

UTRECHT UNIVERSITY  
FACULTY OF SCIENCE  
DEPARTMENT OF INFORMATION AND COMPUTING SCIENCES

MASTER THESIS

---

# Scalable Frequent Pattern Mining using Relational Databases

---

*Author:*  
N.H. Schuiling, B ICT

*Supervisors:*  
Drs. H. Philippi  
Prof. dr. A.P.J.M. Siebes

August 2014

ICA-3971589



**Universiteit Utrecht**

UTRECHT UNIVERSITY

## *Abstract*

Faculty of Science

Department of Information and Computing Sciences

### **Scalable Frequent Pattern Mining using Relational Databases**

by N.H. Schuiling, B ICT

Frequent pattern mining is a computationally expensive preprocessing step for association rule mining and involves major challenges when data sets are large. Many sophisticated algorithms have been proposed since its introduction, however most of them don't address the real difficulty: scalability. In this thesis we study how to leverage relational database management system (RDBMS) integration to improve the scalability of frequent pattern mining. We propose a hybrid framework based on FP-growth, a popular frequent pattern mining algorithm. For FP-tree construction we develop an SQL-based approach and for pattern mining we develop an external application that interfaces with the RDBMS. Subsequently, we adapt this framework to achieve high scalability, by sharding the transaction database and distributing computation over multiple computing instances. Experimental results with MonetDB5 show that this framework is capable of handling large data sets using a scale out strategy. It achieves near-linear speedup when computing instances are added. Finally, we develop an FP-growth based top- $k$  frequent pattern mining method and integrate this method in our framework. To mine the top- $k$  frequent patterns efficiently, we change the FP-growth mining algorithm to mine patterns in support descending order using a level-wise search strategy instead of the depth-first-search strategy of the original algorithm. Experimental results show that this method has surprisingly good performance on a large data set.

# Contents

<b>Abstract</b>	<b>i</b>
<b>Contents</b>	<b>ii</b>
<b>List of Figures</b>	<b>iv</b>
<b>List of Tables</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Goals and motivation . . . . .	2
1.3 Related work . . . . .	3
1.4 Contribution summary . . . . .	3
1.5 Organization . . . . .	4
<b>2 Frequent Pattern Mining</b>	<b>5</b>
2.1 Overview . . . . .	5
2.1.1 Definition . . . . .	6
2.1.2 Complexity analysis . . . . .	6
2.1.3 Algorithms . . . . .	7
2.1.3.1 Apriori . . . . .	7
2.1.3.2 FP-growth . . . . .	8
2.1.4 Drowning in Frequent Patterns . . . . .	9
2.1.5 Constraints . . . . .	9
2.1.6 Applications . . . . .	10
2.2 FP-growth in more detail . . . . .	10
2.2.1 Constructing an FP-tree . . . . .	10
2.2.2 Mining patterns using an FP-tree . . . . .	12
2.2.3 Why use FP-growth? . . . . .	13
2.2.4 Improvements and extensions . . . . .	14
<b>3 Integration of Frequent Pattern Mining in Relational Databases</b>	<b>16</b>
3.1 Input data representation . . . . .	16
3.2 Integration options . . . . .	18
3.3 SQL-based FP-tree construction . . . . .	19
3.3.1 Input data model . . . . .	20
3.3.2 Item counting . . . . .	20

3.3.3	Transaction ordering . . . . .	21
3.3.4	FP-tree construction . . . . .	22
3.4	Mining Frequent Patterns in an external application . . . . .	24
<b>4</b>	<b>Distributed Frequent Pattern Mining</b>	<b>28</b>
4.1	Solving storage issues . . . . .	28
4.2	Computation distribution & parallelization . . . . .	29
4.3	Distributed FP-growth . . . . .	30
4.4	Experimental setup . . . . .	35
4.4.1	Implementation . . . . .	35
4.4.2	Data sets . . . . .	36
4.4.3	Test environment . . . . .	36
4.4.4	Experiments . . . . .	37
4.5	Experimental results . . . . .	38
4.6	Conclusions . . . . .	38
<b>5</b>	<b>Top-<math>k</math> Frequent Pattern Mining</b>	<b>43</b>
5.1	Development of an FP-growth based Top- $k$ Frequent Pattern Mining method . . . . .	43
5.1.1	Generating patterns in support decreasing order . . . . .	44
5.1.2	Pruning the search space . . . . .	45
5.1.3	Full algorithm . . . . .	45
5.1.4	Example . . . . .	45
5.2	Integration in the hybrid FP-growth framework . . . . .	48
5.3	Adding a minimum length constraint . . . . .	48
5.4	Experimental setup . . . . .	50
5.5	Experimental results . . . . .	50
5.6	Conclusions . . . . .	50
<b>6</b>	<b>Future work</b>	<b>54</b>
6.1	Support of condensed pattern representations . . . . .	54
6.2	Distributed Top- $k$ Frequent Pattern Mining . . . . .	54
6.3	Adding a minimum length constraint to Top- $k$ Frequent Pattern Mining . . . . .	55
6.4	Minimize difference between initial and final minimum support threshold in Top- $k$ Frequent Pattern Mining . . . . .	55
6.5	Incremental Frequent Pattern Mining . . . . .	55
6.6	Fault handling in distributed Frequent Pattern Mining . . . . .	56
<b>7</b>	<b>Conclusions</b>	<b>57</b>
	<b>Bibliography</b>	<b>58</b>

# List of Figures

2.1	FP-tree with its associated header table for the transaction database of Table 2.2 [12]	11
2.2	$Tree_p$ : conditional FP-tree of item $p$	13
3.1	FP-tree corresponding with the data of Table 3.6	24
3.2	$Tree_b$ : conditional FP-tree of item $b$	26
3.3	$Tree_c$ : conditional FP-tree of item $c$	26
4.1	Overview of the distributed FP-growth framework	31
4.2	Local FP-trees of $T_1$ and $T_2$	33
4.3	Global FP-tree of $T$	34
4.4	$Tree_b$ : conditional FP-tree of item $b$	34
4.5	$Tree_c$ : conditional FP-tree of item $c$	35
4.6	Results of $pumsb$	39
4.7	Results of $webdocs$	40
4.8	Results of $D14$	41
5.1	FP-tree for $pumsb$ with $\xi = 48193$	47
5.2	Results of $pumsb$	51
5.3	Results of $webdocs$	52

# List of Tables

2.1	Example to illustrate the frequent pattern mining problem: a data set in which $\{beer, cheese\}$ is frequent with $min. support = 2$ .	5
2.2	Example transaction database with sorted frequent items	11
2.3	All conditional pattern bases and trees derived of the FP-tree of Figure 2.2	13
3.1	Example illustrating the single column and multi column layout	17
3.2	Example transaction table $T$	21
3.3	Table $TOTAL\_COUNTS$ : output of the item counting subtask without a minimum support threshold	21
3.4	Table $TOTAL\_COUNTS$ without infrequent items	22
3.5	Table $SORTED\_TRANSACTION$	23
3.6	Table $FP\_TREE$	24
3.7	Prefix paths of item $b$ fetched from $FP\_TREE$	26
3.8	Prefix paths of item $c$ fetched from $FP\_TREE$	26
3.9	All frequent patterns of $FP\_TREE$	27
4.1	$T$ divided in two shards: $T_1$ and $T_2$	32
4.2	Local item frequencies of $T_1$ and $T_2$	32
4.3	Table $TOTAL\_COUNTS$ : global item frequencies of tables $T_1$ and $T_2$ , without infrequent items	32
4.4	Table $SORTED\_TRANSACTION$ of $T_1$ and $T_2$	33
4.5	Table $FP\_TREE$ of $T_1$ and $T_2$	33
4.6	Global $FP\_TREE$ table	34
4.7	Local prefix paths of item $b$ fetched from $FP\_TREE$ of $T_1$ and $T_2$	34
4.8	Global prefix paths of item $b$	34
4.9	Local prefix paths of item $c$ fetched from $FP\_TREE$ of $T_1$ and $T_2$	35
4.10	Global prefix paths of item $c$	35
4.11	Characteristics of the data sets used in the experiments.	36
4.12	Number of frequent items and patterns mined on $pumsb$	38
4.13	Number of frequent items and patterns mined on $webdocs$	38
4.14	Number of frequent items and patterns mined on $D14$	38
5.1	Support descending ordered header table corresponding with the FP-tree of Figure 5.1	47
5.2	Top-6 frequent patterns of $pumsb$	48
5.3	Initial vs final minimum support thresholds of $pumsb$	50
5.4	Initial vs final minimum support thresholds of $webdocs$	50

# Chapter 1

## Introduction

### 1.1 Background

Unprecedented quantities of digital data are everywhere around us and created everyday as a consequence of the information revolution. One of the greatest challenges today for humans is to extract meaningful information from these data; the ability to process these large amounts of data lies far beyond our capabilities. Fortunately computers exist, and sophisticated algorithms can help us with this nontrivial task.

This thesis is about frequent pattern mining, a popular and well researched data mining method. Frequent patterns are combinations of items that appear frequently together in a set of transactions, e.g. originating from supermarket sales records. Frequent pattern mining is related to one of the important categories of data mining problems, namely that of associations between attributes. So-called association rules are employed in many application areas, including product cross-selling, recommendation systems, web traffic analysis and decision problems.

More generally, data mining<sup>1</sup> is concerned with the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data [1]. The overall goal of the data mining process is to extract information from a data set and transform it into an understandable structure for further use. Data mining is an interdisciplinary subfield of computer science, with involving methods at the intersection of artificial intelligence, machine learning, statistics, and database systems [2].

---

<sup>1</sup>Also known as Knowledge Discovery in Databases (KDD).

## 1.2 Goals and motivation

The computation of frequent patterns is a computationally expensive preprocessing step for association rule mining and involves major challenges when data sets are large. Since the introduction of frequent pattern mining in 1993 many sophisticated algorithms have been proposed, however most of them presume that data sets fit in main memory and don't address the real difficulty of frequent pattern mining: scalability [3].

Because frequent pattern mining is an offline system, which means that frequent patterns are not calculated in real-time, the exact speed of execution is not of notable importance. What is important is the ability to handle large data sets from real world applications, that often don't fit in main memory. Considering this, the main goal of our research is to improve scalability of frequent pattern mining.

Also, the integration of frequent pattern mining in relational databases has been a popular research topic for many years [4] [5]. Nevertheless, still most commercial data mining software systems and research prototypes are developed from scratch, use flat (input) files and sophisticated main memory data structures [3]. Often such algorithms are efficient, but lack scalability. Using relational databases, we may achieve scalability by benefiting from provided data abstraction mechanisms, such as buffer management systems, standardized query languages as SQL and powerful built in operators for accessing, filtering and manipulating data. These gives us the advantage of handling large data sets without having to worry about main memory limitations. Furthermore, integrating frequent pattern mining in relational databases might allow ad-hoc mining, because input data often resides in a relational table as part of data warehouse. As such, another goal of our research is to integrate frequent pattern mining in a relational database management system (RDBMS), mainly to achieve scalability without losing abstraction.

Another sub goal of our research is to ease frequent pattern mining by changing the task of mining all patterns with a support value higher than a user specified threshold  $\xi$  to mining only the  $k$  most frequent patterns, which is called top- $k$  frequent pattern mining. In practice it's difficult to determine an appropriate value for the parameter  $\xi$ : high values yield few interesting patterns, while low values might yield an explosion of possibly non interesting patterns. As well, this allows us to benefit from some interesting properties and find the most frequent patterns efficiently.



### 1.3 Related work

Shang et al. [6] partially implemented FP-growth in a RDBMS using SQL. Their work contains useful ideas, but unfortunately the authors don't provide any implementation details about their algorithms. Merely pseudo code is provided, which is not trivial (or even possible) to convert to generic SQL without developing custom extensions or user defined functions. Only SQL code for reordering the transaction table is provided, which is trivial. Another shortcoming of their work is the lack of scalability, e.g. computation distribution and parallelization are not covered in their work. Also, no experimental results on large data sets, such as webdocs (see Section 4.4), are provided. For all that, Shang et al. gave us a viable basis and inspiration for our work.

Li et al. proposed a scalable MapReduce based FP-growth algorithm called PFP in [7]. Their work particularly focuses on the application of supporting query recommendation for search engines. They successfully parallelize FP-growth to deal with large scale data sets and their results show near-linear speedup. The major difference with our work is the use of a MapReduce model and the absence of RDBMS integration, however their results are significant and show that the idea of distributing FP-growth over multiple computing instances is feasible.

Han et al. suggested to change the frequent pattern mining task to mining top- $k$  frequent closed itemsets of length no less than  $min\_l$  in [8]. In their work they present a hash based itemset closure verification scheme and several search space pruning techniques. Regrettably, their implementation depends on a main memory data structure and as a consequence lacks scalability. However, the idea of mining top- $k$  pattern patterns is bright and motivated us to apply this idea to our work.

### 1.4 Contribution summary

In summary, the contributions of this thesis are as follows:

1. We propose a hybrid framework based on FP-growth, that successfully integrates frequent pattern mining in a RDBMS. For FP-tree construction we develop an SQL-based approach and for pattern mining an external application that interfaces with the RDBMS.
2. We adapt this framework to achieve high scalability, by sharding the transaction database into several smaller databases and by distributing computation and parallelizing tasks over multiple computing instances. Experimental results show that

this framework is capable of handling large data sets using a scale out strategy. It achieves near-linear speedup when computing instances are added.

3. We propose an FP-growth based top- $k$  frequent pattern mining method. To be concise, we change the FP-growth mining algorithm to mine patterns in support descending order, such that the  $k$  most interesting patterns are generated efficiently. To do so, the algorithm employs a level-wise search strategy instead of the depth-first-search strategy of the original FP-growth algorithm. In addition, we show how to integrate this method in our hybrid FP-growth framework, that integrates with a RDBMS.

## 1.5 Organization

In Chapter 2 we first formally define frequent pattern mining, analyse its complexity and describe its most popular algorithms. After that, we comprehensively describe the algorithm we based our work on: FP-growth.

Next, in Chapter 3 we show how FP-growth can be integrated in a RDBMS and propose a hybrid framework for this purpose. We discuss several integration options and finally describe how each subtask of FP-growth can be implemented, illustrated by a running example.

The hybrid FP-growth approach is further extended and transformed into a distributed approach in Chapter 4. In this chapter we show how to address the main bottlenecks of FP-growth and we give and discuss the experimental results of our prototype.

In Chapter 5 we change the frequent pattern mining task to top- $k$  frequent pattern mining. We develop an FP-growth based top- $k$  frequent pattern mining method and integrate this method in our hybrid FP-growth framework. Also, we give and discuss the experimental results of our prototype.

Finally, in Chapter 6 we discuss interesting directions for further research and in Chapter 7 we conclude our study.

## Chapter 2

# Frequent Pattern Mining

In this chapter, we introduce frequent pattern mining by giving an overview of the problem, including a formal definition, the description of some practical applications and a survey of the most renowned and influential algorithms proposed for solving this problem. Finally, we study FP-growth in more detail.

### 2.1 Overview

Frequent pattern mining<sup>1</sup> and association rule mining were first introduced in 1993 by Agrawal et al. [4]. Informally speaking, association rules can be seen as if-then rules: e.g. if a person buys cheese, he or she also buys beer. A measure that is often associated with association rule mining is that of support: the fraction of customers for whom the rule holds, or rather the relative number of customers buying all items occurring in the rule (the so-called underlying pattern or itemset) [9]. Basically, the objective is to find those items in a data set that commonly co-occur, based on a certain minimum support value. Besides itemsets, it's also possible to mine more complex patterns, such as trees and graphs.

Transaction 1	beer	cheese	
Transaction 2	cheese	milk	
Transaction 3	beer	cheese	diapers

TABLE 2.1: Example to illustrate the frequent pattern mining problem: a data set in which  $\{beer, cheese\}$  is frequent with  $min. support = 2$ .

---

<sup>1</sup>Also referred to as frequent itemset mining.

### 2.1.1 Definition

The formal definition of frequent pattern mining and association rule mining in a relational setting is the following:

Let  $db$  be a transaction table with schema  $R = \{I_1, \dots, I_n\}$ , in which each  $I_i$  is a binary attribute. The attributes in  $db$  correspond to items and the rows in  $db$  correspond to transactions. A row has value 1 for an attribute (item) if and only if the transaction contains that item.

For  $X, Y \subseteq R$ , with  $X \cap Y = \emptyset$ , let:

- $sup(X)$  denotes the *support* of  $X$ , i.e. the number of rows that have value 1 for all items in  $X$ . Alternatively, support is sometimes expressed as a fraction of the total number of rows, which is called relative support.
- for an *association rule*  $X \rightarrow Y$ , define:
  - the support is  $sup(X \cup Y)$
  - the confidence is  $sup(X \cup Y)/sup(X)$

The main objective is to find all association rules, with respect to a minimum confidence threshold  $\lambda$  and a minimum support threshold  $\xi$ . This problem can be decomposed in two subproblems:

1. Find all frequent patterns  $Z$  in a transaction database  $db$ .  $X$  is called frequent if  $sup(X) \geq \xi$ . This is the frequent pattern mining problem.
2. Use  $Z$  to generate association rules having confidence higher than  $\lambda$ : test all non-empty subsets  $X$  of  $Z$ , whether the rule  $X \rightarrow Z \setminus X$  holds with sufficient confidence.

In this thesis we focus on the frequent pattern mining problem. Research [10] have shown that this problem is significantly computationally more expensive than generating association rules. In the next section we'll further analyse the complexity of this problem.

### 2.1.2 Complexity analysis

A naive way to find all frequent patterns would be to check all subsets of  $R$ . Obviously, this is not feasible when  $|R|$  is large, because this results in an algorithm with a worst case running time of  $O(2^{|R|})$ . For example, suppose that we can check 1024 sets/sec, then:

- For 10 items, we need to check  $2^{10} = 1.024$  subsets and the algorithm is done in 1 sec;
- For 20 items, we need to check  $2^{20} = 1.048.576$  subsets and the algorithm is done in approximately 17 minutes;
- For 100 items, we need to check  $2^{100} = 1.267.650.600.228.229.401.496.703.205.376$  subsets and the algorithm needs roughly  $4 * 10^{18}$  years, which far exceeds the age of the universe (!).

Fortunately, more clever and efficient algorithms exist, but this example does show that the problem's search space is exponential in size.

In addition, Purdom et al. prove the NP-hardness of the frequent pattern problem in [11]. Given a fixed  $l$ , determining if there exists a set of  $l$  items that co-occur in a data set  $k$  times is NP-complete, because the Balanced Complete Bipartite Subgraph Problem reduces to it and the problem itself is in NP. Finding all frequent patterns is a similar problem, but in this case we need to find all frequent itemsets, irrespective of size. Clearly this problem is at least as difficult, hence we conclude that the frequent pattern problem is NP-hard. Consequently, frequent pattern mining algorithms need to cleverly address the problem's NP-hardness, for example using heuristics or effective search space pruning methods.

### 2.1.3 Algorithms

Over time many algorithms for finding all frequent patterns were presented in the literature. In this section we briefly discuss two algorithms that stand out in terms of influence and positive experimental results: Apriori [10] and FP-growth [12].

#### 2.1.3.1 Apriori

In 1993 Agrawal et al. introduced the frequent pattern mining and association rule mining problem in [4]. Shortly after that, in 1994, they introduced the Apriori algorithm [10], that efficiently solves the problem using the anti-monotonic Apriori property:  $Z$  can only be frequent if all of its (non empty) subsets are frequent. In other words: every subset of a frequent pattern must be frequent. This property still forms the basis of many frequent pattern mining algorithms and effectively prunes the search space.

The Apriori property allows the algorithm to perform a level-wise search for frequent patterns, in which the level is the number of items in a pattern. To be concise, Apriori

performs a breadth-first-search by iteratively obtaining candidate itemsets of size  $k + 1$  from frequent itemsets of size  $k$ , and checks their corresponding occurrence frequencies in the database. See Algorithm 1 for a full description of the algorithm. Denote by  $C(i)$  the sets of  $i$  items that are potentially frequent (the candidate sets) and by  $F(i)$  the frequent sets of  $i$  items.

---

**Algorithm 1** Apriori [9]
 

---

**Input:** Transaction database schema  $R$ , minimum support threshold  $\xi$ .

**Output:** The complete set of frequent patterns.

```

1:  $C(1) := R$ 
2:  $i := 1$ 
3: while  $C(i) \neq \emptyset$  do
4:    $F(i) := \emptyset$ 
5:   for each  $X \in C(i)$  do
6:     if  $\text{sup}(X) \geq \xi$  then
7:        $F(i) := F(i) \cup \{X\}$ 
8:     end if
9:   end for
10:   $i := i + 1$ 
11:   $C(i) := \emptyset$ 
12:  for each  $X \in F(i - 1)$  do
13:    for each  $Y \in F(i - 1)$  that shares the first  $i - 2$  items with  $X$  do
14:      if all  $Z \subset X \cup Y$  of  $i - 1$  items are frequent then
15:         $C(i) := C(i) \cup \{X \cup Y\}$ 
16:      end if
17:    end for
18:  end for
19: end while

```

---

### 2.1.3.2 FP-growth

In 2000 Han et al. introduced FP-growth [12]. FP-growth is a more complex, but more efficient algorithm for finding frequent patterns compared to Apriori. The algorithm uses a frequent pattern tree (FP-tree) data structure for storing compressed frequent patterns. This is a prefix tree with links between the branches that link nodes with the same item and a header table for the resulting itemsets.

In (only) two passes over the transaction database an FP-tree and header table are constructed, containing only frequent items. Itemsets are inserted in the FP-tree in support descending order, so the FP-tree provides high compression close to the root of the tree, thus particularly for items with high support.

Subsequently, the FP-tree is recursively processed in a depth-first-search divide-and-conquer scheme and patterns are grown directly, instead of generating candidate items and testing them against the entire database as in Apriori.

The framework proposed in this thesis is based on FP-growth, as such in Section 2.1.6 a more comprehensive description of FP-growth is provided.

#### 2.1.4 Drowning in Frequent Patterns

A common issue with frequent pattern mining is that one often finds too many patterns and, unfortunately, not all patterns are of interestingness. The number of frequent patterns is inversely related to minimum support threshold  $\xi$ . In practice it's difficult to determine an appropriate value for this parameter: high values yield few interesting patterns, while low values might yield an explosion of possibly non interesting patterns.

One way to address this issue is to formulate the problem as a top- $k$  problem, in which only the  $k$  most frequent patterns are mined. Often, this parameter  $k$  is much easier to define than  $\xi$ . In Chapter 5 we'll further discuss this approach.

Another way to cope with the flood of patterns is trying to find less frequent patterns using condensed pattern representations. In this case extra constraints are put on the frequent patterns. Two important concepts in this field are the maximal frequent patterns and the closed frequent patterns [9]:

- A pattern  $l$  is maximal frequent if no proper superset of  $l$  is frequent.
- A pattern  $l$  is closed if no proper superset of  $l$  is frequent and has the same support as  $l$ .

The underlying idea of both concepts is that the set of all maximal/closed frequent patterns represent all frequent patterns, but is far smaller than the set of all frequent patterns [9]. For simplicity, these concepts are not used in this thesis.

#### 2.1.5 Constraints

A third approach to tame the pattern explosion is constraint-based mining, in which only the patterns that satisfy user-specified constraints are mined. Constraints can be categorized into several categories according to their interaction with the mining process. For example, succinct constraints can be pushed into the initial data selection process at the start of mining, anti-monotonic can be pushed deep to restrain pattern growth during mining, and monotonic constraints can be checked, and once satisfied, not to do more constraint checking at their further pattern growth [3].

### 2.1.6 Applications

Possible applications of mined frequent patterns include [3]:

- Improvement of product arrangement in shelves, catalog, on a web page, ...
- Product recommendation: support of cross-selling
- Search query recommendation
- Web traffic analysis
- Input for other data mining methods (e.g. classification [13])
- Indexing and similarity search of complex structured data
- Software bug detection and analysis

## 2.2 FP-growth in more detail

The FP-growth algorithm consists of two steps [12]:

1. Constructing an FP-tree: construct a compact data structure called FP-tree that efficiently stores frequent patterns of a transaction database and enables efficient frequent pattern mining.
2. Mining patterns using an FP-tree: use an FP-tree to recursively mine all frequent patterns.

### 2.2.1 Constructing an FP-tree

An FP-tree is a prefix-tree data structure that stores frequent patterns of a transaction database, where the support (count) of each node is greater or equal to a minimum support threshold  $\xi$ . The data structure is defined as follows [12]:

1. It consists of one root node labeled as "*null*", a set of item prefix subtrees as the children of the root, and a frequent item header table.



TID	Items bought	Sorted frequent items
1	$f, a, c, d, g, i, m, p$	$f, c, a, m, p$
2	$a, b, c, f, l, m, o$	$f, c, a, b, m$
3	$b, f, h, j, o$	$f, b$
4	$b, c, k, s, p$	$c, b, p$
5	$a, f, c, e, l, p, m, n$	$f, c, a, m, p$

TABLE 2.2: Example transaction database with sorted frequent items

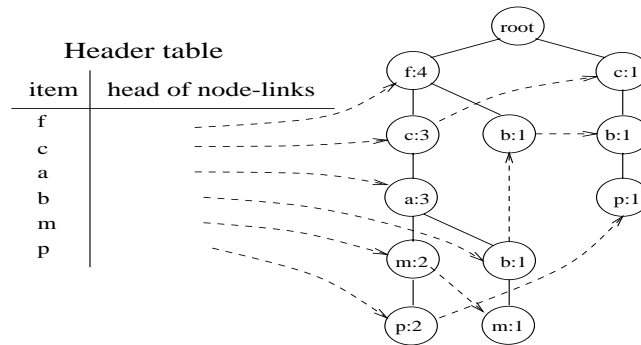


FIGURE 2.1: FP-tree with its associated header table for the transaction database of Table 2.2 [12]

- Each node in the item prefix subtree consists of three attributes: *item-name*, *count* and *node-link*, where *item-name* registers which items this node represents, *count* registers the number of transactions represented by the portion of the path reaching this node, and *node-link* links to the next node in the FP-tree carrying the same *item-name*, or null if there is none.
- The frequent items in each path are stored in support descending order.
- Each entry in the frequent item header table consists of two attributes, (1) *item-name* and (2) *head of node-link*, which points to the first node in the FP-tree carrying the *item-name*.

The construction of an FP-tree requires two scans of the transaction database. The first scan counts the support of each item and filters infrequent items. In the second scan transaction itemsets are sorted in support descending order and inserted in the FP-tree. Refer for a full description to Algorithm 2.

**Running example:** let transaction database  $db$  be the first two columns of Table 2.2 and minimum support threshold  $\xi = 3$ . Figure 2.1 illustrates the FP-tree corresponding with this example.

---

**Algorithm 2** FP-tree construction [12]

---

**Input:** Transaction database  $db$  and minimum support threshold  $\xi$ .**Output:** An FP-tree containing the frequent patterns of  $db$ .

- 1: Scan the transaction database  $db$  once and collect the set of frequent items  $F$  and their supports.
  - 2: Sort  $F$  in support descending order as  $L$ , the list of frequent items.
  - 3: Create the root of an FP-tree,  $t$ , and label it as "null".
  - 4: **for** each transaction  $Trans$  in  $db$  **do**
  - 5:     Select and sort the frequent items in  $Trans$  according to the order of  $L$ .
  - 6:     Let the sorted frequent item list in  $Trans$  be  $[p|P]$ , where  $p$  is the first element and  $P$  is the remaining list.
  - 7:     Call  $insert\_tree([p|P], t)$
  - 8: **end for**
- 

---

**Algorithm 3** Insert\_tree [12]

---

**Input:** The item list to be inserted  $[p|P]$  and an FP-tree node  $t$ 

- 1: **if**  $T$  has a child  $n$  such that  $n.item-name = p.item-name$  **then**
  - 2:     Increment  $n$ 's count with 1
  - 3: **else**
  - 4:     Create a new node  $n$  and let its count be 1, its parent link be linked to  $t$ , and its node-link be linked to the nodes with the same item-name via the node-link structure.
  - 5: **end if**
  - 6: **if**  $P \neq \emptyset$  **then**
  - 7:     Call  $insert\_tree(P, N)$
  - 8: **end if**
- 

### 2.2.2 Mining patterns using an FP-tree

Given an FP-tree, frequent patterns can be extracted using a divide-and-conquer algorithm. This task starts from a frequent length-1 pattern (as an initial suffix pattern), examines only its conditional pattern base (a "subdatabase" that consists of the set of frequent items co-occurring with the suffix pattern), construct its conditional FP-tree and performs mining recursively on such a tree. The pattern growth is achieved via concatenation of the suffix pattern with the new ones generated from a conditional FP-tree. Since the frequent itemset in any transaction is always encoded in the corresponding path of the frequent pattern trees, pattern growth ensures the completeness of the result [12].

Refer for a full description to Algorithm 4. Initially this algorithm is called as  $Mine(FP-tree, \emptyset)$ .

**Running example:** in this example we examine a part of the mining process of the previously constructed FP-tree shown in Figure 2.1. We start the mining process with the least frequent item:  $p$ . For this item we derive pattern  $(p : 3)$  and two prefix paths in

**Algorithm 4** Mining patterns using FP-tree**Input:** An FP-tree  $Tree$  and a set of prefix items  $\alpha$ .**Output:** The complete set of frequent patterns.

```

1: if  $Tree$  contains a single path  $P$  then
2:   for each combination  $\beta$  of the nodes on the path  $P$  do
3:     Generate pattern  $\beta \cup \alpha$  with  $support := \text{minimum support of the nodes in } \beta$ 
4:   end for
5: else
6:   for each item  $a_i$  in the header table of  $Tree$  (in support ascending order) do
7:     Generate pattern  $\beta := a_i \cup \alpha$  with  $support := a_i.support$ 
8:     Construct conditional pattern base of  $\beta$  and then  $\beta$ 's conditional FP-tree
       $Tree_\beta$ 
9:     if  $Tree_\beta \neq \emptyset$  then
10:      Call Mine( $Tree_\beta, \beta$ )
11:    end if
12:   end for
13: end if

```

FIGURE 2.2:  $Tree_p$ : conditional FP-tree of item  $p$ 

item	conditional pattern base	conditional FP-tree
$p$	$\{(f : 2, c : 2, a : 2, m : 2), (c : 1, b : 1)\}$	$\{(c : 3)\}   p$
$m$	$\{(f : 2, c : 2, a : 2), (f : 1, c : 1, a : 1, b : 1)\}$	$\{(f : 3, c : 3, a : 3)\}   m$
$b$	$\{(f : 1, c : 1, a : 1), (f : 1), (c : 1)\}$	$\emptyset$
$a$	$\{(f : 3, c : 3)\}$	$\{(f : 3, c : 3)\}   a$
$c$	$\{(f : 3)\}$	$\{(f : 3)\}   c$
$f$	$\emptyset$	$\emptyset$

TABLE 2.3: All conditional pattern bases and trees derived of the FP-tree of Figure 2.2

the FP-tree:  $(f : 4, c : 3, a : 3, m : 2, p : 2)$  and  $(c : 1, b : 1, p : 1)$ . These two prefix paths forms  $p$ 's conditional pattern base:  $(f : 2, c : 2, a : 2, m : 2)$  and  $(c : 1, b : 1)$ , which is used for constructing  $p$ 's conditional FP-tree. This leads to a tree with only one branch  $(c : 3)$  (see Figure 2.2), since  $\xi = 3$ . Thus, only frequent pattern  $(cp : 3)$  is derived and the mining of patterns associated with  $p$  stops. The algorithm continues this way until all frequent patterns are mined. See Table 2.3 for an overview of all conditional pattern bases and trees generated during the mining process.

### 2.2.3 Why use FP-growth?

The work of this thesis is based on the FP-growth algorithm for mainly three reasons:

1. FP-growth addresses the costly candidate set generation test bottleneck of Apriori-like approaches and is known to perform magnitudes faster [12]. This is important, since scalability is our main goal.
2. One of the advantages of FP-growth over other approaches is that it constructs an FP-tree data structure that compresses the transaction database<sup>2</sup>. An FP-tree is constructed using only two database scans and expensive database scans are avoided in the mining process. As such, FP-growth is suitable for RDBMS integration which is one of our goals.
3. FP-growth-like algorithms are among the most popular and best performing algorithms in the field [3].

### 2.2.4 Improvements and extensions

In the literature many improvements and extensions have been proposed for the original FP-growth algorithm [12] since its introduction. In this section we provide a concise overview of the most interesting improvements.

In 2001, Pei et al. presented a new frequent pattern mining method called H-mine in [14]. This method is based on FP-growth, uses a novel hyper-linked data structure and its mine algorithm doesn't require the creation of any physical projected databases nor conditional FP-trees.

In 2002 and 2003, Liu et al. explored top-down and bottom-up traversal of FP-tree based trees in pattern-growth mining in [15] and [16].

In 2003 and 2004 many improvements and extensions have been published as a result of the Frequent Itemset Mining Implementations workshops (FIMI), held at ICDM'03 and ICDM'04. A very interesting publication is that of Grahne and Zhu in 2003 [17]. They proposed an efficient array based FP-growth technique that reduces the need to traverse FP-trees. Also, Rácz proposed Nonordfp; an algorithm that uses a more compact and memory efficient representation of an FP-tree [18]. This algorithm was principally motivated by the running time and the space required for the FP-growth algorithm. Liu et al. presented AFOPT in [19]. This algorithm aims at improving the FP-growth performance in four perspectives: item search order, conditional database representation, conditional database construction strategy and tree traversal strategy. AFOPT is used as a benchmark in this thesis, together with LCM [20] that was proposed by Uno et al. in 2004.

---

<sup>2</sup>The compression ratio depends on the data set's characteristics.

In 2006, Buehrer et al. presented I/O-conscious optimizations for FP-growth to leverage 64-bit processors in [21]. Their empirical study shows great performance improvements.

## Chapter 3

# Integration of Frequent Pattern Mining in Relational Databases

To achieve scalable frequent pattern mining without losing abstraction, one of our ideas is to integrate FP-growth in a RDBMS. Most previous work adopt Apriori based approaches, but those methods generally suffer from costly candidate generation and testing, especially when mining data sets with long patterns. Though, FP-growth uses a relatively complex frequent pattern tree, which makes this task not trivial (e.g. a straightforward conversion to SQL is not possible).

In this chapter we first give an overview of ways to represent input data in a RDBMS and the architectural options we have to integrate FP-growth in a RDBMS. Finally, we propose a novel hybrid framework for FP-growth that integrates in a RDBMS.

### 3.1 Input data representation

A RDBMS consists of a collection of tables with attributes, all of which is formally described and organized according to the relational model. To mine a transaction database in a RDBMS, first we need to make the transaction data available in a relational table. Commonly, transaction data is stored inside the RDBMS, or if the data is stored elsewhere, it's to be accessible to the RDBMS, e.g. by using a wrapper that provides a relational view on the data.

There are basically two layouts to store transaction data in a relational table (see Table 3.1):

$tid$	$item$
1	beer
1	diapers
2	beer
2	chips
2	diapers
3	chips

(A) Single column layout

$tid$	$item_1$	$item_2$	$item_3$
1	beer	diapers	
2	beer	chips	diapers
3	chips		

(B) Multi column layout

TABLE 3.1: Example illustrating the single column and multi column layout

- The *single column layout* is a layout in which transaction data is transformed into a table  $T$  with the scheme  $(tid, item)$ . When a transaction consists of multiple items, a tuple is created for each item in the transaction.
- The *multi column layout* [22] is a layout in which transaction data is transformed into a table  $T$  with the scheme  $(tid, item_1, item_2, \dots, item_n)$ . In this model  $item_i$  contains the  $i$ th item of a transaction. If  $n$  is the maximum number of items in a single transaction, we have  $n + 1$  attributes in total in this layout.

Previous research shows a relation between data layouts and performance [23], but this depends on the RDBMS used and the chosen implementation level.

In this thesis we opt for the single column layout, because the multi column layout has some major drawbacks:

1. Since the total number of items in a transaction database is typically not known in advance, you first need to analyse the database and create the table afterwards.
2. Dynamic SQL queries are required to cope with the dynamic table scheme.
3. In sparse transaction databases a lot of space is wasted, especially in databases that organize tuples sequentially on disk pages using an N-ary Storage Model (NSM). For example, when the maximum number of items in a transaction database is 1000 and the average number of items in this database is 10, on average you waste 990 columns per tuple. Moreover, this could negatively affect performance.
4. The number of items in a transaction database might exceed the maximum number of attributes supported by a RDBMS. For example, the hard limit for the maximum number of columns per table is 4096 in MySQL, but the effective maximum may even be less for a given table depending on the storage engine used.

## 3.2 Integration options

Integrating the FP-growth algorithm in a RDBMS can be addressed in different ways. In this section we discuss some of the options available, in the order from tight to loose integration. Note that often a combination of integration options is used in practice.

- Query engine level: the query engine of a RDBMS can be extended with custom operators, which can be exposed through SQL. Doing so, you can integrate an algorithm as a first class citizen in the RDBMS, but this task is not trivial and you need to have access to the RDBMS kernel source code.
- User-defined function (UDF): a UDF is a mechanism for extending the functionality of the database server by adding a function that can be evaluated in SQL statements [24]. Often, UDF's can be written in procedural SQL (PSQL)<sup>1</sup> and a variety of high-level programming languages, such as C or C++. In case of the latter, most of the processing happens in the UDF and the RDBMS only provides input data to the UDF. Doing so, it's possible to achieve high performance, but this is at the expense of abstraction; RDBMS query processing is only used for passing data to the UDF.
- Stored-procedure: stored-procedures are similar to UDF's, but the major difference is that UDF's can be used like any other expression within SQL statements, whereas stored procedures must be invoked using the `CALL` statement [25].
- SQL-based: in this approach the algorithm is formulated as one or more SQL queries, which are executed by the RDBMS. You benefit optimally from the abstraction provided by the RDBMS.
- External application: the algorithm is implemented in an external application that reads/queries data from the RDBMS by using a provided client interface, such as Open DataBase Connectivity (ODBC). Possibly, data is cached temporarily in memory or on a local disk to improve performance. When the algorithm is finished, results can be output to the RDBMS. It depends on the executed SQL queries how much you benefit from abstraction provided by the RDBMS.

Our main goal is to achieve high performance and scalability without losing abstraction. Unfortunately, there often is a trade off between performance and data abstraction. Can we do better?

---

<sup>1</sup>Procedural elements have been added to the SQL language in the SQL:1999 and SQL:2003 standards in the part SQL/PSM. This made SQL an imperative programming language [25].



In this thesis we choose for a hybrid approach. For FP-tree construction we use an SQL-based approach. This task consists of the subtasks item counting, transaction ordering and FP-tree construction. For the mining task we use an external application, that interfaces with the RDBMS (e.g. using ODBC). In the next sections we will show in detail how these tasks are implemented. Furthermore, in Chapter 4 we show that this approach can be adapted to a distributed variant that allows high scalability.

Our approach benefits from data abstraction provided by the RDBMS, which would not be the case if we would opt for a query level engine implementation or a UDF/stored procedure written in a high level programming language. However, a UDF/stored procedure written in solely PSQL could also give us the desired data abstraction.

We implemented FP-growth in PSQL and experimented with this approach in Firebird, a popular enterprise open source RDBMS. Unfortunately, experimental results show that competitive performance is not feasible. For example, on *T10I4D100K*, a synthetic data set with 100.000 transactions with an average length of 10 items, a native implementation has a running time of about 4 seconds versus 150 seconds of the PSQL approach. This huge difference is likely caused by the overhead of PSQL interpreting along with the high number of recursive calls in the algorithm. Besides the poor performance, it would be impossible or very difficult to adapt the PSQL implementation to a distributed variant, because communication between RDBMS instances is necessary in this scenario. However, some RDBMS's do support such features in their PSQL implementations.

### 3.3 SQL-based FP-tree construction

The tree construction phase of the FP-growth algorithm roughly consists of the following subtasks:

1. Item counting: scan the transaction database once and collect item occurrence frequencies (i.e. the support values)
2. Transaction ordering: filter infrequent items and order items of transactions in item support descending order
3. FP-tree construction: construct a compressed FP-tree that contains all transactions and their frequent items

In this section we show how we can implement these subtasks in SQL and prepare intermediate results for further mining in an external application (see next section).

### 3.3.1 Input data model

As discussed in Section 3.1 we use a single column layout for input data. In SQL this table is defined as follows:

```
CREATE TABLE T (TID INT, ITEM STRING);
```

### 3.3.2 Item counting

Given a transaction table  $T$ , it's trivial to count item frequencies using the following SQL query<sup>2</sup>:

```
SELECT ITEM, COUNT(*) AS SUPPORT FROM T GROUP BY ITEM ORDER BY 2 DESC;
```

To illustrate the benefits of data abstraction provided by a RDBMS: if we would have a large transaction table that doesn't fit in main memory, we would still be able to execute this query, thanks to sophisticated algorithms in the RDBMS, such as external sorting.

The results of this subtask are input for the next subtask, so we store the results in a temporary table<sup>3</sup> called *TOTAL\_COUNTS*, which is defined as follows:

```
CREATE GLOBAL TEMPORARY TABLE TOTAL_COUNTS (ITEM STRING, SUPPORT int)
ON COMMIT PRESERVE ROWS;
```

Also, the initial query is changed to:

```
INSERT INTO TOTAL_COUNTS
SELECT ITEM, COUNT(*) AS SUPPORT FROM T
GROUP BY ITEM ORDER BY 2 DESC;
```

If we let users specify a minimum support threshold, the query becomes:

```
INSERT INTO TOTAL_COUNTS
SELECT ITEM, COUNT(*) AS SUPPORT FROM T
GROUP BY ITEM HAVING COUNT(*) >= :MINSUPPORT ORDER BY 2 DESC;
```

---

<sup>2</sup>Note that items are sorted in support descending order.

<sup>3</sup>For performance reasons we use temporary tables for storing intermediate results; a temporary table allows a RDBMS to not save tuples on disk. Experiments confirm this assumption.

TID	ITEM
1	d
1	a
1	e
1	g
2	b
2	a
3	h
3	a
3	c
3	b
3	e
4	a
4	b
4	c
4	d
5	c
5	a
5	e
5	b

TABLE 3.2: Example transaction table  $T$ 

ITEM	SUPPORT
a	5
b	4
c	3
e	3
d	2
g	1
h	1

TABLE 3.3: Table  $TOTAL\_COUNTS$ : output of the item counting subtask without a minimum support threshold**Running example:**

Table 3.2 is input to the item counting task. The algorithm counts all items and fills table  $TOTAL\_COUNTS$  as shown in Table 3.3.

**3.3.3 Transaction ordering**

Given the output of the item counting subtask, we can filter infrequent items of  $T$  and order items of transactions in item support descending order while preserving the ascending TID order:

```
SELECT T1.TID, T1.ITEM FROM T AS T1,
```

ITEM	SUPPORT
a	5
b	4
c	3
e	3
d	2

TABLE 3.4: Table *TOTAL\_COUNTS* without infrequent items

```
(SELECT ITEM, SUPPORT FROM TOTAL_COUNTS) AS T2
WHERE T1.ITEM = T2.ITEM ORDER BY T1.TID, T2.SUPPORT DESC, T1.ITEM;
```

Again, we store the output in a temporary table, which is defined as follows:

```
CREATE GLOBAL TEMPORARY TABLE SORTED_TRANSACTION (TID int, ITEM STRING)
ON COMMIT PRESERVE ROWS;
```

And the initial query is changed to:

```
INSERT INTO SORTED_TRANSACTION
SELECT T1.TID, T1.ITEM FROM T AS T1,
(SELECT ITEM, SUPPORT FROM TOTAL_COUNTS) AS T2
WHERE T1.ITEM = T2.ITEM ORDER BY T1.TID, T2.SUPPORT DESC, T1.ITEM;
```

If we didn't filter infrequent items in the previous subtask and want to exclude infrequent items in this stage, we can still do this by removing them from the table *TOTAL\_COUNTS* before executing the transaction ordering query:

```
DELETE FROM TOTAL_COUNTS WHERE SUPPORT <= :MINSUPPORT;
```

### Running example:

Let minimum support threshold  $\xi = 2$ . Given  $\xi$ , items  $g$  and  $h$  are infrequent and removed from *TOTAL\_COUNTS*, as shown in Table 3.4. Finally, table 3.5 is the result of ordering table  $T$  according to the previously computed item frequencies.

### 3.3.4 FP-tree construction

The next subtask is to construct a compressed FP-tree that contains all transactions and their frequent items. In a traditional FP-tree each node is associated with an item

TID	ITEM
1	a
1	e
1	d
2	a
2	b
3	a
3	b
3	c
3	e
4	a
4	b
4	c
4	d
5	a
5	b
5	c
5	e

TABLE 3.5: Table *SORTED\_TRANSACTION*

and a support count. In our implementation each path corresponding with a transaction in an FP-tree is represented as a tuple in table *FP\_TREE* with attributes *PATH* and *COUNT*. This table is defined as follows:

```
CREATE TABLE FP_TREE (PATH string, COUNT int);
```

For a path  $P$  in an FP-tree we define  $FP\_TREE.PATH$  as the semicolon prefixed string concatenation of the items of each node of  $P$  separated with a semicolon. For example, path  $a \rightsquigarrow b \rightsquigarrow c \rightsquigarrow d$  becomes ";a;b;c;d;".

For a path  $P$  in an FP-tree we define  $FP\_TREE.COUNT$  as the count associated with the last node on  $P$ , which is  $P_n$ .

As such, we use the following SQL query to construct the FP-tree and fill table *FP\_TREE*:

```
INSERT INTO FP_TREE
SELECT PATH, COUNT(*) FROM
(SELECT ';' || CONCAT(ITEM, ';') AS PATH FROM SORTED_TRANSACTION GROUP BY TID)
AS GROUPED
GROUP BY PATH;
```

By grouping on *PATH* we compress the results and output as less tuples as possible.

PATH	COUNT
;a;e;d;	1
;a;b;	1
;a;b;c;e;	2
;a;b;c;d;	1

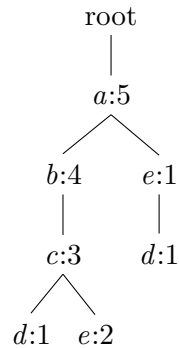
TABLE 3.6: Table *FP\_TREE*

FIGURE 3.1: FP-tree corresponding with the data of Table 3.6

**Running example:**

Given input table *SORTED\_TRANSACTION* (Table 3.4), table *FP\_TREE* is populated as shown in Table 3.6. For each path in the corresponding FP-tree a tuple exists in *FP\_TREE*. The corresponding FP-tree is shown in Figure 3.1.

### 3.4 Mining Frequent Patterns in an external application

The next task is to find the complete set of frequent patterns given table *FP\_TREE*, which is the final output of the SQL-based FP-tree construction task. Unlike the previous task, mining frequent patterns is implemented in an external application, i.e. outside the RDBMS. Basically, we can implemented any known algorithm for mining frequent patterns, provided that we have access to the RDBMS and convert the data of table *FP\_TREE* to an internal data structure that's suitable for the algorithm. We consider two ways of doing this:

1. Initially fetch all tuples of table *FP\_TREE* and build an FP-tree in main memory and thereafter mine all frequent patterns using the original FP-growth algorithm (see Algorithm 5)
2. For each item of table *TOTAL\_COUNTS*, use an optimized SQL query to fetch the conditional pattern base of this item and construct the corresponding conditional

FP-tree, and mine all frequent patterns of this conditional FP-tree (see Algorithm 6)

---

**Algorithm 5** Mine-FP1(*db*)
 

---

**Input:** Database *db* with table *FP\_TREE*.

**Output:** The complete set of frequent patterns.

- 1: Fetch all tuples from table *FP\_TREE* of *db* and construct FP-tree *Tree*
  - 2: Call Mine(*Tree*,  $\emptyset$ ) (Algorithm 4)
- 

---

**Algorithm 6** Mine-FP2(*db*)
 

---

**Input:** Database *db* with tables *TOTAL\_COUNTS* and *FP\_TREE*.

**Output:** The complete set of frequent patterns.

- 1: **for** each item  $a_i$  in table *TOTAL\_COUNTS* of *db* (in support ascending order) **do**
  - 2:     Generate pattern  $a_i$  with  $support = a_i.SUPPORT$
  - 3:     Fetch conditional pattern base of  $a_i$  from table *FP\_TREE* of *db* and construct  $a_i$ 's conditional FP-tree  $Tree_{a_i}$
  - 4:     **if**  $Tree_{a_i} \neq \emptyset$  **then**
  - 5:         Call Mine( $Tree_{a_i}$ ,  $a_i$ ) (Algorithm 4)
  - 6:     **end if**
  - 7: **end for**
- 

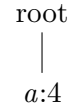
Option 1 has superior performance compared to option 2 as long as sufficient main memory is available, because all tuples from *DB* are transferred to the external application at once and no communication with the RDBMS is required after initialization. Also, *FP\_TREE* is scanned only once by the RDBMS. However, option 1 loads the complete FP-tree in main memory, which is unfeasible for large data sets. Since scalability is one of our main goals, we opt for option 2.

To construct a conditional FP-tree for an item  $a_i$ , we need to find the set of frequent items co-occurring with  $a_i$  in the base FP-tree, i.e. the set of  $a_i$ 's prefix paths. We find these paths using the following SQL query:

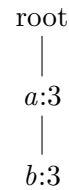
```
SELECT SUBSTRING(PATH, 0, POSITION(';':ITEM||';' in PATH)) AS P, SUM(COUNT)
FROM FP_TREE
WHERE PATH LIKE ';' || ITEM || ';' AND PATH NOT LIKE ';' || ITEM || ';'
GROUP BY P
```

In order to fetch only prefix paths of  $a_i$  from *FP\_TREE*, we select only paths that contain item  $a_i$  and don't start with item  $a_i$ . Also, to reduce the amount of data that is transferred from the RDBMS to a client, we select only the substring of *PATH*, from start till the position of string " $;a_i;$ ". This furthermore reduces the total number of tuples returned by the query, thanks to a likely more effective grouping.

PATH	COUNT
;a;	4

TABLE 3.7: Prefix paths of item  $b$  fetched from  $FP\_TREE$ FIGURE 3.2:  $Tree_b$ : conditional FP-tree of item  $b$ 

PATH	COUNT
;a;b;	3

TABLE 3.8: Prefix paths of item  $c$  fetched from  $FP\_TREE$ FIGURE 3.3:  $Tree_c$ : conditional FP-tree of item  $c$ **Running example:**

We iterate<sup>4</sup> all items of  $TOTAL\_COUNTS$ , starting with item  $a$  with support 5. We generate the first pattern:  $(a : 5)$ . To construct  $a$ 's conditional FP-tree  $Tree_a$ , we fetch all prefix paths from  $FP\_TREE$ .  $a$  is the most frequent item and the only item with support 5 in this example, so the query doesn't return any prefix paths and we continue to the next item. The next item of  $TOTAL\_COUNTS$  is  $b$  with support 4 and again we output this item immediately as a frequent pattern, i.e.  $(b : 4)$ . To construct  $b$ 's conditional FP-tree  $Tree_b$ , we fetch all prefix paths from  $FP\_TREE$  (see Table 3.7). This time one prefix path is returned, which we use to construct  $Tree_b$  (Figure 3.2). This tree contains one single path, thus the pattern  $(ab : 4)$  is generated. The next item is  $c$  with support 3 and we output pattern  $(c : 3)$ . We fetch all prefix paths from  $FP\_TREE$  (see Table 3.8 and Figure 3.3) and find one prefix path. Again, this tree contains one single path, thus the patterns  $(ac : 3)$ ,  $(bc : 3)$ ,  $(abc : 3)$  are generated (all combinations of the items on the path). We continue this way until all frequent patterns are found (see Table 3.9).

---

<sup>4</sup>In this example we iterate the header table in support descending order.



Pattern	Support
a	5
b	4
a b	4
c	3
a c	3
b c	3
a b c	3
e	3
a e	3
b e	2
c e	2
a e b	2
a e c	2
b e c	2
a b e c	2
d	2
a d	2

TABLE 3.9: All frequent patterns of *FP\_TREE*

## Chapter 4

# Distributed Frequent Pattern Mining

One of the major challenges in data mining is the capability to handle large databases. This makes scalability our main goal. FP-growth attempts to address this challenge by compressing transactions in an optimized FP-tree data structure, in the hope that it will fit entirely in main memory. However, data sets from real world applications won't fit in main memory and experimental results have shown that FP-growth has serious performance issues on such data sets and even fails to initialize (i.e. construct the FP-tree) [21].

In this chapter we show how we can adapt the hybrid FP-growth approach of Chapter 3 to achieve high scalability, by cleverly solving the main issues of FP-growth. After that, we show and discuss the results of an experimental prototype.

### 4.1 Solving storage issues

Trie<sup>1</sup> based frequent pattern mining algorithms, such as FP-growth, have the potential to perform bad on large databases. If a trie data structure doesn't fit in main memory, one option is to trash on disk, if the disk has enough capacity. Unfortunately, in this case the efficiency of the data structure is dominated by I/O costs and performance decreases orders of magnitudes.

Regarding FP-growth, our main idea to avoid this problem is to construct several small local FP-trees for subsets of the transaction database. To do so, we first need to split the transaction database into several smaller databases, called shards (also know as

---

<sup>1</sup>Also known as prefix tree.

horizontal partitions). Using the framework described in Chapter 3, we store each shard in a dedicated RDBMS instance<sup>2</sup> and during FP-tree construction we construct a local FP-tree for each shard, i.e. an FP-tree that corresponds only with the transactions of one shard. When constructing a conditional FP-tree for an item in the mining process, we fetch the set of prefix paths from all shards, aggregate them and use them to construct a global conditional FP-tree for the item. Finally, we mine all frequent patterns for the item and continue until all frequent patterns are found.

This method is correct, because the union of all local FP-trees represents the complete transaction database, since for each consecutive shard of the transaction database a local FP-tree was constructed.

Whereas in a traditional FP-tree the global FP-tree needs to completely fit in main memory for optimal performance, in our approach only the conditional FP-tree of a single item needs to fit in main memory. The conditional FP-tree of a single item is expected to be smaller than a global FP-tree. This depends on the data set and the support of an item: e.g. items with high support often have short prefix paths and thus have smaller conditional FP-trees. Meanwhile, items with low support often have long prefix paths and larger conditional FP-trees. When a conditional FP-tree of a single item doesn't fit in main memory, you can generalize this idea and also construct conditional FP-trees using SQL and store them in a RDBMS.

## 4.2 Computation distribution & parallelization

Frequent pattern mining with FP-growth is computationally expensive mainly for two reasons:

1. Constructing an FP-tree takes  $O(|db|)$  time, which is not inefficient, however the size of a transaction database is typically very large.
2. The set of possible frequent patterns is exponential in size, so FP-growth's search space is also exponential. As seen in Section 2.1.1, the frequent pattern mining problem is in the class of NP-hard problems.

A way to improve the performance and scale FP-growth is to distribute computation and parallelize tasks.

---

<sup>2</sup>An alternative would be to use one RDBMS instance and rely on RDBMS capabilities to handle large databases, but in section 4.2 we will show that multiple RDBMS instances are beneficial for distributing computation.

The approach for constructing FP-trees as described in Section 4.1 distributes the FP-tree construction over multiple RDBMS instances that work in parallel. Following this approach, we can also split the mining task into subtasks that can be executed in parallel distributed over multiple instances of the mining process. In the next section we will describe this in more detail.

### 4.3 Distributed FP-growth

The distributed FP-growth framework incorporates five steps to find all frequent patterns in a distributed manner (see Figure 4.1 for an overview):

Step 1

**Sharding:** Divide the transaction database into successive shards of nearly equal size and store each shard on a dedicated RDBMS instance.

Step 2

**Item counting:** Count item frequencies of each shard in parallel as described in Section 3.3.2, merge the counts in the master process (i.e. calculating the sum of the local frequencies of each item), remove infrequent items and finally write the total counts of all frequent items back to each RDBMS instance.

Step 3

**Transaction ordering:** Order items of transactions in support descending order in each shard in parallel as described in Section 3.3.3.

Step 4

**FP-tree construction:** Build local FP-trees for each shard in parallel as described in Section 3.3.4.

Step 5

**Mine frequent patterns:** Mine frequent patterns using the approach described in Section 3.4. Construct an item's conditional FP-tree by fetching conditional pattern bases from the local FP-trees residing in each shard. Doing so, multiple items can be mined in parallel.

**Example:** let  $T$  (Table 3.2, i.e. the data set of the running example of Chapter 3) be the global transaction database and minimum support threshold  $\xi = 2$ . In this example we first divide  $T$  into two successive shards of nearly equal size (step 1):  $T_1$  and  $T_2$  (Table 4.1). Next, we count the local item frequencies of each shard (Table 4.2), merge

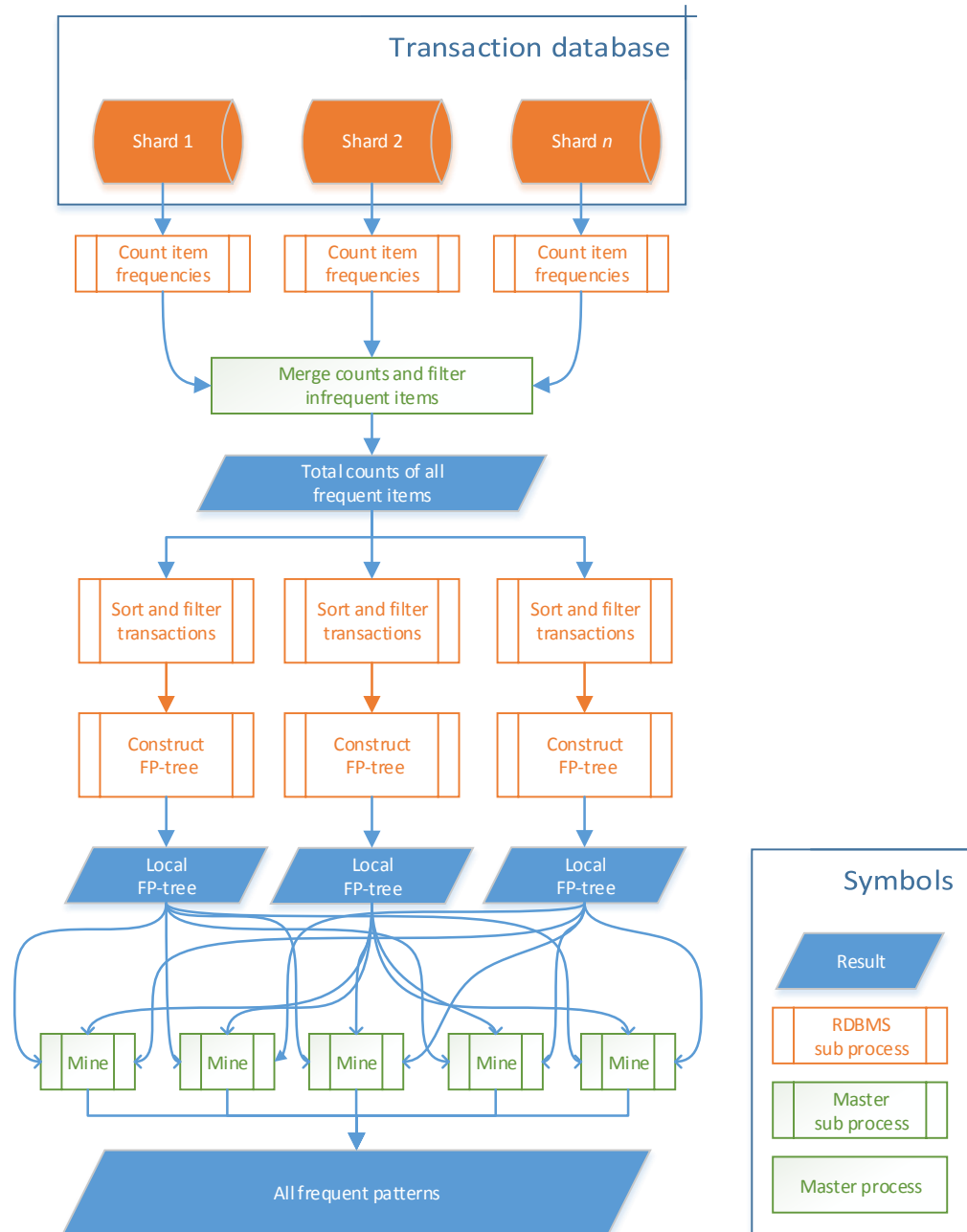


FIGURE 4.1: Overview of the distributed FP-growth framework

TID	ITEM
1	d
1	a
1	e
1	g
2	b
2	a
3	h
3	a
3	c
3	b
3	e

(A)  $T_1$

TID	ITEM
4	a
4	b
4	c
4	d
5	c
5	a
5	e
5	b

(B)  $T_2$

TABLE 4.1:  $T$  divided in two shards:  $T_1$  and  $T_2$ 

ITEM	SUPPORT
a	3
b	2
c	1
e	2
d	1
g	1
h	1

(A)  $T_1$

ITEM	SUPPORT
a	2
b	2
c	2
e	1
d	1

(B)  $T_2$

TABLE 4.2: Local item frequencies of  $T_1$  and  $T_2$ 

ITEM	SUPPORT
a	5
b	4
c	3
e	3
d	2

TABLE 4.3: Table  $TOTAL\_COUNTS$ : global item frequencies of tables  $T_1$  and  $T_2$ , without infrequent items

them, remove infrequent items and write them back to table  $TOTAL\_COUNTS$  (Table 4.3) of each shard (step 2).

In each shard, we reorder the items of all transactions in support descending order using the just computed global item frequencies (step 3). The results of this step are stored in table  $SORTED\_TRANSACTION$  of each shard (Table 4.4).

To finish the tree construction phase, we construct local FP-trees for each shard using table  $SORTED\_TRANSACTION$  (Table 4.5 and Figure 4.2) (step 4). Observe that the

TID	ITEM
1	a
1	e
1	d
2	a
2	b
3	a
3	b
3	c
3	e

(A)  $T_1$

TID	ITEM
4	a
4	b
4	c
4	d
5	a
5	b
5	c
5	e

(B)  $T_2$

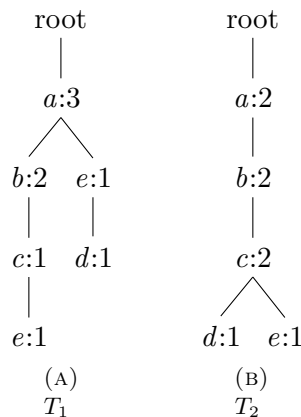
TABLE 4.4: Table *SORTED\_TRANSACTION* of  $T_1$  and  $T_2$ 

PATH	COUNT
;a;e;d;	1
;a;b;	1
;a;b;c;e;	1

(A)  $T_1$

PATH	COUNT
;a;b;c;e;	1
;a;b;c;d;	1

(B)  $T_2$

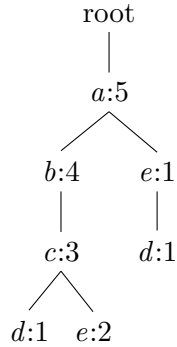
TABLE 4.5: Table *FP\_TREE* of  $T_1$  and  $T_2$ FIGURE 4.2: Local FP-trees of  $T_1$  and  $T_2$ 

union of these local FP-trees represents the global FP-tree of  $T$  (Table 4.6 and Figure 4.3).

In the mining phase (step 5), we iterate table *TOTAL\_COUNTS*, starting with item  $a$  with support 5. We generate the first pattern:  $(a : 5)$ . To construct  $a$ 's conditional FP-tree  $Tree_a$ , we fetch all prefix paths from table *FP\_TREE* of each shard ( $T_1$  and  $T_2$ ) and merge them.  $a$  has no prefix paths, so we continue with the next item.

The next item of *TOTAL\_COUNTS* is  $b$  with support 4, and again we output this item immediately as a frequent pattern, i.e.  $(b : 4)$ . To construct  $b$ 's conditional FP-tree

PATH	COUNT
;a;e;d;	1
;a;b;	1
;a;b;c;e;	2
;a;b;c;d;	1

TABLE 4.6: Global *FP-TREE* tableFIGURE 4.3: Global FP-tree of  $T$ 

PATH	COUNT
;a;	2

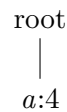
(A)  $T_1$

PATH	COUNT
;a;	2

(B)  $T_2$

TABLE 4.7: Local prefix paths of item  $b$  fetched from *FP-TREE* of  $T_1$  and  $T_2$ 

PATH	COUNT
;a;	4

TABLE 4.8: Global prefix paths of item  $b$ FIGURE 4.4:  $Tree_b$ : conditional FP-tree of item  $b$ 

$Tree_b$ , we fetch all prefix paths from table *FP-TREE* of each shard (Table 4.7) and merge them (Table 4.8). This time one global prefix path is returned, which we use to construct  $Tree_b$  (Figure 4.4). This tree contains one single path, thus the pattern  $(ab : 4)$  is generated.

The next item is  $c$  with support 3 and we output pattern  $(c : 3)$ . Again, we fetch all prefix paths from table *FP-TREE* of each shard (Tables 4.9) and merge them (Table 4.10). Again, the constructed conditional FP-tree  $Tree_c$  (Figure 4.5) contains one single path, thus the patterns  $(ac : 3)$ ,  $(bc : 3)$ ,  $(abc : 3)$  are generated (all combinations of the



PATH	COUNT
;a;b;	1

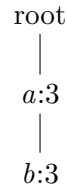
(A)  $T_1$

PATH	COUNT
;a;b;	2

(B)  $T_2$

TABLE 4.9: Local prefix paths of item  $c$  fetched from *FP-TREE* of  $T_1$  and  $T_2$ 

PATH	COUNT
;a;b;	3

TABLE 4.10: Global prefix paths of item  $c$ FIGURE 4.5:  $Tree_c$ : conditional FP-tree of item  $c$ 

items on the path). We continue this way until all frequent patterns are found (see Table 3.9).

## 4.4 Experimental setup

### 4.4.1 Implementation

In order to evaluate the correctness and performance of the distributed FP-growth framework, we implemented the FP-growth mining process in a Microsoft .NET application. This application interfaces through ODBC with MonetDB instances. MonetDB is an open source column-store RDBMS developed at the Centrum Wiskunde & Informatica<sup>3</sup>, that was designed to provide high performance on complex queries against large databases.

To be exact, we implemented the mining task in a .NET 4.5.1 application using C# and Microsoft Visual Studio 2013. This applications runs on each RDBMS instance virtual machine. In our implementation we use the original FP-growth mine algorithm[12], excluding any recent improvements. To coordinate the tree construction task, we developed another application that executes queries through ODBC in MonetDB.

<sup>3</sup>CWI, the Dutch National Research Institute for Mathematics and Computer Science.

	D14	webdocs	pumsb
Total size	13,7 GB	3,7 GB	40 MB
Transactions	5.000.000	1.692.082	49.046
Items	15.000.000	5.267.656	2.113
Average transaction size	180	177	74
Number of tuples	900.000.000	299.887.139	3.629.404

TABLE 4.11: Characteristics of the data sets used in the experiments.

MonetDB provided all necessary SQL features to implement the prototype, except for a string concat aggregate function. To overcome this shortcoming, we implemented<sup>4</sup> this operator in C and recompiled MonetDB.

#### 4.4.2 Data sets

Basically, it's only interesting to benchmark our prototype on large data sets, because the size of a data set makes frequent pattern mining an interesting problem. For this reason we use the *webdocs*[26] data set in the experiments. This data set is the largest publicly available real-life transactional data set, made available to the data mining community through the FIMI repository. This data set is frequently used in publications about frequent pattern mining. In addition, we generated a large synthetic data set called *D14* using the IBM Quest Data Set Generator. However, to also benchmark our prototype on a smaller data set, we also use the *pumsb* data set, containing census data from PUMS (Public Use Microdata Sample). See Table 4.11 for the characteristics of these data sets. Total size denotes the total data set size on disk, represented as an ASCII file with a single column layout. Number of tuples denotes the total number of tuples of this data set in a table with a single column layout.

#### 4.4.3 Test environment

We use virtual machines of the Microsoft Azure cloud computing platform and infrastructure in data center Europe-West (Amsterdam) to run our experiments:

- **Master instance:** 1x A3 (basic tier) virtual machine with 4 CPU cores, 7 GB main memory, 500 IOPS disk and Windows Server 2012 R2 Datacenter OS
- **RDMBS instances:** 4x A6 (basic tier) virtual machine with 4 CPU cores, 28 GB main memory, 500 IOPS disk, Windows Server 2012 R2 Datacenter OS, MonetDB (version Jan 2014-SP1)

<sup>4</sup>At first, we implemented this function in MAL (MonetDB Assembly Language) using an iterative function, but this implementation suffered from bad performance on large data sets (such as *webdocs*).

Notice that these virtual machines have commodity hardware, e.g. Microsoft doesn't even state in their documentation which CPU's and memory types are used. For high performance applications, Microsoft offers A8 and A9 compute intensive virtual machines that features Intel Xeon E5-2670 CPU and fast DDR3-1600 MHz main memory. Nevertheless, we believe that scale out (i.e. use more instances) is the right way to choose for our application, rather than scale up (i.e. using newer, better or more hardware in a single instance).

For making a rough comparison with other FP-growth algorithms we use one extra Linux virtual machine to run experiments with these algorithms:

- 1x A3 (basic tier) virtual machine with 4 CPU cores, 7 GB main memory, 500 IOPS disk and Ubuntu Server 14.04 LTS

#### 4.4.4 Experiments

We evaluated our prototype on data sets *webdocs* and *pumsb* with the same minimum support thresholds as used in the renowned FIMI '04 contest. To compare our results, we make a rough comparison with AFOPT[19] and LCM[20] on the Linux virtual machine described before. These algorithms are among the best performing algorithms of FIMI '04 and their implementation and source code is publicly available. Note that we can't make a fair comparison with these algorithms for numerous reasons. Also, the correctness of our implementation's output is validated using the output of AFOPT and LCM. Since they are both valid submissions of the FIMI contest, we assume their output is correct.

All experiments were run with a so called warm start, opposed to a cold start. This means that the data sets were completely loaded in main memory before the experiments were run. This also applies to the experiments with AFOPT and LCM. Sharding was executed as a pre-processing step and is not included in the experimental results. For our prototype the execution times of FP-tree initialization and mining are measured separately. We consider the mining process done when the algorithm successfully terminates. Statistics are collected from 5 test runs and averages are used for comparison.

To evaluate the scalability of our prototype, we ran experiments with 1 and 4 computing instances.

$\xi$ (minimum support)	Frequent items	Patterns mined
45.000	20	1.164
35.000	34	1.897.480
25.000	51	128.028.883

TABLE 4.12: Number of frequent items and patterns mined on *pumsb*

$\xi$ (minimum support)	Frequent items	Patterns mined
400.000	34	585
350.000	56	1.318
300.000	83	3.468
250.000	122	11.564
200.000	195	58.296
150.000	313	599.975

TABLE 4.13: Number of frequent items and patterns mined on *webdocs*

$\xi$ (minimum support)	Frequent items	Patterns mined
45.000	6.561	51.770
40.000	7.828	62.011
35.000	9.315	77.752

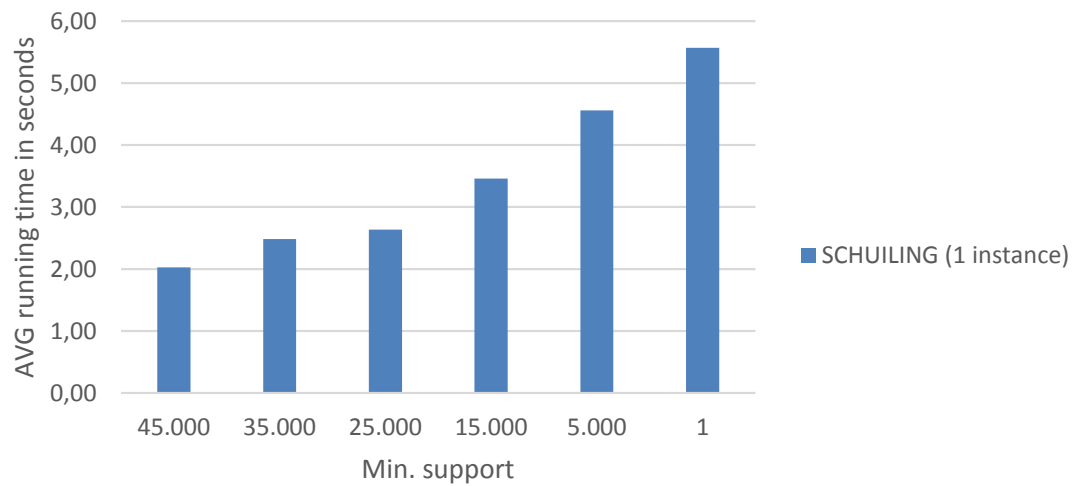
TABLE 4.14: Number of frequent items and patterns mined on *D14*

## 4.5 Experimental results

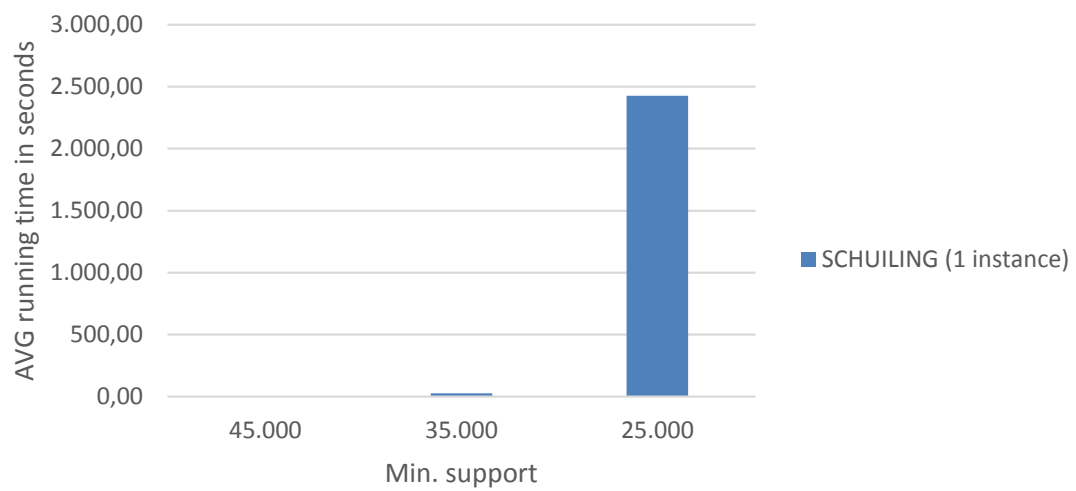
See Tables 4.12, 4.13, 4.14 and Figures 4.6, 4.7, 4.8 for the results of our experiments. Only overall running times of AFOPT and LCM are provided, since for these algorithms it's not possible to measure the running times of tree construction and mining separately. No results of AFOPT and LCM on data set *D14* are provided, because both algorithms ran out of memory in all experiments on this data set.

## 4.6 Conclusions

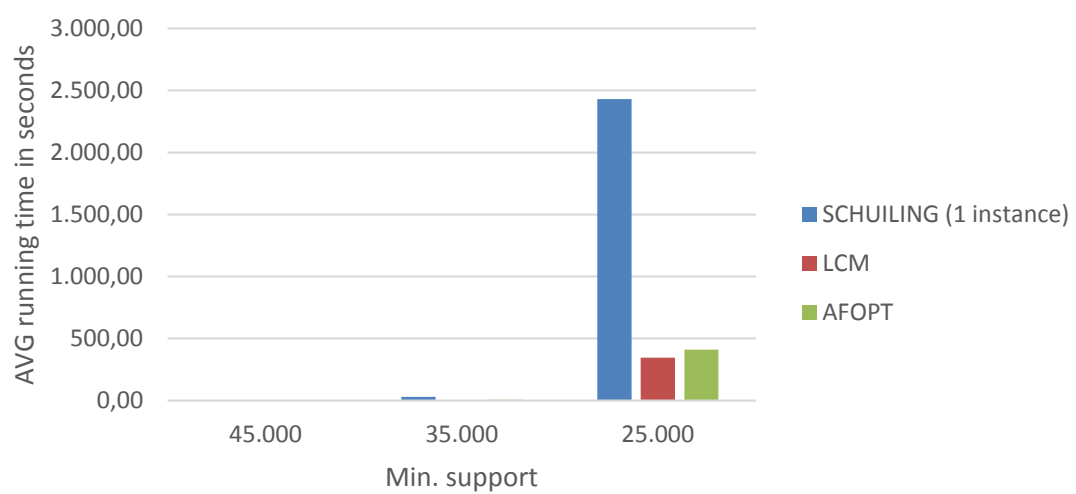
*Pumsb* is a dense data set and therefore great to measure the raw mining performance of our prototype. The number of frequent patterns explodes when the minimum support threshold  $\xi = 25.000$ . We observe that on this data set the mining task is the main bottleneck of our prototype. The overall performance of AFOPT and LCM is superior to our prototype with a factor of  $\sim 6$ . This may be explained by the fact that AFOPT and LCM are highly optimized low-level implementations written in C or C++, in contrast to our naive original FP-growth implementation that is based on .NET, which uses a just-in-time (JIT) compiler.



(A) Average tree construction running times

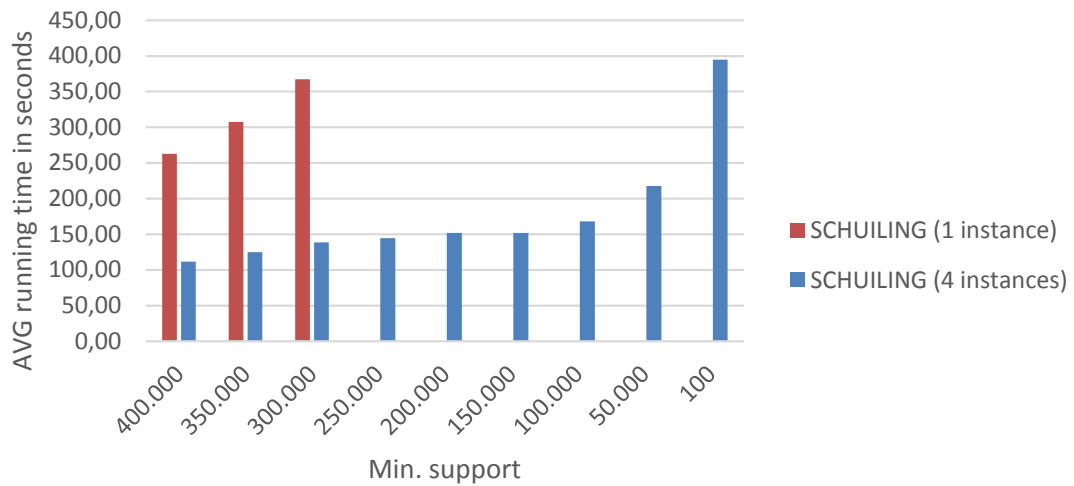


(B) Average mining running times

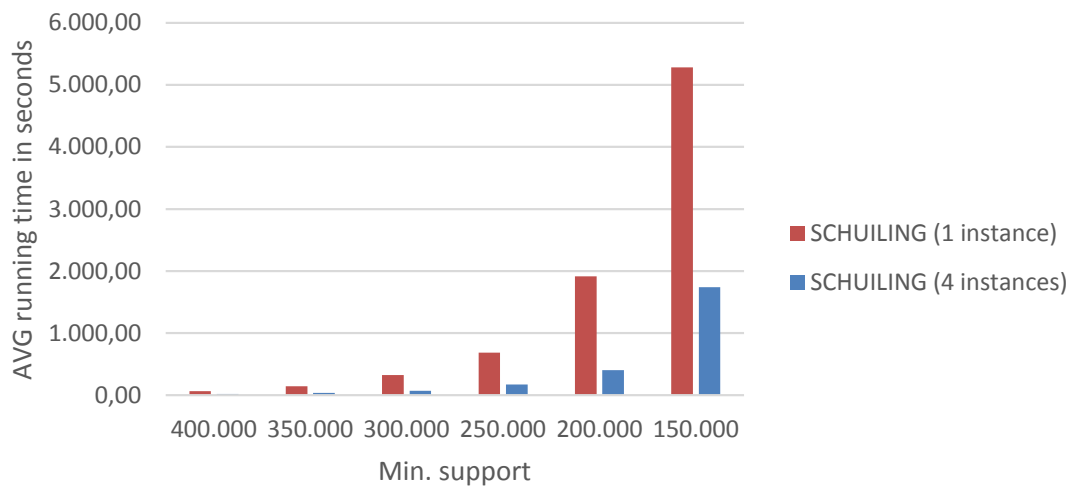


(C) Average total running times

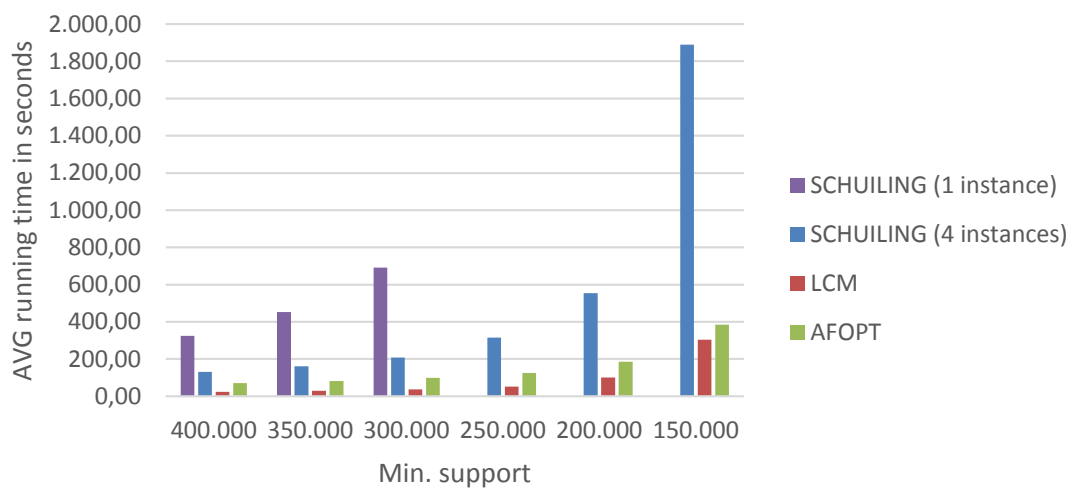
FIGURE 4.6: Results of *pumsb*



(A) Average tree construction running times

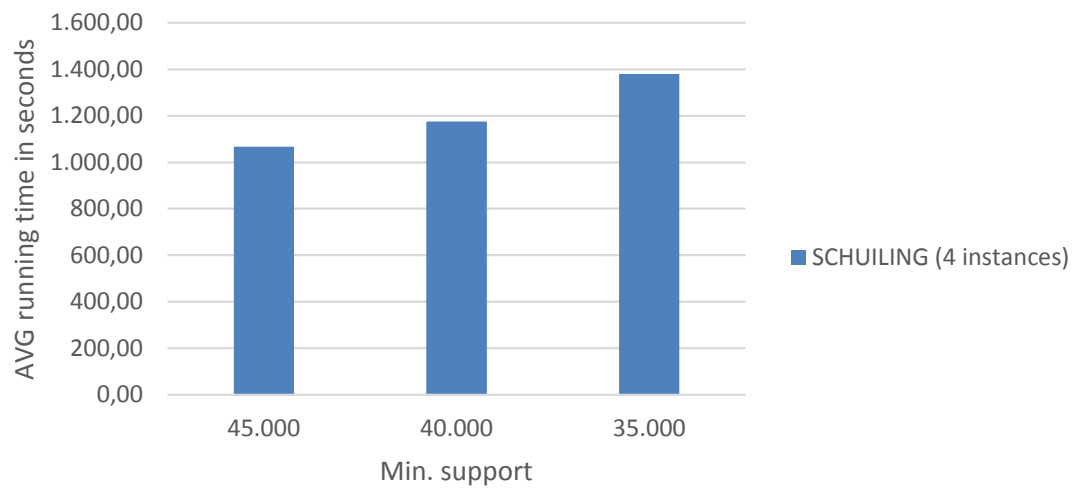


(B) Average mining running times

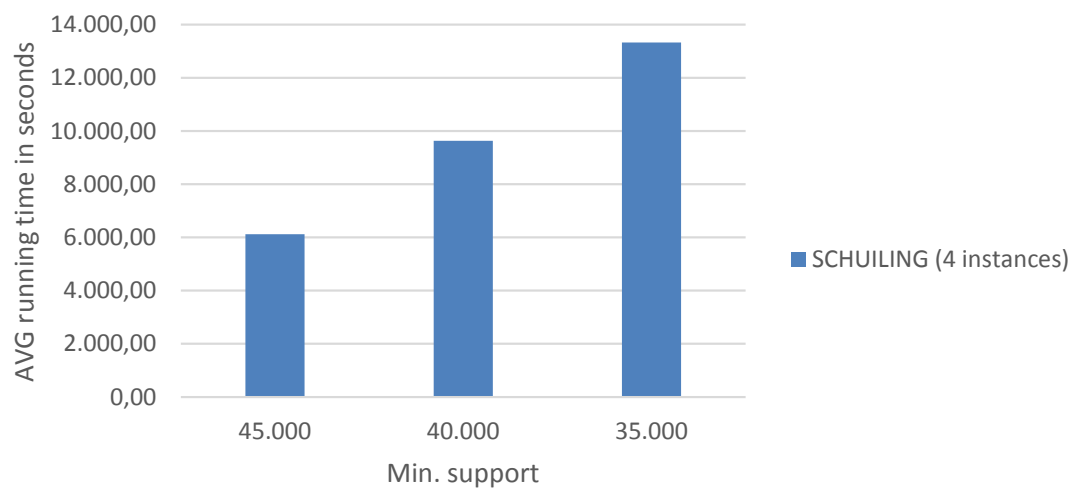


(C) Average total running times

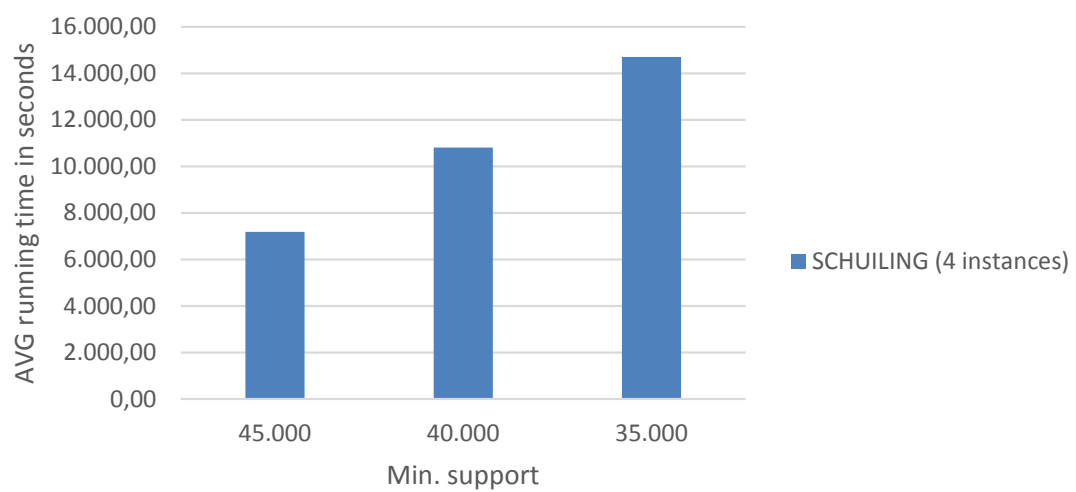
FIGURE 4.7: Results of *webdocs*



(A) Average tree construction running times



(B) Average mining running times



(C) Average total running times

FIGURE 4.8: Results of *D14*

*Webdocs* is a large sparse data set. Considering the tree construction task, on *webdocs* it's possible to initialize an FP-tree with  $\xi = 100$  in reasonable time with 4 instances. Since the number of frequent items affects the cost of joining tables *TOTAL\_COUNTS* and *T*, the effect of the minimum support parameter on the running time is clearly visible. The results show a near-linear speedup when computing instances are added. Also notice that on *webdocs* with 1 instance it's not possible to construct an FP-tree with minimum support values lower than 300.000 due to limitations of MonetDB in our test environment<sup>5</sup>.

The results of the mining task show a near-linear speedup when instances are added. Also, on *webdocs* AFOPT and LCM perform better than our prototype, but by scaling out the performance of our prototype increases and larger data sets can be handled. AFOPT and LCM lack this capability and are limited to one instance.

The experiments on *D14* show that AFOPT and LCM are unable to handle a data set of this size on our test machine, while our prototype has no issues. Likewise on this data set, the mining task takes the most running time.

We conclude that our method is capable of handling large data sets using a scale out strategy. As more instances are added performance increases linearly, e.g. as shown in Figure 4.7a. Our results show that it's feasible to mine large data sets as *webdocs* and *D14* in reasonable time, using a non-optimized FP-growth implementation, with a limited number of instances on commodity hardware.

---

<sup>5</sup>During query processing, MonetDB requires for each single operation during the query execution that all its inputs, its outputs, and possible temporary data structures fit in the address space. MonetDB automatically resorts to virtual memory and memory-mapped files for large intermediate results.[27] However, no swap space is available in our virtual machines, thus the amount of available physical memory can't be exceeded.



## Chapter 5

# Top- $k$ Frequent Pattern Mining

Most frequent pattern mining algorithms require the specification of a minimum support threshold  $\xi$ . However, in practice it's difficult to determine an appropriate value for this parameter. In this chapter we show how we can change the frequent pattern mining task of mining all patterns satisfying a minimum support threshold  $\xi$  to mining the  $k$  most frequent patterns, which is called top- $k$  frequent pattern mining. First, we develop an FP-growth based top- $k$  frequent pattern mining method to efficiently find the set of  $k$  most frequent patterns and we integrate this method in our hybrid FP-growth framework. Finally, we show and discuss the results of an experimental prototype.

### 5.1 Development of an FP-growth based Top- $k$ Frequent Pattern Mining method

In general, a scoring function is used for ranking answers of a top- $k$  query. It's desirable that the top- $k$  answers are found as quickly as possible. Once the top- $k$  set is complete and certainly won't be extended in further continuation of the algorithm, it may terminate.

In our application, the support of a pattern is used as a scoring function. In an ideal scenario our algorithm would mine patterns in monotonically decreasing order of support. Once the  $k$  most frequent patterns are generated, we may stop the algorithm<sup>1</sup>. Is this scenario feasible with FP-growth? And how can we achieve this?

---

<sup>1</sup>To be exact, each pattern whose support is no less than the support of the  $k$ th value in the set of  $k$  most frequent patterns is generated and output. Notice that multiple patterns can have the same support in the transaction database, see Section 5.1.4 for an example.

In the next two sections we develop an FP-growth based top- $k$  frequent pattern mining method. Thereafter we present a full description of the algorithm and provide a detailed example.

### 5.1.1 Generating patterns in support decreasing order

Notice that FP-growth generates conditional FP-trees in a "down-top" manner: the mining algorithm traverses header tables in support ascending order (see Algorithm 4)<sup>2</sup>. This means that the algorithm mines the least frequent item of the global FP-tree first; its conditional FP-tree is generated, the algorithm goes into recursion to mine the just constructed conditional FP-tree and again the header table—this time of the conditional FP-tree—is mined in support ascending order, and so on. Furthermore, observe that FP-growth uses a depth-first-search strategy, as the algorithm goes into recursion immediately.

Our objective is to generate patterns in monotonically decreasing order of support. However, this is not ideal from a practical point of view as we'll show below.

First, we should change the algorithm such that conditional FP-trees are generated in a "top-down" manner; the most frequent items should be mined first. To do so, we change the order in which header tables are traversed from support ascending order to support descending order.

As well, we change FP-growth's depth-first-search strategy to a level-wise search strategy, in which the level corresponds with the number of items in a suffix itemset. This search strategy is based on the following property: a pattern has no proper super patterns with higher support.<sup>3</sup>

This property ensures that mining a conditional FP-tree yields no patterns with higher support than the suffix itemset  $\alpha$  that generated the conditional FP-tree. So, if the algorithm doesn't go into recursion immediately while traversing a header table, patterns will be generated in support decreasing order.

This level-wise search starts on level 2 (see Algorithm 7 and 8) and is implemented using a queue  $Q$ , which holds two-tuples of conditional FP-trees and their corresponding suffix itemsets. On each level  $i$ , the conditional FP-trees generated on level  $i - 1$  are dequeued, patterns of length  $i$  are mined and new conditional FP-trees with their corresponding suffix itemsets are enqueued, and this repeats until  $Q$  is empty. See Algorithm 7 for a full description of this algorithm.

---

<sup>2</sup>In fact, for mining all frequent patterns it makes no difference at all in which order the header tables are traversed.

<sup>3</sup>It's easy to see that this property holds.

Though, using this approach patterns are not yet generated in monotonically decreasing order of support. In the next section we show this is not an issue, if we prune the search space effectively.

### 5.1.2 Pruning the search space

Despite our algorithm has no minimum support input parameter  $\xi$ , we still use the notion of minimum support internally in the algorithm. With this parameter it's possible to effectively prune the search space by raising its value during the mining process. This is based on the fact that patterns can only enter the top- $k$  set if their support is higher or equal to the support of the  $k$ th pattern in the top- $k$  set.

Initially  $\xi$  is set to 1 or to the support value of the  $k$ th item in the header table of the global FP-tree if  $k \leq n$ , in which  $n$  equals the total number of distinct items in  $db$ . The latter is correct, because patterns generated from items in the header table with support less than the support of the  $k$ th item in the header table will certainly not be included in the top- $k$  set. Just like the default FP-growth algorithm, given  $\xi$  an FP-tree is constructed (see Algorithm 3).

Let  $\lambda$  be the value of the  $k$ th item in the set of unique support values corresponding with the top- $k$  set. Once  $k$  patterns are generated,  $\xi$  is raised to  $\lambda$ , if  $\lambda > \xi$ . The mining algorithm only generates a conditional FP-tree when the support of its suffix itemset is higher or equal to  $\xi$ . Also, patterns are only generated when their support is greater or equal to  $\xi$ . Doing so, the search space is pruned in the mining process and the algorithm terminates when top- $k$  patterns are generated.

### 5.1.3 Full algorithm

See Algorithms 7, 8 and 9 for a full description of the FP-growth based top- $k$  frequent pattern mining method.

### 5.1.4 Example

In this example we show how to mine the top-6 patterns of the *pumsb* data set.

Since  $k = 6$ ,  $n = 2113$  and  $k \leq n$ , minimum support threshold  $\xi$  is initialized to the support value of the 5th item of the support descending ordered header table corresponding with  $db$ ;  $\xi$  is set to 48193. Given this parameter FP-tree *Tree* is constructed, as shown

---

**Algorithm 7** Mining top- $k$  patterns based on FP-growth

---

**Input:** An FP-tree  $Tree$ , parameter  $k$ .**Output:** The complete set of frequent patterns.

- 1: Let the header table of  $Tree$  be sorted in support decreasing order
  - 2: Let  $\theta$  be the set of mined patterns
  - 3: Let  $Q$  be a queue
  - 4: Let minimum support threshold  $\xi$  be 1 or the support value of the  $k$ th item in the header table of  $Tree$
  - 5: **for** each item  $a_i$  in the header table of  $Tree$  **do**
  - 6:     **if**  $a_i.support < \xi$  **then**
  - 7:         Break
  - 8:     **end if**
  - 9:     Generate pattern  $\beta := a_i$  with  $support := a_i.support$  (Algorithm 9)
  - 10:     Construct conditional pattern base of  $\beta$  and then  $\beta$ 's conditional FP-tree  $Tree_\beta$
  - 11:     Call  $Mine(Tree_\beta, \beta, Q, k, \theta, \xi)$  (Algorithm 8)
  - 12:     **while**  $Q \neq \emptyset$  **do**
  - 13:          $(Tree_\zeta, \zeta) := Q.Dequeue$
  - 14:         Call  $Mine(Tree_\zeta, \zeta, Q, k, \theta, \xi)$  (Algorithm 8)
  - 15:     **end while**
  - 16: **end for**
  - 17: Retrieve and output top- $k$  patterns in support descending order from  $\theta$
- 

---

**Algorithm 8** Mining top- $k$  patterns based on FP-growth: level- $i$ , with  $i > 1$ 

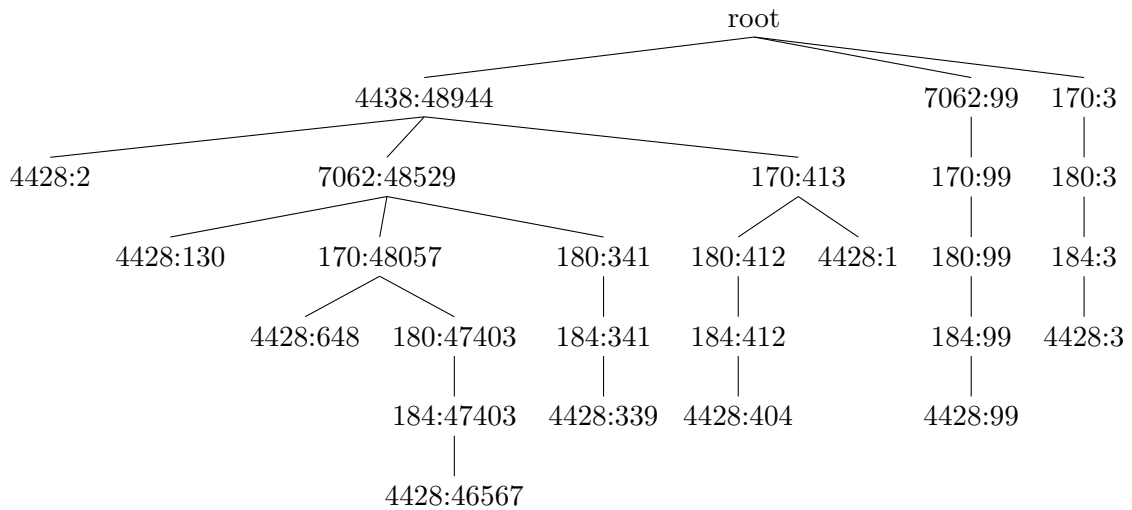
---

**Input:** An FP-tree  $Tree$ , a set of suffix items  $\alpha$ , a queue  $Q$ , parameter  $k$ , pattern set  $\theta$ , minimum support threshold  $\xi$ .

- 1: **if**  $Tree$  contains a single path  $P$  **then**
  - 2:     **for** each combination  $\beta$  of the nodes on the path  $P$  with support  $\geq \xi$  **do**
  - 3:         Generate pattern  $\beta \cup \alpha$  with  $support := \text{minimum support of the nodes in } \beta$  (Algorithm 9)
  - 4:     **end for**
  - 5: **else**
  - 6:     **for** each item  $a_i$  in the header table of  $Tree$  (in support descending order) **do**
  - 7:         **if**  $a_i.support < \xi$  **then**
  - 8:             Break
  - 9:         **end if**
  - 10:         Generate pattern  $\beta := a_i \cup \alpha$  with  $support := a_i.support$  (Algorithm 9)
  - 11:         Construct conditional pattern base of  $\beta$  and then  $\beta$ 's conditional FP-tree  $Tree_\beta$
  - 12:         **if**  $Tree_\beta \neq \emptyset$  **then**
  - 13:             Enqueue  $(Tree_\beta, \beta)$  in  $Q$
  - 14:         **end if**
  - 15:     **end for**
  - 16: **end if**
-

**Algorithm 9** Generate pattern**Input:** Pattern  $\beta$ , parameter  $k$ , pattern set  $\theta$ .

- 1: Append  $\beta$  to  $\theta$
- 2: **if**  $|\theta| > k$  **then**
- 3:    $\lambda :=$ support of the  $k$ th item in  $\theta$
- 4:   Minimum support threshold  $\xi := \lambda$
- 5: **end if**

FIGURE 5.1: FP-tree for *pumsb* with  $\xi = 48193$ 

Item	Support
4438	48944
7062	48628
170	48572
184	48258
180	48258
4428	48193

TABLE 5.1: Support descending ordered header table corresponding with the FP-tree of Figure 5.1

in Figure 5.1. The corresponding support descending ordered header table is shown in Table 5.1.

Next, we start mining the FP-tree. We iterate all items of the header table (level 1), starting with 4438 and the first pattern is generated: (4438 : 48944). 4438's conditional FP-tree is empty, so we continue with the next item: 7062. Mining this item generates the patterns (7062 : 48628) and (4438 7062 : 48529). Notice that still no tuples have been queued in  $Q$ , since all patterns so far were mined on level 1. Notice that even the 2-itemsets so far were generated on level 1, because of their single path property. The next item is 170 and this item generates the patterns: (170 : 48572) and (4438 170 : 48470). The next item is 180 and this item generates the pattern (180 : 48258).

Pattern	Support
4438	48944
7062	48628
170	48572
4438 7062	48529
4438 170	48470
180	48258
184	48258
180 184	48258

TABLE 5.2: Top-6 frequent patterns of *pumsb*

The next item is 184 and this item generates the patterns: (184 : 48258) and (180 184 : 48258). Furthermore, this item enqueues the conditional FP-tree of (180 184) in  $Q$ . This queue item is immediately processed when the mining of 184 is finished, however this conditional FP-tree doesn't generate any frequent pattern. Notice that at this time 6 unique support values are in the set of mined patterns  $\Theta$ . The next item is 4428 and this item generates the pattern: (4428 : 48193). The generation of this pattern raises the minimum support threshold  $\xi$  to 48258, which is the value of the  $k$ th unique support value of  $\Theta$ . The conditional FP-tree of 4428 is mined, but mining terminates immediately because no frequent patterns are left in this conditional FP-tree. The algorithm returns to level 1 and terminates since no frequent items are left in the header table. See Table 5.2 for an overview of the patterns mined.

## 5.2 Integration in the hybrid FP-growth framework

In this section we describe how to integrate the FP-growth based top- $k$  frequent pattern mining method in our hybrid FP-growth framework, that integrates frequent pattern mining in a RDBMS. As explained in the previous section, the tree construction algorithm is unchanged. However, the mining algorithm is changed such that patterns are generated in frequency descending order. Using Algorithm 7 it's trivial to integrate this modification in Algorithm 6, which is shown in Algorithm 10.

## 5.3 Adding a minimum length constraint

Adding a minimum length constraint  $min\_l$  to the top- $k$  pattern mining task, would make this task even more effective. To be exact, with this constraint merely the  $k$  most frequent patterns with minimum length  $min\_l$  will be generated. This constraint

---

**Algorithm 10** Top- $k$  frequent pattern mining in a relational database

---

**Input:** Database  $db$  with tables  $TOTAL\_COUNTS$  and  $FP\_TREE$ , parameter  $k$ .

**Output:** The complete set of frequent patterns.

- 1: Sort  $TOTAL\_COUNTS$  in support decreasing order
- 2: Let  $\theta$  be the set of mined patterns
- 3: Let minimum support threshold  $\xi$  be 1 or the support value of the  $k$ th item in the header table of  $Tree$
- 4: **for** each item  $a_i$  in table  $TOTAL\_COUNTS$  of  $db$  **do**
- 5:     **if**  $a_i.support < \xi$  **then**
- 6:         Break
- 7:     **end if**
- 8:     Generate pattern  $a_i$  with  $support = a_i.SUPPORT$  (Algorithm 9)
- 9:     Fetch conditional pattern base of  $a_i$  from table  $FP\_TREE$  of  $db$  and construct  $a_i$ 's conditional FP-tree  $Tree_{a_i}$
- 10:    **if**  $Tree_{a_i} \neq \emptyset$  **then**
- 11:       Call  $Mine(Tree_{a_i}, a_i, Q, k, \theta, \xi)$  (Algorithm 8)
- 12:    **end if**
- 13:    **while**  $Q \neq \emptyset$  **do**
- 14:        $(Tree_\zeta, \zeta) := Q.Dequeue$
- 15:       Call  $Mine(Tree_\zeta, \zeta, Q, k, \theta, \xi)$  (Algorithm 8)
- 16:    **end while**
- 17: **end for**
- 18: Retrieve and output top- $k$  patterns in support descending order from  $\theta$

---

would for instance give one the ability to mine only long patterns, without first finding numerous, possibly not meaningful, shorter patterns.

It's not trivial to integrate this constraint to our method. The internal minimum support parameter  $\xi$  is initially set to 1 or to the support value of the  $k$ th item in the header table of the global FP-tree if  $k \leq n$ , as described in Section 5.1.2. However, the latter doesn't hold if we add a minimum length constraint with  $min\_l > 1$ , because 1-itemsets will not be included in the top- $k$  set in this case, and thus won't imply any useful knowledge about the top- $k$  set. Consequently, the complete FP-tree needs to be constructed, which is expensive and not desirable. One way to solve this issue, is to incrementally raise  $\xi$  during tree construction and immediately prune the search space. Since FP-tree construction is fully implemented in SQL, this is not convenient.

Though, modifying the mining algorithm such that only patterns of minimum length  $min\_l$  are generated is straightforward, by checking the length of a pattern before generating it. However, we expect that also in the mining algorithm effective search pruning techniques may be implemented.

Unfortunately, we didn't have enough time to solve these challenges and successfully implement this constraint in our method. Therefore, we leave this task as future work (see Section 6.3).

$k$	Initial $\xi$	Final $\xi$	Difference
10	47.640	48.057	417
100	11.798	46.607	34.809
1.000	52	44.290	44.238
10.000	2	34.996	34.994

TABLE 5.3: Initial vs final minimum support thresholds of *pumsb*

$k$	Initial $\xi$	Final $\xi$	Difference
10	677.637	850.323	172.686
100	281.107	553.285	272.178
1.000	60.314	365.144	304.830
10.000	2.460	252.911	250.451

TABLE 5.4: Initial vs final minimum support thresholds of *webdocs*

## 5.4 Experimental setup

The main goal of our experiments is to verify the correctness of our method and evaluate its performance. We experimented with data sets *pumsb* and *webdocs* in the same environment and with the same methodology as in Section 4.4.2. However, in this experiment we only ran experiments with 1 mine instance, since no support for distributed top- $k$  frequent pattern mining has been implemented (yet, see Chapter 6).

In our experiments we tested with  $k$  values 10, 100, 1.000 and 10.000, of which we think they're reasonable for real-life applications. Unfortunately, we didn't find any comparable top- $k$  frequent pattern mining experimental results in the literature. Therefore, a comparison with other algorithms is omitted.

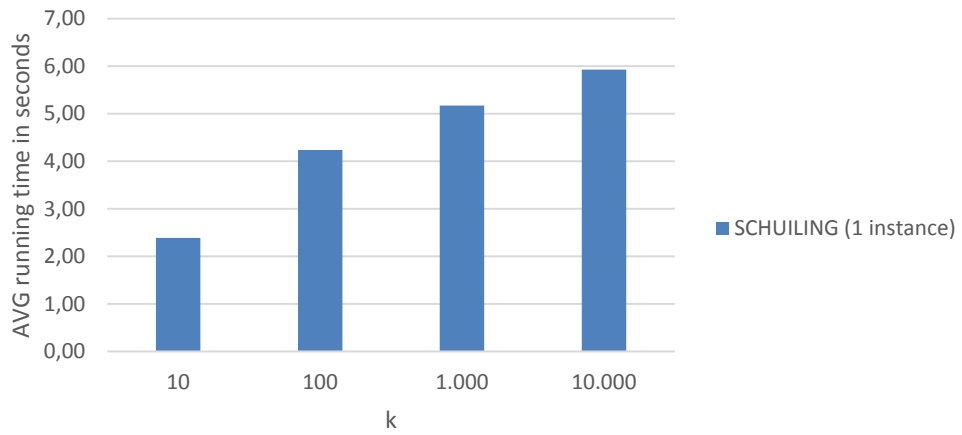
## 5.5 Experimental results

See Tables 5.3, 5.4 and Figures 5.2, 5.3 for the experimental results.

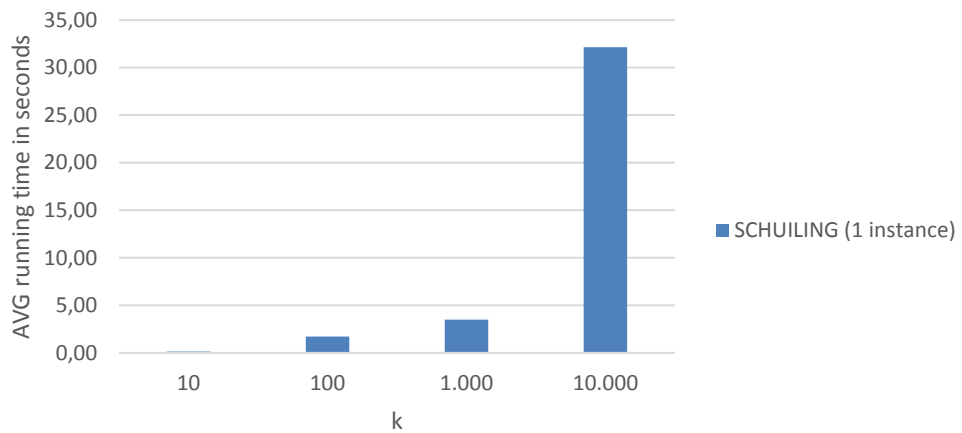
## 5.6 Conclusions

In line with the experimental results of Chapter 4 the running times of the tree construction increase as  $k$  increases on both data sets. This makes sense since the initial minimum support thresholds decrease when  $k$  increases. Again, the results show a less than linear running time increase when the minimum support threshold decreases. Given the results of Chapter 4, we conclude it's feasible to construct an FP-tree corresponding with large  $k$  values in reasonable time.

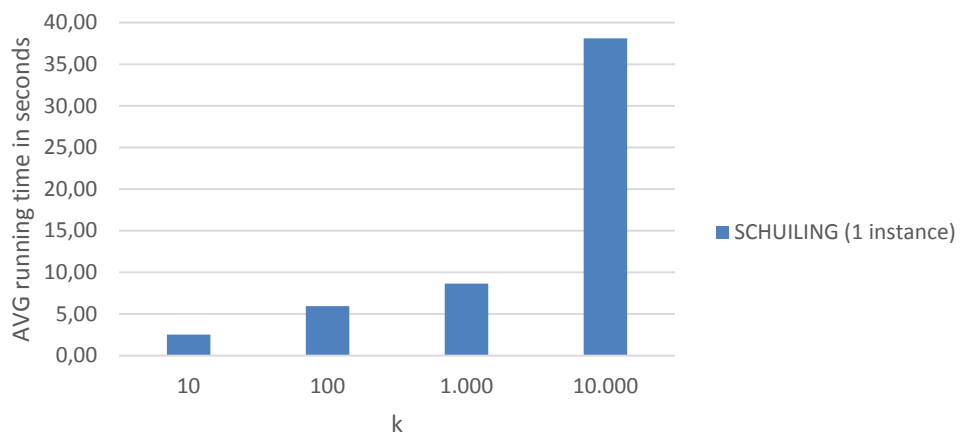




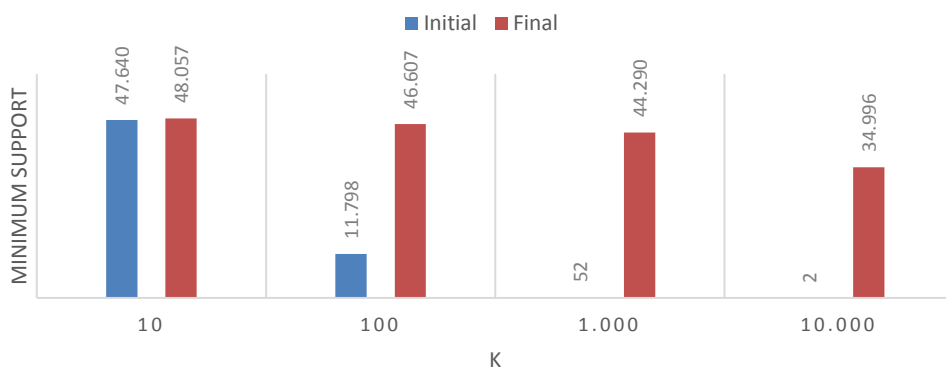
(A) Average tree construction running times



(B) Average mining running times

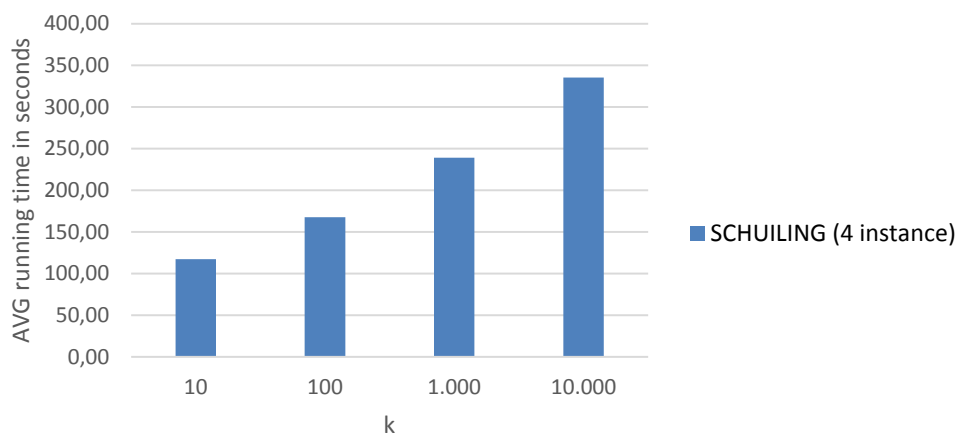


(C) Average total running times

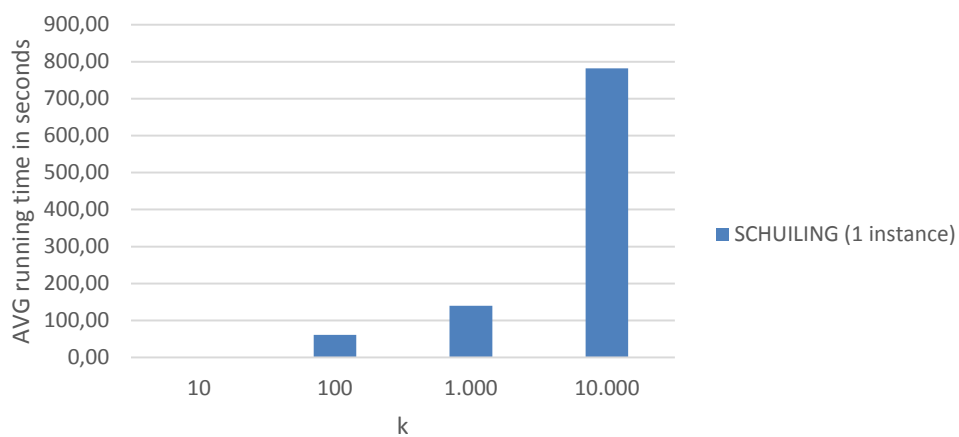


(D) Initial vs final minimum support thresholds

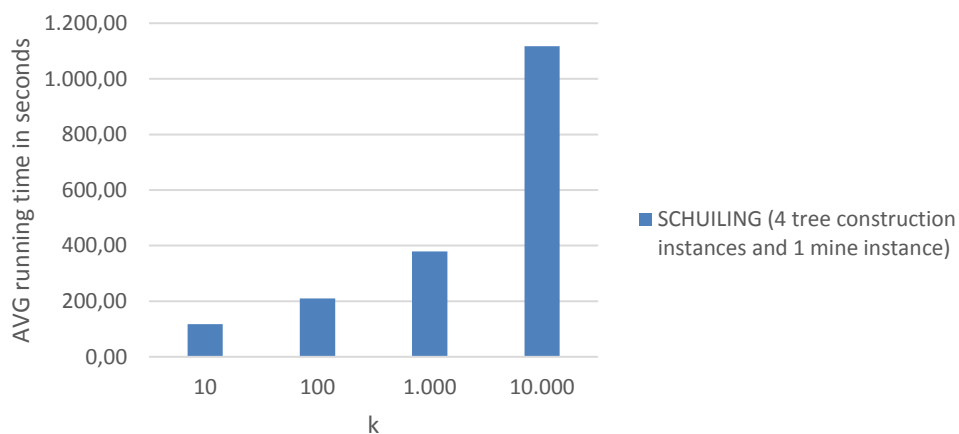
FIGURE 5.2: Results of *pumsb*



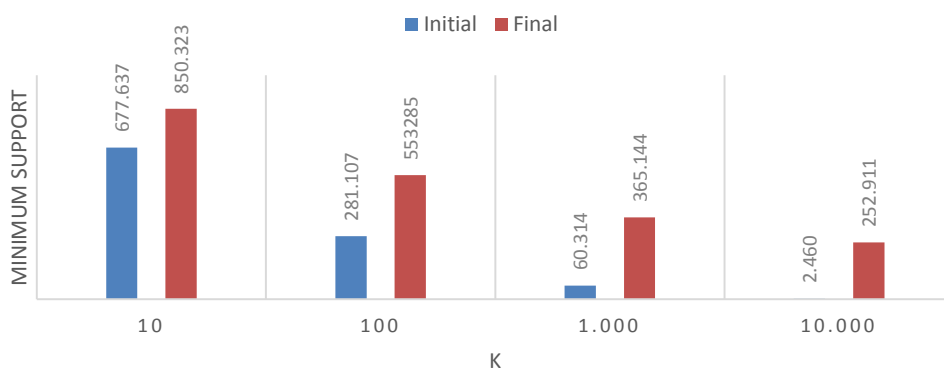
(A) Average tree construction running times



(B) Average mining running times



(C) Average total running times



(D) Initial vs final minimum support thresholds

FIGURE 5.3: Results of *webdocs*

On both data sets, the differences between the initial and final minimum support threshold are quite large. Here is possibly room for improvement (see Chapter 6), as this affects the size of the initial FP-tree and the number of patterns generated in the mining algorithm.

We conclude that our method has good overall performance on both *webdocs* and *pumsb* for  $k$  values varying from 1 up to 10.000.

## Chapter 6

# Future work

In this chapter we discuss some interesting future research directions for our work.

### 6.1 Support of condensed pattern representations

An obvious extension to our work is the support of mining maximal and closed patterns (see Section 2.1.4). This extension is also appropriate for top- $k$  mining. As condensed pattern representations substantially reduce the number of patterns generated without losing any information, it increases both the effectiveness and efficiency of mining. However, we expect there will be some challenges in computation parallelization and distribution.

### 6.2 Distributed Top- $k$ Frequent Pattern Mining

In this thesis we proposed a top- $k$  frequent pattern mining method that supports one computing instance. As scalability is of high significance, this method should be extended to a distributed version. We believe that our framework of Chapter 4 provides a sound basis for this extension, however we expect that additional communication between computing nodes is needed during the mining process in order to synchronize minimum support thresholds and patterns generated.

### 6.3 Adding a minimum length constraint to Top- $k$ Frequent Pattern Mining

As discussed in Section 5.3, adding a minimum length constraint  $min\_l$  to the top- $k$  pattern mining task, would make this task even more effective. Additional research is needed to develop and integrate a search pruning technique in the FP-tree construction algorithm. Also, we expect that further search pruning techniques may be implemented in the mining algorithm.

### 6.4 Minimize difference between initial and final minimum support threshold in Top- $k$ Frequent Pattern Mining

Experimental results of our top- $k$  frequent pattern mining prototype (see Section 5.5) show that the differences between the initial and final minimum support thresholds are quite large in our experiments. By minimizing these differences, we expect to improve the performance of the mining algorithm, as the initial minimum support threshold directly affects the size of the initial FP-tree and the number of patterns generated in the mining algorithm.

We have several ideas about how to achieve this. For instance, one could use a heuristic that 'guesses' the initial minimum support threshold value, based on characteristics of the data set, e.g. obtained by a sampling method. If mining with this threshold not (certainly) yields all top- $k$  patterns, the algorithm restarts with a lower initial minimum support, as in a local search method.

### 6.5 Incremental Frequent Pattern Mining

Since it's hard to pick an appropriate value for the minimum support threshold and  $k$  parameters at once, it would be great if the frequent pattern mining algorithm would support incremental mining, such that a complete restart of the algorithm is not required and already computed results will be reused in a next run. A literature study shows that some publications about this topic are already available.

## 6.6 Fault handling in distributed Frequent Pattern Mining

As the number of computing nodes running a distributed algorithm increases, the probability that a node crashes during execution increases also. Consequently, a distributed application must be fault tolerant and should be able to recover from node failures. Thus, comprehensive fault handling should be implemented in our hybrid FP-growth framework.

## Chapter 7

# Conclusions

In this thesis, we studied how to leverage RDBMS integration to improve the scalability of frequent pattern mining. After providing a thorough introduction of frequent pattern mining, we proposed a hybrid framework based on FP-growth, that successfully integrates frequent pattern mining in a RDBMS. For FP-tree construction we developed an SQL-based approach and for mining we developed an external application that interfaces with the RDBMS.

Subsequently, we adapted this framework to achieve high scalability, by sharding the transaction database into several smaller databases and by distributing computation and parallelizing tasks over multiple computing instances. Experimental results with MonetDB5 showed that this framework is capable of handling large data sets using a scale out strategy. It achieves near-linear speedup when computing instances are added.

Finally, we developed an FP-growth based top- $k$  frequent pattern mining method and integrated this method in our hybrid FP-growth framework. To mine the top- $k$  frequent patterns efficiently, we changed the FP-growth mining algorithm to mine patterns in support descending order using a level-wise search strategy instead of the depth-first-search strategy of the original FP-growth algorithm. Experimental results show that this method has surprisingly good performance on data sets *webdocs* and *pumsb* for  $k$  values varying from 1 up to 10.000.

# Bibliography

- [1] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996. ISBN 0-262-56097-6.
- [2] Soumen Chakrabarti, Martin Ester, and Jiawei Han Shinichi Morishita Gregory Piatetsky-Shapiro Wei Wang Usama Fayyad, Johannes Gehrke. Data mining curriculum: A proposal (version 1.0). 2006. URL <http://www.kdd.org/sites/default/files/CURMay06.pdf>.
- [3] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: Current status and future directions. *Data Min. Knowl. Discov.*, 15(1): 55–86, August 2007. ISSN 1384-5810. doi: 10.1007/s10618-006-0059-1. URL <http://dx.doi.org/10.1007/s10618-006-0059-1>.
- [4] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 207–216, New York, NY, USA, 1993. ACM. ISBN 0-89791-592-5. doi: 10.1145/170035.170072. URL <http://doi.acm.org/10.1145/170035.170072>.
- [5] Tomasz Imielinski and Heikki Mannila. A database perspective on knowledge discovery. *Commun. ACM*, 39(11):58–64, November 1996. ISSN 0001-0782. doi: 10.1145/240455.240472. URL <http://doi.acm.org/10.1145/240455.240472>.
- [6] Xuequn Shang, Kai-Uwe Sattler, and Ingolf Geist. Sql based frequent pattern mining with fp-growth. In Dietmar Seipel, Michael Hanus, Ulrich Geske, and Oskar Bartenstein, editors, *Applications of Declarative Programming and Knowledge Management*, volume 3392 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25560-4. doi: 10.1007/11415763\_3. URL [http://dx.doi.org/10.1007/11415763\\_3](http://dx.doi.org/10.1007/11415763_3).



- [7] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. Pfp: Parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM Conference on Recommender Systems, RecSys '08*, pages 107–114, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-093-7. doi: 10.1145/1454008.1454027. URL <http://doi.acm.org/10.1145/1454008.1454027>.
- [8] Jiawei Han, Jianyong Wang, Ying Lu, and Petre Tzvetkov. Mining top.k frequent closed patterns without minimum support. In *Proceedings of the 2002 IEEE International Conference on Data Mining, ICDM '02*, pages 211–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1754-4. URL <http://dl.acm.org/citation.cfm?id=844380.844747>.
- [9] Arno Siebes and Ad Feelders. Frequent item set mining and association rules. 2012.
- [10] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94*, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. ISBN 1-55860-153-8. URL <http://dl.acm.org/citation.cfm?id=645920.672836>.
- [11] Paul W. Purdom, Dirk Van Gucht, and Dennis P. Groth. Average-case performance of the apriori algorithm. *SIAM J. Comput.*, 33(5):1223–1260, 2004. URL <http://dblp.uni-trier.de/db/journals/siamcomp/siamcomp33.html#PurdomGG04>.
- [12] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD '00*, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-217-4. doi: 10.1145/342009.335372. URL <http://doi.acm.org/10.1145/342009.335372>.
- [13] Kotagiri Ramamohanarao and James Bailey. Discovery of emerging patterns and their use in classification. In Tamas(Tom)Domonkos Gedeon and LanceChunChe Fung, editors, *AI 2003: Advances in Artificial Intelligence*, volume 2903 of *Lecture Notes in Computer Science*, pages 1–11. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-20646-0. doi: 10.1007/978-3-540-24581-0\_1. URL [http://dx.doi.org/10.1007/978-3-540-24581-0\\_1](http://dx.doi.org/10.1007/978-3-540-24581-0_1).
- [14] Jian Pei, Jiawei Han, Hongjun Lu, Shojiro Nishio, Shiwei Tang, and Dongqing Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *Proceedings of the 2001 IEEE International Conference on Data Mining, ICDM '01*, pages 441–448, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1119-8. URL <http://dl.acm.org/citation.cfm?id=645496.657873>.

- [15] Junqiang Liu, Yunhe Pan, Ke Wang, and Jiawei Han. Mining frequent item sets by opportunistic projection. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 229–238, New York, NY, USA, 2002. ACM. ISBN 1-58113-567-X. doi: 10.1145/775047.775081. URL <http://doi.acm.org/10.1145/775047.775081>.
- [16] Guimei Liu, Hongjun Lu, Wenwu Lou, and Jeffrey Xu Yu. On computing, storing and querying frequent patterns. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 607–612, New York, NY, USA, 2003. ACM. ISBN 1-58113-737-0. doi: 10.1145/956750.956827. URL <http://doi.acm.org/10.1145/956750.956827>.
- [17] Gosta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets, 2003.
- [18] B. Rácz. nonordfp: An FP-growth variation without rebuilding the FP-tree. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, volume 126. Citeseer.
- [19] Guimei Liu, Hongjun Lu, and Jeffrey Xu Yu. Afopt: An efficient implementation of pattern growth approach. In *In Proceedings of the ICDM workshop*, 2003.
- [20] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In Roberto J. Bayardo Jr., Bart Goethals, and Mohammed Javeed Zaki, editors, *FIMI*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004. URL <http://dblp.uni-trier.de/db/conf/fimi/fimi2004.html#UnoKA04>.
- [21] Gregory Buehrer, Srinivasan Parthasarathy, and Amol 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*, KDD '06, pages 86–95, New York, NY, USA, 2006. ACM. ISBN 1-59593-339-5. doi: 10.1145/1150402.1150416. URL <http://doi.acm.org/10.1145/1150402.1150416>.
- [22] Karthick Rajamani, Alan L. Cox, Balakrishna R. Iyer, and Atul Chadha. Efficient mining for association rules with relational database systems. In *IDEAS*, pages 148–155, 1999. URL <http://dblp.uni-trier.de/db/conf/ideas/ideas1999.html#RajamaniCIC99>.
- [23] Ralf Rantza. Frequent itemset discovery with sql using universal quantification. In Rosa Meo, Pier Luca Lanzi, and Mika Klemettinen, editors, *Database*

- Support for Data Mining Applications*, volume 2682 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2004. ISBN 3-540-22479-3. URL <http://dblp.uni-trier.de/db/conf/cinq/cinq2004.html#Rantzau04>.
- [24] Wikipedia. User-defined function — wikipedia, the free encyclopedia, 2014. URL [http://en.wikipedia.org/w/index.php?title=User-defined\\_function&oldid=601426234](http://en.wikipedia.org/w/index.php?title=User-defined_function&oldid=601426234). [Online; accessed 16-June-2014].
- [25] Wikipedia. Stored procedure — wikipedia, the free encyclopedia, 2014. URL [http://en.wikipedia.org/w/index.php?title=Stored\\_procedure&oldid=611940771](http://en.wikipedia.org/w/index.php?title=Stored_procedure&oldid=611940771). [Online; accessed 16-June-2014].
- [26] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. Web-docs: a real-life huge transactional dataset. In Roberto J. Bayardo Jr., Bart Goethals, and Mohammed Javeed Zaki, editors, *FIMI*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004. URL <http://dblp.uni-trier.de/db/conf/fimi/fimi2004.html#Lucchese0PS04>.
- [27] MonetDB. Recipes book. 2014. URL <http://www.monetdb.org/Documentation/RecipesBook>.