Robust and Distributed Top-N Frequent-Pattern Mining With SAP BW Accelerator

Thomas Legler
SAP AG
69190 Walldorf, Germany
thomas.legler@sap.com

Wolfgang Lehner
Technische Universität Dresden
01307 Dresden, Germany
wolfgang.lehner@tu-dresden.de

Jan Schaffner
Jens Krüger
Hasso-Plattner-Institut
14482 Potsdam, Germany
{first}.{last}@sap.com

ABSTRACT

Mining for association rules and frequent patterns is a central activity in data mining. However, most existing algorithms are only moderately suitable for real-world scenarios. Most strategies use parameters like minimum support, for which it can be very difficult to define a suitable value for unknown datasets. Since most untrained users are unable or unwilling to set such technical parameters, we address the problem of replacing the minimum-support parameter with top-n strategies. In our paper, we start by extending a top-n implementation of the ECLAT algorithm to improve its performance by using heuristic search strategy optimizations. Also, real-world datasets are often distributed and modern database architectures are switching from expensive SMPs to cheaper shared-nothing blade servers. Thus, most mining queries require distribution handling. Since partitioning can be forced by user-defined semantics, it is often forbidden to transform the data. Therefore, we developed an adaptive top-n frequent-pattern mining algorithm that simplifies the mining process on real distributions by relaxing some requirements on the results. We first combine the PARTITION and the TPUT algorithms to handle distributed top-n frequent-pattern mining. Then, we extend this new algorithm for distributions with real-world data characteristics. For frequent-pattern mining algorithms, equal distributions are important conditions, and tiny partitions can cause performance bottlenecks. Hence, we implemented an approach called MAST that defines a minimum absolute-support threshold. MAST prunes patterns with low chances of reaching the global top-n result set and high computing costs. In total, our approach simplifies the process of frequent-pattern mining for real customer scenarios and data sets. This may make frequent-pattern mining accessible for very new user groups. Finally, we present results of our algorithms when run on the SAP NetWeaver BW Acceleratorwith standard and real business datasets.

VLDB '09, August 24-28, 2009, Lyon, France

1. INTRODUCTION

The importance of data mining is widely acknowledged today. Mining for association rules and frequent patterns is a central activity in data mining [21]. Three main strategies are available for such mining: *APRIORI* [1], FP-tree-based approaches like FP-GROWTH [8, 15], and algorithms based on vertical data structures and depth-first mining strategies like *ECLAT* and *CHARM* [22, 23, 24].

Unfortunately, most of these algorithms are only moderately suitable for many "real-world" scenarios because their usability and the special characteristics of the data are two aspects of practical association rule mining that require further work.

1.1 Expert vs. Non-Expert

All mining strategies for frequent patterns use a parameter called *minimum support* to define a minimum occurrence frequency for searched patterns. This parameter cuts down the number of patterns searched to improve the relevance of the results. In complex business scenarios, it can be difficult and expensive to define a suitable value for the minimum support because it depends strongly on the particular datasets. Users are often unable to set this parameter for unknown datasets, and unsuitable minimum-support values can extract millions of frequent patterns and generate enormous runtimes. For this reason, it is not feasible to permit ad-hoc data mining by unskilled users. Such users do not have the knowledge and time to define suitable parameters by trial-and-error procedures. Our discussions with users of SAP software have revealed great interest in the results of association-rule mining techniques, but most of these users are unable or unwilling to set very technical parameters. Given such user constraints, several studies have addressed the problem of replacing the minimum-support parameter with more intuitive top-n strategies [7, 9, 10, 14].

We have developed an adaptive mining algorithm to give untrained SAP users a tool to analyze their data easily without the need for elaborate data preparation and parameter determination. Previously implemented approaches of distributed frequent-pattern mining were expensive and time-consuming tasks for specialists. In contrast, we propose a method to accelerate and simplify the mining process by using top-n strategies and relaxing some requirements on the results, such as completeness. Unlike such data approximation techniques as sampling, our algorithm always returns exact frequency counts. The only drawback is that the re-

sult set may fail to include some of the patterns up to a specific frequency threshold.

1.2 Research vs. Business

Another aspect of real-world datasets is the fact that they are often partitioned for shared-nothing architectures, following business-specific parameters like location, fiscal year, or branch office. Users may also want to conduct mining operations spanning data from different partners, even if the local data from the respective partners cannot be integrated at a single location for data security reasons or due to their large volume.

Almost every data mining solution is constrained by the need to hide complexity. As far as possible, the solution should offer a simple user interface that hides technical aspects like data distribution and data preparation. Given that SAP users have such simplicity and distribution requirements, we have developed an adaptive mining algorithm to give unskilled SAP users a tool to analyze their data easily, without the need for complex data preparation or consolidation.

For example, SAP NetWeaver Business Intelligence scenarios often partition large data volumes by fiscal year to enable efficient optimizations for the data used in actual workloads. For most mining queries, more than one data partition is of interest, and therefore, distribution handling that leaves the data unaffected is necessary.

The algorithms presented in this paper have been developed to work with data stored in the SAP NetWeaver BW Accelerator. A salient feature of the SAP BW Accelerator is that it is implemented as a distributed landscape that sits on top of a large number of shared-nothing blade servers. Its main task is to execute OLAP queries that require fast aggregation of many millions of rows of data. Therefore, the distribution of data over the dedicated storage is optimized for such workloads. Data mining scenarios use the same data from storage, but reporting takes precedence over data mining, and hence, the data cannot be redistributed without massive costs. Distribution by special data semantics or user-defined selections can produce many partitions and very different partition sizes. The handling of such real-world distributions for frequent-pattern mining is an important task, but it conflicts with the requirement of balanced partition sizes [21]. To the best of our knowledge, no algorithm is currently available for efficient and robust top-n frequent-pattern mining on uneven distributed datasets.

1.3 Structure

The remainder of the paper is organized as follows. Section 2 introduces some mining algorithms and extends them heuristically to improve their performance. That section focuses on top-n algorithms for *ECLAT*-based solutions.

Section 3 considers the goal of handling distributed data. In general, business datasets may be unevenly distributed, but initially, we focus on equally distributed datasets. We combine two well-known algorithms: the *PARTITION* algorithm [17] for distributed frequent-pattern mining and the *TPUT* algorithm [3] for distributed top-n aggregations.

Section 4 considers real-world distributions with different partition sizes. The runtime of regular minimum-support-based frequent-pattern mining algorithms increases dramatically for small absolute minimum-support values. In particular, the small partitions can be the bottleneck for unevenly distributed datasets. One way to handle low absolute support values is to define a minimum absolute-support threshold. By pruning local patterns with small absolute-support values, we can make a trade-off between runtime and result quality. We implemented this approach as the Minimum Absolute Support Threshold (*MAST*) algorithm.

Section 5 evaluates the results of running our algorithms on the SAP NetWeaver BW Accelerator. We used a mixture of well-known artificial and real-world datasets in different distributions and partitionings. Section 7 summarizes our results and concludes the paper.

1.4 Formal Preliminaries

To complete the preliminaries, we define the basic concepts and terminology for the study of top-n patterns.

Let $I = \{i_1, i_2, \ldots, i_g\}$ be a set of items. An itemset or pattern P is a non-empty subset of I. The length of itemset P is the number of items contained in P. P is called an l-itemset or l-pattern if its length is l.

A tuple (tid, P) is called a transaction, where tid is a transaction identifier and P is a pattern. A transaction database DB is a set of transactions. A pattern P is contained in transaction (tid, Y) if $P \subseteq Y$.

Given a transaction database DB, the support of P, denoted as $support(P)$, is the number of transactions in DB that contain P. The minimum support threshold for P to become part of the result set is called $minSupport$.

A pattern P is a top-n frequent itemset if there exist no more than $n-1$ patterns whose support is higher than that of P. Patterns with the same support as the n-th pattern are top-n results, too.

2. SIMPLE MINING PARAMETERS

In this section, we focus on efficient and simple frequent-pattern mining techniques. We start by introducing a top-n version of ECLAT. Then, we extend this algorithm heuristically to improve its search strategy and performance.

The problem of mining for the top-n frequent patterns is a well-studied area of research [7, 9, 10, 14, 21]. There are solutions for most of the common association-rule mining algorithms. In this paper, we focus on a top-n algorithm for *ECLAT*-based solutions. Like the classic *APRIORI*-based approaches, *ECLAT* uses the well-known downward closure property: the support of each subset of a pattern has at least the support of the pattern itself. Thus, if a pattern is frequent, each of its subsets is frequent as well.

2.1 Problem Statement

In contrast to breadth-first algorithms like *APRIORI*, the *ECLAT* algorithm works in a depth-first way. The algorithm starts with 1-patterns and adds items recursively until the minimum support is no longer reached. The function *Support*() returns the frequency of patterns in the database. These support values can be determined for all subnodes in one single scan per node. This algorithm is very efficient for regular mining calls using a user-defined minimum support threshold. Top-n minings work in a different way. Such an implementation lacks efficiency if the minimum-support value, which increases from 1 to the final top-n support during the algorithm's runtime, grows slowly. In that case, a lot of top-n candidates are identified and stored but only a fraction reach the final top-n result set. This can dramatically increase the storage and calculation effort.

TID	Items
T01	a b c d
T02	c
T03	d
T04	d
T05	d
T06	d
T07	b c d
T08	d
T09	a b c
T10	a b c
T11	a b c d
T12	c

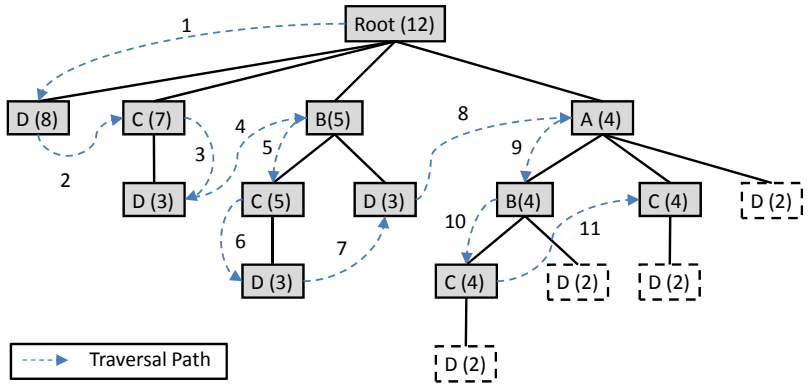


Figure 1: Example and processing of ECLAT

Example:
The following example demonstrates how the top-n *ECLAT* algorithm works. We wish to determine the top five patterns for the dataset shown in Figure 1 in tabular format and visualized as a pattern tree. The dotted line indicates how the algorithm traverses the pattern tree. The numbers in brackets represent the transaction count. For each node, the list of valid transactions can be used to drill down efficiently to its child nodes because the transaction list of each node is a subset of the parent's transaction list. Note that it is difficult to return to already visited and processed nodes because the required information to count the support of such nodes efficiently has already been removed.

Step	Top-List	minSupport
1	[**D:8**]	1
2	[**D:8, C:7**]	1
3	[**D:8, C:7, CD:3**]	1
4	[**D:8, C:7, B:5, CD:3**]	1
5	[**D:8, C:7, B:5, BC:5,CD:3**	3
6	[**D:8, C:7, B:5, BC:5, CD:3, BCD:3**]	3
7	[**D:8, C:7, B:5, BC:5, CD:3, BCD:3, BD:3**]	3
8	[**D:8, C:7, B:5, BC:5, A:4**, CD:3, BCD:3]	4
9	[**D:8, C:7, B:5, BC:5, A:4, AB:4**, CD:3, BCD:3]	4
10	[**D:8, C:7, B:5, BC:5, A:4, AB:4, ABC:4**, CD:3, BCD:3]	4
11	[**D:8, C:7, B:5, BC:5, A:4, AB:4, ABC:4, AC:4**, CD:3, BCD:3]	4

Table 1: Processing example

Table 1 and Figure 1 together show how to find the top-5 frequent patterns. Starting from the root node, the pattern tree is processed from left to right, following the dotted line. The number of the line represents the steps in Table 1. Patterns with a level of support lower than the minimum are skipped, and patterns with the same support as the n-th pattern reach the top-n result. The bold entries in Table 1 represent the current top-n patterns for each step. The other patterns are top-n candidates pushed out of the top-n set. One can easily see that two patterns (CD:3, BCD:3) have been stored as candidates without reaching the final top-n result set. Unfortunately, real datasets generate much bigger overheads.

2.2 Idea Overview

One method to avoid mining unnecessary patterns has been proposed in [9]. The basic idea is to start with 1-itemsets and move on to the next-level child nodes of the currently most frequent unexpanded top-n node. The algorithm stops when the child nodes of all active top-n nodes have been explored. That is possible because each pattern depends only on its parent nodes and not on parallel branches. This algorithm is efficient and fast for small datasets and/or small n. For large n, high memory consumption becomes a problem because each unexpanded node has to store all its transaction IDs to enable the steps down to the child nodes efficiently. Although this structure can be held in a compressed form, such as a sparse array or a bit vector, the memory footprint can still be large for real-world scenarios. For typical business datasets and customer requirements with millions of transactions and n set to $10,000$ or more, such a solution becomes very expensive. However, it is still important to find an intelligent way of traversing the pattern tree efficiently. A solution based on level-wise search strategies like APRIORI, called MTK, has been introduced by Chuang et.al. in [6]. MTK splits the search space into partitions with user-defined memory constraints to minimize the number of required database scans.

We propose an algorithm called *FASTINC* to solve this problem. *FASTINC* uses a heuristic branch-and-bound [12] premining step to define a lower bound lb for the final n-th pattern support value and for performing the common mining call using $minSupport = lb$ instead of $minSupport = 1$. The increased initial minimum support prevents the examination of sub-branches that have no chance of reaching the final top-n set. Since the premining step works approximately, the result quality after this step is poor and this intermediate result does not fit most user requirements. Nevertheless, the support of the n-th pattern is always lower and already close to the final n-th support. The mining step of *FASTINC* can use the intermediate n-th support to increase the initial minimum support. This optimization

substantially reduces the number of temporarily generated, stored, and replaced patterns. The runtime of the premining and mining steps is better than regular one-step top-n *ECLAT* processing.

Since the second mining step is a regular top-n mining with increased initial minimum support, we focus on the premining step and on how to find an adequate threshold.

2.3 Determine the Lower Bound

Figure 1 illustrates that the support value for specific patterns shrinks with increasing level in the pattern tree. It is easy to see that top-n patterns are mostly close to the root node. We use a common mining call with some very rough restrictions as a premining step to find patterns with a better chance of reaching the final top-n set. In that step, on the one hand, the final top-n patterns could be missed, but on the other hand, the support values for the explored patterns are correct. Therefore, the support value of the n-th premining pattern can represent the lower bound lb of the final top-n.

Patterns on higher levels can reach better support values than patterns on deeper levels on other branches, like *BC:5* and *A:4*. For such behavior, the heuristic mining problem is reduced to determine the relevance of each node for the final top-n result set. This problem is hard to solve in an efficient way because branches on the right of the current branch are unknown, and it is impossible to return to previous branches without storing immediate results. We propose some heuristics to stop a branch's exploration in order to proceed with its neighbor. Strategies for when to stop the recursion include:

1. if current pattern length $\geq$ predefined maximum pattern length m,
2. if current support $\leq \gamma * support(Y)$, for $0 \leq \gamma \leq 1$ and the next relevant item Y,
3. if current pattern does not reach at least rank $n * \beta$, for a defined $0 \leq \beta \leq 1$.

All conditions rely on the fact that for frequent-pattern trees, the probability of finding top-n values diminishes with an increasing level in the tree and with decreasing frequency of single items (or from left to right in the tree).

1. The first condition proposes a static way of focusing on the most frequent patterns. Such an implementation stops examining a pattern tree branch at a user-defined maximum level in the pattern tree. This is a robust way to control the algorithm runtime on most datasets, and it is fast and easy to implement because there is no need for dynamic or complex decisions. The maximum-pattern-length parameter m can also be defined dynamically for each mining call. The effect of parameter m depends on the number of distinct values and n. Due to specific data characteristics and data dependencies, such estimations fail in most cases. Experiments show that a fixed $m = 2$ is a reasonable value for most real-world business data because of usually low average transaction sizes in retail datasets.
2. The second condition is more dynamic and depends on the dataset. Each branch is examined until the support value is smaller than the support of the next single item on the next level. Thus, a stored pattern is very likely to be more frequent than many patterns examined on upcoming branches, starting from the root node. This provides a very rough pruning that discards many potential final top-n patterns. This intermediate result produces a low lb value and therefore results in bad performance.
3. The third condition is based on the heuristic that a final top-n pattern reaches a higher ranking than other patterns at insertion time. Patterns at the end of the ranking have just a small chance of staying in the top-n set, and therefore, the algorithm skips them by default. This condition's behavior is similar to that of the second condition, and it depends on β. The correct definition of β is difficult. Thus, this condition should not be used.

This premining step could be iterated multiple times to raise the final minimum support level close to the support of the final n-th pattern. Experiments show that the benefit of performing the premining step multiple times shrinks significantly with the number of runs. For regular datasets, the lower bound lb after a premining down to 2 patterns reaches 75%...90% of the final minimum support. Increasing the premining depth to 3 patterns lifts the lower bound to 80%...100% and nearly doubles the runtime.

2.4 Summary

In this paper, we address the problem of parameter reduction and tuning for top-n frequent-pattern mining. Based on an implementation of *ECLAT*, we propose an algorithm called *FASTINC* to improve the mining process by estimating the result and using this information to speed up the mining process. In the next section, we will focus on the efficient handling of real-world data distributions in combination with top-n frequent-pattern minings.

3. DISTRIBUTED TOP-N ASSOCIATION RULE MINING

In addition to our goal of improving top-n data mining by replacing user-unfriendly parameters with more intuitive ones, we also want to handle real-life distributed datasets efficiently, if only to make full use of the distributed architecture of the SAP NetWeaver BW Accelerator. First, we introduce the handling of equally distributed datasets, and in the subsequent section, we extend this algorithm to work with real-life, unevenly distributed SAP datasets.

Naive solutions handle complex operations on distributed datasets by merging partitions or redistributing the data. We are unable to use such techniques because the distribution is part of the given semantics in many of our real-world scenarios (e.g., a global star schema is often partitioned into multiple star schemas by fiscal year) or because either data security or resource bottlenecks preclude any modifications of the physical design or the replication of the datasets.

3.1 Basic Strategy

A *prima facie* attractive adaptation of common top-n algorithms for distributed landscapes is to run local top-n calculations on each partition and to combine these partial results to a final global top-n set. However, most of the top-n algorithms cannot be used that way without making

generally unwarranted assumptions about the characteristics of the data and its distribution. Such algorithms fail or block the landscape for bad conditions. A robust solution should be able to handle that.

A problem occurs for distributed data with distinct top-n results on each partition. If a final top-n pattern is frequent but never reaches a local result set, this pattern will be globally missed. For example, the top-3 bestsellers for an apparel manufacturer may be completely different from season to season, but the fourth bestseller could be the same every time. For the complete year, it may be that *underwear* is the global number-one bestseller, but it may miss the global top-3.

To attack this problem, we extend the known algorithm *PARTITION* [17] with functionality of the distributed top-n aggregation algorithm *TPUT* [3]. The local mining part on each partition is independent from a specific frequent-pattern mining algorithm. Therefore, we use our *FASTINC* algorithm. Obviously, *FP-GROWTH*, *APRIORI* etc. are also possible.

The *PARTITION* algorithm (for patterns based on minimum support) works as follows:

1. Determine all patterns on all nodes m with the defined minimum support.
2. Send all intermediate results $I_{1,\ldots,m}$ to a control operator to merge the results.
3. Since each global frequent pattern must be frequent on at least one partition, the final result must be a subset of the merged patterns $M = \bigcup I_{1,\ldots,m}$ (maybe some partial counts are still missing).
4. Search on all nodes m for patterns $P \in M, P \notin I_m$.
5. Merge again, filter patterns with valid global minimum support, and return results.

The *TPUT* algorithm goes like this:

1. Use a top-n pre-aggregation step to determine a lower bound lb for the global top-n result values.
2. Search all local results for all values x, possibly $aggregation(x) \geq lb$.
3. Merge intermediate results.
4. Determine missing local aggregations.
5. Merge again and resolve top-n results.

The steps of these two algorithms are very similar and it seems surprisingly easy to combine the distributed frequent-pattern mining idea of *PARTITION* with the distributed top-n handling of *TPUT*. However, there are important differences in the behavior of aggregation and frequent-pattern mining. The first and the second step of *TPUT* are fast and efficient for aggregations because aggregating twice to find distributed top-n results is acceptable. This holds especially for *TPUT*, where aggregation costs are very small in contrast to communication costs. Mining for frequent patterns twice, first for local top-n patterns to define lb and again with $minSupport = lb$, intolerably increases the overall runtime in many scenarios. Moreover, on some datasets, a low threshold lb can produce many more than just n results for both aggregation and frequent-pattern mining. The consequences for aggregation are large intermediate result sets and thereby higher network traffic and merge efforts. Frequent-pattern mining generates more partial results, too, but requires much more effort to determine the local result sets; and it may even fail completely. Therefore, a naive adaptation of *TPUT* for distributed frequent-pattern mining is possible in principle but problematic in practice because the runtime is hard to predict and the usability is poor. Thus, an adaptation of *TPUT* for distributed top-n frequent-pattern mining is not reasonable for many scenarios. To define a baseline for the evaluation, we refer to this less robust algorithm as *TPARTITION*.

3.2 Improved Strategy

In contrast to aggregations, frequent-pattern mining is usually a task that does not require completeness. The majority of frequent-pattern mining tasks in real-life scenarios are slightly unspecific by nature. To find the top-10 most important pieces of information is not necessary if the extracted information overall is still very interesting for the user. For these reasons, we propose an efficient and robust algorithm called Save-Threshold (*STH*) to perform stable frequent-pattern mining on real-world distributions. The algorithm produces an approximate result set with exact support values but potentially missed top-n patterns. Such behavior is much more interesting for customers in comparison to synopsis-based approximations like sampling with estimated support values.

Figure 2 shows the exact but less robust algorithm *TPARTITION* for frequent-pattern mining. The *STH* algorithm, as part of *TPARTITION*, is used to define a lower bound for searching global top-n results without losing results. Using both phases, the algorithm determines the exact top-n patterns. Using just *STH*, it returns an approximate top-n result. Note that *STH* is only responsible for half of the runtime. As the evaluation will show, running the full *TPARTITION* is much more expensive.

Starting with *STH* and using the local top-n minings to build the global top-n patterns still generates a large fraction of the exact final top-n patterns for most datasets and data distributions. Due to these cost savings, the use of only the robust *STH* instead of the full *TPARTITION* seems reasonable. There is high probability, but no specific probability bound, for a global top-n pattern to reach at least one local top-n set. Such patterns will be found by *STH*, and missed counts on some partitions after phase $S1$ will then be determined in phase $S2$. Therefore, it is impossible to get an incorrect final support count, and all support values in the top-n result set of *STH* are correct. Only patterns missed on each partition may miss the global top-n result set and patterns beyond the global top-n will be returned instead. Nevertheless, it is possible to define a threshold to divide the result set into guaranteed and non-guaranteed top-n patterns. In the next section, we will consider how to define an upper bound for the support value of a possibly missed top-n pattern.

3.3 Computing the Save Threshold of STH

Since the final support values are exact, a specific maximum support value s of missed patterns can be determined. Therefore, all patterns P with $support(P) > s$ are guaran-

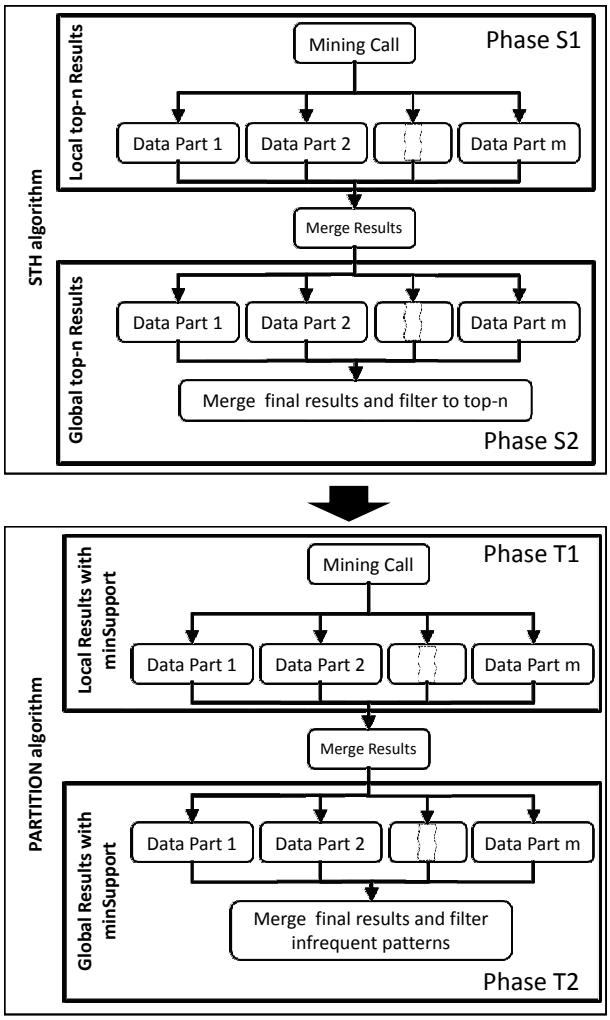


Figure 2: Distributed top-n frequent-pattern mining with TPARTITION

teed to have the right rank in the top-n result list because no missed pattern can reach a higher position. The same argument also shows that all patterns P with $support(P) > s$ are found by the algorithm.

The question is how to define s. Let W be the most frequent globally missed pattern. The only way W can fail to reach the global top-n is by missing the local top-n on each partition. Let $X_{1\ldots m}$ be the n-th result pattern on each partition $1 \ldots m$. The maximum support value ms that W can reach is

$$ms(W) = \sum_{i=1}^{m} (support(L_i) - 1) = s$$

.

In this case, all final patterns P with $support(P) > s$ can be guaranteed because they have both the right rank and correct support values. Tests with artificial and real data and different distribution strategies suggest that $ms(W)$ values are close to the support of the n-th rank pattern, so most of the results are guaranteed. By mining the local top-$(n * \alpha)$ patterns (for $\alpha > 1$) and returning just the global top-n, the save threshold can be lowered down to the support of the global n-th pattern without massive processing overhead.

TID	Items
T01	a b c d
T02	c
T03	d
T04	d
T05	d
T06	d

TID	Items
T07	b c d
T08	d
T09	a b c
T10	a b c
T11	a b c d
T12	c

Table 2: A partitioned transaction database DDB

Guarantees to find the completely correct results by choosing a specific α are not possible.

Example:
Within this example, the task is to mine for the top-2 frequent patterns of the distributed dataset in Table 2.

1. Running a top-2 mining on each partition returns $I_1 = [D : 5, C : 2]$ and $I_2 = [C : 5, B : 4]$.
2. Merging local results builds $M = I_1 \cup I_2 = [C : 7, D : 5, B : 4]$.
3. Missed support values are support for pattern B on the first data partition and support for pattern D on the second one.
4. The smallest local supports are 2 and 4, so the save threshold is $s = (2 - 1) + (4 - 1) = 4$.
5. Merging formerly missed and already known support values (from steps 2 and 3) produces
   $M = [C : 7, D : 5, B : 4] \cup [D : 3] \cup [B : 1]$
   $M = [\mathbf{D:8,\ C:7}, B : 5]$.
6. The final result for the global top 2 is $M = [D : 8, C : 7]$.
7. Since $s = 4$ and for all $P \in M, support(P) > s$, the global result set is complete.

The worst case of this algorithm appears in datasets with almost disjoint data partitions, with local but not global frequent patterns, and in those where most of the real global top-n patterns miss all local top-n result sets. In this case, many of the missed patterns could be locally sparse in any location. These patterns can still be more frequent globally than the algorithmically selected global top-n patterns, which renders the whole result set uncertain. However, in most real-world scenarios, there are globally applicable semantics for all the data partitions, so many frequent patterns appear globally. From this point of view, extremely disjoint local result sets probably indicate better results by individual pattern minings for each partition.

4. HANDLING OF REAL-WORLD DATA DISTRIBUTIONS

In contrast to most artificially generated datasets, real-world distributions often have different partition sizes, tiny partitions, and uneven data characteristics. Distributions based on business-related characteristics, such as time or country, can differ in size by orders of magnitude. Even in the case of equally distributed datasets, a reasonable business query can

select a very unevenly distributed subset of the data. Discussions with SAP customers show that most of their mining calls target such uneven distributions. Unfortunately, equal partition sizes are an important prerequisite for most distributed frequent-pattern mining algorithms [21].

We start by describing the problem of very different partition sizes with regular minimum-support-based frequent-pattern mining, and we use the example of the *PARTITION* algorithm. We propose a solution for algorithms based on minimum support and top-n. The standard *PARTITION* algorithm mines for locally frequent patterns on each partition with the same relative minimum support $minSupport$ as defined for the global dataset. For example, a global mining call to determine frequent patterns P with $support(P) \geq 20\%$ will search on each partition for frequent patterns reaching a local 20% support value [17].

The runtime of frequent-pattern mining increases dramatically for small absolute minimum-support values due to the exponential complexity of such algorithms [21]. If $|DB|$ is the number of used transactions, we define the *absolute minimum support* as

$$minSupportAbs = \lceil minSupport * |DB| \rceil$$

The mining process uses the absolute minimum support. For example, $minSupport = 10\%$ for a dataset with 50 transactions means $minSupportAbs = 5$ and patterns with a frequency of 5 or more will be found. Setting $minSupport = 1\%$ will produce $minSupportAbs = 1$. In this case, the algorithm will declare, store, and return all possible patterns as frequent. The consequence is a complex and expensive operation because the runtime scales up exponentially with shrinking $minSupport$ but improves only linearly with a reduction of $|DB|$ [21]. The same parameters for a partition with $10,000$ transactions give $minSupportAbs = 100$, and only a small fraction of all patterns will reach this frequency. Therefore, a mining process on a small dataset is often much slower than one with the same relative $minSupport$ but $minSupportAbs \gg 1$ on larger datasets. For unevenly distributed datasets, surprisingly, it is the small partitions that can be the bottleneck.

An approach to handle such unintended behavior is required, and for usability reasons, it should be automated. One way to handle small partitions is a minimum threshold for $|DB|$. All data partitions that do not reach this minimum number of transactions return an empty result set by default. Otherwise, the calculation of the final frequent-pattern support values can occur as before. Final result sets built like this can miss patterns that are only seen in small data partitions and not in large ones. Such scenarios are possible but improbable because global frequent-pattern supports are mostly dominated by the large partitions. The final support values are still correct because all patterns found in larger partitions will be determined after the first merge step for both small and large partitions. Experiments show mostly correct results, but only very weak guarantees are possible for this approximation.

Another way to handle low absolute support values is to define a minimum absolute support threshold. Most of the local top-n results on small partitions with an absolute support value close to 1 are not included in the local results on larger partitions. Therefore, they will not significantly influence the final result set. Therefore, by pruning patterns with small local absolute support values, we should be able to achieve an efficient trade-off between runtime and result quality. We implemented an additional parameter t to define a lower bound for the absolute minimum support value on each partition. If the regular absolute minimum support is below this threshold, then

$$minSupportAbs = \max(\lceil minSupport * |DB| \rceil, t).$$

The optimal value for t depends on many characteristics, such as the global transaction count, data dependencies, and more. Nevertheless, for most scenarios, a default value of $t = 2$ reduces the risk of excessive runtimes and an explosion of local result sets for small partitions. We found this approach attractive and we implemented it as the *Minimum Absolute Support Threshold* ($MAST$) algorithm. To adapt this kind of risk handling for distributed top-n frequent-pattern mining, the local mining calls have to start with the setting $minSupportAbs = t$ on each partition (instead of $minSupportAbs = 1$).

In our experience, this optimization is useful for many real-world scenarios. Nevertheless, without knowing anything about the patterns we may have missed, we cannot make any judgment on the final result quality. By closing the gap between the STH approach and the handling of real-world data distributions, the $MAST$ approach can address the quality issue caused by the increased local minimum support values within the STH algorithm. With the same arguments as in Section 3.3, the most frequent pattern P missed on all partitions $1 \ldots m$ can reach

$$support(P) \leq \sum_{j=1}^{m} (minSupportAbs_j - 1) = b.$$

As with the save threshold s, it is possible to use regular minimum-support-based and top-n algorithms to define an upper bound b for the maximum support of a missed pattern. Therefore, all patterns $P \in M$ with $support(P) > b$ are found, correct, and guaranteed. For real datasets with many small partitions, b gives uncertain results because b increases with the number of increased local supports to t. Thus, datasets with many small partitions using t and only a few very large partitions produce a high value for b. This procedure creates result sets with small global absolute support values and mostly non-guaranteed top-n results. However, experiments with real-world datasets and distributions show that in most such cases the global result is dominated by the big partitions. Although the results are not guaranteed, our evaluation will show that they are still very close to being correct.

5. EVALUATION

We implemented our algorithms within the SAP NetWeaver BW Acceleratorand exploited its existing data structures and system resources. Due to limited space, we are not able to give more technical details in this paper. Please, have a look at [13, 16] for details and further information. Our experiments used a blade-server architecture with up to three blades and four CPUs with 2.6 GHz per blade. Each blade had 8 GB RAM and ran Microsoft Windows Server 2003 Enterprise x64.

5.1 Test Data

Table 3 describes the experimental datasets. We used a mixture of well-known artificial and real-world datasets. The

Name	Type	Partitions	Transactions	Avg. Trans. Length	Distinct Items
SynthA	artificial/open	1, 2, 4, 6, 8, 10, 40	800 000	19.9	772
SynthB	artificial/open	1	980 000	10.2	24 000
Retail	real/open	1, 2	85 146	9.6	16 398
CustomerA	real/closed	1, 2, 4, 10, 22	134 167	3.0	72 252
CustomerB	real/closed	40	33 542 000	3.3	72 025

Table 3: Dataset characteristics

Top-100	Parts	lb	Support of n-th Pattern	Uncertain	Wrong	α For Save Top-100
	1		17 361	0	0	1.00
	2	17 378	17 361	1	0	1.03
	4	17 446	17 361	6	0	1.03
	6	17 426	17 361	5	0	1.05
	8	17 420	17 361	5	0	1.03
	10	17 398	17 361	3	0	1.03
	40	17 510	17 361	11	0	1.07
Top-1000						**Top-1000**
	1		9 227	0	0	1.00
	2	9 213	9 227	0	0	1.00
	4	9 228	9 227	1	0	1.01
	6	9 232	9 227	2	0	1 02
	8	9 233	9 227	2	0	1.02
	10	9 232	9 227	2	0	1.01
	40	9 249	9 227	4	0	1.06
Top-10000						**Top-10000**
	1		2 638	0	0	1.00
	2	2 636	2 638	0	0	1.00
	4	2 633	2 638	0	0	1.00
	6	2 634	2 638	0	0	1.00
	8	2 632	2 638	0	0	1.00
	10	2 631	2 638	0	0	1.00
	40	2 623	2 638	0	0	1.00

Table 4: Scaling of *STH's* lb for evenly distributed artificial *SynthA* dataset

SynthA dataset is generated by a modification of the well-known IBM data generator [11]. As a real and public dataset, we tested our approach with the open *Retail* dataset [2]. The distributed version of *Retail* is divided into two parts storing short and long transactions to show scenarios with uneven data characteristics.

Given the commercial context of this work, we were primarily interested in achieving good performance for typical SAP business data; thus, for our tests, we used real business datasets called *CustomerA* and *CustomerB* in different distributions and partitionings. The customer datasets are different to the other dataset with respect to average transaction length and number of distinct items. These characteristics cause a high number of short itemsets. The partitioning is fixed for each dataset for all evaluations. Usually, the support of determined patterns is quite low ($< 0.1\%$) because of weak (but real) dependencies. Thus, using this data, the support of frequent patterns is usually not significantly higher than that of unfrequent patterns. For *Save-Threshold* experiments, all datasets were used with a different number of partitions, distributed by transaction ID. All datasets were stored in a vertical data format and each transaction item used one table row.

Only patterns with two or more items are able to produce association rules. Therefore, top-n mining means mining n patterns with length ≥ 2 in our evaluation. And hence, top-n mining can return more than n results because any pattern of size 1 with higher support than the n-th pattern becomes part of the top-n result.

5.2 Results for FASTINC

The first evaluation shows centralized top-n frequent-pattern mining and the effect of premining on the definition of a lower bound for the n-th pattern support. Figures 3(a) and 3(b) illustrate the effect of premining and the maximum premining level used. The results for *Basic* represent the standard usage without premining, those for *FASTINC-2* show premined patterns with a maximum length of two items, and those for *FASTINC-3* show premined patterns with a maximum length of three. All these experiments use unpartitioned datasets.

The effect of the *FASTINC* parameter depends strongly on the underlying dataset and configuration. For the real business data shown in Figures 3(a) and 3(b), the premi-

Top-n	Parts	lb	Support of n-th Pattern	Uncertain	Overall Patterns	Wrong
100	22	45	24	87	104	0
250	22	37	17	223	256	0
1.000	22	27	8	1.148	1.223	0
5.000	22	22	4	5.543	5.768	1
10.000	22	22	3	11.754	12.258	1
100	5	36	24	70	101	0
250	5	20	17	112	253	0
1.000	5	10	8	621	1.222	0
5.000	5	5	4	2508	5.764	1
10.000	5	5	3	9855	12.111	1

Table 5: Unbalanced partitionings on *CustomerA* using $minSupportAbs = 2$ for *MAST*

ning step produces diverse runtimes. For small values of n, the standard mining without premining outperforms the versions using *FASTINC* by a factor of up to 3. Nevertheless, *FASTINC* scales better and outperforms the standard algorithm for larger n. Since our customers set $n > 10000$ to provide a solid basis for further analysis, *FASTINC-2* is a useful optimization for their mining scenarios.

Figure 4 shows the decreased numbers of recursion calls needed to find a top-n result. This number represents the number of evaluated patterns, too. The *TOP-n ECLAT* graph represents a standard TOP-n ECLAT call, *FASTINC-2* shows the corresponding FASTINC-2 call. *Initialscan* presents the number of recursions needed for premining. For this example, *FASTINC-2* already includes the *Initialscan*. The graphs promise a significant speedup, but *FASTINC-2* reduces a lot of recursion with low support. Very frequent and therefore expensive patterns are still part of the result. Implementations with higher costs per recursion call can reach much better optimization. Figure 4(b) shows the number of saved recursion calls for the three different premining strategies. The $\gamma = 40\%$ graph shows the number of saved recursions by storing only patterns reaching the top-$(n * \gamma)$. The $\beta = 20\%$ graph visualizes the effect of using a pattern for premining if its support is higher than $\beta * support(X)$ and X is the next single item. Other values for *gamma* and β show similar scalings. It is easy to see that FASTINC outperforms the strategies using γ or β. If the β-stategy is not able to fill the top-n and lift the support for the second scan, the algorithm performs poorly. It is hard to choose a good value for β, thus using this strategy is not a robust solution.

5.3 Results for STH

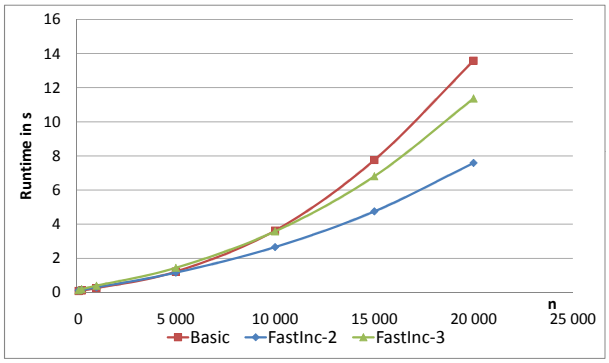
To evaluate distributed top-n mining, we looked at the impact of partitioning, the value of n, data characteristics for the save threshold, the number of uncertain patterns in the top-n list, the absolute *minSupport* value to resolve top-n sets, and the number of incorrect or missed final results. Tables 4, 5, and 6 represent this evaluation for different datasets. One characteristic is the decreasing number of uncertain patterns with increasing n. Another characteristic is a small α factor to reach completely guaranteed results. The speedup factor in Table 6 represents the runtime of *STH* in comparison to a full *TPARTITION*. The minimum support of *TPARTITION* is the value of column *MinSup for TPARTITION*. As assumed, the speedup has approximately a factor of 2 for balanced partitioned datasets. Figure 3(c) shows the runtime scaling for an uneven partition with different data characteristics on each part. It is easy to see that *STH* outperforms the naive integration of TPUT and PARTITION by orders of magnitude for big n.

Note that the save threshold per dataset for each n is largely independent of the distribution. The impact of partitioning is small in relation to the support values. In particular, the number of partitions does not significantly influence lb. The reason for the fluctuation of lb can be found in changing dependencies and data characteristics caused by modified data distributions. Figure 3(d) shows the consequences of increased partitioning for the result quality. The number of uncertain patterns increases with the number of partitions, but it stabilizes above a certain threshold.

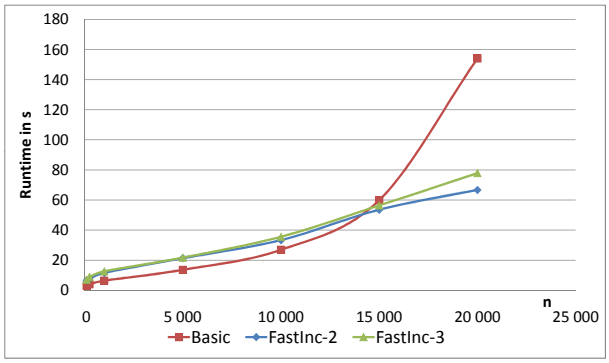
Table 5 shows an evaluation for uneven partition sizes. For this experiment, the *CustomerA* dataset was partitioned by date into 22 parts; 20 of them were very small and 2 of them were large. Regular mining calls cannot handle this distribution efficiently because most of the small partitions are unable to produce local top-n sets with an absolute support > 1. Each evaluation presented in Table 5 runs one to eight seconds. The same configuration without *MAST* using *TPARTITION* was stopped after 20 hours. This represents a speedup factor of at least $9,000$ between MAST and TPARTITION. This behavior is independent of the underlying mining algorithm because the runtime and the number of local results generated on such partitions increase massively. For these experiments, we defined a minimum absolute support of 2 on each partition. With 20 small parts using the lower bound, most final result patterns are uncertain. Nevertheless, most of them are still correct. That is, the large partitions dominate the final result set and the small partitions do not substantially influence the output quality.

6. RELATED WORK

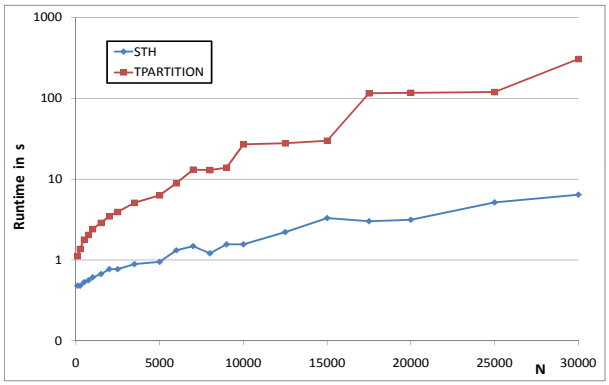
This paper touches on numerous different problem definitions that have been well-studied in many papers in the database community. [5, 18, 21] give an interesting overview including algorithms, complexity, and a problem definition for frequent-pattern mining. There are plenty of similarities between aggregation and frequent-pattern mining, which is reflected in related work. An extension of *PARTITION* [17] for top-n frequent-pattern mining applies a principle close to that of distributed top-n aggregations using *TPUT* [3].



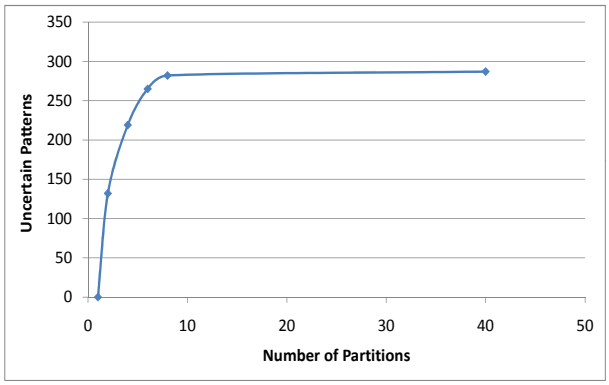
(a) Top-n scaling for *Retail* dataset



(b) Top-n scaling for *CustomerB* dataset



(c) Top-n scaling for partitioned *Retail* dataset



(d) Uncertain patterns (top-1000, *CustomerB)*

Figure 3: Top-n scalings

Top-n patterns with specific characteristics are addressed in [7]. This work considers searching for top-n patterns with a predefined minimum length. Such restrictions are irrelevant for our work because we know from SAP system users that pattern mining in real-world retail datasets tends to focus on small patterns. Therefore, we only need to consider frequent patterns up to a specific size. Furthermore, shorter patterns are used to determine the importance of generated association rules, and so they cannot be pruned. Nevertheless, the main idea represents a solution for efficient top-n pattern mining. The mining engine holds a list of active top-n frequent nodes of the pattern tree and drills down on the most frequent unresolved node for new top-n candidates until all active nodes have been processed. This is efficient in the number of generated patterns, but it conflicts with the principle of using as much information as possible about generated patterns and dropping unused information quickly. In real scenarios, such top-n lists have to hold n-transaction lists for a large number of stored transactions. A similar solution for *CLOSET* can be found in [9].

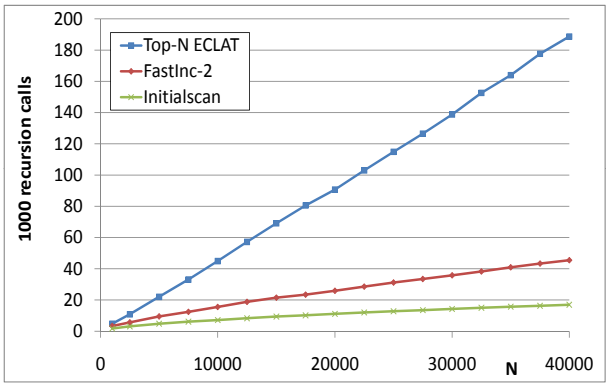
There are a number of distributed mining frameworks, such as [4, 5, 14, 19], and papers on the importance of the semantics of data distributions [20, 25]. A similar solution to the *FASTINC* algorithm proposed in this paper can be found in [10], where - in contrast to our solution - a dynamic threshold t is used to decide between going down and proceeding with the next branch in the pattern tree. Here, the definition and management of t become difficult, which makes the solution weak for real datasets and data mining scenarios.

7. SUMMARY AND CONCLUSION

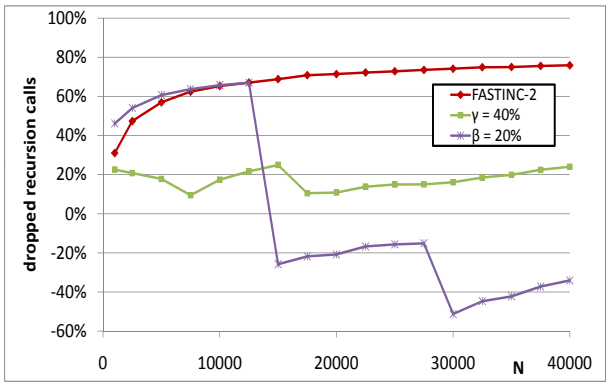
In real-world data mining scenarios, a minimum-support parameter can be difficult to set and the data is often distributed into differently sized partitions. Therefore, we explored the applicability of more intuitive top-n strategies. We developed an adaptive mining algorithm that avoids the need for complex data preparation or prior parameter determination. Our method accelerates and simplifies the mining process by relaxing some requirements on the results, such as completeness. Our algorithm always returns exact frequency counts, but the top-n set may fail to include some of the patterns in the data.

To avoid mining unnecessary patterns, we proposed a premining step to find a lower bound lb for the final n-th pattern support value and performed the mining call using $minAbsSupport = lb$. The increased initial minimum support prevents the examination of sub-branches that have no chance of reaching the final top-n set. We implemented this as the *FASTINC* algorithm. The premining step is heuristic and this result cannot be returned to the user. But a second step with increased initial minimum support value substantially reduces the runtime for both steps together. As premining step, we used a mining call to find patterns with a raised chance of reaching the final top-n set and we proposed three heuristic conditions.

To handle equally distributed datasets, one option is to run local top-n calculations on each partition and to combine these partial results to a final global top-n set. However, this involves making mostly unwarranted assumptions



(a) Recursion calls Top-N ECLAT vs. FASTINC-2



(b) dropped recursion calls

Figure 4: Recursion calls with FASTINC on *SynthB*

Top-n	Partitions	lb	Support of n-th Pattern	MinSup For TPARTITION	Uncertain	Wrong Results	Speedup
100	2	23	24	22.7	0	0	2.10
250	2	16	17	15.1	0	0	2.01
1 000	2	7	8	6.3	0	1 missed	2.15
5 000	2	2	4	1.9	0	0	2.12
10 000	2	2	3	1.9	0	0	2.17
100	4	24	24	21.9	0	0	1.94
250	4	15	17	13.1	0	0	2.10
1 000	4	8	8	7.2	0	15 missed	1.71
5 000	4	2	4	0.0	0	0	2.05
10 000	4	0	3	0.0	0	0	2.17
100	10	22	24	18.1	0	0	2.99
250	10	13	17	9.0	0	0	1.98
1 000	10	9	8	1.0	276	0	2.03
5 000	10	0	4	1.0	0	0	2.02
10 000	10	0	3	1.0	0	0	2.15

Table 6: Behavior of *STH* on real dataset *CustomerA*

about the data. To weaken this problem, we combined two algorithms, *PARTITION* and *TPUT*, and we tried to use the advantages of both. Since frequent-pattern mining does not usually require exact results and completeness, we proposed an algorithm called *Save-Threshold* to perform stable frequent-pattern mining on distributed data and to build an approximate result set with exact support values but potentially missed top-n patterns. We considered how to define a lower bound for the support value of a missed top-n pattern. We defined a specific maximum support value that a missed pattern can reach by noting that the only way the pattern can fail to reach the global top-n is by missing the local top-n on each partition.

Real-world distributions often have very different partition sizes. This causes a problem but we proposed a solution. For unevenly distributed datasets, the small partitions can be a performance bottleneck, since the runtime of frequent-pattern mining increases dramatically for small absolute minimum support values. One way to handle low absolute support values is to define a minimum absolute support threshold. By pruning patterns with small absolute support values, we made a trade-off between runtime and result quality. We implemented a parameter called t to define such a lower bound and found that for most scenarios, a default value of 2 reduces the risk of excessive runtimes and an explosion of local result sets for small partitions. We implemented this approach as the *MAST* algorithm. We defined an upper bound b for the maximum support of a missed pattern in such a way that all patterns with support above b are correct and guaranteed. Since real datasets with many small partitions and only a few very large partitions usually produce a high b, they create result sets with small global absolute support values in the top-n, and most of the results are not guaranteed. However, our tests showed that in most such cases, the global result is dominated by the big partitions and therefore still usable.

We evaluated our algorithms by running them on the SAP NetWeaver BW Accelerator. We used a mixture of artificial and real-world datasets in different partitionings. We tested the effect of premining and of different premining levels. The effect of the *FASTINC* parameter depends strongly on the underlying dataset. In cases with many dependencies and long average transaction times, premining does not significantly raise the lower bound for the n-th pattern support.

For small values of n, standard mining without premining outperforms mining using *FASTINC* by a factor of 1 to 3, but *FASTINC* outperforms the standard algorithm for larger n. In practice, *FASTINC-2* is a useful optimization for many real-world scenarios.

To evaluate distributed top-n mining, we looked at the impact of partitioning, the value of n, data characteristics for the save threshold, the number of uncertain patterns in the top-n list, the absolute *minSupport* value to resolve top-n sets, and on the number of incorrect or missed final results. The more common data distributions only rarely deliver incorrect patterns. The save threshold per dataset for each n is largely independent of the distribution. The impact of partitioning is small in relation to the support values, and the number of partitions does not significantly influence the lower bound. We tested the consequences of partitioning for the result quality. The number of uncertain patterns increases with the number of partitions, but it stabilizes above a certain threshold. We also tested uneven partition sizes. For those, most final result patterns were uncertain but still correct.

Although this paper discusses problems that have been well studied in the community, to our knowledge, the aspects that we considered have not previously been examined in such detail and combination. These aspects are decisive in practice for extracting the full power of real-world frequent-pattern mining.

8. ACKNOWLEDGMENTS

The authors would like to thank the SAP NetWeaver EIM development team TREX for their numerous contributions to the preparation of this paper. Special thanks go to Christian Kuehrt and Andrew Ross for very helpful discussions.

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. of the 20th Intl. Conf. On Very Large Data Bases*, pages 487–499. Morgan Kaufmann, 1994.

[2] T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets. Using association rules for product assortment decisions: A case study. In *Knowledge Discovery and Data Mining*, pages 254–260, 1999.

[3] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *Proc. of Intl. Symposium on Principles Of Distributed Computing*, pages 206–215, 2004.

[4] J. Chattratichat, J. Darlington, Y. Guo, S. Hedvall, M. Köler, and J. Syed. An architecture for distributed enterprise data mining. In *Proc. of the 7th Intl. Conf. On High-Performance Computing And Networking*, pages 573–582, 1999.

[5] D. W.-L. Cheung and Y. Xiao. Effect of data skewness in parallel mining of association rules. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 48–60, 1998.

[6] K.-T. Chuang, J.-L. Huang, and M.-S. Chen. Mining top-k frequent patterns in the presence of the memory constraint. *The VLDB Journal*, 17(5):1321–1344, 2008.

[7] S. Cong. Mining the top-k frequent itemset with minimum length m, 2001.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of the 19th Intl. Conf. on Management of Data*, pages 1–12, 2000.

[9] J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *Proc. of the ICDM'02*, December 2002.

[10] Z. He. Mining top-k approximate frequent patterns. Technical Report TR-2005-0315, 2005.

[11] The IlliMine Project, http://illimine.cs.uiuc.edu.

[12] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[13] T. Legler, W. Lehner, and A. Ross. Data mining with the SAP NetWeaver BI accelerator. In *Proc. of the 32nd Intl. Conf. On Very Large Data Bases*, pages 1059–1068, 2006.

[14] S. Michel, P. Triantafillou, and G. Weikum. KLEE: a framework for distributed top-k query algorithms. In *Proc. of the 31st Intl. Conf. On Very Large Data Bases*, pages 637–648. VLDB Endowment, 2005.

[15] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[16] A. Ross. *SAP NetWeaver BI Accelerator*. Galileo Press, 2009.

[17] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *The VLDB Journal*, pages 432–444, 1995.

[18] D. B. Skillicorn. Parallel frequent set counting. *Parallel Computing*, 28(5):815–825, 2002.

[19] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, pages 648–659, August 2004.

[20] R. Wirth, M. Borth, and J. Hipp. When distribution is part of the semantics: A new problem class for distributed knowledge discovery. In *Proc. of the PKDD 2001 Workshop on Ubiquitous Data Mining for Mobile and Distributed Environments*, pages 56–64, Freiburg, Germany, 2001.

[21] M. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, 1999.

[22] M. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proc. of the 2003 Intl. Conf. on Knowledge Discovery and Data Mining*, pages 326–335, 2003.

[23] M. Zaki and C. Hsiao. Charm: an efficient algorithm for closed association rule mining. Technical report, 1999.

[24] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. Technical Report TR651, 1997.

[25] Y. Zhao, H. Zhang, F. Figueiredo, L. Cao, and C. Zhang. Mining for combined association rules on multiple datasets. In *Proc. of the 2007 Intl. Workshop On Domain Driven Data Mining*, pages 18–23, 2007.