Twig: An Adaptable and Scalable Distributed FPGrowth

Rubens Moreira Bruno Coutinho Filipe Arcanjo Rodrigo Rocha
Wagner Meira Jr. Dorgival Guedes Renato Ferreira
Department of Computer Science
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
{*rubens, coutinho, filipe, rcor, meira, dorgival, renato*}*@dcc.ufmg.br*

***Abstract*—Despite the advances in recent decades, achieving high performance for several parallelized applications is still a challenge. Data mining algorithms are often hard to parallelize and scale as a consequence of being irregular and intensive in terms of both computation and I/O. One example of such challenge is Frequent Pattern Mining (FPM), which is one of the most popular and relevant tasks in the area. It allows the extraction of co-occurring patterns (e.g., itemsets, sequences, graphs) in the input data set. As the problem is #P-complete, distributed approaches must strive to succeed against the pitfalls of load imbalance, avoiding that the execution be delayed by a few overloaded computers. This paper presents Twig, a novel strategy for parallelizing a state-of-the-art frequent itemset mining algorithm, FPGrowth. Our solution consists of two main phases: (i) the building of a partitioned FPTree representation of the entire data set and (ii) the parallel extraction of frequent itemsets, with a distributed version of the FPGrowth algorithm. We considerably reduced primary memory requirements, as the FPTree may be fully partitioned transaction-wise among computers while incurring a low communication overhead. Our solution also exhibits sub-linear communication growth on scalability experiments and outperforms very optimized sequential and distributed approaches. It achieves up to 78% efficiency on a cluster of 9 machines while processing 279GB of spam data.**

***Keywords*-data mining, fpgrowth, load balancing, distributed processing, parallel programming, scalability**

I. Introduction

Despite the evolution of parallel architectures and programming languages, achieving high performance for several sets of parallel applications is still challenging. An important such set is that of data mining applications. They are computationally complex, rarely scale linearly with the input size [1], and are often hard to parallelize and scale as a consequence of being irregular and I/O and computation-intensive. We may consider this issue along three dimensions: distribution/parallelization (assigning parts of the problem to multiple machines), skewing/irregularity (dealing with different I/O and compute demands across partitions), and scalability (avoiding excessive overheads as the number of machines rises). We may achieve distribution by partitioning the data and the tasks in a way that different tasks can be executed in parallel, with minimal data access conflicts. Irregularity can be handled by incorporating adaptation mechanisms that adjust the partitions to balance computation. Finally, scalability can be achieved by minimizing communication overheads among partitions, *i.e.*, improving the reference locality of the resulting distributed application. These three dimensions are hard to deal with in isolation; together, they present an even more difficult challenge.

Itemset mining is a class of data mining applications which too are subjected to these challenges. Given a multiset of transactions of items, the goal is to determine the itemsets (*i.e.*, sets of items) that co-occur frequently enough (for some threshold). This problem is #P-complete and proposed strategies aim to reduce the search space of the problem and to improve the efficiency and scalability of known algorithms [1]–[3]. Many such strategies try to reach their goals through parallel and distributed computing. However, parallel itemset mining is a challenging task. In solving it, algorithms are required to find relationships between transactions and itemsets, so it is not efficient to simply partition data across one of those dimensions, since that would result in either data replication or costly mappings in terms of communication when computations focus on transactions or patterns (itemsets). As the frequency of patterns is data-dependent, it is not possible to easily predict the load due to any partition pattern beforehand, so it is necessary to adjust partitions to balance load. On the other hand, the fact that the application has to evaluate multiple potential patterns simultaneously is a parallelization opportunity to be exploited. A natural cost-effective method is to parallelize the computation on clusters of machines, exploiting benefits of aggregate memory and disk space, along with parallel I/O and processing capabilities [1].

The issue we address in this paper is how to partition the data in such a way that accesses to both transactions and patterns present good reference locality, while the computation load associated with partitions is evenly distributed across the processors. Such partitioning scheme must be easy to calibrate w.r.t. data and running environment characteristics. Our solution is **Twig**, a novel distributed algorithm to compute frequent itemsets. It consists of an FPTree partitioning scheme and a distributed version of the FPGrowth algorithm [3]. Twig works by building a partitioned FPTree representation of the entire data set,

which is later used to extract frequent patterns (i.e., itemsets) in a distributed fashion.

The main contributions of this paper are: (i) an adaptable and *data-conscious* partitioning scheme at the granularity of transactions, which permits a complete and balanced distribution of the dataset, as well as of the tree that the algorithm builds and its associated projections, with a low communication overhead; (ii) Twig, which is an implementation of the proposed scheme in the filter-labeled stream paradigm, on top of the Watershed programming framework, and (iii) extensive experimental evaluation of Twig using two real data sets (spam and twitter), where we observed sub-linear scalability of communication with the number of computers, better performance than both very optimized sequential [4] and distributed [5] FPM implementations, and near linear speedups, with an efficiency of 78%, while processing a real spam data set of $279\texttt{GB}$ on a cluster of 9 machines.

II. Frequent Itemset Mining

The frequent itemset mining problem (FIM) was originally defined by Agrawal *et al.* [6] in the context of association rule mining. Formally, let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of n items, $D = \{T_1, T_2, \ldots, T_m\}$ be a multiset of m transactions where $T_i \subseteq I$ for $i = 1, ..., m$. A non-empty subset of I containing exactly k items is a k-itemset. The support of a k-itemset is the number of transactions from D in which that itemset occurs. The frequent pattern mining problem is to find all itemsets, among the potential $2^n - 1$ possible ones, which have a support greater than, or equal to a given threshold $\mathrm{minsupp}$.

The first practical solution to FPM was the Apriori algorithm, proposed in 1994 by Agrawal *et al.* [7]. Apriori is able to strip-off some sets from the powerset of I by using an algebraic structure named lattice, but requires several data base scans to check the frequency of each itemset. Later, on year 1997, Zaki *et al.* [2] proposed the algorithm Eclat, which reduces the number of data base scans by storing, on main memory, inverted indexes of transactions that contain the itemsets represented in the lattice. The solution is more efficient than the previous approach, but requires a large enough memory to keep the inverted indexes, and generates non-frequent candidate itemsets, which incur in non-productive work.

In 2000, Han *et al.* [3] proposed an algorithm named FPGrowth, along with a data structure named FPTree. FP-Growth provides data compression, while also guaranteeing that only frequent itemsets are considered, *i.e.*, without candidate itemset generation. Based on this approach, many parallel and distributed solutions have been proposed. Buerher et al. [8], in 2007, developed a partitioning scheme that could process 1 terabyte of data in a cluster of 48 computers. Despite the optimizations performed, the partitioning policy adopted may increase the required amount of primary memory as new machines are added to the cluster. Our distributed strategy comprises an efficient partitioning of the FPTree, as well as a parallel version of FPGrowth for distributed frequent itemset mining. In the next section, we present an overview of FPGrowth.

A. FPGrowth Algorithm

We begin our overview by defining an FPTree, the data structure essential to FPGrowth. An FPTree is prefix-trie structure in which the nodes have been augmented with integers representing frequencies. Figure 1 depicts an FPTree built from a sample data set D.

FPTree ($minsupp = 2$)				
Frequencies		Transactions		
Item	$f(i)$	id	Initial	Sorted
a	6	1	b a	a b
b	4	2	c a b e	a b c e
c	4	3	a f	a
d	3	4	c d a	a c d
e	2	5	c d b	b c d
f	1	6	g c a	a c
g	1	7	e d b a	a b d e

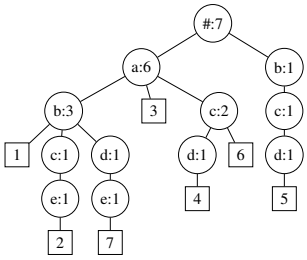

Figure 1: The FPTree is built by inserting each sorted transaction in the tree. Round nodes are items $\{\text{label}_i : \text{freq}(i)\}$, and square nodes are ids of transactions within each path.

To build an FPTree, one must perform two scans on the data set: a first one to determine the frequency of each item, and a second one to process each transaction based on those frequencies and insert them in the FPTree. During the second scan, infrequent items are discarded from each transaction. The remaining ones are sorted in non-increasing order of frequencies, resulting on a string $s(T_i)$ that represents the transaction. The FPTree is built by inserting each transaction string $s(T_i)$ on an augmented trie. In addition to symbols, each node on this trie contains an integer counter for the number of strings $s(T_i)$ passing through that node. The FPTree also contains lateral pointers, linking nodes that share the same item label. Such links can be used to improve the performance of the algorithm during the extraction of frequent itemsets.

The frequencies along a path are always monotonically non-increasing, *i.e.*, a child node is either as frequent as its parent node, or less frequent. By placing the most frequent items close to the root, the FPTree exploits the overlaps among very frequent itemsets, thus reducing the tree size. It also permits pruning of the itemset search space, which both speeds up the execution — as fewer itemsets have to be processed — and mitigates the impact of data skewness while dealing with parallel variations of the algorithm.

Once the tree is built, we must extract frequent itemsets. Notice that a simple enumeration of sub-paths in the tree does not yield the correct result. For instance, in Fig. 1, consider the itemset $\{\texttt{a}, \texttt{c}\}$, which is frequent, since we

defined $\mathrm{minsupp} = 2$, and $\{\mathtt{a},\mathtt{c}\}$ occurs three times (in Transactions $2, 4$ and 6). A simple DFS would traverse the path $\mathtt{\#},\mathtt{a},\mathtt{b},\mathtt{c}$, which contains Transaction 2, but would not consider it frequent, since it would not lead directly to the sub-path $\{\mathtt{a},\mathtt{c}\}$. Later in the DFS we would find $\{\mathtt{a},\mathtt{c}\}$ in the path $\mathtt{\#},\mathtt{a},\mathtt{c}$ as frequent, but that only holds Transactions 4 and 6, *i.e.*, the frequency count of such itemset would be off by 1.

FPGrowth addresses these corner cases by recursively projecting the FPTree. Given an itemset X and an FPTree $\mathtt{FP}$, we project $\mathtt{FP}$ over X by building a new FPTree $\mathtt{FP}_X$, containing all transactions from $\mathtt{FP}$ in which X occurs. $\mathtt{FP}_X$ is also known as conditional-FPTree, or CFPTree — considering the $\mathtt{FP}$ conditioned to the occurrence of X. Consequently, each CFPTree has an associated itemset, which may or may not be frequent w.r.t. $\mathrm{minsupp}$.

In conclusion, to extract frequent itemsets, the FPTree is initially projected on all 1-itemsets, which are known to be frequent — infrequent items were already discarded. Each resulting CFPTree is projected recursively, generating associated k-itemsets. Whenever the generated itemset X occurs less than $\mathrm{minsupp}$ times, its associated CFPTree is discarded, since no itemset that contains X can be frequent in the data set. On the other hand, if X is frequent, the projection continues until the tree is a linear graph, from which all subsets are enumerated. The support of each itemset is the frequency of its least frequent item.

B. Challenges

We identified three major challenges while parallelizing the FPGrowth algorithm.

1) Performance Scalability: As we shall discuss in Section V, very efficient single-node solutions to FPM cannot scale over large data sets, since hardware limitations are reached rather soon. Since data sets and associate meta data structures may even exceed the disk capacity of a single machine [8], the execution of local solutions are limited by I/O performance, regardless of eventual optimizations. Not only the efficiency is compromised, but depending on the size required to represent the data set in memory, it may be the case that only out-of-core approaches can be applied to single-computer strategies. In conclusion, it is hard to effectively scale FPM using a single computing node, and thus distributing the computation among computers in a cluster seems absolutely necessary.

2) Data and Computation Distribution: The first challenge for parallelizing the FPGrowth algorithm is that it traverses the tree in a depth-first search fashion, which has been proved to be P-complete and hard to parallelize [9]. Regarding the data, we access three data organizations during the execution of the algorithm: transaction, FPtree and projection. The transactions are accessed for counting the frequency of items and building the tree, and the usual distribution is to break the transaction database horizontally, that is, each partition contains a set of transactions. The FPtree, as mentioned, represents the item co-occurrence and has to be partitioned while it is being built, which induces a vertical partition of it. Finally, the projection has to merge sub-trees that contain a given itemset in a recursive fashion. In this case, the challenge is not only to perform the partitions, but also to transition from one partition scheme to the other.

3) Load balancing: The transactions are initially split into disjoint chunks, evenly distributed among computers. This partitioning is simple, and does not guarantee load balancing, *i.e.*, some chunks may contain many distinct items, and yield rather complex FPTrees. Overloaded or slower nodes, a.k.a. *stragglers*, tend to delay the entire execution, by causing synchronization problems and idleness in the remaining computers. Thus, it is important that no computer gets hindered by load imbalance (data skewness). Although estimating the complexity of each FPTree would require a prior characterization of the input, one may derive heuristics to address load imbalance during the execution of a distributed FPM solution. Since the goal is to even out the initial distribution of work, there is no way to escape exchanging data among computer nodes. Therefore, the crucial point is to establish a nice compromise between communication and load balancing.

III. Twig: Distributed FPGrowth

There are many challenges in distributed itemset mining. As discussed in section II-B, distributed solutions must withstand hardware limitations, such as disk, memory, and network restraints, as well as avoid the pitfalls of data replication and skewness. To address such issues, we propose Twig, an adaptable and efficient distributed FIM solution.

A. Twig Algorithm Overview

Twig is a distributed algorithm for mining frequent itemsets from very large data bases. In a nutshell, the algorithm has two main stages: first, we build a global FPTree, partitioned over the available computing nodes; later, we extract frequent itemsets using a distributed version of the FPGrowth algorithm. Figure 2 depicts a high-level execution of Twig, which takes the following parameters: an input data set $D = \bigcup_{i=1}^{p} D_i$, partitioned over p processors; a frequency threshold $\mathrm{minsupp}$; and two partitioning levels, μ and ρ. To simplify the understanding, let us first summarize the main steps that compose our approach.

We begin by counting, at each processor p, the frequencies of items from the data set partitions D_i (step 1, *InputReader*). To aggregate local counts into a global table of frequencies, each pair $\langle \mathrm{item}, \mathrm{frequency} \rangle$ is mapped by the item to a specific processor (step 2, *ItemCounter*). The resulting table is broadcast to *FPTreeBuilders*, to build local FPTrees (step 3). Since each processor now has the global table of frequencies, we are able to build FPTrees that

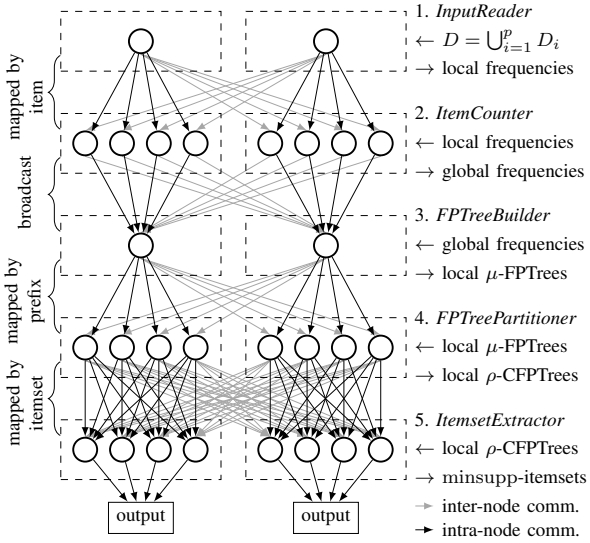

Figure 2: High-level Twig Algorithm flow. This diagram depicts the execution of Twig with two computing nodes, each having 4 processing units. Each stage of the algorithm is executed by 4 different processes, except for *InputReader* and *FPTreeBuilder*, which would have threads serialized since they perform heavy disk access. We label each step, as well as associated data input (←) and output (→) flow.

only store items with frequency greater than, or equal to $\mathrm{minsupp}$. Twig uses three main data structures: μ-FPTrees, composed of FPTree prefixes and subtrees; a global FP-Tree, partitioned over available computers; and ρ-CFPTrees, which are basically FPTree projections.

Firstly, consider FPTrees built from two different chunks: the first has many inner branches, while the second is a linear graph. Depending on the algorithm, the running time may vary greatly among trees. To mitigate such load imbalance, we partition local FPTrees into prefixes and subtrees, which are distributed among computing nodes. Subtrees are mapped by associated prefixes to *FPTreePartitioners* (step 4). We define prefixes and subtrees as follows:

Definition 1: Consider a chunk $D_i = \{T_j, T_{j+1}, ..., T_x\}$ of the input data set, and a maximum length $\mu \in \mathbb{N}$. A path P, beginning at the root node of an FPTree $\mathtt{FP}$, is a prefix:

- if $|P| < \mu$, and $\exists\ T_k \in D_i \mid P = T_k$;
- or if $|P| = \mu$ (P has exactly μ nodes).

Definition 2: Consider an FPTree $\mathtt{FP}$, and one of its prefixes, P. A subtree $\mathtt{FP}_P$ is a tree structure rooted at the last node of P, and the pair $\langle P, \mathtt{FP}_P \rangle$ is a μ-FPTree.

The distribution of prefixes and subtrees, or μ-FPTrees, results in the second data structure used by Twig, a global FPTree, partitioned across the cluster of machines. For any pair of transactions $T_i, T_j \in D$, we have that prefix of T_i equals prefix of T_j if, and only if, T_i and T_j belong to the same partition. This property, which comes as a consequence of our load balancing solution, allows us to improve data compression: we exploit the fact that transactions with prefixes in common may also have common remaining items, which will be represented only once in the same FPTree. Finally, *FPTreePartitioners* project each partition up to ρ-itemsets, i.e., projections yielding $1, 2, ..., \rho$-itemsets. Associated CFPTrees and itemsets compose the third main data structure used in our solution: ρ-CFPTrees, which are mapped by itemsets to *ItemsetExtractors* (step 5). Once all ρ-CFPTrees associated with an itemset X are merged by the assigned computing node, frequent itemsets $F_i \supseteq X$ are extracted and written to the output device (step 5). Sections III-B and III-C explain in detail the two main stages of Twig: the building of a global partitioned FPTree, and the distributed frequent itemset extraction.

B. FPTree Partitioning

The first partitioning of our solution is employed over local FPTrees (step 3 to 4). The motivation is the potential load imbalance from the input data set: a particular chunk may be *harder* to process, yielding more complex tree structures, and therefore lagging the execution of the associated thread. In order to mitigate imbalance, we build FPTrees locally, and later split such structures into subtrees, which are then distributed among computers. The subtree extraction is parameterized, confering greater adaptability to Twig, in face of the many possible input data distributions.

1) FPTreeBuilders: To build local FPTrees, each *InputReader* (step 1) reads its assigned data base chunk, calculating local frequencies of items. Next, each pair $\langle \mathrm{item}, \mathrm{frequency} \rangle$ is mapped by item to *ItemCounters*, to determine global frequencies. The resulting table is broadcast to *FPTreeBuilders*, which perform another data base scan, building local FPTrees. The process is shown in algorithm 1. Assume data base D to have m transactions, n distinct items,

Algorithm 1: Frequency counting and local FPTrees

```
1 freqs_local ← count_items(D_i);
2 SEND freqs_local;
3 freqs_global ← RECEIVE_FROM_ALL;
4 fptree_local ← FPTree(D_i, minsupp, freqs_global);
```

and consider we run p parallel processes. Using a hash table, local frequencies are computed in $O(mn/p)$ time, and global frequencies are reduced in $O(np)$. Given $m \gg n$, we may consider $O(mn/p + np) = O(mn/p)$. Each local chunk has m/p transactions, each holding at most n items. Thus, local FPTrees are built in $O(mn^2/p)$, as each item insertion in the FPTree requires a $O(n)$ time scan in the child list of its parent node[1]. The time complexity of building local

[1]Notice this complexity may be reduced by using a hash data structure, but it may not be cost-effective, since n is small, and hashing keys may become a costly overhead.

FPTrees is $O(mn^2/p)$, which requires $O(mn/p)$ space to store nodes in each local tree.

Recall the sample data base from Figure 1. We partition the data into two chunks, the first with transactions 1 to 4, and the second, transactions 5 to 7. Figure 3 shows FPTrees built from such chunks, as well as the subtrees extracted with a maximum prefix length $\mu = 2$.

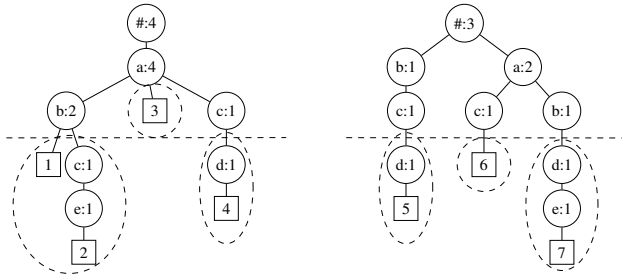

Figure 3: Each *FPTreeBuilder* extracts $\langle \text{prefix}, \text{subtree} \rangle$ pairs, according to a prefined prefix length ($\mu = 2$). Square nodes simply indicate which transactions end at each path.

Algorithm 2 is executed by *FPTreeBuilders*, which map pairs $\langle \text{prefix}, \text{subtree} \rangle$ by using prefix items as key. Such pairs are mapped to *FPTreePartitioners*. The label is computed as a hash function over the prefix. Regard *tids* as the list of transactions ending at a given node, and *serialize()* as a compressed integer array representation of a tree. The tree encoding is akin to that presented by Buerher et al. [8].

Algorithm 2: Extraction of μ-FPTrees

```
1 for each node ← DFS(fptree_local) do
2 |   if (node.level < μ) and (node.tids ≠ ∅) then
3 |   |   SEND serialize(⟨node.prefix, node.tids⟩);
4 |   else if node.level = μ then
5 |   |   SEND serialize(⟨node.prefix, node.subtree⟩);
6 |   end
7 end
```

2) FPTreePartitioners: Once at *FPTreePartitioners*, μ-FPTrees are merged into global FPTree partitions. Merging is the key operation of our solution: pairs $\langle \text{prefix}, \text{subtree} \rangle$ are traversed in DFS, and merged onto local partitions of the global FPTree. As we present in algorithm 3, each node from the μ-FPTree is *inserted*, either by creating a new node in the local partition, or by adding the received frequency to an existing node.

As discussed in section II-B, load balance contributes greatly to performance. Our partitioning strategy is a compromise between load balancing and data replication: whenever a prefix with less than μ items exists, its nodes may be stored by two different partitions. However, beyond the improved load distribution, common subtrees from different processors may be merged into single structures, thus reducing storage and processing requirements. These contributions are possible due to the following property:

Algorithm 3: Global FPTree Partitions from μ-FPTrees

```
1 node ← partition.root;
2 for each in_node ← DFS(μ-FPTree) do
3 |   while in_node.level ≤ node.level do
4 |   |   node ← node.parent;
5 |   end
6 |   node ← node.insert(in_node);
7 end
```

Proposition 1: Given a data set $D = \{D_1, D_2, ..., D_p\}$, and any pair of transactions $T_i, T_j \in \texttt{DB} \mid T_i = T_j$. Prefix of T_i equals prefix of T_j if, and only if, T_i and T_j belongs to the same partition.

Proof: Local FPTrees are built based on a global ordering, common to all computing nodes. Any transactions $T_i, T_j \mid T_i = T_j$ would, thus, have items in the same order, regardless of its source tree. Therefore, any prefix P, with maximum length μ, and whose subtree contains T_i, would have an equal prefix P', whose subtree contains T_j. Since transactions are assigned to global partitions via a hash function over the prefix, and $P = P'$, T_i and T_j will be assigned to the same partition of the global FPTree. ■

Using our sample data base, we show the resulting global partitions in Figure 4. Such partions are projected on up to ρ-itemsets, and associated CFPTrees, or ρ-CFPTrees, are mapped to *ItemsetExtractors*, using the ρ-itemset as key. The process is analogous to the extraction of μ-FPTrees, having prefixes as ρ-itemsets, and subtrees, associated ρ-CFPTrees.

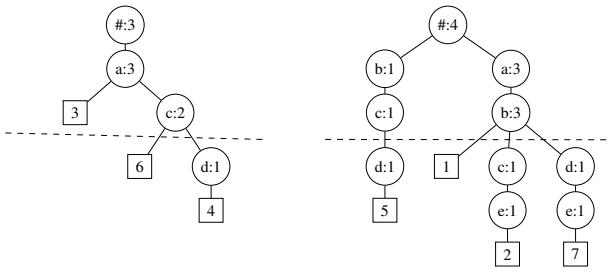

Figure 4: Global partitioned FPTree, resultant of merging pairs $\langle prefix, sub\text{-}tree \rangle$, namely μ-FPTrees. Each FPTree partition is placed on a different computing node.

C. Large-Scale Distributed Itemset Mining

Upon arrival at *ItemsetExtractors*, ρ-CFPTrees and related itemsets are merged into global CFPTrees. The global structure contains all occurrences of $\{1, 2, ..., \rho\}$-itemsets X in the data set, and can thus be mined for frequent itemsets $F_i \supseteq X$, based on minsupp. The global FPTree partitions,

built at step 4, are projected up to ρ-itemsets: if $\rho = 1$, we have one 1-itemset and 1-CFPTree for each frequent item; in case $\rho = 2$, we have $\{1,2\}$-itemsets and related $\{1,2\}$-CFPTrees. In general, $\binom{n'}{\rho}$ itemsets and CFPTrees may be created for a data set with n' frequent items. Such initial projections occur without pruning, as we only know the global frequencies of 1-itemsets at the time.

Conditional FPTrees, once merged into global CFPTrees, can be mined for frequent itemsets in parallel. So, we now set up another trade-off between data replication and load balancing: low values of ρ yield fewer CFPTrees, and processors may be overloaded; on the other hand, higher ρ values generate more CFPTrees, but also replicates common subsets between conditional trees, requiring more data exchange. At the end, global CFPTrees are further projected, and resulting frequent itemsets are recorded.

IV. Implementation Details

A. Watershed Framework

The application was written in C++, using the distributed computing framework Watershed [10] [2]. Watershed is a distributed filter-stream processing system for massive data sets, inspired by the data-flow model. Watershed defines data processing components as filters, and the communication channel as a data stream. The version of the framework used in this work uses OpenMPI as a low-level communication interface.

B. Performance Tuning

The in-memory representation of the FPTree is very compact: we use three pointers and two integers to represent each node (32 bytes per node). We also optimize memory allocation by means of user-level memory management, which reduces the number of system calls. The implementation, as many other FPGrowth solutions, maps item labels to linked lists of nodes, in order to efficiently project FPTrees. Regarding communication aspects, messages are aggregated into 8KB (8192 bytes) packets, which reduces the amount of context switches during data exchange.

We employ a very efficient serialization of the FPTree, similar to Buehrer's [8] solution. The serialization reduces the communication costs of building a distributed FPTree, and of sending FPTree projections (CFPTrees). The encoding applied consists of representing each node as two integers and a selector byte: one integer for the label, another one for the frequency, and the selector byte states whether there was a change of path in the DFS representation of the FPTree. Whenever the traversal changes, we send another integer with the new current level of the DFS. Each encoded node requires 13 bytes on average.

[2] `http://www.speed.dcc.ufmg.br/watershed`

V. Experimental Evaluation

The cluster used in the following experiments is composed of 9 machines, each containing a quad-core Intel Xeon X3440 with Hyperthreading and 8 MB cache, 16 GB RAM, a 1 TB 7200 RPM SATA disk, running 64-bit Linux 3.2.0 The nodes are connected using Gigabit Ethernet. To evaluate the performance of Twig, we used two real world data sets: a spam data base, aggregating messages from over 10 different countries; and a set of Twitter posts, storing tweets crawled using the Twitter API. The main characteristics of these data sets are shown in Table I.

	SpamMining	Twitter
# Transactions	2.78bi (2777879353)	233mi (233259498)
# Distinct items	113mi (112706424)	64mi (63883303)
Avg. trans. length	27 (26.6208)	8 (7.7638)
Total size	279GB	7.9GB

Table I: We tested Twig on two real bases: a large set of spam messages (SpamMining), and a set of Twitter posts (Twitter).

In this section, we evaluate Twig in terms of scalability, speedup, load balancing, and communication costs. We also compare Twig with sequential and distributed implementations of FPGrowth.

A. Speedup

We first analyze Twig in terms of speedup, which shows how much the performance improves as a function of the number of machines used to process a given data set. To run the experiments, the data was partitioned equally among the processors for each execution. Figure 5 shows the speedup

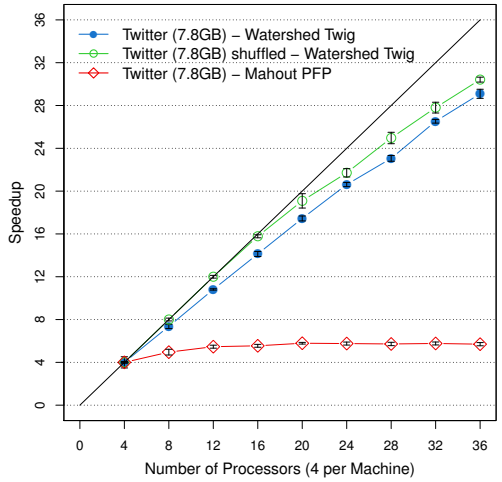

Figure 5: Speedup on Twitter base with $\text{minsupp} = 0.25\%$.

results for the original (blue), and shuffled (green) versions of the Twitter data set ($\text{minsupp} = 0.25\%$). To obtain the shuffled version, we shuffled the transactions on the

original data set before partitioning, which improves load balancing. Figure 6 shows how our algorithm behaves with the SpamMining data set (minsupp $= 1.00\%$). In both speedup experiments, we obtain near linear speedups, a strong evidence that Twig performs well on different data sets.

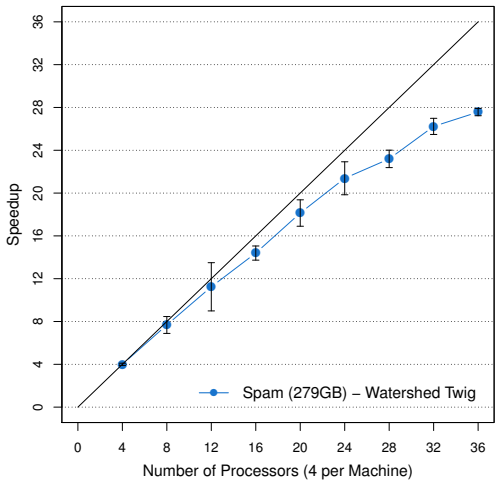

Figure 6: Speedup on SpamMining data set.

B. Load balancing based on prefix length

This test uses all 9 computers, over which the input is evenly distributed. The goal is to evaluate the effect of load balancing on execution times. Figure 7 shows the results, along with the number of independent parallel tasks for different maximum prefix lengths μ.

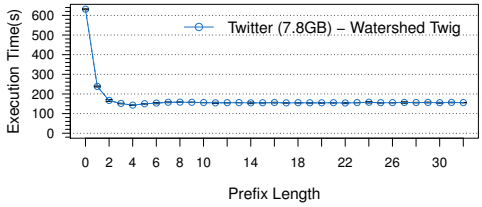

Figure 7: Execution times and number of independent parallel tasks for different maximum prefix lengths μ.

Initially, execution times drop steadily, since very short prefixes lead to large differences in terms of tree size at each machine, hurting load balancing among processors, which results in longer execution times. Further increases in the prefix length — between lengths 4 and 8 — slightly increase execution time, as there is a small growth in the communication overhead in that case. After a certain prefix length, the execution time does not vary significantly.

C. Communication costs

During the speedup experiments with both data sets, we observed how communication varied as machines were added to the computation. The behaviors of both the average exchange per host and the global message exchange are shown in Figure 8. The increase in communication is a natural drawback of parallelization: to maintain the correctness of algorithms, it is often necessary to exchange more information as the number of machines grows. The results show that, despite the expected behavior in communication (Figure 8, top), the average costs (Figure 8, bottom) continuously decrease with more computers. This turns out to reduce the global communication time, since costs of data exchange grows sub-linearly with the number of computing nodes, and less time will be spent sending/receiving data at each processor.

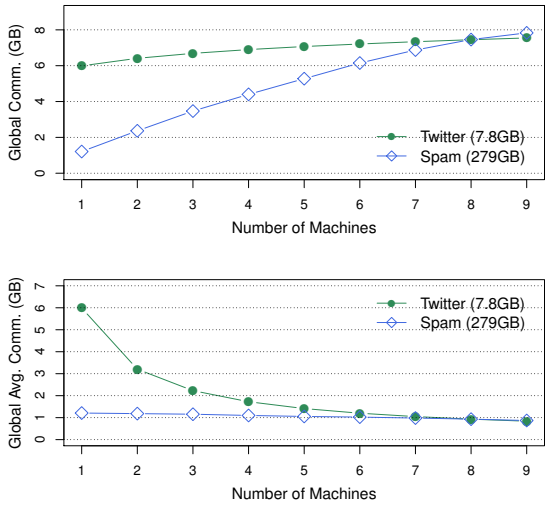

Figure 8: Variation of both global (top) and average communication (bottom) as new machines are added to the cluster.

D. Baseline comparisons

We now present performance comparisons between Twig and well known FIM solutions. First, we compare Twig with an efficient sequential implementation [4], analyzing the scalability of both solutions. Later, we compare the speedup capability and execution time of both Twig and the distributed implementation of the FPGrowth algorithm [5], named PFP.

1) Borgelt's Efficient Sequential Implementation: Borgelt [4] describes an efficient C implementation of the FPGrowth algorithm. In that implementation, projected FPTrees are pruned by removing items that have become infrequent during projection. This technique is known as FPBonsai [11] pruning, and may reduce the size of the FPtree considerably, leading to faster projections.

Borgelt provides a very efficient sequential implementation of FPGrowth, yet it fails to handle massive data sets, given that hardware limitations on a single machine are reached rather soon, both in terms of storage and memory

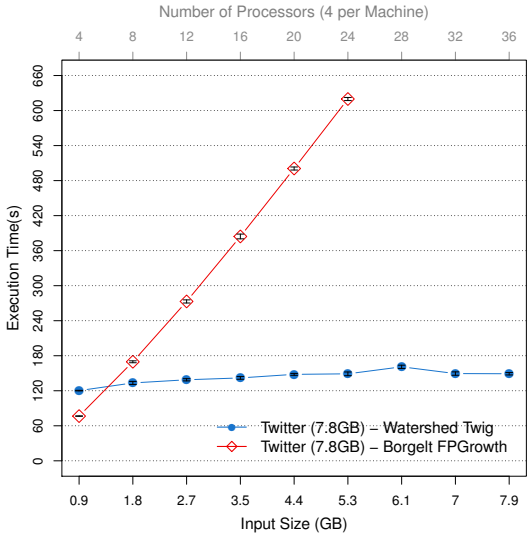

Figure 9: Scalability comparison between the Twig and Borgelt's FPGrowth implementations.

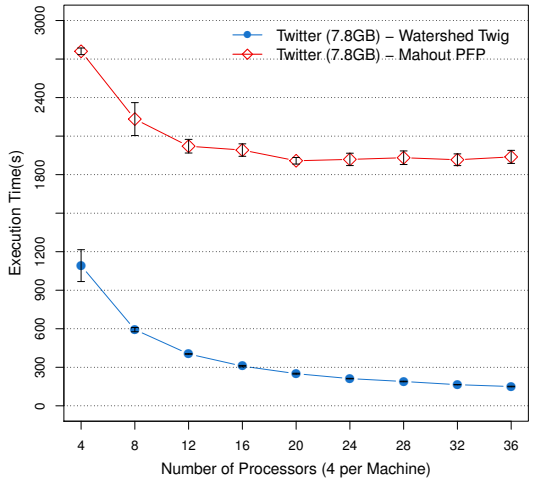

Figure 10: Execution time comparison between Twig and Apache Mahout's parallel FPGrowth implementations.

resources. During execution, Borgelt's FPGrowth exceeded the memory limitation of our machines (16GB). As depicted in Figure 9, it could properly execute up to a sample of the Twitter data set with just 5.3GB.

2) Comparison with Apache Mahout PFP: Apache Mahout[3] is a library of scalable data mining and machine-learning algorithms, implemented on top of Apache Hadoop, using the MapReduce [12] paradigm. Once we have the data set stored in the Hadoop Distributed File System (HDFS) [13], the execution of the parallel FPGrowth (PFP) [5], offered by the Mahout library, performs three MapReduce jobs for generating the top-k frequent patterns for each item. In HDFS, the data is split into 256MB blocks, with three replicas per block.

The first MapReduce job counts global items frequencies. The second job runs a parallel FPGrowth to calculate the top-k items of group dependent shards. Mappers output one or more key-value pairs, where each key is a group-id, and its corresponding value is a generated group-dependent transaction. The combiner takes each group of dependent transactions and builds an FPTree, used by the single reducer instance to run the FPGrowth algorithm with the FPBonsai [11] pruning. The top-k frequent patterns are produced for each group, and the third and final job aggregates different top-k patterns by items, thus calculating the final top-k frequent patterns for each item.

Mahout's parallel FPGrowth is able to handle data sets much larger than those that can be handled with sequential implementations. However, as Figure 5 shows, the speedups obtained by it are low. PFP has also poor performance in terms of execution time when compared to Twig, as Figure 10 shows. We present the ratio between PFP and Twig as we vary the number of processors on Figure 11. As it can be seen there, Twig can be up to 13 times faster than PFP, a result which is obtained with only nine machines (36 processors). Further, the fact that the ratio seems to increase linearly with the number of processors suggests that Twig scales better as we add machines.

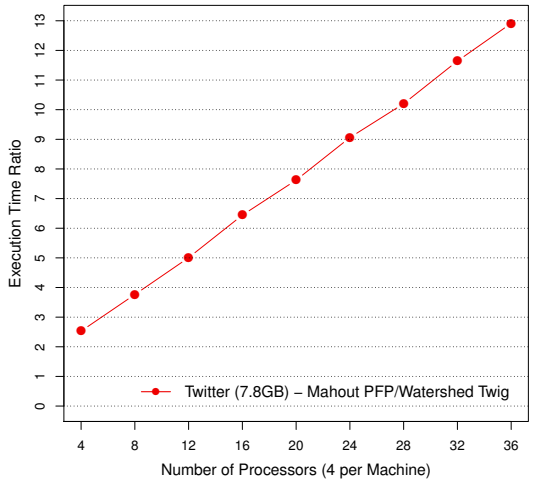

Figure 11: Execution time ratio between the Twig implementation and Apache Mahout parallel FPGrowth.

[3]http://mahout.apache.org

VI. Related Work

FPGrowth is a state-of-the-art solution to frequent itemset mining: it provides substantial data compression, and only yields frequent itemsets. However, mining massive data sets is still a challenging problem, requiring solutions to be both efficient and scalable. Many approaches towards parallelizing frequent itemset generation have been proposed, and FPGrowth is often used as a processing step for more complex tasks. Proposals range from distributed computing and architecture specific solutions [5], [8], [14], [15], to peripheric optimization and approximate algorithms [16]–[18].

Some of the approaches have focused on memory and multi-core optimizations. Cache-conscious in-memory FP-Tree representations include CCTree [19], providing fast bottom-up traversals; and FPArray [17], which effectively exploits data locality and benefits from hardware and software prefetching. Teodoro et al. [15] presented efficient parallelizations of the TreeProjection [20] algorithm on multi-core and GPU architectures. To overcome single-node memory restraints, out-of-core and compression alternatives have also been proposed: in the first, disk performance limits execution times of out-of-core solutions [21]; in the later, compressed data structures, such as Schlegel's CFPTree and CFPArray [16], reach computers memory limits rather soon, for very large data sets.

PFP [5] is the main MapReduce parallel FPM solution, made publicly available through the Apache Mahout library. Moens et al. [22] proposed two MapReduce solutions for mining large data sets: Dist-Eclat, a parallel Eclat focused on speed, and BigFPM, a hybrid algorithm, combining principles from Apriori and Eclat, and optimized for really massive data sets. PARMA [18], an approximate FPM algorithm, was also developed on top of MapReduce. Balanced Parallel FPGrowth (BPFP) [23] improves PFP by incorporating a load balancing strategy. In contrast to PFP, BPFP does aggregate the final itemsets, as it aims to discover all — not only the top-k — frequent itemsets. Among the distributed approaches, Buerher et al. [8] proposed a global partitioned representation of the FPTree, which was capable of processing a synthetic data set of 1TB over 48 machines, connected via Infiniband. However, the partitioning scheme works item-wise, and may require a computing node to keep the entire FPTree locally.

VII. Conclusion

In this paper we presented Twig, an adaptable and *data-conscious* partitioning scheme at the granularity of transactions. It permits a complete and balanced distribution of the data set as well as the FPtree that the algorithm FPGrowth builds, and associated projections, still with a low communication overhead. The algorithm is based upon a global, irregular data structure that can grow quite large. Therefore, FPGrowth, despite being fast, is limited by memory to handle large data sets. Parallelizing the algorithm, therefore, can mitigate the memory constraint by leveraging the global memory of a cluster of machines. Such parallelization, however, is hard to achieve as the computation is irregular and data-dependent.

Our proposed parallelization relies on carefully partitioning the main data structure, the FPTree, which is then used for extracting the frequent patterns in a distributed fashion. This approach offers an effective load balancing, which notably reduces overall execution time and memory requirements. Our approach yields high levels of parallelization of independent computations with minimal communication and data replication overhead.

To evaluate our parallelization, we experimented with two data sets derived from the real world: SpamMining, an aggregate of over 2 billion spam messages collected from ten different countries, and Twitter, a data set of Twitter messages. In terms of scalability, we achieved a 78% efficiency for 36 processors mining a 279GB spam dataset. We compared our approach to a well-known MapReduce-based parallel FPGrowth called Mahout PFP, and demonstrated that our solution is capable of managing massive data sets while presenting near linear speedups. In fact, with nine computers, we finish the frequent itemset extraction in only 7.26% of the time taken by Mahout PFP with the same number of nodes.

As future works, we intend to employ the same approach to other projection-based algorithms, in particular for mining frequent sequences and graphs. We also intend to investigate mechanisms such as message coalescing and others that may improve the performance of Twig. Finally, we intend to experiment with larger and more diverse data sets.

Acknowledgements

This work was partially supported by CNPq, CAPES, Fapemig, and InWeb – National Institute of Science and Technology for the Web.

References

[1] S. R. Upadhyaya, "Parallel approaches to machine learning-A comprehensive survey," *J. Parallel Distrib. Comput.*, vol. 73, no. 3, pp. 284–292, Mar. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2012.11.001

[2] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New algorithms for fast discovery of association rules," in *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, ser. KDD '97. AAAI Press, 1997, pp. 283–286.

[3] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, May 2000. [Online]. Available: http://doi.acm.org/10.1145/335191.335372

[4] C. Borgelt, "An implementation of the fp-growth algorithm," in *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*. ACM, 2005, pp. 1–5.

[5] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "Pfp: parallel fp-growth for query recommendation," in *Proceedings of the 2008 ACM conference on Recommender systems*, ser. RecSys '08. New York, NY, USA: ACM, 2008, pp. 107–114. [Online]. Available: http://doi.acm.org/10.1145/1454008.1454027

[6] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Rec.*, vol. 22, no. 2, pp. 207–216, Jun. 1993. [Online]. Available: http://doi.acm.org/10.1145/170036.170072

[7] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499. [Online]. Available: http://dl.acm.org/citation.cfm?id=645920.672836

[8] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz, "Toward terabyte pattern mining: an architecture-conscious solution," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '07. New York, NY, USA: ACM, 2007, pp. 2–12. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229432

[9] J. H. Reif, "Depth-first search is inherently sequential," *Information Processing Letters*, vol. 20, no. 5, pp. 229–234, 1985.

[10] T. L. A. de Souza Ramos, R. S. Oliveira, A. P. de Carvalho, R. A. C. Ferreira, and W. Meira, "Watershed: A high performance distributed stream processing system," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on*. IEEE, 2011, pp. 191–198.

[11] F. Bonchi and B. Goethals, "Fp-bonsai: the art of growing and pruning small fp-trees," in *Advances in Knowledge Discovery and Data Mining*. Springer, 2004, pp. 155–160.

[12] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[14] A. Veloso, W. Meira, Jr., R. Ferreira, D. G. Neto, and S. Parthasarathy, "Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining," in *Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, ser. PKDD '04. New York, NY, USA: Springer-Verlag New York, Inc., 2004, pp. 422–433. [Online]. Available: http://dl.acm.org/citation.cfm?id=1053072.1053111

[15] G. Teodoro, N. Mariano, W. Meira Jr., and R. Ferreira, "Tree projection-based frequent itemset mining on multicore cpus and gpus," in *Proceedings of the 2010 22nd International Symposium on Computer Architecture and High Performance Computing*, ser. SBAC-PAD '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 47–54. [Online]. Available: http://dx.doi.org/10.1109/SBAC-PAD.2010.15

[16] B. Schlegel, R. Gemulla, and W. Lehner, "Memory-efficient frequent-itemset mining," in *Proceedings of the 14th International Conference on Extending Database Technology*, ser. EDBT/ICDT '11. New York, NY, USA: ACM, 2011, pp. 461–472. [Online]. Available: http://doi.acm.org/10.1145/1951365.1951420

[17] L. Liu, E. Li, Y. Zhang, and Z. Tang, "Optimization of frequent itemset mining on multiple-core processor," in *Proceedings of the 33rd international conference on Very large data bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 1275–1285. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325997

[18] M. Riondato, J. A. DeBrabant, R. Fonseca, and E. Upfal, "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*, ser. CIKM '12. New York, NY, USA: ACM, 2012, pp. 85–94. [Online]. Available: http://doi.acm.org/10.1145/2396761.2396776

[19] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey, "Cache-conscious frequent pattern mining on a modern processor," in *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 577–588.

[20] R. C. Agarwal, C. C. Aggarwal, and V. Prasad, "A tree projection algorithm for generation of frequent item sets," *Journal of parallel and Distributed Computing*, vol. 61, no. 3, pp. 350–371, 2001.

[21] G. Buehrer, S. Parthasarathy, and A. Ghoting, "Out-of-core frequent pattern mining on a commodity pc," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 86–95.

[22] S. Moens, E. Aksehirli, and B. Goethals, "Frequent itemset mining for big data," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 111–118.

[23] L. Zhou, Z. Zhong, J. Chang, J. Li, J. Huang, and S. Feng, "Balanced parallel fp-growth with mapreduce," in *Information Computing and Telecommunications (YC-ICT), 2010 IEEE*