Mining Frequent Patterns from Uncertain Data with MapReduce for Big Data Analytics

Carson Kai-Sang Leung[1,⋆] and Yaroslav Hayduk[1,2]

[1] University of Manitoba, Winnipeg, MB, Canada
[2] Université de Neuchâtel, Neuchâtel, Switzerland
kleung@cs.umanitoba.ca

Abstract. Frequent pattern mining is commonly used in many real-life applications. Since its introduction, the mining of frequent patterns from *precise data* has drawn attention of many researchers. In recent years, more attention has been drawn on mining from *uncertain data*. Items in each transaction of these uncertain data are usually associated with existential probabilities, which express the likelihood of these items to be present in the transaction. When compared with mining from precise data, the search/solution space for mining from uncertain data is much larger due to presence of the existential probabilities. Moreover, we are living in the era of Big Data. In this paper, we propose a tree-based algorithm that uses MapReduce to mine frequent patterns from Big uncertain data. In addition, we also propose some enhancements to further improve its performance. Experimental results show the effectiveness of our algorithm and its enhancements in mining frequent patterns from uncertain data with MapReduce for Big Data analytics.

1 Introduction

Frequent pattern mining aims to discover implicit, previously unknown, and potentially useful knowledge—in the form of frequently occurring patterns—from large amounts of data. Since its introduction [2], the research problem of mining frequent patterns has drawn attention of many researchers. Over the past two decades, numerous methods have been proposed to mine and visualize frequent patterns [8, 20] as well as other related patterns [10, 14, 18, 24]. Examples of these methods include the classical Apriori algorithm [3] and the tree-based FP-growth algorithm [9]. Both algorithms mine frequent patterns from transaction databases of *precise* data.

However, there are situations in which users are uncertain about the presence or absence of some items or events [13, 15]. For example, a physician may highly suspect (but cannot guarantee) that a patient suffers from some specific diseases. The uncertainty of such suspicion can be expressed in terms of *existential probability*. As a concrete example, a physician may suspect that a patient has (i) a 75% likelihood of suffering from a flu and (ii) a $33\frac{1}{3}\%$ likelihood of suffering from

⋆ Corresponding author.

a cold (regardless of having or not having the flu). Here, in this uncertain dataset of patient records, each transaction represents a patient's visit to a physician's office. Note that a patient may suffer from multiple diseases at the same time (i.e., multiple items may appear together in the same transaction). Each item (representing a potential disease) in the transaction is associated with an existential probability expressing the likelihood of a patient having that disease in that visit. With this notion, each item in a transaction in traditional databases of precise data can be viewed as an item with a 100% likelihood of being present in the transaction.

Other examples of uncertain data include datasets of satellite images, where each item in a transaction expresses the likelihood of the presence of an object captured in an image. As the third example, each transaction in a dataset for an election expresses the likelihood of a collection of candidates chosen by (the secret ballot of) a voter. These are just a few examples of many real-life situations in which data are uncertain. Hence, efficient algorithms for mining uncertain data are in demand. Over the past few years, a few algorithms [1, 15–17] have been proposed to mine frequent patterns *serially* from static datasets of uncertain data. Note that the presence of existential probabilities in these uncertain datasets leads to a huge number of possible worlds [13] when using probabilistic-based mining of frequent patterns. In other words, the search space for frequent pattern mining from *uncertain data* can be much larger than that from *precise data*.

The situation has been worsen as we have moved into the era of *Big Data* [23]. When mining from vast amounts of Big Data, more efficient approaches (besides *serial* approach) are needed. To handle Big Data, some researchers proposed the use of *MapReduce*, which mines the search space with distributed or parallel computing. However, earlier works on MapReduce focused on data processing [7] or data mining tasks other than frequent pattern mining (e.g., clustering [5], outlier detection [11], structure mining [27]). Although two recent works [21, 25] were proposed to mine frequent patterns, both of them mine *precise* data (instead of *uncertain* data).

Hence, some natural questions to ask are: Can we use MapReduce to mine *uncertain* data? Can we use MapReduce to perform tree-based mining of uncertain data? How can we further speed up the mining process? In response to these questions, we propose a tree-based algorithm called **MR-growth**, which uses **M**ap**R**educe to mine frequent patterns from uncertain data in a pattern-**growth** fashion for Big Data analytics. Moreover, our additional *key contributions* of this paper include the three enhancements to the MR-growth algorithm.

This paper is organized as follows. The next section gives background and related work. In Section 3, we propose our MR-growth algorithm for mining frequent patterns from uncertain data using MapReduce. Section 4 discusses three enhancements to our MR-growth algorithm. Evaluation results and conclusions are presented in Sections 5 and 6, respectively.

2 Background and Related Work

In this section, we provide background on frequent pattern mining from uncertain data and on MapReduce. We also discuss some related works.

2.1 Mining Frequent Patterns from Uncertain Data

When using probabilistic-based mining [6, 15, 19] with the "possible world" interpretation [13], a pattern is considered *frequent* if its expected support is no less than the user-specified *minsup* threshold. When items within a pattern X are independent, the *expected support* of X in the database DB can be computed by summing (over all transactions $t_1, \ldots, t_{|DB|}$) the product (of existential probabilities within X):

$$expSup(X) = \sum_{i=1}^{|DB|} \left(\prod_{x \in X} P(x, t_i) \right), \tag{1}$$

where $P(x, t_i)$ is an existential probability of item x in transaction t_i. With this definition of expected support, the existing tree-based UF-growth algorithm mines frequent pattern from uncertain data as follows. The algorithm first scans the dataset once to compute the expected support of all domain items (i.e., singleton itemsets). Infrequent items are pruned as their extensions/supersets are guaranteed to be infrequent. The algorithm then scans the dataset the second time to insert all transactions (with only frequent items) into an UF-tree. Each node in the UF-tree captures (i) an item x, (ii) its existential probability $P(x, t_i)$, and (iii) its occurrence count. At each step during the mining process, the frequent patterns are expanded recursively.

2.2 The MapReduce Programming Model

MapReduce [7] is a high-level programming model for processing vast amounts of data. Usually, MapReduce uses parallel and distributed computing on clusters or grids of nodes (i.e., computers). The ideas behind MapReduce can be described as follows. As implied by its name, MapReduce involves two key functions: "map" and "reduce". The input data are read, divided into several partitions (subproblems), and assigned to different processors. Each processor executes the *map function* on each partition (subproblem). The map function takes a pair of $\langle key, value \rangle$ data and returns a list of $\langle key, value \rangle$ pairs as an intermediate result:

$$\text{map: } \langle key_1, value_1 \rangle \mapsto \text{list of } \langle key_2, value_2 \rangle,$$

where (i) key_1 & key_2 are keys in the same or different domains, and (ii) $value_1$ & $value_2$ are the corresponding values in some domains. Afterwards, these pairs are shuffled and sorted. Each processor then executes the *reduce function* on (i) a single key value from this intermediate result together with (ii) the list of all values that appear with this key in the intermediate result. The reduce function "reduces"—by combining, aggregating, summarizing, filtering, or transforming—the list of values associated with a given key (for all k keys) and returns a list of k values:

$$\text{reduce: } \langle key_2, \text{list of } value_2 \rangle \mapsto \text{list of } value_3,$$

or returns a single (aggregated or summarized) value:

$$\text{reduce: } \langle key_2, \text{list of } value_2 \rangle \mapsto value_3,$$

where (i) key_2 is a key in some domains, and (ii) $value_2$ & $value_3$ are the corresponding values in some domains. Examples of MapReduce applications include the construction of an inverted index as well as the word counting of a document.

2.3 Related Work

Earlier works on MapReduce focused either on data processing [7] or on some data mining tasks other than frequent pattern mining (e.g., outlier detection [11], structure mining [27]). Recently, Lin et al. [21] proposed three Apriori-based algorithms called SPC, FPC and DPC to mine frequent patterns from *precise data*. Among them, SPC uses single-pass count to find frequent k-itemsets at the k-th pass of the database scan (for $k \geq 1$). FPC uses fixed-passes combined-counting to find all k-, $(k+1)$-, ..., $(k+m)$-itemsets in the same pass of database scan. On the one hand, this fix-passes technique fixes the number of required passes from K (where K is the maximum cardinality of all frequent itemsets that can be mined from the precise data) to a user-specified constant. On the other hand, due to combined-counting, the number of generated candidates is higher than that of SPC. In contrast, DPC uses dynamic-passes combined-counting, which takes the benefits of both SPC and FPC by taking into account the workloads of nodes when mining frequent itemsets with MapReduce. Like these three algorithms, our proposed MR-growth algorithm also uses MapReduce. However, unlike these three algorithms (which mine frequent itemsets from *precise data* using the *Apriori-based approach*), our proposed MR-growth algorithm mine frequent itemsets from *uncertain data* using a *tree-based approach*. Note that the search/solution space for frequent pattern mining from uncertain data is much larger than frequent pattern mining from precise data due to presence of the existential probabilities.

Riondato et al. [25] proposed a parallel randomized algorithm called PARMA for mining *approximations* to the top-k frequent itemsets and association rules from *precise data* using MapReduce. Although PARMA and our MR-growth algorithm both use MapReduce, one key difference between the two algorithms is that we aim to mine *truly frequent* (instead of approximately frequent) itemsets. Another key difference is that we mine from *uncertain data* (instead of precise data).

3 Our MR-Growth Algorithm for Mining Frequent Patterns from Uncertain Data with MapReduce

In this section, we propose our MR-growth algorithm, which uses MapReduce to mine frequent patterns from huge amounts of uncertain data in a tree-based pattern-growth fashion. The algorithm can be divided into multiple stages.

First, MR-growth reads a huge dataset of uncertain data. As each item in the dataset is associated with an existential probability, MR-growth aims to compute the expected support of all domain items (i.e., singleton itemsets) by using MapReduce. The expected support of any itemset can be computed by using Equation (1). Moreover, when computing singleton itemsets, such an equation can be simplified to become the following:

$$expSup(\{x\}) = \sum_{i=1}^{|DB|} P(x, t_i), \tag{2}$$

where $P(x, t_i)$ is an existential probability of item x in transaction t_i. Specifically, MR-growth divides the uncertain dataset into several partitions and assigns them to different processors. During the *mapping phase* of this stage, the mapper function receives ⟨transaction ID, content of that transaction⟩ as input. For every transaction t_i, the mapper function emits a $\langle key, value \rangle$ pair for each item $x \in t_i$.

What should be the emitted pair? A naive attempt is to emit $\langle x, 1 \rangle$ for each occurrence of $x \in t_i$. It would work well when mining *precise* data because each occurrence of x leads to an actual support of 1. In other words, occurrence of x is the same as the actual support of x when mining precise data. However, this is *not* the case when mining *uncertain* data. The occurrence of x can be different from the expected support of x when mining uncertain data. For instance, consider an item a with existential probability of 0.9 that appears only in transaction t_1. Its expected support may be higher than item b that appears seven times but with an existential probability of 0.1 in each appearance. Then, $expSup(\{a\}) = 0.9 > 0.7 = expSup(\{b\})$. Hence, instead of emitting $\langle x, 1 \rangle$ for each occurrence of $x \in t_i$, MR-growth emits $\langle x, P(x, t_i) \rangle$ for each occurrence of $x \in t_i$. In other words, the *mapper* function can be specified as follows:

```
For each transaction $t_i \in$ partition of the uncertain dataset do
    for each item $x \in t_i$ do
        emit $\langle x, P(x, t_i) \rangle$.
```

This results in a list of $\langle x, P(x, t_i) \rangle$ pairs for many different x and $P(x, t_i)$. Afterwards, these pairs are shuffled and sorted. Each processor then executes the reduce function on the shuffled and sorted pairs to obtain the expected support of x. In other words, the *reducer* function can be specified as follows:

```
Set $expSup(x) = 0$;
For each $x \in \langle x$, list of $P(x, t_i) \rangle$ do
    for each $P(x, t_i) \in \langle x$, list of $P(x, t_i) \rangle$ do
        $expSup(x) = expSup(x) + P(x, t_i)$.
    emit $\langle x, expSup(x) \rangle$.
```

Table 1. A sample transaction dataset of uncertain data

TID	Itemsets
t_1	$\{a$:0.5, b:0.5, c:1.0, d:1.0, u:0.5$\}$
t_2	$\{a$:0.5, b:0.5, p:0.5$\}$

Example 1. Let us consider an uncertain dataset as shown in Table 1 with *minsup*=1.0 For the first transaction t_1, the mapper function outputs $\langle a, 0.5\rangle$, $\langle b, 0.5\rangle, \langle c, 1.0\rangle, \langle d, 1.0\rangle, \langle u, 0.5\rangle$. Similarly, for the second transaction t_2, the mapper function outputs $\langle a, 0.5\rangle, \langle b, 0.5\rangle, \langle p, 0.5\rangle$. These pairs are then shuffled and sorted. Afterwards, the reducer reads $\langle a, [0.5, 0.5]\rangle$, $\langle b, [0.5, 0.5]\rangle$, $\langle c, [1.0]\rangle$, $\langle d, [1.0]\rangle, \langle p, [0.5]\rangle, \langle u, [0.5]\rangle$ and outputs $\langle a, 1.0\rangle, \langle b, 1.0\rangle, \langle c, 1.0\rangle, \langle d, 1.0\rangle, \langle p, 0.5\rangle$, $\langle u, 0.5\rangle$ (i.e., items and their corresponding expected support). □

Next, MR-growth reads the singleton items and their associated existential supports, and prunes the infrequent items. Then, it splits the list containing frequent singletons into distinct groups and assigns a unique ID to each group. The new list containing group-to-singleton mappings is called a group list (G-list). To summarize, this stage identifies which conditional trees should be mined together on one computing node.

Example 2. Let us continue with our example. At this stage, MR-growth prunes items u and p because their existential support equals to $0.5 < 1.0$=*minsup*. Given that we want to split the remaining (frequent) items a, b, c and d into two groups (the number of items which are mapped to a given group can be determined automatically depending on the number of computing nodes), this stage yields $\langle Group_1$: $a, b\rangle$ and $\langle Group_2$: $c, d\rangle$. □

The next stage is an important and computationally intensive stage. Here, MR-growth identifies all group-dependent transactions. First, on each machine executing the mapper functions, MR-growth loads the G-list into main memory, and creates a reverse map that maps singletons to their corresponding group ID. Then, each mapper receives $\langle key = groupID,\ value = DB(groupID)\rangle$ as input. For every transaction t_i in $DB(groupID)$, MR-growth substitutes all transaction items with their corresponding group IDs from the reverse map, creating a new list I of the same size as the transaction size. For each groupID in I, MR-growth locates the rightmost appearance L of $groupID$ in I, and emits a new (truncated) transaction, in the form of $\langle key' = groupID,\ value' = t_i[1]t_i[2] \ldots t_i[L]\rangle$.

Afterwards, MR-growth receives group-dependent transactions in the form of $\langle key = groupID,\ value = \{t_1 \ldots t_n\}\rangle$ and inserts them into a tree, creating a compressed tree-based representation of these group-dependent transactions. MR-growth then collects the group-dependent UF-trees and merges them into one single tree, from which frequent patterns can be mined.

Example 3. Let us continue with our example. After replacing the transaction items with their corresponding group IDs, we get the two lists as summarized in Table 2.

Table 2. Group lists

Group List	groupIDs
1	$\langle 1, 1, 2, 2\rangle$
2	$\langle 1, 1\rangle$

For the first transaction t_1, the rightmost appearance of groupID 1 is 2 (indicating the appearance of a, b ends in the 2nd position of transaction t_1). So, $\langle 1, [a{:}0.5, b{:}0.5]\rangle$ is emitted. Similarly, the rightmost appearance of groupID 2 is 4 (indicating the appearance of c, d ends in the 4th position of transaction t_1). So, $\langle 2, [a{:}0.5, b{:}0.5, c{:}1.0, d{:}1.0]\rangle$ is emitted. In a similar fashion, for the second transaction t_2, the rightmost appearance of groupID 1 is 2 (indicating the appearance of a, b ends in the 2nd position of transaction t_2). So, $\langle 1, [a{:}0.5, b{:}0.5]\rangle$ is emitted.

MR-growth then merges both group-dependent transactions in the UF-tree. Notice that we can merge the two received group-dependent transactions into one branch of the UF-tree for $Group_1$: $(a{:}0.5){:}2$ and $(b{:}0.5){:}2$ when using the (*item*:*probability*):*count* notation. For $Group_2$, MR-growth receives $\langle 2, [a{:}0.5, b{:}0.5, c{:}1.0, d{:}1.0]\rangle$ as input and inserts the single group-dependent transaction into a new UF-tree. To summarize, the algorithm emits $\langle 1, [(a{:}0.5){:}2, (b{:}0.5){:}2]\rangle$ and $\langle 2, [(a{:}0.5){:}1,(b{:}0.5){:}1,(c{:}1.0){:}1,(d{:}1.0){:}1]\rangle$.

Afterwards, for $Group_1$, the reducer function receives one UF-tree in the $\langle 1, [(a{:}0.5){:}2,(b{:}0.5){:}2]\rangle$ format, mines patterns having items a and b from that tree, but it does not discover any *frequent* patterns. Similarly, for $Group_2$, the reducer function receives $\langle 2, [(a{:}0.5){:}1, (b{:}0.5){:}1, (c{:}1.0){:}1, (d{:}1.0){:}1]\rangle$ as input, mines that tree for patterns having items c and d, and emits $\langle \{c, d\}{:}1.0\rangle$ as the only frequent pattern. □

4 Enhancements to MR-Growth

While our proposed MR-growth algorithm efficiently mines frequent patterns from uncertain data, we propose three enhancements in this section to further speed up the mining process.

4.1 Enhancement #1: Multi-core Processors in the ForkJoin Framework

To increase the mining speed, we exploit machines having multi-core processors by using the *ForkJoin framework* [12]. The main goal of the ForkJoin framework is to split computationally intensive tasks into multiple pieces, which can then be performed in parallel, to minimize the execution time of the algorithm. Unlike MapReduce (in which the developer does *not* need to explicitly control the work distribution process), ForkJoin requires the developer to explicitly control the work distribution process. In the Java programming language, the ForkJoin framework also uses the concept of *work stealing*, where the thread (which completes the work assigned to it) can *steal* tasks from other threads and assign the tasks to idle threads as efficiently as possible.

Each thread maintains a task queue and repeatedly takes the next available task from the head of its queue until the task queue becomes empty. Each time when a thread does not have any pending work to complete (i.e., its queue is empty), the thread becomes a *thief*. It selects a different thread at random, and tries to *steal* a task from the tail of the queue of the chosen thread. Once the task is completed, this process is repeated (i.e., steal a task from the tail of the queue of some random thread, which can be the previously selected one).

To summarize, we enhance our MR-growth algorithm by taking the following steps:

1. detect the number N_{cores} of processing cores on a multi-core processor;
2. divide the list of group-dependent items G_i into approximately equal parts such that each thread (running on a different processing core) is responsible for mining $\frac{|G_i|}{N_{cores}}$ items, and insert these items into the queue of each thread;
3. when any given thread finishes constructing and mining conditional trees for all its assigned singletons, the algorithm attempts to steal a singleton from the queue of a random thread.

This enhancement is particularly beneficial in MapReduce infrastructures having a limited number of computing nodes. Each group ID is mapped to many singleton items. Hence, each computing node is responsible for the construction and the mining of conditional trees for a number of domain items (i.e., singleton itemsets).

4.2 Enhancement #2: Efficient Conditional Tree Construction

In this section, we discuss the next enhancement, which allows us to construct conditional trees without constructing projected trees first. Recall that to construct a conditional tree, the MR-growth algorithm first constructs a projected tree and then prunes the locally-infrequent nodes from it to create a conditional tree.

Our enhancement for conditional tree construction to our proposed MR-growth algorithm can be described as follows. The conditional tree is constructed using two traversals of the main tree. The first bottom-up traversal accumulates the counts of all encountered items on the path, flagging all visited nodes. Then, by traversing the same path again but *top-down*—which can be accomplished by recursively visiting only the flagged child nodes, the second scan traverses the main tree in a depth-first manner to build a conditional tree. Specifically, the algorithm performs the following steps:

1. For each item x in the header table, traverse the tree bottom-up and count the occurrence of each encountered item on the tree path;
2. If the parent node of the current node has more than one child, flag the current node with item x (i.e., `childNode.flag`$=x$);

Table 3. A sample transaction database of precise data

TID	Itemsets	(Ordered) Frequent Itemsets
t_1	$\{a, b, c, d, u\}$	$\{a, b, c, d\}$
t_2	$\{a, b, p\}$	$\{a, b\}$
t_3	$\{a, j, b, c, i, d\}$	$\{a, b, c, d\}$
t_4	$\{g, a, n, b\}$	$\{a, b\}$
t_5	$\{l, a, b, m, c\}$	$\{a, b, c\}$
t_6	$\{a, c, q, t, u, g, w, d\}$	$\{a, c, q, t, u, w, d\}$
t_7	$\{h, a, q, t, u, w\}$	$\{a, q, t, u, w\}$
t_8	$\{b, c, q, u, t, w, k\}$	$\{b, c, q, t, u, w\}$

3. Determine which items are locally frequent, and insert them into the new header table, which will be associated with the conditional tree;
4. Traverse the tree again in a top-down fashion. When a node with multiple children is encountered, visit each of the children and flag it with item x;
5. For each visited node, check if it is frequent. Add the frequent nodes to the new conditional tree; and
6. Stop traversing the current branch if all children of the current node are guaranteed to be infrequent.

Example 4. Let us consider the dataset shown in Table 3. Without loss of generality, when building the $\{d\}$-conditional tree, MR-growth with Enhancement #2 first traverses each $\{d\}$-link (circled nodes denote $\{d\}$-link nodes) bottom-up. Then, it accumulates the count of the encountered items in the header table. For nodes having multiple children, it flags each child node with $\{d\}$. Fig. 1 demonstrates the process of building a $\{d\}$-conditional tree. Nodes visited during the first traversal are surrounded by squares, and nodes visited during the second traversal are bolded. During the second traversal, we can stop the traversal of both branches early (e.g., after visiting a node with item c) because, thanks to the information collected during the first traversal of the tree, we know that any items after item c in *any* path are guaranteed to be infrequent.

Using the tree in Fig. 1, let us compute the amount of allocated memory and calculate the number of visited node required by the original version of MR-growth vs. the version enhanced by this efficient conditional tree construction. During the first bottom-up traversal, MR-growth (w/ Enhancement #2) visited 9 nodes in the main tree and did not allocate any new nodes. Then, MR-growth (w/ Enhancement #2) traversed the tree once again in the top-down fashion. It visited 4 nodes in the main tree and allocated 2 nodes for the new conditional tree. To summarize, MR-growth (w/ Enhancement #2) traversed 13 nodes and allocated 2 new nodes; it did not perform any memory deallocations. In contrast, the original version of MR-growth visited 9 nodes (in the main tree) + 8 nodes (in the projected tree) = 17 nodes as well as allocated 9 new nodes for the projected tree, 7 of which needed to be deallocated in the conditional tree. □

As observed from the above example, the benefits of employing Enhancement #2 include the following:

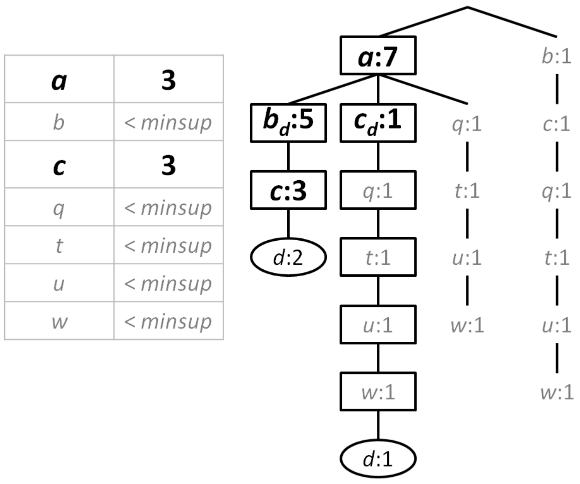


Fig. 1. The efficient construction of a $\{p\}$-conditional tree

1. its efficient tree node allocation, which avoids the need of (i) allocating memory for infrequent nodes in a projected tree and (ii) freeing it when pruning a projected tree or a conditional tree;
2. its efficient tree traversal, which visits all of the tree nodes only twice in the worst case.

4.3 Enhancement #3: Sampling

Sampling is a commonly used technique in many data mining tasks, especially when trying to find an *approximate* solution (instead of an accurate one). It was observed [26] that the UF-tree is usually bigger than the FP-tree because the former captures both items and their existential probabilities from the uncertain datasets whereas the latter captures only the items from the precise databases. As the tree gets bigger, it takes longer to build and traverse. Consequently, mining with UF-trees usually takes longer than mining with FP-trees.

The idea of our Enhancement #3 is that, instead of building a UF-tree during the mining process, we adopt the Concatenating Sample method [4]. For every $\langle x, P(x, t_i)\rangle$-pair in each transaction t_i, we generate a random real number r in the range (0,1]. If r is $\geq P(x, t_i)$, then we include x in the current sample. Otherwise (i.e., when $r < P(x, t_i)$, we omit x from the current sample. At the end of this sampling process, we obtain a "possible" world instance for the uncertain dataset. For example, $t_1 = \{a, b, d\}$ and $t_2 = \{b, p\}$ can be one such "possible" world instance. We can then mine frequent patterns from such an instance in the same way that we mine frequent patterns from precise data (e.g., using FP-growth [9]).

As one sample is subject to bias, we repeat the above process to obtain a few samples and mine frequent patterns from each of these samples. Given these samples, we apply the enhanced version of MR-growth to mine frequent patterns. As we are dealing with "possible" world instances in the same way that we mine precise data, we need to modify the mapper and reducer function for this enhancement. For example, instead of letting mapper emit $\langle x, P(x, t_i)\rangle$, we modify the mapper to emit $\langle x, 1\rangle$.

As this mining process gives an approximate solution, a post-processing step (which requires one more—i.e., the third—scan of the dataset). So, we also need to modify the reducers to execute this post-processing step.

5 Experimental Results

In this section, we evaluate our proposed MR-growth algorithm and its enhancements. Experiments were run using either a single machine or the Amazon EC2 cluster. Specifically, some experiments were executed on a machine with an Intel Core i7 4-core processor (1.73 GHz) and 8 GB of main memory, running a 64-bit Windows 7 operation system. All versions of the algorithm were implemented in the Java programming language. The stock version of Apache Hadoop 0.20.0 was used. As for the datasets for experiments, we used those benchmarks (e.g., `accidents`, `connect4` and `mushroom`) from the UCI Machine Learning Repository (`http://mlearn.ics.uci.edu/MLRepository.html`) and the FIMI repository (`http://fimi.ua.ac.be`). Some other experiments were run on the Amazon EC2 cluster—specifically, 11 `m2.xlarge` computing nodes (`http://aws.amazon.com/ec2`). As Calders et al. [4] suggested that two samples per transaction were sufficient to approximate (i.e., with less than 0.02% error) the expected supports of the mined patterns for most datasets, we also used two samples per transaction in the experiments.

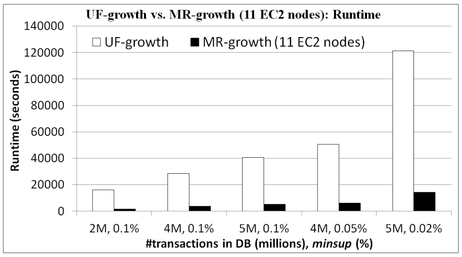
In addition to the above real-life benchmark datasets, we also generated three new synthetic datasets using the IBM Quest Dataset Generator [3] for our evaluations. The generated data ranges from 2M to 5M transactions with an average transaction length of 10 items from a domain of 1K items. As these datasets originally contained only precise data, we assigned to each item contained in every transaction an existential probability from the range (0,1].

5.1 Evaluation of MR-Growth

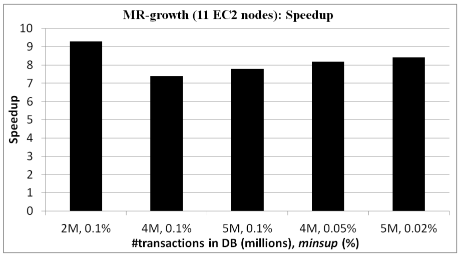
In this experiment, we executed our MR-growth algorithm in the MapReduce environment with 11 nodes. Fig. 2(a) shows that, while the sequential version of the UF-growth algorithm took more than 120,000 seconds to execute, its corresponding version required less than 20,000 seconds in the MapReduce environment.

Observed from Fig. 2(b), when the total execution time of the MR-growth algorithm was low, speedup of 7 to 8 times over its sequential version was achieved. When we increased the dataset size, the algorithm achieved a speedup of approximately 8.5 times on 11 nodes.

In terms of accuracy, as an exact algorithm, our MR-growth algorithm found the same sets of truly frequent patterns as those returned by UF-growth [15].



(a) Runtime



(b) Speedup

Fig. 2. UF-growth vs. MR-growth

5.2 Evaluation of Enhancement #1: MR-Growth with ForkJoin

This experiment demonstrates the effect of employing multiple threads for mining frequent patterns with our MR-growth algorithm. As the tests were executed on a 4-core machine, we varied the number of threads from 1 to 4.

For the accidents dataset, Fig. 3(a) shows that, when the execution time was short, the algorithm took longer on multiple threads than on a single thread. The reason is that, as there is not enough work to do in the parallel, the algorithm cannot take advantage of multiple cores. When the algorithm executed for a longer period of time (e.g., more than 100 seconds), sub-linear speedup was achieved as shown in Fig. 3(b). Fig. 3 also shows the experimental results for other datasets (e.g., connect4 and mushroom).

As this enhancement aims to speed up the mining process, it does not change the accuracy of the mining results.

5.3 Evaluation of Enhancement #2: Efficient Conditional Tree Construction

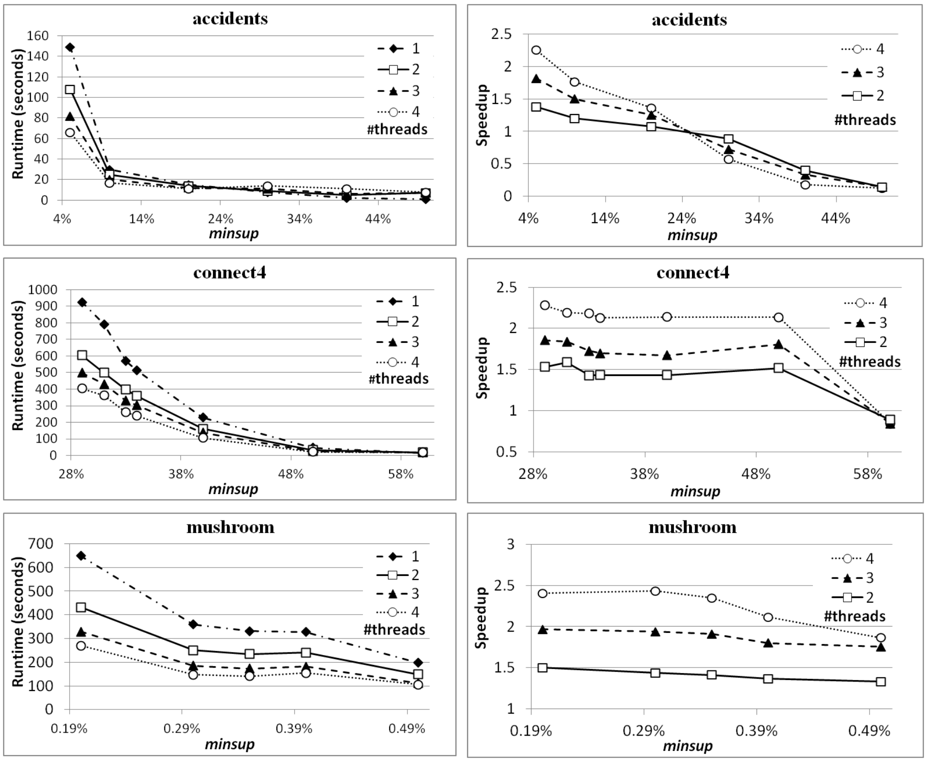
In this experiment, we compared the execution time of the original version of MR-growth with the version enhanced with the efficient conditional tree construction as discussed in Section 4.2.

Fig. 4 shows that, for small *minsup* values, both versions yielded the final result in less than 200 seconds for the accidents dataset. The performance differences became apparent only when *minsup* was lowered to 30%. As for the connect4 and mushroom datasets, the enhanced version of the MR-growth algorithm outperformed the original version. Fig. 4 also highlights that the enhanced MR-growth algorithm performed better (e.g., the difference were more than 300 seconds in some cases).

In terms of accuracy, frequent patterns mined by MR-growth with Enhancement #2 were identical to those mined by MR-growth without this enhancement.

5.4 Evaluation of Enhancement #3: MR-Growth with Sampling

We evaluated Enhancement #3 by comparing the execution times of the original version of MR-growth with the version enhanced with sampling. Fig. 5(a)



(a) Runtime (b) Speedup

Fig. 3. MR-growth with ForkJoin

illustrates that MR-growth (w/ sampling) consistently yielded the final result quicker than the original version of MR-growth.

Moreover, to test the capacity of the MR-growth algorithm to further offload work to different processor cores by using the ForkJoin framework (Enhancement #1), we compared the execution times of MR-growth with one thread on ForkJoin vs. MR-growth with two threads on ForkJoin. We observed from Fig. 5(b) that, while the execution time of MR-growth on two threads was lower, the overall benefits of distributing work to multiple threads was not too significant. This behaviour is expected because Amazon uses virtualization to expose *virtual processing cores* on shared hardware resources, which degrades potential speedup. Significant performance improvements could be observed in MapReduce environments built from high-performance machines [22].

As MR-growth with Enhancement #3 approximated expected supports, the mined frequent patterns were not identical to those mined by MR-growth without this enhancement. However, the differences were not too significant. In other words, the algorithm produced an acceptable approximation to truly frequent patterns.

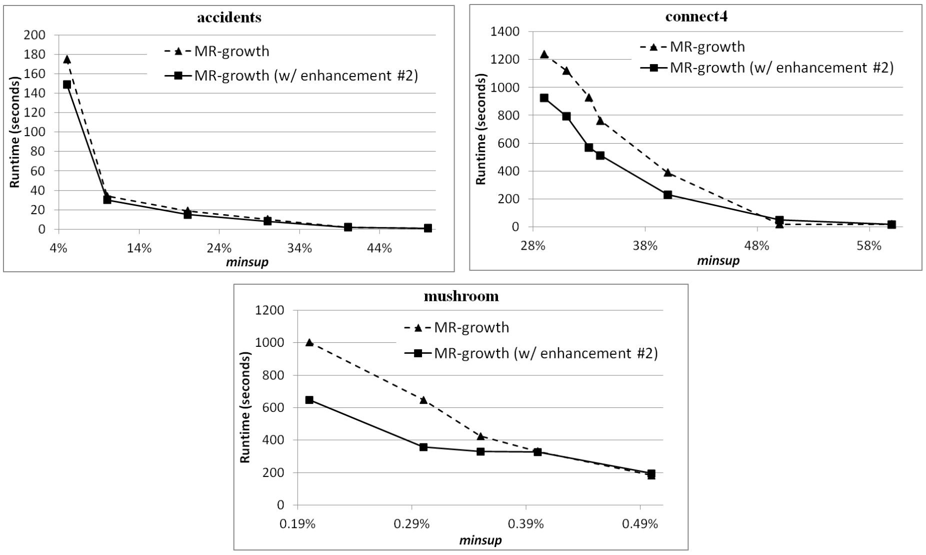
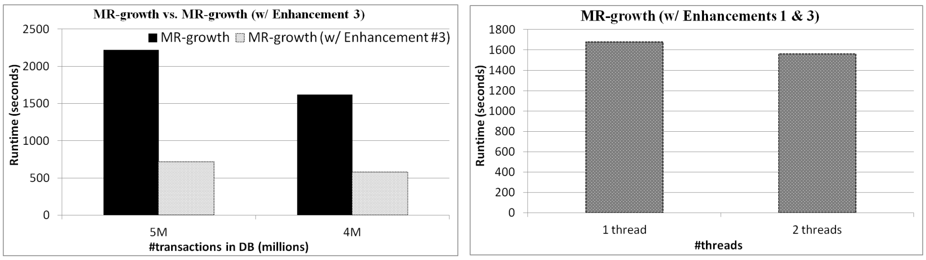


Fig. 4. MR-growth (without vs. with Efficient Conditional Tree Construction)



(a) MR-growth: With Enhancement #3 (b) With Enhancements #1 & #3

Fig. 5. MR-growth with Sampling (Enhancement #3) + 1 or 2 threads in ForkJoun (Enhancement #1)

6 Conclusions

There are many real-life situations in which we observe uncertain data (e.g., in temperature and wind speed readings, patient diagnosis, and satellite imaging). Given the probabilistic nature of these data, it may take a long time and more resources to mine frequent patterns from uncertain data. Currently, many state-of-the-art algorithms for mining frequent patterns from uncertain data may not provide superb performance because most of them are not crafted to execute in parallel. In this paper, we introduced our MR-growth algorithm, which provides the possibility to construct and mine smaller-sized UF-trees on distributed machines. Our experimental results demonstrate the effectiveness of employing the

MapReduce programming model for mining frequent patterns from uncertain data for Big data analytics. Moreover, the use of MapReduce yields significant speedups to our MR-growth algorithm. As for the use of the ForkJoin framework, it is beneficial for networks where computing nodes contain multi-core processors. As ongoing and future work, we plan to conduct more extensive experiments (e.g., evaluate the effect of the number of nodes in the MapReduce environment on the runtime). We also target at finding a framework, possibly an extension of MapReduce, which would allow us to recursively build sub-trees and schedule their mining on available computation resources.

Acknowledgements. This project is partially supported by NSERC (Canada) and University of Manitoba.

References

1. Aggarwal, C.C., Li, Y., Wang, J., Wang, J.: Frequent pattern mining with uncertain data. In: ACM KDD 2009, pp. 29–38 (2009)
2. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. In: ACM SIGMOD 1993, pp. 207–216 (1993)
3. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: VLDB 1994, pp. 487–499 (1994)
4. Calders, T., Garboni, C., Goethals, B.: Efficient pattern mining of uncertain data with sampling. In: Zaki, M.J., Yu, J.X., Ravindran, B., Pudi, V. (eds.) PAKDD 2010, Part I. LNCS (LNAI), vol. 6118, pp. 480–487. Springer, Heidelberg (2010)
5. Cordeiro, R.L.F., Traina Jr., C., Traina, A.J.M., López, J., Kang, U., Faloutsos, C.: Clustering very large multi-dimensional datasets with MapReduce. In: ACM KDD 2011, pp. 690–698 (2011)
6. Chui, C.-K., Kao, B., Hung, E.: Mining frequent itemsets from uncertain data. In: Zhou, Z.-H., Li, H., Yang, Q. (eds.) PAKDD 2007. LNCS (LNAI), vol. 4426, pp. 47–58. Springer, Heidelberg (2007)
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. CACM 51(1), 107–113 (2008)
8. Eavis, T., Zheng, X.: Multi-level frequent pattern mining. In: Zhou, X., Yokota, H., Deng, K., Liu, Q. (eds.) DASFAA 2009. LNCS, vol. 5463, pp. 369–383. Springer, Heidelberg (2009)
9. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: ACM SIGMOD 2000, pp. 1–12 (2000)
10. Kiran, R.U., Reddy, P.K.: An alternative interestingness measure for mining periodic-frequent patterns. In: Yu, J.X., Kim, M.H., Unland, R. (eds.) DASFAA 2011, Part I. LNCS, vol. 6587, pp. 183–192. Springer, Heidelberg (2011)
11. Koufakou, A., Secretan, J., Reeder, J., Cardona, K., Georgiopoulos, M.: Fast parallel outlier detection for categorical datasets using MapReduce. In: IEEE IJCNN 2008, pp. 3298–3304 (2008)
12. Lea, D.: A Java fork/join framework. In: ACM Java 2000, pp. 36–43 (2000)
13. Leung, C.K.-S.: Mining uncertain data. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 1(4), 316–329 (2011)
14. Leung, C.K.-S., Jiang, F., Sun, L., Wang, Y.: A constrained frequent pattern mining system for handling aggregate constraints. In: IDEAS 2012, pp. 14–23 (2012)

15. Leung, C.K.-S., Mateo, M.A.F., Brajczuk, D.A.: A tree-based approach for frequent pattern mining from uncertain data. In: Washio, T., Suzuki, E., Ting, K.M., Inokuchi, A. (eds.) PAKDD 2008. LNCS (LNAI), vol. 5012, pp. 653–661. Springer, Heidelberg (2008)
16. Leung, C.K.-S., Sun, L.: Equivalence class transformation based mining of frequent itemsets from uncertain data. In: ACM SAC 2011, pp. 983–984 (2011)
17. Leung, C.K.-S., Tanbeer, S.K.: Fast tree-based mining of frequent itemsets from uncertain data. In: Lee, S.-g., Peng, Z., Zhou, X., Moon, Y.-S., Unland, R., Yoo, J. (eds.) DASFAA 2012, Part I. LNCS, vol. 7238, pp. 272–287. Springer, Heidelberg (2012)
18. Leung, C.K.-S., Tanbeer, S.K.: Mining popular patterns from transactional databases. In: Cuzzocrea, A., Dayal, U. (eds.) DaWaK 2012. LNCS, vol. 7448, pp. 291–302. Springer, Heidelberg (2012)
19. Leung, C.K.-S., Tanbeer, S.K., Budhia, B.P., Zacharias, L.C.: Mining probabilistic datasets vertically. In: IDEAS 2012, pp. 99–204 (2012)
20. Leung, C.K.-S., Jiang, F.: RadialViz: An orientation-free frequent pattern visualizer. In: Tan, P.-N., Chawla, S., Ho, C.K., Bailey, J. (eds.) PAKDD 2012, Part II. LNCS (LNAI), vol. 7302, pp. 322–334. Springer, Heidelberg (2012)
21. Lin, M.-Y., Lee, P.-Y., Hsueh, S.-C.: Apriori-based frequent itemset mining algorithms on MapReduce. In: ICUIMC 2012, art. 76 (2012)
22. Lloyd, W., Shrideep, P., Olaf, D., Lyon, J., Mazdak, A., Ken, R.: Migration of multi-tier applications to infrastructure-as-a-service clouds: an investigation using Kernel-based virtual machines. In: IEEE/ACM GRID 2011, pp. 137–144 (2011)
23. Madden, S.: From databases to big data. IEEE Internet Computing 16(3), 4–6 (2012)
24. Rashid, M. M., Karim, M. R., Jeong, B.-S., Choi, H.-J.: Efficient mining regularly frequent patterns in transactional databases. In: Lee, S.-g., Peng, Z., Zhou, X., Moon, Y.-S., Unland, R., Yoo, J. (eds.) DASFAA 2012, Part I. LNCS, vol. 7238, pp. 258–271. Springer, Heidelberg (2012)
25. Riondato, M., DeBrabant, J., Fonseca, R., Upfal, E.: PARMA: a parallel randomized algorithm for approximate association rules mining in MapReduce. In: ACM CIKM 2012, pp. 85–94 (2012)
26. Tong, Y., Chen, L., Cheng, Y., Yu, P.S.: Mining frequent itemsets over uncertain databases. In: PVLDB, vol. 5(11), pp. 1650–1661 (2012)
27. Yang, S., Wang, B., Zhao, H., Wu, B.: Efficient dense structure mining using MapReduce. In: IEEE ICDM Workshops 2009, pp. 332–337 (2009)