Israel Open University
Department of Mathematics and Computer Science

Graduate Project in Computer Science

"Performance Comparison of MapReduce Algorithms for Frequent Itemsets Mining"

Advisor: Prof. Ehud Gudes

Semester A, 2017

Student name: Alexander Khovansky

Table of Contents

Abstract

Frequent Itemset Mining (FIM) is an important technique to extract knowledge from data, aiming to find frequently co-occurrent items. The found frequent itemsets can then be used for further analysis, notably for Association Rules Mining. Some FIM algorithms focus on improving the frequent itemsets generation efficiency for a single-processor execution. However, the growing size of the datasets and advances in parallel execution technologies caused an emergence of parallel FIM algorithms. As the Map/Reduce frameworks such as Hadoop and Spark became the main parallel execution platforms, most parallel algorithms employ the Map/Reduce paradigm. The most famous Map/Reduce FIM algorithm is PFP-Growth, which is also implemented in the standard Apache Spark distribution. This work creates a new application, '***fim-cmp***', that uses Spark to implement two other promising Map/Reduce algorithms, 'Big FIM' and 'Parallel dFIN'. This paper describes the application and the implemented algorithms, discusses some important implementation details and improvements, and finally compares the algorithms' performance with that of the PFP-Growth.

Index Terms: frequent itemset mining, association rules generation, mapreduce, spark

Introduction

Project Summary

The project goal is to implement two promising distributed Map/Reduce FIM algorithms and then compare their performance with that of the classic Parallel FP-Growth algorithm ([13]).

The chosen algorithms are:

1. 'Big FIM' ([12]).
2. A Map/Reduce version of the 'dFIN' algorithm ([8], [14]).

   The Map/Reduce approach to parallelize a single-processor 'FIN' algorithm ([7]) is described in ([14]), but instead of 'FIN' a newer version, 'dFIN' ([8]), is used.

Both algorithms are implemented in Apache Spark 2 ([19]) using Java 8, and their performance is compared to the standard implementation of the Parallel FP-Growth algorithm shipped as part of the Spark distribution.

The experiments have been carried out on Amazon EMR ([17]).

The implementation, '***fim-cmp***', allows the user to run all the three algorithms on his/her own dataset and to choose other parameters for the runs via a command line. The list of allowed parameters per algorithm is available via online help. In addition, example scripts are provided to run each algorithm with typical parameters either on the local network or in Amazon EMR.

The entire application code is available in a public GitHub repository at https://github.com/Khovansa/fim .

Background Information

Data mining, a process of discovering patterns in large data sets, is becoming an increasingly important field as more data are becoming available. Association Rules Mining (ARM) is one of the most important methods in data mining. Its purpose is to discover interesting relations between variables in large data sets. A classic example of ARM is market data analysis that looks for rules such as 'if a customer bought items X and Y then s/he is likely to buy item Z', where the likelihood is above a user-defined minimal confidence level. The ARM is usually done in two steps: first, find all Frequent Itemsets (FI) whose frequency is above a user-defined minimal threshold and second, find the association rules from these frequent itemsets. The first step is considered more computationally expensive and most algorithms in ARM are targeting the FI generation. In the above example the first step would be to find FIs {XY, XYZ} and the second step would be to detect a rule 'if XY then Z'.

The input data for FI mining algorithms usually come in the form of a table with items as columns and transactions as rows. Following the market data example, each row would contain all the items bought by one customer at a specific time. FI mining algorithm time and memory efficiency is very important since the input data might be huge. Initially, the research effort was to design an efficient deterministic sequential algorithm. Some of the best known examples are Apriori [1], Eclat [2], FP-Growth [3], D-CLUB [5], and Node-set based algorithms ([6], [7], and [8]). A Closet [4] family of algorithms strives to reduce the output (and improve efficiency) by focusing only on Closed FIs, that is FIs that can't be increased without decreasing their frequency. As the input size continued to grow, more ways of FI mining of large data sets were explored. One way is to use a smart random sampling and run ARM/FI mining on this sample, e.g. PARMA ([9]). Another way is to do FI mining in parallel. Usually, a known sequential algorithm is adjusted to run on multiple threads of execution. Well known examples are Parallel Apriori, such as YAFIM ([10]), Par-Eclat ([11]), Dist-Eclat and BigFIM([12]), Parallel FP-Growth ([13]), and PFIN ([14]). Parallel algorithms might assume either shared memory (targeting multi-threaded execution on a single machine), assume no shared memory (targeting distributed execution on multiple machines) or try a hybrid approach, such as [11]. Moreover, distributed algorithms might either assume that each single machine can store a representation of the entire data set in its main memory or try to deal with the input for which it is not possible.

Most distributed algorithms assume a MapReduce framework for implementation. Each MapReduce operation consists of 3 steps: 'map', 'shuffle', and 'reduce'. The 'map' step processes the input to produce a list of (key, value) pairs; if the input data is split into shards located on multiple machines, each machine would normally execute the map operation on its own shard. The 'shuffle' operation is performed by the framework (no programmer code is required); it takes the output of the 'map' operation as input and produces (key, list of values) pairs as output. Finally, the 'reduce' operation uses the output of the 'shuffle' operation as the input and produces the final output, normally as (key, values list) pairs. The best known MapReduce framework is Hadoop ([18]). A newer alternative is Apache Spark ([19]), which offers simpler API and is usually much more efficient than Hadoop mostly due to its reduced work with the disk.

The most widely used distributed algorithm is Parallel FP-Growth ([13]). It assumes a MapReduce framework and is already implemented in the standard Apache Spark distribution. Most new algorithms usually compare their performance to that of Parallel FP-Growth.

Choice of the Algorithms to Compare

The choice of the algorithms was driven by a subjective estimation of their chances to outperform the Parallel FP-Growth algorithm in some aspect. Another consideration was the diversity of the algorithms, i.e. to choose algorithms from different 'families'.

As PFP-Growth outperformed all the preceding parallel algorithms, the search has been confined to algorithms designed after PFP-Growth.

Algorithms running on a sample of the input data, such as PARMA ([9]), have been excluded from consideration as not contributing to the performance comparison, as once the sample is chosen, any algorithm could be used to run on this sample.

Distributed algorithms based on Apriori, such as YAFIM ([10]), have been excluded as Apriori is reported to be much slower than FP-Growth and Eclat.

There are a number of algorithms focusing solely on closed FIs generation (Closet [4], CloPN [20]). The algorithms aim to improve performance by not spending time on itemsets having supersets with the same support. Yet they are reported (e.g. in [5]) to bring benefits only to very dense, mostly artificial, datasets with long patterns, but yield less savings on other dataset types.

FiDoop ([22]) explores the idea to use FP-Trees only for itemsets of the same length, but the paper does not report much performance comparisons data.

Initially promising was a distributed version of the D-CLUB algorithm ([5], [15]). It is reported in [5] to outperform FP-Growth and cater for both dense and sparse datasets.

Similarly to Eclat, it uses 'vertical' dataset representation, i.e. representing a dataset as a map (item → transactions bitset) for each itemset which can be stored in a bit matrix ('bitmap'). It then uses sophisticated techniques to mine itemsets by producing child bitmaps and to reduce each bitmap by removing entire rows or columns. The goal is to offer an adaptive technique that would bring performance benefits similar to Closet for extremely dense datasets, but also offer improvements for less dense datasets. Yet initial experimentation with the algorithm has shown that D-CLUB is still much slower than FP-Growth on standard FIMI ([21]) dense datasets. Besides, one of the other algorithms chosen for comparison, PFIN, also applies techniques to detect closed FIs.

Eventually, the choice has been made for the following two distributed algorithms:

1. BigFIM described in [12]. The algorithm is based on Eclat with diffsets and Apriori. It is reported in [12] to outperform Parallel FP-Growth on large datasets, due to smaller memory requirements on each individual node and better distribution of load between the nodes.
2. A Map/Reduce version of the 'dFIN' algorithm ([8], [14]). As mentioned earlier, the '*fim-cmp*' application uses the map/reduce approach from [14], but implements a newer underlying sequential algorithm described in [8].

Outline of the Algorithms

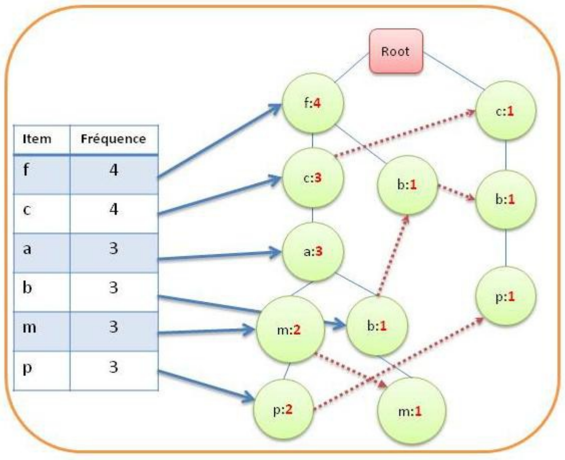
Parallel FP-Growth

The sequential version, FP-Growth, is described in [3], and its adaptation to the Map/Reduce framework, Parallel FP-Growth, is described in [13].

FP-Growth

The algorithm first builds a compact FP-Tree structure kept in memory and then uses it to directly extract the frequent itemsets, thus avoiding costly DB scans.

FP-Tree is a trie-like structure whose nodes are items with counts. Building it requires two dataset scans. The first scan creates a list of frequent items, L_1, sorted by descending support count. In the second dataset scan, items in each transaction are first sorted according to their L_1 order and then inserted into the FP-Tree. The insertion of a transaction to the FP-Tree is similar to an insertion of a word to a trie, where the items correspond to letters, and transactions correspond to words. For each item, either a new node is created with counter 1, or a node counter is incremented if the node already exists. FP-Tree also maintains pointers between nodes containing the same item. Below is an example of a FP-Tree after insertion of the transactions [f, c, a, m, p], [f, c, a, m, p], [f, c, a, b, m], [f, b], [c, b, p]:



(Image copied from **Khaled Tannir** blog http://blog.khaledtannir.net/2012/07/lalgorithme-fp-growth-construction-du-fp-tree-23)

Once the FP-Tree is created, it is used to extract the frequent itemsets. The process is divide-and-conquer: for each item s in the reversed L_1 find all FIs containing s and not containing the less frequent items. In the example above, it will first try to find all FIs containing 'p', then all FIs containing 'm', but not 'p', then all FIs containing 'b', but neither 'm' nor 'p' etc. Below is an outline of a recursive routine that finds all FIs from the given FP-Tree and a given prefix (initially empty).

find_FIs(tree, prefix):

```
For each item s in (reversed $L_1$ \ prefix) having a leaf node in 'tree':
1. Check if s is frequent by adding the counts along its nodes. If yes, output (prefix $\cup$ {s}) and continue, otherwise move to the next item.
2. Build a conditional FP-Tree, tree_s, for s. The conditional FP-Tree is the FP-Tree that would be built if we only consider transactions containing s and then remove s from these transactions. In practice, it can be built from the FP-Tree by updating the counters in 'tree' along all paths leading to s.
3. If tree_s is not empty, call find_FIs(tree_s, prefix $\cup$ {s})
4. Remove all leaf nodes containing s from 'tree'.
```

FP-Tree is a highly-compressed representation of the original dataset. The greater the overlap between the transactions, the higher the compression achieved by the FP-Tree. The main problem with the FP-Growth algorithm is that the FP-Tree might be huge for large datasets, so it might not fit into the main memory and even disk on a single machine.

Parallel FP-Growth

The Parallel FP-Growth algorithm uses the Map/Reduce approach and strives to solve the above problem by splitting the original dataset into smaller ones by introducing a concept of a *group-dependent dataset*. The sorted list of frequent items, L_1, is split into N groups where N is a user-defined parameter. It is also possible to assign each frequent item to its own group. Given a transaction t and a group g, the *group-dependent transaction* of t is created by sorting t by L_1 order and dropping all the items from t that appear after the last occurrence of an item from g. A transaction that does not contain an item from g is skipped. For instance, if we assign each item into its own group, and continue with the example dataset from above, a group-dependent transaction of transaction [f, c, a, m, p] and group 'a' is simply [f, c, a].

Rationale:

Informally, less frequent items might not significantly reduce the size of each transaction, but appear in less transactions, while more frequent items appear in more transactions, but will significantly reduce the individual transaction length. Thus the expectation is that the group-dependent datasets will not significantly differ in their size.

The algorithm outline:

1. Compute L_1 using the standard Map/Reduce parallel counting. Each machine might contain only a part (shard) of the entire dataset.
2. Assign a group-id to each frequent item from L_1.
3. The main Map/Reduce phase:
   (a) Map: for each transaction t and for each group-id output a pair (group-id, group-dependent transaction of (t, group-id))
   (b) Reduce: each reducer will receive a pair (group-id, all group-dependent transactions related to this group-id). It will then run the standard FP-Growth on the received group-dependent dataset.

Big FIM

The Big FIM algorithm is described in [12]. It strives to improve the load balancing of the Parallel FP-Growth algorithm and also reduce the memory requirements on each individual machine.

Big FIM Overview

The algorithm outline:

1. Run Parallel Map/Reduce Apriori to compute FIs up to the given size, k. Normally, k is either 3 or 4.
2. Compute longer FIs in another Map/Reduce phase:
   (a) Map: for each k-FI find its *tid-list*, that is, ids of all transactions containing this FI. Then output a pair (key = (k-1)-prefix of the k-FI, value = {k-FI, the tid-list of the k-FI}).
   (b) Reduce: each reducer will receive a (k-1)-prefix as a key and list of {k-FI, the tid-list of the k-FI}) as a value. It will run the standard Eclat algorithm on the 'value' part, 'and'-ing the tid-lists to produce longer FIs.

Load balancing:

The algorithm authors observe that the Parallel FP-Growth can only split the original dataset based on individual frequent items, which might be too coarse-grained and lead to poor load balancing between the machines. Big FIM splits the original dataset based on pairs or triples, meaning much shorter individual reducer jobs and a better load balancing between the machines as a result.

Memory requirements:
In Parallel FP-Growth each reducer has to build the entire FP-Tree for its group-dependent dataset. Although it is smaller than the FP-Tree of the entire original dataset, it might still be too large if the original dataset is big enough. In contrast, Big FIM significantly reduces the memory requirements on each individual machine by sending much smaller datasets in each reducer job. In addition, Eclat can be implemented to require less memory than FP-Growth.

The Apriori and Eclat algorithms are outlined below.

Apriori

Apriori ([1]) is based on *anti-monotone property*: if some k-FI is not frequent, then all its (k+1)-FIs can't be frequent. So it first finds L_1, a list of frequent items, then uses it to find L_2 – a list of 2-FIs, then uses L_2 to find L_3 – a list of 3-FIs etc. In general, to find L_{k+1}, a naïve implementation of Apriori scans the complete dataset and picks and counts all itemsets of size (k+1) that contain two different itemsets from L_k and that don't contain any subset not from L_k.

Map/Reduce Apriori

Map/Reduce implementation of Apriori can efficiently use partitioning of the original dataset between the machines. Each machine can contain only a shard, i.e. a subset of the original dataset. The 'map' phase produces the candidate itemsets of size k and their counts in a particular shard. The 'reduce' phase sums up the counts for each candidate k-itemset and chooses the frequent k-FIs. The process repeats to find FIs of the next size and so on, until no more FIs of size n could be found.

Eclat

Eclat algorithm ([2]) usually assumes a *vertical layout* of the dataset: unlike a more frequent *horizontal layout* which lists items for each transaction, in a vertical layout each item is mapped to its *tid-list*, i.e. ids of the transactions that contain the item. A support of an item is simply the length of its tid-list. The tid-list of 2-itemset is an intersection of the tid-list of its items. In general, to compute a tid-list of a k-itemset it's enough to compute an intersection of any two of its subsets of length (k-1). There is no restriction on how to explore the space of the frequent itemsets: it could be either in BFS order, i.e. first finding all k-FIs and then using them to generate all (k+1)-FIs or in any other order. To quickly check if some FI is frequent it's enough to 'and' the tid-lists of all its items and check the length of the resulting tid-list.

The tid-lists can be very long for a large dataset, inducing a cost on both processor and memory. One possible optimization is to only keep the difference in a tid-list of a k-FI to one of its contained (k-1)-FIs and some additional information required to compute the support of the k-FI. The tid-list of differences is called *diffset* and the technique is described in [23]. This technique has been used in the '*fim-cmp*' implementation of Big FIM.

A well known method to split the Eclat FI computation between multiple machines is to send all itemsets with the same prefix to a separate machine ([11]). As described earlier, Big FIM uses this technique as well.

Parallel dFIN

A single-processor FI mining algorithm, dFIN, is described in [8]. The Map/Reduce approach to parallelize a similar single-processor algorithm, 'FIN' ([7]), is described in [14].

dFIN

The algorithm uses the following definitions and observations:

- *PPC-Tree*: PPC-Tree is similar to FP-Tree, but each node also holds its pre-order and post-order tree traversal numbers.
- *Nodeset(i)* = {all nodes in the PPC-Tree holding an item i, sorted by increasing pre-order number}.
  - support(i) = $\sum_{n \in Nodeset(i)} count(n)$ (same as in FP-Growth)
- *Nodeset($[i_1, ..., i_k]$)* is defined as Nodeset($[i_1,...,i_{k-2}, i_{k-1}]$) $\cap$ Nodeset($[i_1,...,i_{k-2}, i_k]$).
- *DiffNodeset(i_1, i_2)* is defined as {x $\in$ Nodeset(i_1) | $\neg\exists$y $\in$ Nodeset(i_2) that is ancestor of x in the PPC-Tree}.
- Lemma 1: support($[i_1, i_2]$) = support(i_1) - $\sum_{n \in DiffNodeset(i1,\, i2)} count(n)$

Let's denote itemset $[i_1,...,i_k]$, k>2, as P, itemset $[i_1,...,i_{k-2}, i_{k-1}]$ as P1, and itemset $[i_1,...,i_{k-2}, i_k]$ as P2.

- *DiffNodeset(P)* is defined as Nodeset(P1) \ Nodeset(P2)
- Lemma 2: DiffNodeset(P) = DiffNodeset(P2) \ DiffNodeset(P1)
- Lemma 3: support(P) = support(P1) - $\sum_{n \in DiffNodeset(P)} count(n)$

To avoid repeated computations, the algorithm computes support of DiffNodeset(any itemset) as soon as the DiffNodeset is created.

'Equivalent items' and closed itemsets:

The algorithm has a step to detect what it calls '*equivalent items*' to skip traversal on entire branches of a solution space.

If support($[i_1,...,i_{k+1}]$) = support($[i_1,...,i_k]$), then i_{k+1} is added to the list of '*equivalent items*' of DiffNodeset($i_1,...,i_k$) and the creation of DiffNodeset($i_1...i_{k+1}$) and all its subtree is skipped.

Once all 'equivalent items' $i_{p1}...i_{ps}$ for a given itemset $[i_1,...,i_k]$ are identified, the algorithm outputs $\{[i_1,...,i_k] \cup S, S \in$ power set of $\{i_{p1},...i_{ps}\}\}$.

- This way the algorithm is in fact able to quickly identify all **closed itemsets**, as $[i_1,...,i_k,i_{p1},...,i_{ps}]$ is a closed itemset. However, the paper does not explicitly references this concept.

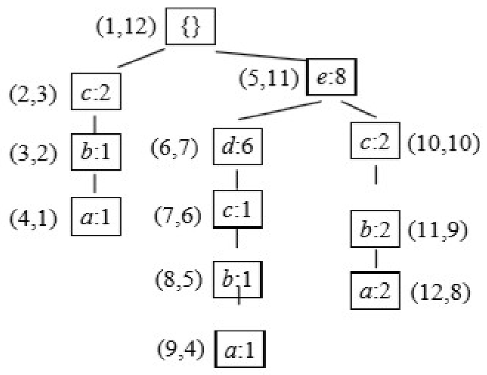
The algorithm outline is below:

1. Scan the dataset and find a list of frequent items sorted by decreasing frequency.
2. Scan the dataset and build PPC-Tree.
3. For each frequent item i, build Nodeset(i).
   Once done, the PPC-Tree is no longer required and can be discarded.
4. For each pair (i_1, i_2) of frequent items, build DiffNodeset(i_1, i_2).
   Despite the fact that the definition references the PPC-Tree, the algorithm does not use the PPC-Tree in practice: it is possible to construct DiffNodeset(i_1, i_2) simply from Nodeset(i_1) and Nodeset(i_2) by checking the pre-order and post-order numbers on their nodes.
5. DiffNodeset of k-itemset $[i_1, ..., i_k]$ and its support can be directly computed from DiffNodesets of $[i_1,...,i_{k-2}, i_k]$ and $[i_1,...,i_{k-2}, i_{k-1}]$ using Lemmas 2 and 3.
   Similarly to Eclat, the solution space can be explored in different order.
   The algorithm described in the paper uses DFS traversal.

Also not explicitly mentioned in the paper, there is no need to keep all the computed DiffNodesets all the time. In fact, the DFS traversal allows to keep only a small subset of FIs' DiffNodesets at any given point in time. This point will be elaborated in the section describing the dFIN implementation details.

Example:

The example is taken from [8] with infrequent items omitted for brevity.
Let the input dataset be: [c], [a, b, c], [a, b, c, d, e], 2 x [a, b, c, e], 5 x [d, e].
Then the L_1, sorted by descending frequency, is: **[e, d, c, b, a]**.
The PPC-Tree is:



The numbers in parentheses are the pre-order and post-order tree traversal numbers, and the numbers after a colon are the counts.
As a pre-order number uniquely identifies a PPC-Tree node, let's use the pre-order numbers to designate the nodes, e.g. '4' to designate a node {(4, 1), a:1}. For convenience, a node count might follow its pre-order number the after a colon, e.g. '4:1'.
Then the items' nodesets are:

- Nodeset(a)={4:1, 9:1, 12:2}
- Nodeset(b)={3:1, 8:1, 11:2}
- Nodeset(c)={2:2, 7:1, 10:2}
- Nodeset(d)={6:6}
- Nodeset(e)={5:8}

Let's generate Nodesets and **DiffNodesets** for some pairs:

- Nodeset(ce)={7, 10} and **DiffNodeset**(ce)={2}, since nodes '7' and '10' have '5' as their ancestor in the PPC-Tree, while '2' does not.

- ◦ By Lemma 1, support(ce) = support(c) - $\sum_{n \in DiffNodeset(ce)} count(n)$ =

  $(count(node2)+count(node7)+count(node10))-(count(node2))=5-2=3$
- Nodeset(cd)={7} and **DiffNodeset**(cd)={2, 10}, as '7' have '6' as its ancestor, while '2' and '10' do not.
  - ◦ support(cd) = support(c) - $\sum_{n \in DiffNodeset(cd)} count(n)$ = 5 – (2 + 2) = 1

Nodesets and DiffNodesets of itemsets longer than two can be quickly obtained from the shorter ones, e.g.:

- Nodeset(cde) = Nodeset(cd) $\cap$ Nodeset(ce) = {7} $\cap$ {7, 10} = {7}
- **DiffNodeset**(cde) = DiffNodeset(ce) \ DiffNodeset(cd) = {2} \ {2, 10} = $\varnothing$
- By Lemma 3, support(cde) = support(cd) - $\sum_{n \in DiffNodeset(cde)} count(n)$ = 1 – 0 = 1

An example of an itemset with equivalent items is 'a' with equivalent items 'b' and 'c', as support(a) = support(abc) = 4.

Parallel FIN / dFIN

The way to parallelize the computation via Map/Reduce is almost identical to that of the Parallel FP-Growth. That is, a sorted list of frequent items, L_1, is split into N groups and a group-dependent dataset is created and processed for each group in a separate machine.

Implementation

General Notes

Innovation

All the sub-sections prefixed with either “*Efficient Data Structures:* “, “*Algorithm Modification:* “, or “*Algorithm Extension:* “ are contribution of this work. To the best of the author's knowledge, the solutions described in these sections have not been mentioned in other works.

Technology Choice

The algorithms are implemented in Apache Spark 2 ([19]) using Java 8.
Apache Spark is usually much more efficient than an earlier Map/Reduce framework, Hadoop, mostly due to its reduced work with the disk.
Java 8 offers function references and lambda functions features which are very handy in the function-oriented Spark environment.

Spark allows running the programs both in 'local' mode on a single machine and via a variety of *cluster managers*, including Spark's own 'standalone' cluster manager. The implemented application provides direct support and has been tested in the following modes:

- Local mode
- Running on multiple nodes via the Spark 'standalone' cluster manager
- Running in Amazon Web Services (AWS) environment

Building the Application

The 'fim-cmp' uses *maven* to build the application. The standard '*mvn install*' should be used to build the application that can be run in the 'local' Spark mode using the embedded Spark engine.

To be able to submit the application to an external Spark process, a 'fat jar' should be generated that would include not only the project's classes, but also its dependencies (except Spark). The command to generate such the 'fat jar':
mvn -Pshade clean package

See the next section on details of how to run the application.

Command-Line Interface

The '*fim-cmp*' application's intended use is to submit it to a running Spark cluster. The choice of the algorithm, the Spark master URL, the path to the input file and various algorithm-specific options are specified as command-line flags. In addition, its '*scripts*' folder contains example scripts to submit FI mining jobs, one script per supported algorithm, and also scripts to start a Spark master and a Spark worker. Each invocation of the Spark worker script creates a Spark worker on the machine from which it is invoked, i.e. it can run either on the same machine or on a different machine. A typical Amazon Web Services JSON configuration is also provided.

Command-line interface structure:

<algorithm-name> <algorithm-specific options>

The algorithm names can be listed with '*--help*' option, the currently supported ones are:
BIG_FIM | FIN | FP_GROWTH .

The algorithm-specific options can be listed with '*--help*' after the algorithm name, e.g. '*BIG_FIM --help*'.

The main class:

The main class is ***org.openu.fimcmp.cmdline.CmdLineRunner***.

Output options:

The output options allow to choose whether to output only the total number of found FIs or all the found FIs. In the second case the FIs are printed in a canonized format which allows to validate that all algorithms produce the same result.

All the supported options are listed and described in Appendix A.

Java Package Structure

The algorithm implementation code resides in the '*algs*' package with sub-packages dedicated to specific algorithms: *'apriori', 'bigfim', 'eclat', 'fin', 'fpgrowth'*, and a package '*algbase*' for common algorithms-related code, such as e.g. finding a list of frequent items etc.
Additional utility packages include '*result*' containing classes able to receive the resulting FIs, '*itemset*' for itemset-related classes, '*cmdline*' for command-line arguments parsing, and '*util*' for smaller-scale general utility classes, such as a memory-efficient and minimalistic *BitArray* class.

Correctness Verification

To verify the correctness of each algorithm there is an option to print all the found FIs sorted in some fixed order. As the FP_GROWTH algorithm option uses the standard Spark implementation of Parallel FP-Growth algorithm which can be assumed to be correct, it is possible to validate the correctness of other algorithms by printing their results and comparing them with those of FP_GROWTH. This has been done on several inputs to make sure all the algorithms find precisely the same set of FIs.

Efficient Use of Spark

There are several general measures that can drastically improve the run time of Spark programs. The most important ones are listed below:

- **Efficient use of Spark partitions**: prefer '*mapPartitions*' over 'map' and also prefer partition-aware aggregate functions over the partition-unaware ones. This point is further elaborated in a sub-section on Big FIM implementation details.
- **Prefer arrays of primitives such as *int[], long[]* etc. in Spark RDDs** over more advanced Java classes (and in particular, over classes of java.util package). This is because the number of objects in Spark RDD might be very large for a big dataset meaning significant influence on both processor time, memory, and network traffic.
- **Prudent storage of intermediate results**: Spark will recompute everything on each aggregate operation, so some intermediate results that don't take much volume are better to be stored either in memory or in memory and disk.
- **Use Kryo serialization library** as much as possible as it is reported to be more efficient than the standard Java serialization

Big FIM Implementation Details

The Big FIM algorithm has been implemented according to [12]. The main tasks related to Big FIM implementations are:

1. Implementing parallel map/reduce Apriori to compute FIs up to the given size (typically, 4)
2. Finding tid-lists for the chosen k-FIs and grouping the k-FIs by their common prefix of length (k-1)
3. Running Eclat with diffsets on each group of k-FIs

A number of steps have been applied to maximize the run-time efficiency of the implementation. In addition, several options have been provided to tune the algorithm. The main efficiency-maximizing steps and the main algorithm-tuning options are described below.

Reasonable Use of Spark Partitions

Spark provides partition-aware functions that can drastically reduce a number of 'shuffle' operations and significantly improve the run time.

Consider a classic word counting problem. Many map/reduce examples of a word counting problem show a mapper producing a pair (word, 1) and a basic reducer summing the second element of the pair. Similarly, a naïve implementation of Apriori at step k would be to produce pairs (k-itemset, 1) at the 'map' step and to sum the counters per k-itemset at the 'reduce' step. It turns out that this implementation is extremely inefficient due to a huge number of 'shuffle' operations for any k greater than 2. This is because the number of the 'shuffle' operations is proportional to the number of generated pairs which might be very large even for a single transaction for dense datasets.

Much more efficient approach is to employ Spark partition-aware operations, such as '*mapPartitions*'. This means to do most of the heavy work inside a partition, utilizing a task-specific efficient data structure to hold the intermediate results per partition and then apply the 'reduce' operation on these data structures. The number of different partitions should be small and proportional to the number of machines. Spark guarantees to only merge the results of per-partition computation after fully completing the computation for the partition. In other words, the number of 'reduce' operations will be proportional to the number of partitions. This also applies to partition-aware '*reduce*' operations that accept two functions: one that adds an element to an 'accumulator' and another one that merges two accumulators. It is guaranteed that the number of 'merge' operations will also be proportional to the number of partitions.

This approach has been applied both to find k-FIs with map/reduce Apriori and to find the tid-lists for the k-FIs. More details will be provided in subsequent sections.

Efficient Data Structures: Use Item and Itemset Ranks

After a list of frequent items, L_1, is found and sorted, each item can be associated with its sequential number in this list or '*rank*'. Integer ranks are more efficient way to represent items than strings. This is how the Parallel FP-Growth algorithm is implemented by the Spark distribution. In Apriori, the same approach can be applied not only for items, but also for itemsets. Once a list of frequent items, L_1, is found, each item is replaced by its rank in L_1 and each 2-itemset can be represented by a pair of integers. Similarly, once a list of 2-FIs, L_2, is found, each 2-FI can be replaced by its rank in L_2 and each 3-itemset can be represented by a pair of integers, namely (frequent item rank, 2-FI rank). In general, once a list of k-FIs, L_k, is found on step k of Apriori, each k-FI is replaced by its rank in L_k and any (k+1)-FI can be represented as a pair (frequent item rank, k-FI rank).

This allows, in particular, to represent a mapping k-itemset → count simply as a 2-dimensional primitive integer array that maps (frequent item rank, (k-1)-FI rank) to a count. This is the data structure used to represent the Apriori counting operation result per partition.

Efficient Data Structures: Use Bitsets to Hold Sets of K-Itemsets

Each step of Apriori produces candidate k-itemsets that then counted and filtered out. The number of candidate k-itemsets per transaction might be large (such as hundreds of thousands or even millions) in dense datasets for $k > 2$ and low minimal support. Moreover, as mentioned earlier, it is advisable to store intermediate computation at each Apriori step to prevent Spark to recompute everything on the next step.

The previous section described how each k-FI could be represented as a single integer rank. This allows to represent a set of k-FIs per transaction as a **bitset**. A bitset is simply a java long[] array in which each 'long' number holds a yes/no bit for the next 64 values. It means up to 64 times reduction in memory usage compared with an array of integers. Once all k-FIs have been identified, '*fim-cmp*' stores a pair (frequent item ranks as int[], set of k-FIs as a bitset, i.e. as long[]) per transaction. (Naturally, all the pairs are represented as another Spark RDD). Concise representation of these data mean saving in both processor time, memory, and network traffic.

Moreover, some operations that formerly contained 'if' logic can now be accomplished with simple and fast bit-wise operations such as AND, NOT, and XOR. For instance, at each

Apriori step, before starting to work with transactions, '*fim-cmp*' pre-computes a map (frequent item rank, k-FI rank) → whether the (k+1)-itemset represented by this pair has a chance to be frequent. This map is represented as an array of bitsets, which can be then 'and'-ed with transaction k-FIs to quickly remove (k+1)-itemset candidates that have no chance to be frequent. This technique is used several times throughout the application.

Experiments have shown that the above idea, combined with using partition-aware Spark functions resulted in performance improvements by a factor of more than ten.

Algorithm Modification: Generate TID Lists for FIs Only

This simple optimisation is not mentioned in the original Big FIM paper ([12]). Generating the tid-list per itemset requires more time than simply counting the itemset frequency. So '*fim-cmp*' first identifies k-FIs and only then generates tid-lists for the k-itemsets already known to be frequent.

Summary: K-Itemset Counting and Tid-List Generation

Combining the techniques described in the previous section leads to the following:

- Results of the k-itemset counting per partition are represented as a single int[][] array which holds a mapping (frequent item rank, (k-1)-FI rank) → count
- Before the next Apriori step (or the tid-list generation step), a pair (frequent item ranks, bitset of k-FIs) is computed and stored per transaction
- Results of the tid-list generation per partition are represented as a single long[][] array which holds a mapping (k-FI rank) → bitset of ids of transactions containing this k-FI

Algorithm Extension: Switching to Eclat Options

Switching from Apriori to Eclat is controlled by an option specifying the length of the itemset prefix to group the FIs to be sent to Eclat. A typical value is either 2 or 3, as suggested in the Big FIM paper. A value of 3 means that the algorithm finds all 4-FIs, computes their tid-lists, and then sends all 4-FIs with the same 3-length prefix to Eclat.

In addition, there is an option to continue with Apriori for sparse datasets. On sparse datasets, the parallel Apriori has been noticed to outperform Big FIM's submissions of tid-lists to the Eclat-running workers **on some inputs**. This is because the (parallel) Apriori moves much less data around as each machine just continues to run on its original input partition and only counts per k-FI are exchanged. By default, a dataset is considered sparse enough to continue with Apriori if

|all k-FIs| / |all (k-1)-FIs| ≤ 1.1, i.e. if the number of k-FIs is small relative to the number of (k-1)-FIs. Yet on other inputs a normal switch to Eclat yields better performance, hence a conservative ratio.

Eclat Implementation Details

According to the Big FIM algorithm, only a single-processor Eclat algorithm needs to be implemented. An algorithm receives a list of all k-FIs with the same (k-1)-size prefix and a tid-list for each k-FI. The Eclat algorithm will produce all the itemsets longer than k by 'and'-ing the tid-lists of the existing and new FIs. The solution space is explored in BFS order. The algorithm has an option to use the '*diffsets*' optimization mentioned in the section describing the Eclat algorithm. This option is usually enabled.

In addition, a D-CLUB paper ([5]) has inspired some experimentation with further reducing the tid-list sizes. In particular, there is an option to 'squeeze' tid-lists by removing the columns that are all 0s. This initially sounded promising as the number of k-FIs arriving to the Eclat algorithm is usually small (20 is a typical number). Unfortunately, this did not yield any noticeable improvement. One of possible reasons is that the tid-lists are represented as bitsets, so it is required to remove a significant number of columns to justify an additional processing time required to squeeze the tid-lists. This option is normally disabled.

There is also some logic regarding the number of Spark partitions to be used for Eclat. Ideally, it should be as large as the number of different prefixes, but this number might be too large, meaning unnecessary overhead. So this number is determined as a minimum of the following values: {number of different prefixes, number of machines x number of processors, user-defined maximum}.

FIN Implementation Details

Both the single-processing algorithm, dFIN, and the way to parallelize it are described in detail in the appropriate papers ([8], [14]). Although [14] uses an older underlying single-processing algorithm on each machine, utilizing it for the newer one is relatively straightforward. There are two fine points in 'dFIN' algorithm that are not explicitly addressed in [8]. These points are elaborated in the sections below. In addition, an option has been provided to apply the work splitting approach of Big FIM to the dFIN algorithm which might be useful for mining FIs on relatively small dense datasets with a particularly low minimal support. This option is also described below.

Algorithm Clarification: Disposal of Unneeded DiffNodesets

The original paper on the 'dFIN' algorithm ([8]) talks of the 'set enumeration tree' to explore the solution space. The search starts with nodes representing a single item or 2-FI and then goes down the tree by producing the child nodes. In fact, there is no need to store the entire tree. To be able to proceed to the next step to find the node's children **it is only required to have the right siblings of the current node and of all its parents**. To make use of this fact the application uses DFS traversal order to explore the solution space and only store a very small subset of all the nodes at any specific moment. This point is extremely important for the algorithm's performance, but it is not explicitly mentioned in the original paper. Foe example, a tree of depth 10 with each node's degree of 5 will require to store no more than 50 nodes (and usually less) at any moment of time comparing to 5^{10} of total tree nodes.

Algorithm Modification: Equivalent Items Collection

The dFIN algorithm described in [8] uses the 'equivalent items' mechanism to detect **closed itemsets** thus significantly reducing the search space. However, it starts collecting the 'equivalent items' only after the 2-items nodes are created. (Specifically, the procedure 'Build_2-itemset_DN' does not collect the equivalent items, while the procedure 'Constructing_Pattern_Tree' does). This means that an opportunity is missed to cut large sub-trees from the solution space earlier.

For example, suppose both items *u* and *v* are frequent and always appear together, and suppose there are two more items *p* and *m* that are more frequent than *u* and *v*. The original dFIN algorithm will create DiffNodeset nodes for *'up', 'um', 'vp', 'vm'* itemsets and will then fully explore their descendants. However, *'v'* is clearly an 'equivalent item' of *'u'*, meaning that only nodes *'uvp'* and *'uvm'* should be explored.

The '*fim-cmp*' implementation fixes this by starting collecting the 'equivalent items' from the single-item node, thus cutting large sub-trees from the solution space earlier.

Algorithm Modification: Fast Processing of Closed Itemsets

The dFIN algorithm produces closed itemsets in the form that has all the required information to generate subsets with the same support. This is one of great advantages of dFIN over e.g.

FP-Growth which can't efficiently identify closed itemsets. Yet the original dFIN algorithm ([8]) explicitly generates all such subsets. In contrast, the '*fim-cmp*' implementation of the dFIN algorithm does not generate these subsets by default, but allows a specific result holder implementation to choose what to do with this information. This means that it is possible to avoid wasting time and memory on generating unneeded itemsets. The user is almost never interested in generation of such itemsets: reporting a closed itemset $(u_1...u_n)v_1...v_k$ where the equivalent items' part is clearly marked shows all the information in concise form.

Algorithm Extension: Option to Split the Work By Longer Prefixes

The work splitting approach described in [14] is similar to the approach of Parallel FP-Growth, which means the work can only be split by a single frequent item. The Big FIM paper ([12]) claims that this is not fine-grained enough leading to unequal load distribution between the machines. As already described, it proposes to split the work by pairs or triples of frequent items instead.

The '*fim-cmp*' tries to explore this idea in the context of the Parallel dFIN algorithm as well. In addition to the 'normal' implementation of the Parallel FIN algorithm described in [14], it provides an option to split the work by pairs or triples of frequent items. If the option is enabled, it first runs the dFIN algorithm on the master node, but traverses the search space in BFS order, thus creating all DiffNodeset nodes up to a user-defined level k (e.g. 2). Then it submits each k-level node to the workers, similarly to the Big FIM. It means that the load balancing between nodes is based on k-length prefix instead of a single-item prefix.

Unfortunately, all the work to compute DiffNodeset nodes up to the level k is done on the driver machine. This includes building and keeping the PPC-Tree for the entire dataset on the driver machine as well, meaning this option is unsuitable for large datasets. However, this option might be preferable to explore smaller dense datasets with a particularly low minimum support.

Experiments

Hardware

All the experiments have been conducted on the publicly-available Amazon EMR framework ([17]). This, combined with the publicly-available source code, yields a complete transparency of the experiments results, as they can be verified by anyone with a relative ease. The details of machines and their configuration are described in *Tables#1* and *#2* below.

Table 1. Experiments' Machines

Purpose	**Type**	**Hardware**
Driver	*m2.xlarge*	2 vCPU, 17.1 GiB memory, 420 SSD GB storage, EBS Storage:none
Executor	*m3.xlarge*	8 vCPU, 15 GiB memory, 80 SSD GB storage, EBS Storage:none
Executor	*m3.2xlarge*	16 vCPU, 30 GiB memory, 160 SSD GB storage, EBS Storage:none

The type, the number of instances, and the specific configuration of the executor machines vary and are specified per experiment.

Table 2. Machine Configuration

Software	emr-5.0.0 stack with Hadoop and Spark
Cluster configuration	[{"classification":"spark-defaults","properties": {"maximizeResourceAllocation":"true","spark.app.name":"fim-cmp"}}]
Step configuration	--class org.openu.fimcmp.cmdline.CmdLineRunner Might also constrain the number of Spark executors, the number of executor cores, and the executor memory via the appropriate parameters, e.g. for 21 cores with 3GB of memory each: --num-executors 7 --executor-cores 3 --executor-memory 3g

Datasets

The experiments have been conducted on 3 datasets from the FIMI repository ([21]).

The dataset details are summarized in the *Table #3* below:

Table 3. Experiments' Datasets

Dataset	**Number of Transactions**	**Size**	**Type**
pumsb	49046	15.9MB	Very Dense
connect	67557	8.8MB	Dense
webdocs	1692082	1.4GB	Sparse

The datasets have been used by other sources to measure the performance of the Big FIM algorithm ([12]) and the Parallel FIN algorithm ([14]).

Comparison Details

Compared Algorithms

Table 4. Compared Algorithms

Name	**Description**
FIN PAR	*fim-cmp* implementation of the Parallel dFIN algorithm
FIN SEQ	*fim-cmp* implementation of dFIN with work splitting inspired by the Big FIM algorithm, as described in the '*Splitting the Work By Longer Prefixes*' section
BIG_FIM	*fim-cmp* implementation of the Big FIM algorithm
FP_GROWTH	Implementation of Parallel FP-Growth algorithm by standard Spark distribution

Comparison Method

The algorithms have been run with the 'counting-only' option enabled, meaning the frequent itemsets are counted, but not explicitly generated. This way the time and complexity of storing a huge number of frequent itemset is avoided and only the time required to compute the itemsets is measured. Other works use similar approach, e.g. [12].

In addition, as mentioned in the 'FIN Implementation Details' section, dFIN has not been asked to explicitly generate all the same-support subsets of closed itemsets. The justification is that the user is almost never interested in generation of such itemsets. Reporting a closed itemset $(u_1...u_n)v_1...v_k$ where the equivalent items' part is clearly marked shows all the required information. Yet the total number of the frequent itemsets is correctly reported, i.e. dFIN does report the same **number** of frequent itemsets as other algorithms. (As mentioned in the 'Correctness Verification' section, there is an option to generate all frequent itemsets, including the same-support subsets of closed itemsets in some canonical order to make sure that all the algorithms produce precisely the same results. This option was not enabled during the performance comparison runs.)

Performance Comparison Experiments

- The algorithm names used below are described in *Table#4*

'Pumsb' Dataset

Table 5. 'Pumsb' dataset: common parameters	
Hardware	1 *m2.xlarge* machine for the driver 7 *m3.xlarge* machines for the executors
Spark nodes configuration	21 total executor cores, 3 GB per executor core
Number of input partitions	20
BIG_FIM split prefix length	3
FIN SEQ split prefix length	2

Table 6. 'Pumsb' dataset: results

Algorithm	Min Support	Time	Result count	Comment
FIN PAR	0.6	23s	19,529,991	
FIN SEQ	0.6	20s	19,529,991	
FP_GROWTH	0.6	37s	19,529,991	
BIG_FIM	0.6	55s	19,529,991	Apriori: 6,350 (23s) Eclat: 19,523,641 (32s)
FIN PAR	0.52	26s	98,620,291	
FIN SEQ	0.52	24s	98,620,291	
FP_GROWTH	0.52	1m 22s	98,620,291	
BIG_FIM	0.52	3m 28s	98,620,291	Apriori: 66,100 (23s) Eclat: 98,554,191 (3m 5s)
FIN PAR	0.44	47s	990,556,543	
FIN SEQ	0.44	1m 20s	990,556,543	
FP_GROWTH	0.44	10m 22s	990,556,543	
BIG_FIM	0.44	15m 52s	990,556,543	Apriori: 127,719 (26s) Eclat: 990,428,824 (15m 26s)
FIN PAR	0.36	3m 20s	12,248,052,423	
FIN SEQ	0.36	3m 28s	12,248,052,423	
FP_GROWTH	0.36	> 3 hours		Stopped after 3 hours
BIG_FIM	0.36	-	-	Not run

'Connect' Dataset

Table 7. 'Connect' dataset: common parameters	
Hardware	1 *m2.xlarge* machine for the driver 7 *m3.xlarge* machines for the executors
Spark nodes configuration	21 total executor cores, 3 GB per executor core
Number of input partitions	20
BIG_FIM split prefix length	3
FIN SEQ split prefix length	2

Table 8. 'Connect dataset: results

Algorithm	Min Support	Time	Result count	Comment
FIN PAR	0.6	19s	21,250,671	
FIN SEQ	0.6	20s	21,250,671	
FP_GROWTH	0.6	25s	21,250,671	
BIG_FIM	0.6	1m 20s	21,250,671	Apriori: 33,410 (21s) Eclat: 21,217,261 (59s)
FIN PAR	0.52	23s	66,899,663	
FIN SEQ	0.52	18s	66,899,663	
FP_GROWTH	0.52	44s	66,899,663	
BIG_FIM	0.52	3m 33s	66,899,663	Apriori: 45,848 (22s) Eclat: 66,853,815 (3m 11s)
FIN PAR	0.44	18s	198,370,615	
FIN SEQ	0.44	21s	198,370,615	
FP_GROWTH	0.44	1m 40s	198,370,615	
BIG_FIM	0.44	8m	198,370,615	Apriori: 59,269 (23s) Eclat: 198,311,346 (7m 37s)
FIN PAR	0.36	22s	582,153,327	
FIN SEQ	0.36	17s	582,153,327	
FP_GROWTH	0.36	4m 40s	582,153,327	
BIG_FIM	0.36	-	-	Not run

'Webdocs' Dataset

Table 9. 'Webdocs' dataset: common parameters	
Spark nodes configuration	FIN, FP_GROWTH: 7 total executor cores, 16GB per executor core BIG_FIM: default configuration (no restrictions)
Number of input partitions	80
BIG_FIM split prefix length	3
FIN SEQ	Not run

Notes:

- Both FIN and FP_GROWTH failed with memory issues on *m3.2xlarge* (32GB) instances when allowed to use multiple cores per executor machine
- Both FIN and FP_GROWTH failed with memory issues on *m3.xlarge* (16GB) instances even when restricted to use only a single core per executor machine
- BIG_FIM had no memory issues on any executor machine even when allowed to use all available processors

Table 10. 'Webdocs' dataset: results					
Algorithm	**Executor machine memory**	**Min Support**	**Time**	**Result count**	**Comment**
FIN PAR	30GB	.175	17m 30s	3,761	
FP_GROWTH	30GB	.175	23m 30s	3,761	
BIG_FIM	30GB	.175	58s	3,761	Apriori: 2,599 (54s) Eclat: 1,162 (4s)
FIN PAR	30GB	.150	> 40m	-	Stopped after 40 minutes
FP_GROWTH	30GB	.150	> 40m	-	Stopped after 40 minutes
BIG_FIM	30GB	.150	1m 03s	10,388	Apriori: 5,846 (55s) Eclat: 4,542 (8s)
BIG_FIM	30GB	.125	1m 18s	38,221	Apriori: 14,981 (1m 1s) Eclat: 23,240 (17s)
BIG_FIM	15GB	.175	1m 12s	3,761	Apriori: 2,599 (1m 6s) Eclat: 1,162 (6s)
BIG_FIM	15GB	.150	1m 17s	10,388	Apriori: 5,846 (1m 5s) Eclat: 4,542 (12s)
BIG_FIM	15GB	.125	1m 45s	38,221	Apriori: 14,981 (1m 17s) Eclat: 23,240 (28s)

Notes on 'webdocs' dataset results (*Table #10*):

(a) [14] reports slightly better performance on both Parallel FIN and Parallel FP-Growth on similar (but not identical) hardware, most probably due to smarter Hadoop and Spark configuration. Yet the reported time in [14] is still much larger than that of the Big FIM implemented by '*fim-cmp*'.

(b) Executor machines' log files on both FIN and FP_GROWTH show extensive garbage-collection activity. This suggests that the reason of very slow performance of these algorithms is insufficient memory.

(c) Most of the time of the BIG_FIM algorithm (on 'webdocs' dataset) is spent on Apriori and preparation for Eclat. The Eclat part only takes several seconds to complete. In contrast, in 'pumsb' and 'connect' dense datasets Eclat computation took most of the time as it computed the most of the FIs.

Scalability Experiments

There were less experiments on algorithms' scalability than on algorithms' performance. The reasons are:

- Cost: the cost of running experiments on a large number of Amazon EMR machines is above the allotted budget
- Benefit: Parallel dFIN algorithm uses the same mechanism of work split as Parallel FP-Growth, so both algorithms should have similar scalability figures. Big FIM has even stronger work split mechanism: the scalability of the parallel Apriori depends on the tunable number of input partitions, as it can process each partition locally and only share counts; the split between Eclat jobs is tunable and can be based on a prefix of any length, unlike PFP-Growth that can only use a single item for the work split.

Yet a limited number of experiments have been performed to check the *fim-cmp* implementation. The goal was to verify the claim that

(A) the scalability of the *fim-cmp* implementation of Parallel dFIN algorithm is similar to that of Parallel FP-Growth, and the scalability of the *fim-cmp* implementation of Big FIM is similar or better than that of Parallel FP-Growth.

For this purpose each algorithm is run on some chosen dataset with fixed parameters on 2, 4, and 8 machines (specifically, on 1, 3, and 7 *m3.xlarge* executor machines and one *m2.xlarge* driver). Then the improvement ratio of each algorithm is compared to that of Parallel FP-Growth. The ***improvement ratio*** = (time of run with 1 executor) / (time of the current run).

Table 11. Scalability experiments: run parameters per algorithm

FIN PAR	'pumsb' dataset, min support = 0.44 3 cores per executor machine, 3GB per executor core
FP_GROWTH	'pumsb' dataset, min support = 0.52 3 cores per executor machine, 3GB per executor core
BIG_FIM on pumsb	'pumsb' dataset, min support = 0.52, splitting prefix length = 3 3 cores per executor machine, 3GB per executor core
BIG_FIM on webdocs	'webdocs' dataset, min support = 0.175, splitting prefix length = 3 Default configuration (no restrictions)

Table 12. Scalability experiments: results

Algorithm	**# of executor instances**	**Time**	**Improvement ratio**
FP_GROWTH	1	230s	-
FIN PAR	1	150s	-
BIG_FIM on pumsb	1	521s	-
BIG_FIM on webdocs	1	243s	-
FP_GROWTH	3	108s	230/108=2.13
FIN PAR	3	85s	150/85=1.76
BIG_FIM on pumsb	3	214s	521/214=2.43
BIG_FIM on webdocs	3	114s	243/114=2.13
FP_GROWTH	7	82s	230/82=2.8
FIN PAR	7	47s	150/47=3.2
BIG_FIM on pumsb	7	148s	521/148=3.52
BIG_FIM on webdocs	7	72s	243/72=3.375

Although is not possible to draw any certain conclusions due to a small number of experiments, the results in *Table#12* do support the claim (A). FIN PAR shows a better ratio than FP_GROWTH in one of the two cases. BIG_FIM shows the same or better ratio than FP_GROWTH in all the cases.

Evaluation

Dense Datasets

The 'Parallel dFIN' algorithm is clearly the fastest on dense datasets.

As can be seen in *Tables #6* and *#8*, it runs faster than the standard Spark implementation of the Parallel FP-Growth and much faster than the Big FIM. The lower the minimum support the clearer the advantage of the Parallel dFIN over the PFP-Growth. It took only about 3 and half minutes for the Parallel dFIN to identify more than 12 billion of FIs on the 'pumsb' dataset with minimum support of 0.36 on 8 machines. The PFP-Growth could not complete the task within 3 hours.

The main reason is of course that the (single-processor) dFIN algorithm is more efficient than the (single-processor) FP-Growth. The most obvious causes of this are:

- The DiffNodesets structure allows generating the child itemsets faster as it does not require creation of conditional FP-Trees
- dFIN is able to skip entire branches of computation due to its ability to identify 'equivalent items', i.e. unlike FP-Growth, it is able to identify closed itemsets

The Big FIM is clearly the slowest algorithm to run on dense datasets.
The reason is that its underlying Eclat algorithm has much **larger cost per generated FI** than either FP-Growth or dFIN, once they generate the FP-Tree/PPC-Tree. To generate a new FI Eclat needs to 'and' two tid-lists which is usually proportional to the total number of transactions, even if Eclat with diffsets is used. In contrast, both FP-Growth and dFIN run on **compressed data structures**. Generating a new FI with dFIN only requires traversal of all the PPC-Tree nodes of the same item, which is normally a fast operation.

The application of the prefix-based work split strategy in the context of dFIN algorithm, i.e. the 'FIN SEQ' has shown results comparable to those of the 'normal' Parallel dFIN algorithm on small and dense datasets. It might still outperform the 'normal' Parallel dFIN on larger number of machines due to a more powerful split strategy, but this is purely a speculation. As already stated, the 'FIN SEQ' algorithm as its stands now is not applicable to large datasets whose PPC-Tree can't be hold on a single machine.

Sparse Datasets

As can be seen in *Table#10*, the Big FIM algorithm was clearly the fastest on large sparse datasets. The most probable reason is its much lower memory requirements (see note (b) to *Table#10*).

Both the PFP-Growth and the Parallel dFIN algorithm are required to build the entire FP-Tree/PPC-Tree on each group-dependent dataset. These trees' memory footprint might not be significantly smaller than the memory footprint of the FP-Tree/PPC-Tree of the entire original dataset. In contrast, the memory requirements per machine of the Big FIM could be very small on sparse datasets. Let's shortly examine both its Parallel Apriori and Eclat phases:

- Parallel Apriori: this phase is required to generate tid-lists for each k-FI, where k is a user parameter which is normally either 3 or 4. It first either counts the itemset frequency or generates the tid-list per partition. The partitions could be made as small as required to fit the memory requirements. Then the tid-lists from each partition are merged. The total number of bits required to hold all the tid-lists of all k-FIs is
  (total number of k-FIs) x *(total number of transactions).*
  The expectation is that sparse datasets don't have a huge number of the k-FIs (i.e. 3-FIs or 4-FIs).
- Eclat: each executor core receives a group of k-FIs with the same (k-1)-length prefix and runs Eclat on this group to generate the rest of FIs in BFS order. As implemented by *fim-cmp*, the maximal number of FIs that might exist for the given k at the same time is:
  $\max_{m>k-1}$ *((total number of m-FIs with the same (k-1)-length prefix)* x
  (total number of (m+1)-FIs with the same (k-1)-length prefix))
  Again, this number is usually relatively small for sparse datasets for k that is either 3 or 4.

The slowest algorithm on large sparse datasets was PFP-Growth with the Parallel dFIN coming second. The reasons are most probably the same as listed in the section on the dense datasets evaluation.

Conclusion

Two modern Map/Reduce algorithms, 'Big FIM' ([12]) and 'Parallel dFIN' ([8], [14]) have been implemented in Spark framework and their performance have been compared with that of 'Parallel FP-Growth' algorithm ([13]) using Amazon EMR cloud ([17]) on FIMI datasets ([21]). A number of innovative steps has been made to ensure the efficiency of the implementation.

The implemented application, '*fim-cmp*', allows the user to run the three supported algorithms, 'Parallel dFIN', 'Big FIM', and 'PFP-Growth', on any Spark cluster and further investigate their performance through a number of command-line options to tune each algorithm.

The experiments have shown that:

1. **The 'Parallel dFIN' outperforms the PFP-Growth on all types of input datasets and might be much faster on very dense datasets and low minimal support.**
2. **The 'Big FIM' is much slower on dense datasets than either 'Parallel dFIN' or PFP-Growth. At the same time the 'Big FIM' was much faster on large sparse datasets than the other two algorithms due to its much lower memory requirements.**

The reasons for these observations have been briefly examined.

References

1. Agrawal Rakesh, Srikant Ramakrishnan (September 1994). Fast algorithms for mining association rules in large databases. In Bocca Jorge B., Jarke Matthias, and Zaniolo Carlo (Eds.). In Proceedings of the 20th International Conference on Very Large Data Bases, pp. 487-499. Santiago, Chile.
2. Zaki M. J. (2000). Scalable algorithms for association mining. IEEE Transactions on Knowledge and Data Engineering. 12 (3), pp. 372–390.
3. Han Jiawei (2000). Mining Frequent Patterns Without Candidate Generation. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. SIGMOD '00: 1–12.
4. Pei Jian, Han Jiawei, Mao Runying (2000). CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. In Proceedings 2000 ACM-SIGMOD Int. Workshop Data Mining and Knowledge Discovery, pp. 21- 30.
5. Li Jianwei, Choudhary Alok N., Jiang Nan, Liao Wei-keng (April 2006). Mining Frequent Patterns by Differential Refinement of Clustered Bitmaps. In Proceedings of the Sixth SIAM International Conference on Data Mining, April 20-22, 2006, Bethesda, MD, USA.
6. Deng Z. H., Wang Z., Jiang J. (2012). A New Algorithm for Fast Mining Frequent Itemsets Using N-Lists. SCIENCE CHINA Information Sciences, 55 (9), pp. 2008 - 2030
7. Deng Z. H., Lv S. L. (2014). Fast mining frequent itemsets using Nodesets. Expert Systems with Applications, 41(10), pp. 4505–4512.
8. Deng Z. H. (2016). DiffNodesets: An Efficient Structure for Fast Mining Frequent Itemsets. Applied Soft Computing, 41, April 2016, pp. 214-223.
9. Riondato Matteo, DeBrabant Justin, Fonseca Rodrigo L. C., Upfal Eli (2012). PARMA: A Parallel Randomized Algorithm for Approximate Association Rules Mining in MapReduce. In Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM 2012).
10. Qiu Hongjian, Gu Rong, Yuan Chunfeng, Huang Yihua (May 2014). YAFIM: A Parallel Frequent Itemset Mining Algorithm with Spark. In 2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW).
11. Zaki M. J. (2000). Hierarchical Parallel Algorithms for Association Mining. Rensselaer Polytechnic Institute, Troy, NY 12180, USA.

12. Moens S., Aksehirli E., Goethals B. (October 2013). In Big Data, 2013 IEEE International Conference.
13. Li Haoyuan, Wang Yi, Zhang Dong et al (January 2008). PFP: Parallel FP-Growth for Query Recommendation. In Proceedings of the 2008 ACM Conference on Recommender Systems, RecSys 2008, Lausanne, Switzerland, October 23-25, 2008.
14. Lin Chen, Gu Junzhong (June 2016). PFIN: A Parallel Frequent Itemset Mining Algorithm Using Nodesets. International Journal of Database Theory and Application, Vol.9, No.6 (2016), pp.81-92.
15. Liu Ying, Liao Wei-keng, Choudhary Alok, Li Jianwei (December 2007). Parallel Data Mining Algorithms for Association Rules and Clustering. pp. 9-14.
16. https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home
17. https://aws.amazon.com/emr/
18. http://hadoop.apache.org/
19. http://spark.apache.org/
20. Mehdi Zitouni, Reza Akbarinia, Sadok Ben Yahia, Florent Masseglia (September 2015). A Prime Number Based Approach for Closed Frequent Itemset Mining in Big Data. DEXA'2015: 26th International Conference on Database and Expert Systems Applications, Sep 2015, Valencia, Spain.
21. Frequent Itemset Mining Implementations Repository. http://fimi.cs.helsinki.fi/.
22. Yaling Xun, Jifu Zhang, Xiao Qin (January 2015). FiDoop: Parallel Mining of Frequent Itemsets Using MapReduce.
23. M. J. Zaki, K. Gouda (2003). Fast vertical mining using diffsets. In Proceedings of the ACM SIGKDD, pages 326–335.

Appendix A: Command-Line Interface Options

Command-Line Interface Structure

The general structure is: <algorithm-name> <algorithm-specific options>

The algorithm names can be listed with '*--help*' option, the currently supported ones are:

BIG_FIM | FIN | FP_GROWTH .

The algorithm-specific options can be listed with '*--help*' after the algorithm name, e.g. '*BIG_FIM --help*'.

BIG_FIM Options

Essential options:

--spark-master-url <arg>	Spark master URL
--input-file-name <arg>	Either an absolute path or a path relative to 'FIM_CMP_INPUT_PATH' environment variable
--min-supp <arg>	Min support
--input-parts-num <arg>	Number of partitions to read the input file

Algorithm-tuning options:

--persist-input <arg>	Whether to persist the input RDD
--start-eclat-prefix-len <arg>	Prefix length to switch from Apriori to Eclat
--curr-to-prev-res-ratio-threshold <arg>	Threshold to determine sparse datasets for which continue with Apriori, default: 1.1
--eclat-parts-num <arg>	Number of partitions to use for Eclat
--eclat-use-diff-sets <arg>	Eclat: whether to enable the diffsets
--eclat-use-squeezing <arg>	Eclat: whether to enable the squeezing
--use-kryo	Whether to use kryo serialization library

Output options:

--eclat-cnt-only <arg>	Whether to only count the FIs instead of actually collecting them
--print-intermediate-res <arg>	Whether to print F1, F2, ..., and also some progress info
--print-all-fis <arg>	Whether to print all found frequent itemsets

Usage example:

%SPARK_HOME%\bin\spark-submit --class org.openu.fimcmp.cmdline.CmdLineRunner --master spark://192.168.1.68:7077 --driver-memory 1200m --executor-memory 1200m file://c/projects/fim-cmp/target/fim-cmp-1.0-SNAPSHOT.jar BIG_FIM --spark-master-url spark://192.168.1.68:7077 --input-file-name pumsb.dat --min-supp 0.8 --start-eclat-prefix-len 3 --input-parts-num 3 --eclat-parts-num 6 --eclat-cnt-only true

FIN Options

Essential options:

--spark-master-url <arg>	Spark master URL
--input-file-name <arg>	Either an absolute path or a path relative to 'FIM_CMP_INPUT_PATH' environment variable
--min-supp <arg>	Min support
--input-parts-num <arg>	Number of partitions to read the input file

Algorithm-tuning options:

--persist-input <arg>	Whether to persist the input RDD
--run-type <arg>	SEQ_PURE_JAVA | SEQ_SPARK | PAR_SPARK
--itemset-len-for-seq-processing <arg>	The required itemset length of the nodes processed sequentially on the driver machine, e.g. '1' for items
--use-kryo	Whether to use kryo serialization library

Output options:

--cnt-only <arg>	Whether to only count the FIs instead of actually collecting them
--print-intermediate-res <arg>	Whether to print F1, F2, ..., and also some progress info
--print-all-fis <arg>	Whether to print all found frequent itemsets

Usage example:

%SPARK_HOME%\bin\spark-submit --class org.openu.fimcmp.cmdline.CmdLineRunner --master spark://192.168.1.68:7077 --driver-memory 1200m --executor-memory 1200m file://c/projects/fim-cmp/target/fim-cmp-1.0-SNAPSHOT.jar FIN --spark-master-url spark://192.168.1.68:7077 --input-file-name pumsb.dat --min-supp 0.8 --input-parts-num 3 --run-type PAR_SPARK --cnt-only true

FP_GROWTH Options

Essential options:

--spark-master-url <arg>	Spark master URL
--input-file-name <arg>	Either an absolute path or a path relative to 'FIM_CMP_INPUT_PATH' environment variable
--min-supp <arg>	Min support
--input-parts-num <arg>	Number of partitions to read the input file

Algorithm-tuning options:

--persist-input <arg>	Whether to persist the input RDD
--use-kryo	Whether to use kryo serialization library

Output options:

--cnt-only <arg>	Whether to only count the FIs instead of actually collecting them
--print-intermediate-res <arg>	Whether to print F1, F2, ..., and also some progress info
--print-all-fis <arg>	Whether to print all found frequent itemsets

Usage example:

%SPARK_HOME%\bin\spark-submit --class org.openu.fimcmp.cmdline.CmdLineRunner --master spark://192.168.1.68:7077 --driver-memory 1200m --executor-memory 1200m file://c/projects/fim-cmp/target/fim-cmp-1.0-SNAPSHOT.jar FP_GROWTH   --spark-master-url spark://192.168.1.68:7077 --input-file-name pumsb.dat --min-supp 0.8 --input-parts-num 3 --persist-input false --cnt-only true