functions are defined in a straightforward manner. Second, *AprioriS* does not require user to specify a support threshold. This can help expedite the experiments in discovering frequent itemsets by omitting the process of finding appropriate thresholds, which tends to be time consuming in practice. Furthermore, the execution time of *AprioriS* is robust to the support threshold, which can impact the amount of pruning and the resulting itemsets. Thus, given unlimited resources, *AprioriS*'s execution time is less impacted by pruning compared to those of *AprioriPMR* and other naive MapReduce-based Apriori. Third, *AprioriS* solution is complete in that all frequent itemsets will be produced given enough resources in the MapReduce environment. Finally, and most importantly, based on both theoretical and empirical results, *AprioriS* is highly effective in performance while producing the same accuracy. It requires one scan of database and a single phase of MapReduce. In fact, *AprioriS* is level free. This is a distinctive feature that contributes to its effectiveness. As shown in the Complexity Analysis in "Case description" section D, we showed that *AprioriS* and *AprioriPMR* have the same asymptotic upper bound. Thus, theoretically, execution time of both algorithms will differ by some constant factor. This depends on the application and application domain. We have added this clarification in "Discussion and evaluation" section.

In general, implementing an algorithm on MapReduce framework starts by designing two independent Map and Reduce functions that convert the datasets to the form of (*key, value*) pairs to be opportunistically processed in parallel. In MapReduce programming, all the Map and Reduce functions are executed on different machines independently to maximize the use of parallel computation. However, the final result is obtained only after the completion of Reduce step. An algorithm may require single or multiple MapReduce phases. For an algorithm that is recursive, naive implementation is usually translated to multiple phases of MapReduce to obtain the final result. In designing the

MapReduce-based algorithm to fully utilize the parallel computation, it is desirable to minimize the wait between each round of multiple phases of MapReduce (from Reduce step).

MapReduce provides an efficient and simple model to scale algorithms for large computational problems. Like other techniques, MapReduce comes with advantages and limitations depending on which type of problems we are using it for. Previous research [16, 25, 33] has indicated that some data analytic algorithms are inherently parallel and can be adapted to MapReduce paradigm naturally [33]. However, some are not and the transitions of these algorithms to MapReduce paradigm have shown to be much more complex or ineffective [16]. Examples of such algorithms include multiple iterative algorithms, some of which require a chained of data to be processed for convergence or to be updated after each iteration [25, 33]. This clearly adds overhead in communication and data movement. To parallelize these algorithms, we do not necessarily follow the naive MapReduce-based implementation that mimics original sequential processes but to look for alternative solutions that effectively exploit parallelism.

Conclusion

This paper presents a study of the applicability of MapReduce for scaling data analytic and machine learning algorithms to "Big algorithms" for Big Data. Most previous studies have focused on improving naive MapReduce-based algorithms or MapReduce frameworks [1, 16, 18, 39, 41]. Some aim to identify the characteristics of sequential analytic algorithms that are not easily amendable to parallelism [25, 33]. We instead aim to identify the characteristics of parallel algorithms that will maximize the use of parallel computing in MapReduce framework for scaling their performance. We address this issue by examining a family of Apriori algorithms, for finding frequent itemsets, which is a crucial element of association rules mining problem.

We have introduced a small class of MapReduce-based Apriori algorithms that computes frequent itemsets using a small number of MapReduce phases and are free or almost free of level-wise process. Specifically, our study compares the performance of *AprioriPMR*, an existing one-level MapReduce-based Apriori with our proposed *AprioriS*, a simple non-naive level free MapReduce-based algorithm. By mapping each of all possible subsets of items in all transactions (as a *key*) to its corresponding frequency of occurrence (*value*), *AprioriS* requires only a single MapReduce phase and a single database scan while *AprioriPMR* requires two phases.

The findings support our conjecture that to fully exploit parallelism, the MapReduce-based algorithm should be designed to be free of dependent iterations. Here *AprioriS* has no dependent iteration while *AprioriPMR* has one, and *AprioriS* performs better than *AprioriPMR* using multiple machines in MapReduce framework. Our studies use a small number of machines. However, the results obtained so far are promising. Our future work intends to increase the scale of our experiments to a larger scale of clusters. The results confirm that effective MapReduce implementation should avoid dependent iterations, such as that of the original sequential Apriori [3]. These findings could lead to many more alternative non-naive MapReduce-based "Big algorithms".

Abbreviations
ApiroriPMR: Apriori Powerset MapReduce; *AprioriS*: Apriori Simplified; *Conf*: confident; *k*-itemset: itemset of size *k*; *sup*: support.

Acknowledgements
Not applicable.

Authors' contributions
RH designed the study and wrote the draft manuscript. PK performed analysis of MapReduce result. GC implemented the proposed algorithm, performed the experiments and assisted in the analysis of the data. All authors read and approved the final manuscript.

Funding
Not applicable.

Availability of data and materials
The datasets used and/or analyzed during the current study are available from the corresponding author on reasonable request.

Competing interests
The authors declare that they have no competing interests.

Author details
[1] Department of Electrical and Computer Engineering, Naresuan University, NU, Phitsanulok, Thailand. [2] Department of Computer Science, Texas Tech University, TTU, Lubbock, USA.

Received: 13 June 2019 Accepted: 10 November 2019
Published online: 30 November 2019

References

1. Afrati F, Sarma A, Salihoglu S, Ullman J. Vision paper: towards an understanding of the limits of Map-Reduce computation. arXiv:1204.1754v1. 2012.
2. Agrawal R, Imielinski T, Swami A. Mining association rules between sets of items in large databases. In: Proceedings of ACM SIGMOD conf. management of data, Washington, D.C. 1993. p. 207–16.
3. Agrawal R, Srikant R, et al. Fast algorithms for mining association rules. In: Proc. 20th int. conf. very large databases, VLDB, vol. 1215. 1994. p. 487–99.
4. Agrawal R, Shafer JC. Parallel mining of association rules. IEEE Trans Knowl Data Eng. 1996;8(6):962–9.
5. Apache Hadoop. 2019. http://hadoop.apache.org. Accessed 20 Mar 2019.
6. Archenaa J, Anita EM. A survey of big data analytics in healthcare and government. Procedia Comput Sci. 2015;50:408–13.
7. Bhatotia P, Wieder A, Akkus I, Rodrigues R, Acar U. Large-scale incremental data processing with change propagation. HotCloud. 2011.
8. Brin S, Motwani R, Ullman JD, Tsur S. Dynamic Itemset counting and implication rules for market basket data. ACM SIGMOD Record. 1997;26(2):255–64.
9. Bu Y, Howe B, Balazinska M, Ernst M. HaLoop: efficient iterative data processing on large clusters. Proc VLDB Endowm. 2010;3(1–2):285–96.
10. Castro E, Maia T, Pereira, M, Esmin A, Pereira D. Review and comparison of Apriori algorithm implementations on Hadoop-MapReduce and Spark. Knowl Eng Rev. 2018; 33.
11. Chao CM, Chen PZ, Yang SY, Yen CH. An efficient mapreduce-based apriori-like algorithm for mining frequent itemsets from big data. Wireless internet, social informatics and telecommunications engineering. 2018. p. 76–85
12. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–13.
13. Dhanya S, Vysaakan M, Mahesh AS. An enhancement of the MapReduce Apriori algorithm using vertical data layout and set theory concept of intersection. Adv Intell Syst Comput. 2016;385:225–33.
14. Farooqi MM, Shah MA, Wahid A, Akhunzada A, Khan F, ul Amin N, Ali I. Big Data in healthcare: a survey. In: Applications of intelligent technologies in healthcare. 2019. p. 143–52.
15. Fier F, Augsten N, Bouros P, Leser U, Freytag JC. Set similarity joins on mapreduce: an experimental survey. Proc VLDB Endowm. 2018;11(10):1110–22.
16. Grolinger K, Hayes M, Higashino WA, L'Heureux A, Allison DS, Capretz M AM. Challenges for mapreduce in big data. In: Proceedings of IEEE World Congress on Services. 2014. p. 182–9.
17. Imran A, Ranjan P. Improved Apriori Algorithm Using Power Set on Hadoop. In: Proceedings of the first international conference on computational intelligence and informatics, advances in intelligent systems and computing, vol 507, Singapore; 2017.
18. Karloff H, Suri S, Vassilvitskii S. A model of computation for MapReduce. In: Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms. 2010. p 938–48.
19. Khader M, Awajan A, Al-Naymat G. Sentiment analysis based on MapReduce: a survey. In: Proceedings of the 10th international conference on advances in information technology. 2018. p. 11.
20. Khezr SN, Navimipour NJ. MapReduce and its applications, challenges, and architecture: a comprehensive review and directions for future research. J Grid Comput. 2017;15(3):295–321.
21. Kovacs F, Illes J. Frequent Itemset Mining on Hadoop. In: Proceedings IEEE 9th international conference on computational cybernetics (ICCC), Hungry. 2013. p. 241–45.

22. Li L, Zhang M. The strategy of mining association rule based on cloud computing. In: Proceedings IEEE international conference on business computing and global informatization (BCGIN). 2013. p. 29–31.
23. Li N, Zeng L, He Q, Shi Z (2012) Parallel Implementation of Apriori Algorithm based on MapReduce. In: Proceedings 13th ACIS international conference on software engineering, artificial intelligence, networking and parallel & distributed computing, IEEE, p. 236–41.
24. Li R, Hu H, Li H, Wu Y, Yang J. MapReduce parallel programming model: a state-of-the-art survey. Int J Parallel Prog. 2016;44(4):832–66.
25. Lin J. Mapreduce is good enough? If all you have is a hammer, throw away everything that's not a nail! Big Data. 2013;1(1):28–37.
26. Lin MY, Lee PY, Hsueh SC. Apriori-based Frequent Itemset Mining Algorithms on MapReduce. In: Proceedings 6th international conference on ubiquitous information management and communication (ICUIMC '12). New York: ACM; 2012. p. 76.
27. Luna JM, Padillo F, Pechenizkiy M, Ventura S. Apriori versions based on mapreduce for mining frequent patterns on big data. IEEE Trans Cybern. 2017;48(10):2851–65.
28. Mao W, Guo W. An improved association rules mining algorithm based on power set and Hadoop. In: IEEE information science and cloud computing companion (ISCC-C). 2013. p. 236–41.
29. Mauro AD, Greco M, Grimaldi M. Understanding big data through a systematic literature review: the ITMI model. Int J Inform Technol Decis Mak. 2019;18(04):1433–61.
30. Mongo DB. 2019. https://www.mongodb.com. Accessed 20 Mar 2019.
31. Oruganti S, Ding Q, Tabrizi N. Exploring HADOOP as a platform for distributed association rule mining. In: Future computing 2013 the fifth international conference on future computational technologies and applications, 2013. p. 62–7.
32. Park JS, Chen MS, Yu PS. Using a Hash-based method with transaction trimming for mining association rules. IEEE Trans Knowl Data Eng. 1997;9(5):813–25.
33. Parker C. Unexpected challenges in large scale machine learning. Proc. of the 1st international workshop on Big Data, streams and heterogeneous source mining: algorithms, systems, programming models and applications. 2012. p. 1–6.
34. Savasere A, Omiecinski E, Navathe S. An efficient algorithm for mining association rules in large databases. In: Proceedings 21st VLDB conference, Switzerland; 1995. p. 432–44.
35. Singh S, Garg R, Mishra PK. Review of Apriori based algorithms on MapReduce framework. In: Proceedings of the international conference on communication and computing (ICC-2014), Bangalore, India; 2017. p. 593–604.
36. Singh S, Garg R, Mishra PK. Performance optimization of MapReduce-based Apriori algorithm on Hadoop cluster. Comput Electr Eng. 2018;67:348–64.
37. Yadranjiaghdam B, Pool N, Tabrizi N. A survey on real-time big data analytics: applications and tools. In: 2016 international conference on computational science and computational intelligence (CSCI). 2016. p. 404–9.
38. Yang XY, Liu X, Fu Y. MapReduce as a Programming Model for Association Rules Algorithm on Hadoop. In: Proceedings 3rd international conference on information sciences and interaction sciences (ICIS). vol. 99, no 102. 2010. p. 23–5.
39. Zaki MJ. Parallel and distributed association mining: a survey. In: Concurrency, IEEE, 1999. vol 7, no 4, p. 14–25.
40. Zaki MJ, Parthasarathy S, Li W, Ogihara M. Evaluation of Sampling for Data Mining of Association Rules. In: Proceedings IEEE 7th international workshop on research issues in data engineering. 1997. p. 42–50.
41. Zhang Y, Gao Q, Gao L, Wang C. PrIter: a distributed framework for prioritized iterative computations. IEEE Trans Parallel Distrib Syst. 2011;24(9):1884–93.

Publisher's Note