Dieses Dokument ist eine Zweitveröffentlichung (Postprint) /

This is a self-archiving document (accepted version):

Benjamin Schlegel, Tim Kiefer, Thomas Kissinger, Wolfgang Lehner

pcApriori: scalable apriori for multiprocessor systems

Erstveröffentlichung in / First published in:

SSDBM '13: Conference on Scientific and Statistical Database Management, Baltimore 29.07. – 31.07.2013. ACM Digital Library, Art. Nr. 20. ISBN 978-1-4503-1921-8

DOI: https://doi.org/10.1145/2484838.2484879

Diese Version ist verfügbar / This version is available on:

https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-806412

SLUB Wir führen Wissen.

TECHNISCHE UNIVERSITÄT DRESDEN

QUCOSA Quality Content of Saxony

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 20, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484879

pcApriori: Scalable Apriori for Multiprocessor Systems

Benjamin Schlegel, Tim Kiefer, Thomas Kissinger, Wolfgang Lehner
Database Technology Group
TU Dresden
Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

Frequent-itemset mining is an important part of data mining. It is a computational and memory intensive task and has a large number of scientific and statistical application areas. In many of them, the datasets can easily grow up to tens or even several hundred gigabytes of data. Hence, efficient algorithms are required to process such amounts of data. In the recent years, there have been proposed many efficient sequential mining algorithms, which however cannot exploit current and future systems providing large degrees of parallelism. Contrary, the number of parallel frequent-itemset mining algorithms is rather small and most of them do not scale well as the number of threads is largely increased. In this paper, we present a highly-scalable mining algorithm that is based on the well-known Apriori algorithm; it is optimized for processing very large datasets on multiprocessor systems. The key idea of pcApriori is to employ a modified *producer–consumer* processing scheme, which partitions the data during processing and distributes it to the available threads. We conduct many experiments on large datasets. pcApriori scales almost linear on our test system comprising 32 cores.

General Terms

Data mining, Association rule mining

Keywords

Frequent itemset mining, Parallel Apriori

1. INTRODUCTION

Frequent-itemset mining is a popular part of data mining with the goal of finding values or items that co-occur frequently in a dataset. It has many application areas like market-basket analysis, web-mining, gene-expression analysis, mining in astronomy, etc. Frequent-itemset mining is further the foundation of many other pattern mining variants; e.g., sequence mining [2], graph mining [26] or rare itemset mining [18]. Thus, optimizations on the base algorithms can often be applied with minor changes to these extensions.

Frequent-itemset mining can be described as follows: Let $\mathcal{I} = \{ a_1, \ldots, a_m \}$ be a set of *items* and $\mathcal{D} = (T_1, \ldots, T_n)$ be a database of *transactions*, where each transaction $T_i \subseteq \mathcal{I}$ consists of a set of items. The *relative support* of an *itemset* $I \subseteq \mathcal{I}$ denotes the percentage of transactions that contain the itemset I. The goal of itemset mining is to find all itemsets that satisfy a certain *minimum relative support* ξ. The chosen ξ value thereby influences the effort for mining; it becomes more expensive as ξ decreases because more frequent itemsets are found.

There exists a large variety of algorithms tackling the challenge of finding frequent itemsets. Basically,—as discussed by various authors [25, 12, 15]—none of them is superior over all other algorithms; the dataset being mined and the chosen ξ value determine which algorithm performs best. The most popular frequent-itemset mining algorithms are FP-growth [14] and Eclat [28] for which many optimizations and variants were proposed. The Apriori [1] algorithm is often wrongly considered as inferior compared to these mining algorithms. This basically stems from the way how Apriori processes the data. It requires k scans of the base data to obtain size-k frequent itemsets; clearly this is expensive if the scans are performed on disks, however, scanning in main memory is in comparison almost for free. Apriori thus works well on currently available multiprocessor systems because they provide large main memory capacities; e.g., some systems are equipped with 2TB of main memory.

Apriori typically performs better than the other mining algorithms on datasets that have many small transactions (e.g., datasets from retail business) or when the result set contains mainly short and only few long frequent itemsets. There further exist several Apriori variants, which perform differently depending on the used datasets and chosen ξ values. The original Apriori algorithm [1], for example, represents the transaction database using simple lists whereas other Apriori variants [6, 5] use prefix trees for that purpose. The former is usually faster for rather large ξ values since no expensive building of prefix trees is required. Contrary, the latter benefit from the sorted transactions when the ξ values are rather small.

©2013 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *SSDBM* '13, July 29 - 31 2013, Baltimore, MD, USA
https://doi.org/10.1145/2484838.2484879

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 20, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484879

Compared to the large number of sequential APRIORI variants [1, 6, 5, 20, 27], there exist only few parallel versions of it that are tailor-made for multiprocessor systems. These algorithms were moreover developed at a time, when the number of available threads in a single system was rather low so that scalability was not the major concern. The parallel algorithms [27, 9, 16] rely on either expensive synchronization primitives (e.g., locks) or replicated data structures. The former lead to contention and furthermore induce serious overhead because communication between the cores (e.g., bus snooping, request for ownership changes of cachelines) is required for them. Replicated data structures indeed minimize communication between cores, however, they are also unsuitable because their space usage explodes as the number of threads is largely increased. This furthermore lead to large pressure on the shared caches since all threads try to keep their data within them.

In this paper, we propose PCAPRIORI, which is a highly scalable version of APRIORI. PCAPRIORI incorporates efficient existing as well as novel components to achieve a high single-threaded performance as a basis for a fast parallel algorithm. Thereby, we chose a data layout that is simple and allows a fast parallel dataset conversion and load balancing during processing. Multi-threaded PCAPRIORI relies on partitioned data structures, which however require communication between the threads. To minimize communication costs, we propose a variant of the producer–consumer processing model [17]. We conduct a large number of experiments on large synthetic datasets. This includes a comparison with highly efficient existing algorithms and scalability experiments.

In what follows, we assume that the dataset being mined is already fully available in main memory (e.g., in an in-memory database or loaded previously from disks or network). In all other cases, loading the datasets can introduce a large sequential fraction so that most of our optimizations would be meaningless.

The rest of the paper is organized as follows: In Section 2, we explain the basic components of PCAPRIORI. This includes details to the data layout, sequential support counting, and employed database pruning techniques. We discuss parallel support counting in Section 3. We start with an overview about parallel counting techniques and thereafter explain our variant of the producer–consumer processing scheme in more detail. Section 4 provides the results of our experimental evaluation. We compare the single-threaded PCAPRIORI with existing implementations and evaluate the scalability of multi-threaded PCAPRIORI. Section 5 gives an overview about the related work of this paper; we mainly review and discuss sequential and parallel APRIORI-based algorithms. Finally, we conclude the paper in Section 6.

2. BASIC ALGORITHM

PCAPRIORI has basically the same core algorithm as APRIORI; it iteratively generates and tests *candidate itemsets* to obtain frequent itemsets of increasing cardinality. This works as follows: The input dataset is parsed to obtain the frequent items F_1. Based on these items, the dataset is transformed into a tailor-made data layout, which represents the dataset's transactions. Thereby, infrequent items are removed from the transactions, i.e., they are *filtered*. After the dataset has been fully converted, the algorithm executes two phases repeatedly to obtain all frequent itemsets.

Candidate generation and pruning: Generate the candidates C_k within the k-th iteration using the frequent itemsets F_{k-1}. For that purpose, merge all itemsets in F_{k-1} with all other itemsets in F_{k-1} that share the same first $k-2$ items. The number of candidates C_k is minimized to reduce the costs of the second phase.

Support counting: Scan the transaction database and generate for each transaction all of its size-k subsets. Search each of these subsets within the indexed candidates. If a subset is found, increase the support value of the respective candidate. After the scan is complete, the candidates in C_k that fulfill ξ form the frequent itemsets F_k of length k. Increase k by one and continue with the first phase.

The algorithm stops as soon as C_k is empty. APRIORI thus requires k iterations (i.e., scans over the converted database) to obtain frequent itemsets of length k. If k denotes the size of the largest found frequent itemset, then the output of the algorithm is given by $\bigcup_{i=1}^{k} F_i$.

For large datasets, the time required for candidate generation on APRIORI-suited datasets is negligible, so we use existing candidate generation and pruning techniques [24]. The support counting step, however, is computational expensive and forms APRIORI's performance hotspot.

2.1 Data layout

The internal transaction representation has a large impact on support counting since all transactions are scanned during an iteration. For PCAPRIORI, we employ a *page-based transaction representation*, where the filtered transactions are stored clustered in pages based on their length, i.e., all filtered transactions that share a page have the same length. Hence, the length of the transactions within a page must not be stored multiple times; each page stores the length of its containing transactions only once. Thus, less memory is required for storing small transactions because the length field requires as much memory as a single item. This amounts in a space reduction of 33% for length-2 transactions and 25% for length-3 transactions which leads to significant overall savings since filtered transactions of such lengths occur most frequently in datasets suitable for APRIORI. Nevertheless, it is not useful to maintain pages for each transaction length. Especially for long transactions, there is almost no space reduction and the pages for certain lengths (e.g., for more than 20 frequent items) usually contain only few transactions. For this reason, we only partition the transactions up to a length $p_{max} = 16$. All filtered transactions that contain more than p_{max} items are stored in pages where each of them has an explicit length information assigned. For mining, we maintain p_{max} lists where each list connects pages that contain transactions of the same length.

The filtered transactions are represented in the pages with only as many bytes as required to encode the number of

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 20, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484879

frequent items. Such an optimization was also applied by various other authors (e.g., [21]) within their particular representations and requires a recoding from the actual item id $I \in \mathcal{I}$ to an internal id that lies in the range from 0 to $n-1$ where n denotes the number of frequent items. Hence, for 256 or less items, each transaction item is 8-bit encoded, whereas otherwise the transaction items are 16-bit encoded. More than 65,536 frequent items are uncommon for frequent-itemset mining but in such a case, the items would be encoded using larger data types.

The conversion from a dataset to the page-wise layout is simple and cheap; p_{max} pages are maintained in which filtered transactions are inserted depending their length. Whenever a page is full, a new page is created and the full page is enqueued in one global page list. Parallel conversion works similar, except that each thread maintains p_{max} pages and parses independent parts of the dataset. To reduce synchronization costs of the enqueue operation, we set the page size to 1MB. Notice that the filtered transactions have a different order in the page-based layout, which, however, does not influence the mining result.

Besides the memory reduction and cheap conversion, the page-wise layout eases load balancing for multi-threaded support counting. The pages provide a proper granularity for work distribution and the load of a page can be predicted based on the length of the transactions it contains. Pages with large transactions induce more load because longer transactions have more subsets that need to be counted. For this reason, these pages are processed first to avoid load imbalance at the end of an iteration.

2.2 Memory management

Memory management is an essential part of pcApriori because large amounts of memory are repeatedly allocated and deallocated during mining, i.e., pcApriori stores the filtered transactions in large arrays and the indexed candidates and their support values using many small chunks of memory. Hence, the sizes of the allocations vary strongly. The standard memory allocation functions like `new` and `delete` within C++ or `malloc` and `free` within C, however, are intended for applications with simple allocation pattern and thus cause problems. First of all, the functions all maintain complex data structures (e.g., free-lists), which are updated whenever memory is allocated or deallocated. This leads to serious performance degradations when a large number of allocations is performed. Memory fragmentation—the second problem—increases the memory footprint of pcApriori. It is caused by the varying sizes of the allocations and worsens when multiple threads are used. Lastly, calls to the allocation functions are internally serialized, which hinders scalability.

Our tailor-made memory management avoids the problems of the standard allocators. We use lightweight memory pools, which provide memory that grows and shrinks like a stack. The pools provide a simple interface with basically only two functions. The function `grow` returns a chunk of memory from the end of the stack and increases the stack by the size of the allocated chunk. Contrary, the `shrink` function reduces the stack to a certain size. The memory allocation and deallocation of each of pcApriori's core data structures is thus realized with this interface. For that purpose, each thread has multiple own memory pools assigned from which it allocates memory without any synchronization. For example, each thread grows an own *page stack* by the size of a page whenever a new page must be created.

Each memory pool is maintained using only a few variables. The *base pointer* holds the address of a continuous memory chunk (i.e., the start of the stack) whereas the *stack pointer* refers to the end of the stack. Each pool has a large chunk of virtual memory assigned, which is provided by a memory-mapped file. These files are mapped on consecutive addresses with a large distance between each pair of pools (e.g., 16GB). The pools are increased or decreased using the thread-safe linux function `mremap`. To avoid repeated calls of it, we internally increase or decrease the stack chunk-wise—usually using at least 128MB chunks. Because virtual memory is not mapped to physical memory until it is touched (i.e., read or written), this does not increase the physical memory usage.

2.3 Support counting

Within support counting, all $\binom{n}{k}$ subsets of a transaction are created, where n denotes the length of the transaction and k the number of the current iteration. These subsets are then searched within the indexed candidates and—for each that is found—a respective counter is increased. Like other Apriori variants do, we employ different data structures for maintaining the candidates and their counter depending on the number k of the current iteration.

In the first two iterations ($k=1$ and $k=2$), pcApriori employs the *direct counting* technique [23]; i.e., a single array is used to represent the candidates. The basic idea is to transform a subset being counted into an integer value and use this value as index in the array. For example, if the size-2 itemset bd from a set of 10 frequent items $\{a, b, \ldots, j\}$ is counted, it can be mapped to $(b \to 1) \cdot 10 + (d \to 3) = 13$ and whenever it occurs in a transaction, the count value in the array at position 13 is incremented. The actual hash function is more elaborate to avoid unused entries for bb or db, which do not occur as subsets. Direct counting works well since only a *single access* per item ($k=1$) or 2-itemset ($k=2$) into an array is required.

For the later iterations ($k>2$), pcApriori employs trie-based counting [6, 5]. Its basic idea is to organize all candidates within a prefix-hash tree [10]. In the simplest variant of such a trie, each node consists of n pointers where n denotes the number of frequent items; the trie's depth is given by the length of the candidates. Only the *leaf nodes* do not contain pointers; they hold integer values representing the count values of the candidates. Each candidate (with items ordered ascending) forms a path within the trie. At the root node, the first item identifies which pointer of the root node should be followed; the second item identifies which pointer of the next node should be followed; and so forth. The candidate's last item identifies its associated `count` value at the leaf node. The transactions' subsets (again ordered ascending) are counted in a similar fashion. Their items identify a path within the trie; if the path is available, then the subset's last item is used to increase a count value within the leaf node.

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 20, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484879

Borgelt [6] and Bodon [5] employ the trie for indexing the candidates and—at the same time—for storing so far found frequent itemsets. Each node has therefore besides the n pointers also n count values, which denote the support of a itemset represented by the path to a count value. In each iteration, the trie is extended by a new level, which holds the candidates of this iteration. These new leaf nodes do not contain pointers until the next level is added in the next iteration. Borgelt [6] further stores at each node two variables that denote the first item and last item within a node; they are used to avoid unused pointers within the nodes.

In pcApriori, we divide the single trie into an *itemset trie* for storing so far found frequent itemsets and a *count trie*, which holds only the candidates of a single iteration and has `count` fields only in the leaf nodes. The former is thus solely used for candidate generation and gets a new level whenever new frequent itemsets are obtained whereas the latter is solely used for support counting. The separation of the tries greatly reduces the number of unused data elements (i.e., count fields and unused pointers in the inner nodes) within the caches during counting. Only the count trie has to be rebuild in each iteration, which however is negligible compared to the costs for support counting when large datasets are mined.

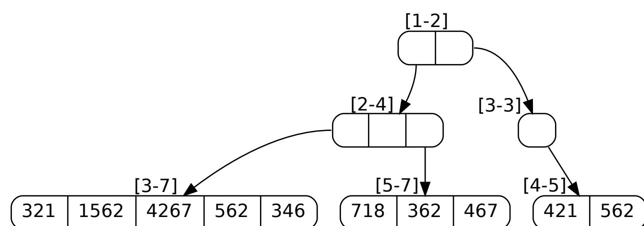


Figure 1: A count trie used in pcApriori

Figure 1 depicts the count trie that contain three inner nodes and represents 10 candidates. Each node maintains the range of items for which there exist pointers or count fields. The candidate *abd*, for example, is represented by the path $\{a \to 1, b \to 2, d \to 4\}$ and has the count value 1562.

We further speed up counting by reducing the size of the trie nodes. Almost all of them consist of only pointers to child nodes. On a 64-bit machine, a single pointer requires 64 bit or 8 bytes and can address 2^{64} bytes of memory. The count trie, however, is typically much smaller so that 32-bit pointers or sometimes even 16-bit pointers are sufficient to address its nodes. We thus adjust the size of the pointers to the size of the trie. Each pointer holds only an offset to a base address, which refers to the continuous memory area provided by the trie's memory pool. Whenever a pointer needs to be dereferenced, it is computed by adding the offset to the base address.

To minimize the effort for subset counting, Borgelt [6] proposed a recursive counting procedure, which ensures that each node is loaded only once when the subsets of a single transaction are counted. Basically, the function calls itself for each non-empty suffix of a passed *transaction substring* in a single recursion call and thereby the trie is traversed. All items within transaction substrings that reach leaf nodes, are counted in the leaf nodes' counter.

In pcApriori, we employ a loop-based version of the recursive function to avoid its many function calls. For that purpose, we transform it to multiple loop-based functions; each of these functions is tailor-made for a certain trie depth whereby it is sufficient to provide functions for tries with three ($k \geq 3$) up to six levels ($k \leq 6$). The remaining iterations often amount to only a small fraction of the overall runtime on Apriori-suited datasets so that the recursive function is sufficient in these iterations. Besides reducing the number of function calls, the transformation enables the compiler to apply further optimizations.

2.4 Database pruning

Database pruning techniques can further reduce the time for support counting. pcApriori scans in each iteration the converted database completely. Each filtered transaction that is removed using *pruning* thus reduces the scan time in the later iterations, i.e., subset counting for removed transactions is not required.

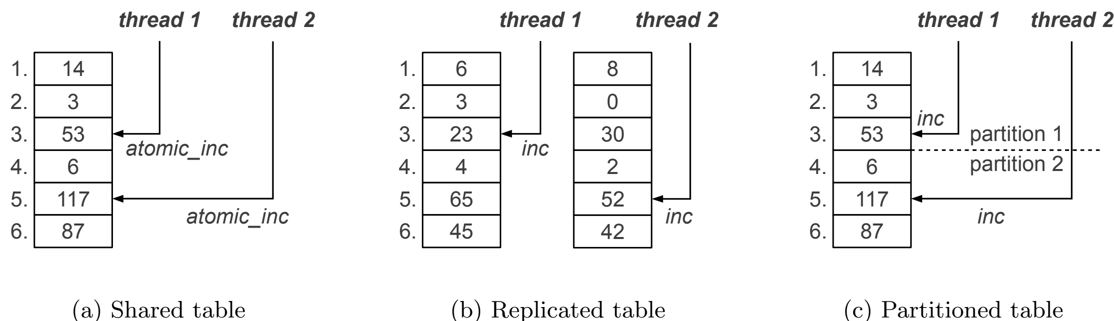
A simple transaction pruning technique is to remove transactions that are too small during an iteration. For example, the transactions with only *two* items are not required for obtaining the support of candidates with *three* and more items. pcApriori's page-wise layout eases pruning of such transactions; all lists with pages that contain transactions with insufficient length are simply skipped during support counting. Since none of the skipped pages (and thus skipped transactions) is physically accessed, there is no overhead for parsing the transaction's length, branching, or copying transactions. Such overhead might occur when pruning is applied to other data layouts. Notice that the pages with explicit length information could not simply be skipped, however, the number of iterations is typically smaller than p_{max} so that these transactions will not be skipped anyway.

Trimming transactions, i.e., removing items that are not required in later iterations from the transactions, is often even more beneficial than database pruning. Even a small decrease of the average transaction length has a large impact on the costs of subset counting, i.e., 120 size-3 subsets have to be counted for a transaction with 10 items whereas only 56 size-3 subsets have to be counted for a transaction with 8 items. However, reducing a transaction's length from 5 to 4 items might not be useful so that some trimming techniques with perfect results but large overhead are rather counterproductive.

Trimming is already very effective if it is based on the generated candidates within an iteration. After the candidates are generated, we exploit that not all frequent items occur in them. In pcApriori, the ids are assigned to the frequent items based on their frequency; less frequent items get a larger id. We observed that the largest id of the *candidate items*—in what follows denoted as *borderline id*—is often much smaller than the largest id of the frequent items since the less frequent items have a higher probability to be removed from all candidates. *Length trimming* removes all items with an id larger than the borderline id from the transactions being counted because these items are not part

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 20, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484879



(a) Shared table (b) Replicated table (c) Partitioned table

Figure 2: Parallel direct-count approaches

of any of the iteration's candidates and thus are never be counted. *Dead-item trimming* goes even further than length trimming. It removes all items that are not part in any of the candidates from the transactions.

3. PARALLEL SUPPORT COUNTING

So far, we have explained how support counting in pcApriori is performed using only a single thread. We will now discuss how the direct counting and trie-based counting can be performed in parallel using multiple threads.

3.1 Parallel direct counting

As discussed earlier, direct counting is used for obtaining the frequent items and frequent 2-itemsets within pcApriori. In both cases, counting is performed by increasing values in a large lookup table. If the frequent items should be obtained, the size of the lookup table depends on the number of distinct items that are within the dataset being mined. For market-basket analysis datasets, the number of distinct items is typically below 100,000. For web-mining datasets, however, the number of distinct items easily reaches several millions. Hence, the lookup table used for counting could require less than a megabyte but also up to hundreds of megabytes. Similarly, the number of frequent items from which the frequent 2-itemsets are obtained is typically in the range from several hundred up to a few ten thousands. If each ordered pair of frequent items is mapped to a 4-byte counter, then the respective lookup table varies from several kilobytes up to a few gigabytes. Thus, the lookup table for direct counting fits in some cases in the processor's caches and is in other cases quite large.

While the direct counting technique is straight-forward to implement for sequential processing, it is more involved for parallel processing. In general, there are three parallel direct counting approaches that differ in the way how they deal with parallel increments. Using a single *shared table* for all threads is prohibitively expensive because synchronization (e.g., locks or atomic increments) is required to avoid inconsistent results caused by race conditions. Figure 2(a) illustrates the shared table approach. The *replicated table* approach—shown for two threads in Figure 2(b)—maintains a local table for each thread in which the thread can increment the items' count values independently from other threads. Synchronization is only required after the counting to merge the local tables into a single table that contains the final result. Besides its simplicity, the replicated table approach has a much higher memory consumption than its sequential counterpart. Hence, it cannot be used for very large tables and/or for a very large number of threads. Furthermore, even for small tables with only a few megabytes, it has a limited scalability because the local tables may not fit into the caches anymore while a single global table would fit (at least in the large last-level cache).

The *partitioned table* approach uses only a single table for all threads, however, it is partitioned so that each thread increments the count values for those items that lie within a certain ranges. In Figure 2(c), for example, the first thread counts all values from 1 to 3 while the second thread counts values from 4 to 6. For realistic scenarios, the ranges are usually much larger and the threads have multiple ranges assigned for a better load balancing. Furthermore, for 32-bit count values, the size of the ranges must be a multiple of 16 because otherwise false sharing might occur. The actual parallel counting of the items can be done in two ways: (1) all threads scan the complete dataset and count only their assigned values and (2) each thread has an assigned part of the dataset and distributes scanned values to the responsible threads. The former variant is not useful since the threads perform much redundant work, i.e., each thread checks for each value whether it is in its range or not. The latter, however, requires communication between the threads. We achieve this using a modified producer–consumer scheme, which we explain below in more detail. Although partitioning induces communication overhead, which is not necessary when the lookup table is replicated, it requires only as much space as the shared approach or sequential processing. Compared to replicated processing, the caches are thus much better exploited, which pays off for large tables.

There are also hybrid approaches between these base approaches possible. For example, a replicated table could be assigned to each processor whereas threads that run on the same processor share a partitioned table. As soon as the overhead for partitioning is paid, however, it seems more efficient to use only a single lookup table for all threads of a system because then each processor has only to hold parts of the full table, which improves cache efficiency. Only when communication between the processors is expensive (in terms of bandwidth) or too many partitions are required, then hybrid approaches might beneficial.

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 20, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484879

3.2 Processing model

In the following, we describe our multiple-producer/multiple-consumer processing model, which is derived from the basic producer–consumer model [17]. It is employed for (1) partitioning the subsets being counted and (2) communicating them to the threads responsible for increasing their respective count values in the partitioned count table. The basic idea is that each thread is always in one of two states. As *producer*, it partitions the input data based on a certain partitioning function whereas as *consumer*, it processes data in its partition. For parallel direct counting, the transactions from the input dataset ($k = 1$) or the page-wise layout ($k = 2$) form the input data and processing means to increase the count values in the count table.

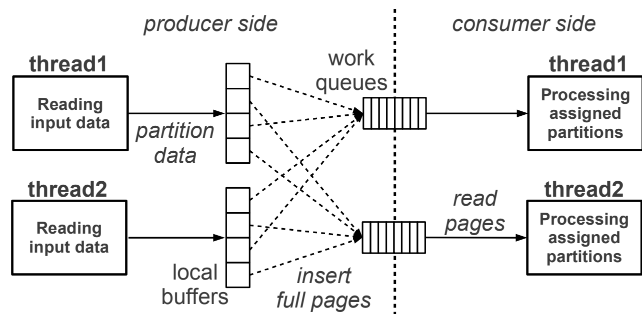


Figure 3: Multiple-producer/multiple-consumer processing with four partitions and two threads

Synchronized *work queues* are used to communicate partitioned data between the producers and the consumers. Each thread has a single work queue assigned that contains the partitioned data the thread is responsible for, i.e., data that lies in the thread's assigned partitions. To avoid excessive communication overhead that would occur if a producer enqueues single data elements, the producers collect data in *local buffers* and enqueue chunks of data. For that purpose, the local buffers have pages assigned that float in cycles during processing. They are (1) filled by a producer, (2) put into a work queue, (3) processed by a consumer, (4) put into an *empty-pages queue*, and finally (5) obtained by a producer from the latter queue to start the cycle again. Figure 3 illustrates the components of the processing model (except for the empty-pages queue) when two threads are employed that divide the input data into four partitions. Notice that the work queues are filled by multiple threads but each of them is read by only a single thread.

For parallel direct counting, each producer calculates the position of the subsets' count values and communicates these positions to the consumers. For $k = 1$, the item itself is used as position while for $k = 2$, it is calculated from the two items as discussing by Perego et al. [23]. A consumer thus only increases the count values using the calculated positions that are in the pages of its work queue. The full algorithm works as follows:

Producer A thread that is in the producer mode continuously reads chunks of transactions; as long as a single chunk contains further transactions, it repeats the following steps:

1. Read a transaction from the input chunk.
2. For each size-k subset s of the transaction repeat the following steps:
   a) Obtain the partition id p_{id} and the count-table position c for s.
   b) If the page of the local buffer assigned for p_{id} is full, then enqueue the page into the work queue assigned for p_{id} and get a new page from the empty-pages queue and assign it to the local buffer.
   c) Insert c into the page of the local buffer assigned for p_{id}.

If the thread's work queue is *not* empty, it switches to *consumer mode*. Otherwise, it reads the next chunk from the input data and the cycle starts again.

Consumer A thread in the consumer mode is responsible for increasing the count values for all subsets that are in its partition. It repeats the following steps:

1. If the thread's work queue is empty, switch back to *producer mode*.
2. Dequeue the next page p from the assigned work queue.
3. For each count-table position c in p, increase the c-th value in the global count table.
4. Enqueue p into the empty-pages queue.

The algorithm finishes, as soon as there are no further transaction chunks and all pages in all work queues are processed. The questions that remain are (1) which partition function should be employed, (2) how to balance the load and distribute the partitions, and (3) how many partitions should be used.

Partitioning the items in the first iteration is performed based on the items themselves. The count-table position of an item i is inserted into the partition x using $x = (i/256) \mod p$ where p denotes the number of partitions. The division is implemented using a shift instruction and is required to avoid false sharing that occurs when two consecutive count values are in different partitions but are in the same cache line. The item pairs are similarly partitioned in the second iteration: The count-table position of a pair of items i and j is inserted into the partition $x = i \mod p$. By using only the first item for obtaining the partition id of an item pair, we reduce the calculation effort during counting since we only need to calculate the partition id once for a set of subsets generated from a transaction, e.g., the subsets ac, ad, and af for a transaction $acdf$ all fall in the same partition.

The partitions themselves are distributed in a round-robin manner. This may lead to load imbalance, which, however, is compensated by our processing model. All threads can always become producers for which load is always available. Only a single consumer should not be overloaded since their work queues would otherwise overflow—leading to contention for all threads inserting in such a queue. For this reason, it is always the thread's highest priority to process

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 20, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484879

pages of its work queue because this work cannot be distributed to other threads (at least as long as the partitions are not redistributed). It thus switches also to be a consumer when it is blocked by a full queue. Fortunately, the tasks of the consumer are cheaper than the tasks of the producer; this limits the possibility of queue contention.

Finally, the number of partitions, work queue sizes, number of floating pages, and size of a page influence the efficiency and overhead of the processing model. If, for example, the pages are too large and too many partitions are used, then the size of the local buffers explodes. We empirically found that 256 partitions, a page size of 32KB, a task queue with at most 256 entries and $1024 \cdot t$ floating pages where t denotes the number of threads works best on our system. These variables may be different on other systems.

Summing up, the processing model employed for direct counting is well suited for multiprocessor systems. All threads can hold the parts of the global count table they are responsible for in the processor's memory on which they are running. Only the pages in the work queues need to be exchanged between the processors for which the interconnects provide sufficient bandwidth. The processing model thus should scale very well to a large number of processors and threads. Nevertheless, the model's major drawback is the effort required for partitioning the data. Many operations need to be performed per item whereas basically only a single memory request is required per item for sequential direct counting. This might limit its applicability for small count tables. For large count tables, however, the partitioning weakens this drawback because it increases spatial and temporal locality, which increases cache line utilization. This is also suggested by our experiments in this paper.

3.3 Parallel trie-based counting

For counting subsets using tries in parallel, we have basically the same options as for parallel direct counting. The *shared count-trie* approach requires the least changes to the algorithms. All threads traverse the same trie and increase the `count` values within the leaf nodes using atomic increments or small critical regions. Clearly, this approach has again high synchronization costs, which strongly limit its applicability. The *replicated count-trie* approach is similar to the full replication employed by [16]. The count trie is split into a shared upper part and a replicated lower part. The shared part comprises all the inner nodes, which are only read by the threads, while the lower part comprises the leaf nodes, which are updated during counting. As mentioned before, the counting without synchronization comes at the price of a multiple times larger memory consumption compared to the single-threaded execution. For this reason, replicated counting is only useful when the number of leaf nodes is rather low, i.e., for moderate or high ξ values or in the later iterations. The *partitioned count-trie* has the same memory footprint as the trie used for single-threaded counting because only a single trie is used. The count values in the trie, however, are partitioned. Each thread has an assigned number of partitions in which it increases the count values without synchronization. To avoid that all threads scan all pages, we use the previously described producer–consumer scheme to exchange the positions of `count` values that need to be increased by another thread. Recall that the producer–consumer scheme incurs overhead but may increase the data locality when the count values in a certain area are increased.

The implementation of replicated counting in pcApriori is quite simple. A single memory pool is used to provide memory for the shared upper part of the count trie. After the candidates are inserted, all inner nodes are stored within the pool's respective memory area and pointers to inner nodes contain always the offset to the starting address of this area. Pointers to leaf nodes, however, contain an offset to the address 0x0. Since the leaf nodes are basically only a large array of integer values, the address of a count value can be calculated using these offsets and the base address of a thread's replicated leaf array. Hence, each thread adds the leaf offsets to the address of its own memory, which is obtained from the thread's memory pool. Clearly, all count fields within this area have to be set to zero before counting.

Partitioned trie-based counting in pcApriori is performed similar to partitioned direct counting. Only the partition criteria is different: All `count` values of leaf nodes that have the same value $x = i \bmod p$ are within the partition, where i denotes the second last item of a candidate and p denotes the number of partitions. For example, the candidates *acd*, *ace*, and *bcd* are in the same partition.

4. EXPERIMENTS

This section contains the results of our experimental evaluation of pcApriori. We start with explaining the setup—including details about our test system and the employed datasets. Thereafter, we discuss the outcome of the single-threaded and multi-threaded experiments.

4.1 Setup

In the following experiments, we use a four-socket NUMA multiprocessor system that consists of four Intel E7-4830 processors, each equipped with eight cores and a 24MB last-level cache. Each processor has 32GB of main memory assigned so that 128GB are available for the complete system. The cores run at a frequency of 2.13GHz and support dynamic frequency scaling. In favour of more consistent results, we turned this feature off and run the cores always with 2.13GHz. The E7-4830 processor supports Hyperthreading so that up to 64 threads can be run in parallel.

We employ a 64-bit linux (Ubuntu server 10.04) as operation system. pcApriori is implemented in C++ and compiled using Intel Parallel Composer 2011; we used `-fast` as only optimization flag. OpenMP pragmas were used to enable thread-level parallelism. We implemented—similar as [21]—most of our components using templates and select an appropriate variant at runtime depending on the number of frequent items. If, for example, the number of frequent items is below 257, then all filtered transactions can be represented using 8-bit arrays and the 8-bit versions of dataset conversion and support counting functions are selected. We measure in all experiments the wall clock time by using linux `gettimeofday` (within the code of our algorithms) or linux `time` (at process level for existing algorithms).

In all experiments, we mine datasets that are well-suited for Apriori-based algorithms, i.e., they consist of mainly short

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 20, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484879

transactions. Unfortunately, available realistic datasets are all very small. The datasets **retail** [8], **kosarak** [11], and BMS-POS [29], for example, fit together in the cumulated L3-cache (4×24MB) of our test system. Hence, mining these realistic datasets is not challenging.

To provide larger datasets for our experiments, we generated synthetic datasets using the IBM Quest Dataset Generator [3]; it allows to build datasets of arbitrary size. We use it to build datasets that follow the same characteristics (e.g., average cardinality, number of distinct items) of the realistic datasets while containing more transactions. For example, the **quest-retail** dataset has the same average cardinality and number of distinct items as the **retail** dataset but it has about 1000x more transactions, i.e., instead of 4MB it comprises 4GB. Despite the same characteristics, the synthetic datasets behave differently compared to the real-world datasets because the Quest generator is based on a simple model. For this reason, we built a tool that allows us to combine a realistic with a synthetic dataset. It exchanges the n most-frequent distinct items of each transaction of the synthetic dataset by the n most-frequent distinct items of a randomly chosen transaction from the respective realistic dataset. Thereby, n was set to three quarters of the average transaction cardinality but at most to 25. For example, we exchanged the 7 most-frequent distinct items of each **quest-retail**'s transactions because this dataset has an average transaction cardinality of 10. Table 1 gives an overview about the synthetic datasets used in this paper. All of these datasets comprise 4GB.

	# items	avg. card.	# transactions
quest-retail	17,000	10	86 million
quest-kosarak	41,000	8	115 million
quest-BMS-POS	2,000	7	128 million

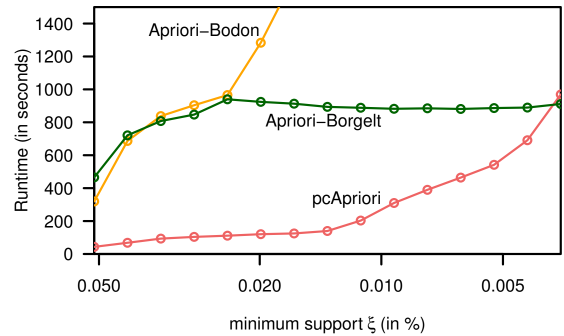
Table 1: Characteristics of the used datasets

To provide a fair starting situation for all competing algorithms, we store the datasets within a RAM disk. We thus do not need to change their file interfaces—i.e., whether they use **mmap**, **read**, or **gets**—for employing them to load the datasets with in-memory speed. In pcApriori, we load the data using linux **mmap** because it eases access to a file.

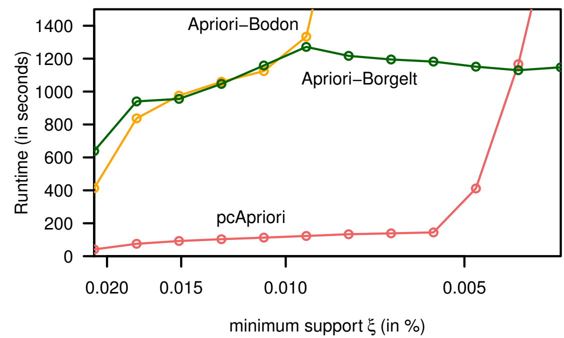
4.2 Single-threaded

In the first set of experiments, we evaluate pcApriori's single-threaded performance. The purpose of these experiments is to show that pcApriori is competitive with existing mining algorithms. This is a precondition for multi-threaded pcApriori because running an algorithm with multiple threads is not reasonable when its single-threaded version is several orders of magnitude slower than an other single-threaded algorithm that solves the same task.

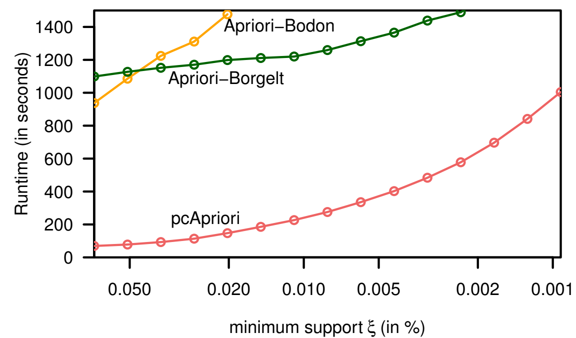
As competitors, we use two highly optimized sequential Apriori implementations from Bodon [4] and Borgelt [7]. The former (Apriori-Bodon) relies on C++ standard data structures, uses the standard allocators for memory allocation, and stores all transactions in a tree in which duplicate transactions are stored only once. Apriori-Borgelt is implemented in C and also uses the standard memory allocators.



(a) quest-retail



(b) quest-kosarak



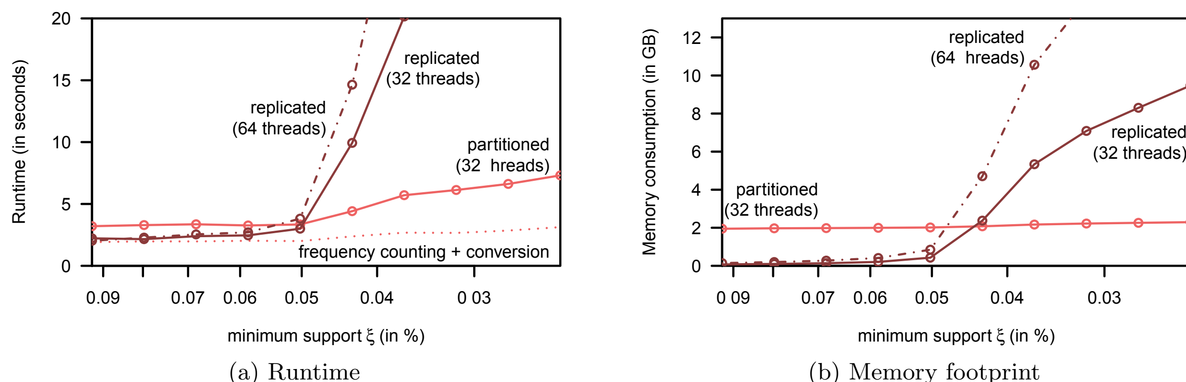
(c) quest-BMS-POS

Figure 4: Single-threaded performance on various datasets and varying ξ values

It requires only a single run over the dataset because it converts the complete dataset into an internal representation. Thereby, however, the transactions are not filtered because the frequent items are first obtained after the conversion. Apriori-Borgelt also builds a trie-based representation from the filtered transactions. Subset counting is solely performed on this trie, which greatly reduces the time for counting. We further tried Goethal's Apriori implementation, an FP-growth implementation [13], and Borgelt's Eclat implementation [6]. All three showed a poor performance on the used datasets so we do not provide results for them. The latter two algorithms, however, are superior on datasets with larger transactions (cf. the discussion in Section 5).

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 20, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484879



(a) Runtime

(b) Memory footprint

Figure 5: Parallel direct 2-counting on `quest-retail` for varying ξ values

Figure 4(a) illustrates the execution time of the algorithms under test on `quest-retail`. As can be seen, pcApriori is always the fastest of the three algorithms for rather large ξ values. For $\xi = 0.042\%$, it requires only 68s to finish whereas the better of both other algorithms requires 687s; this amounts to a speedup of 10x. Compared to the other algorithms, pcApriori's strengths are its phases before the actual mining, i.e., the tasks before the expensive subset counting. Obtaining the frequent items and the dataset conversion takes only 30s. Contrary, Apriori-Borgelt requires 460s before it starts mining the frequent itemsets. This time includes, however, all the steps to build the trie-based transaction representation, which greatly speeds up subset counting and leads to an almost constant total execution time until very small ξ values are used. The execution time even decreases on this dataset for $\xi < 0.024\%$—caused by caching effects—until it increases again to 1100s for $\xi = 0.001\%$ (not shown). Apriori-Borgelt's efficient subset counting is also the reason that it is faster than pcApriori for $\xi < 0.003\%$. Apriori-Bodon is always slower than pcApriori but performs better than Apriori-Borgelt for $\xi > 0.04\%$. It has high costs during subset counting because it does not apply any pruning technique and has a rather inefficient recursive subset counting procedure. Furthermore, the initial parsing and conversion of the dataset is also much slower than for pcApriori because it uses the standard C++ allocators.

We observe similar results on the datasets `quest-kosarak` (Figure 4(b)) and `quest-BMS-POS` (Figure 4(c)). On these two datasets, pcApriori performs best for rather larger ξ values. It achieves speedups of up to 16x within the measured ξ value range. Apriori-Borgelt is faster than pcApriori on `quest-kosarak` as ξ gets smaller than 0.0035%.

4.3 Multi-threaded

In the next set of experiments, we evaluate the performance of multi-threaded pcApriori. We first provide results for parallel direct counting. For that purpose, we implemented the shared (shared), replicated (replicated), and partitioned (partitioned) count table approach for it.

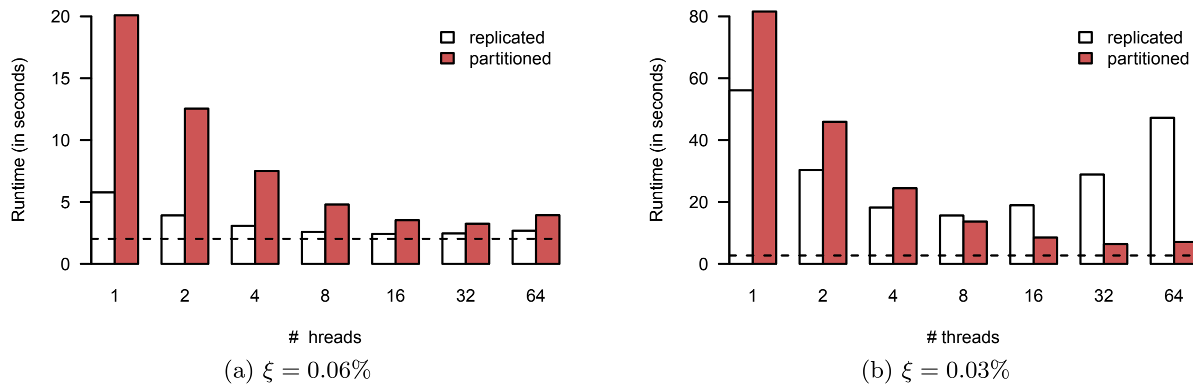
We observed that replicated performs best within the first scan of pcApriori, i.e., when the frequent item are obtained (not shown). For all three datasets, the number of distinct items is smaller than 41,000 so that each of the replicated count arrays requires at most 200KB and thus fits always in the L2-cache of each core. Therefore, replicated scales almost linear as the number of threads is increased. Partitioned scales also well, but has much higher costs since the data must be partitioned and transfered to the threads. For web-mining datasets with many distinct items, however, partitioned might be faster than replicated. The shared count approach does not scale at all. It is already for a single thread much slower than both other counter and the performance even decreases as the number of threads is increased. For this reason, we do not consider this approach in the following experiments.

The results are different when the frequent 2-itemsets are obtained in the second iteration. Figure 5(a) illustrates the respective runtime on `quest-retail` with ξ in the range from 0.10% to 0.02% and 32 threads. We further plot the runtime of replicated for 64 threads and time required for obtaining the frequent items and the dataset conversion. As can be seen, replicated is only faster than partitioned for $\xi > 0.05\%$. For such ξ values, the number of frequent items is below 1300, which limits the number of size-2 candidates to 844,350; a single count table thus requires less than 3MB so that the size of the replicated count tables remains acceptable. In this ξ range, however, counting the size-2 candidates accounts for only a small fraction of the overall runtime. As ξ further decreases, more frequent items are obtained and, with that, the count tables holding the size-2 candidates grow considerably. Partitioned is then much more efficient than replicated because more candidates remain in the caches during counting. Interestingly, both counting approaches do not benefit from Hyperthreading for any of the measuring points; the overhead for the additional replicated tables or the queues is not offset by the SMT threads.

The memory usage of the two counting approaches (excluding the size of the converted dataset) is depicted in Figure 5(b). Partitioned has higher memory requirements than replicated for high ξ values because it has to maintain the pages (i.e., that are within the queues and free page queue) required by the producer–consumer processing

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 20, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484879



(a) $\xi = 0.06\%$ (b) $\xi = 0.03\%$

Figure 6: Parallel direct 2-counting on `quest-retail` for various ξ values and a varying number of threads

scheme. This memory, however, remains constant as ξ decreases. Only the single count table grows by about 300MB from $\xi = 0.09\%$ to $\xi = 0.02\%$. Contrary, the memory footprint of REPLICATED grows from 74MB to 5.75GB in the same range. If 64 threads are used, then even twice as much memory is required. Notice that both approaches require roughly the same amount of memory for $\xi = 0.043\%$, but PARTITIONED is at this point already much more efficient than REPLICATED because of its better cache utilization.

To evaluate the scalability of both count table approaches, we varied the number of threads for certain ξ values from 1 to 64. Figure 6(a) illustrates the time required for obtaining the frequent 2-itemsets on `quest-retail` with $\xi = 0.06\%$. We include the time for the earlier phases—illustrated using a dashed line—, which are still run using 32 threads. As can be observed, REPLICATED performs best; it is more than four times as fast as PARTITIONED when only a single thread is used because the count table is quite small so that the partitioning overhead does not pay off, i.e., a single table contains count values for "only" 818,560 candidates on this dataset when $\xi = 0.06\%$. Nevertheless, PARTITIONED scales well up to 32 threads whereas REPLICATED scales only up to 16 threads. This effect is increased when ξ decreases to 0.03% (Figure 6(b)). The count table then contains 51,974,110 candidates generated from 10,196 frequent items. REPLICATED scales only up to 8 threads; if more threads are employed, the runtime even increases until it is almost as high as for a single thread. Again, PARTITIONED scales well up to 32 threads and is with 8 threads already faster than REPLICATED.

We obtain similar results for the dataset `quest-kosarak`. If the size-2 candidates for more than 2220 ($\xi \approx 0.02\%$) frequent items are counted, then PARTITIONED is more efficient than REPLICATED. On `quest-BMSPOS`, however, REPLICATED performs always best since this dataset contains only 2000 distinct items.

Summing up, REPLICATED is more efficient for high ξ values where the count tables are rather small and candidate counting typically is not expensive. As soon as the count tables grow considerably, the pressure on the shared caches increases, which limits REPLICATED's scalability. PARTITIONED is then more efficient; it scales well with an increasing number of threads. Nevertheless, it has higher initial memory requirements for the producer–consumer scheme, so one should switch from REPLICATED to PARTITIONED depending on the number of frequent items. The threshold for switching should be set to 2200 frequent items on our test system, but may be different on other systems that have larger caches or allow more threads to run in parallel.

We obtain similar results for the overall mining procedure. The runtime for a varying number of threads is illustrated in Figure 7(a). REPLICATED scales only well up to 4 threads whereas PARTITIONED scales well up to 32 threads. For this reason, PARTITIONED is multiple times faster than REPLICATED when a large number of threads is used. The overall memory consumption is illustrated in Figure 7(b). As can be seen, the PARTITIONED's memory footprint grows strongly as the number of threads is increased. For 64 threads, it requires more than 25GB while the converted dataset comprises only about one GB. PARTITIONED's memory footprint increases too—caused by the floating pages—but remains acceptable even for 64 threads. If necessary, it could further be decreased if the floating page size is reduced at the cost of a slightly worse runtime.

The overall runtime of PCAPRIORI on `quest-kosarak` and `quest-BMS-POS` is similar (not shown). REPLICATED performs worse than PARTITIONED as soon as the count trie exceeds about 15–25MB. Moreover, REPLICATED is unusable when the count trie is larger than 300MB. Hence, only PARTITIONED can be employed if a large number of threads is used and the count trie reaches a certain size; REPLICATED is thus only useful for very small count tries.

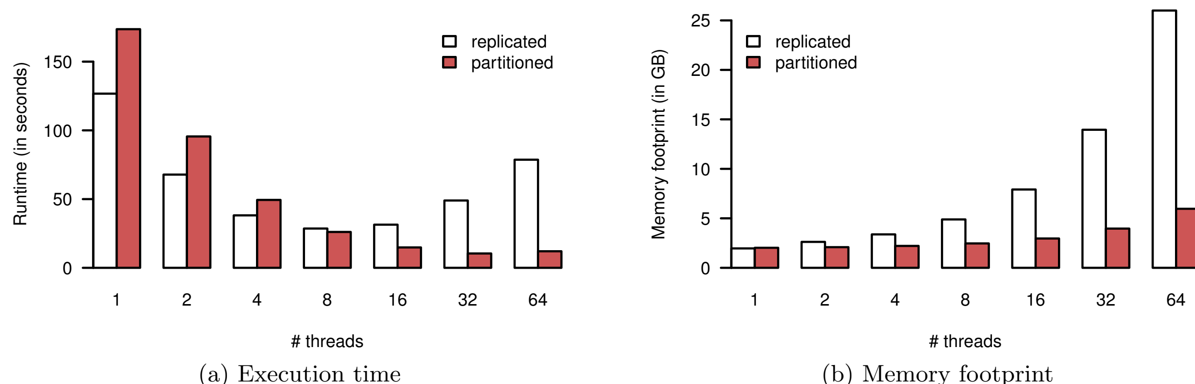
5. FURTHER RELATED WORK

We will now review related work that is not already covered in this paper so far. We organize it in three parts: We review (1) sequential APRIORI versions, (2) parallel APRIORI versions, and (3) other frequent-itemset mining algorithms.

Optimizations for sequential APRIORI mostly target the support counting step or the transaction representation. Zaki et al. [27], for example, improve the original hash tree used for counting. Other APRIORI variants [20, 6, 5] employ prefix

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 20, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484879



(a) Execution time

(b) Memory footprint

Figure 7: Overall performance on `quest-retail` and $\xi = 0.03\%$

trees for this purpose, which typically are superior over the employed hash trees. Perego et al. [23] proposed the direct count technique, which—as discussed earlier—uses an array for counting the frequency of the distinct items or candidate 2-itemsets. They further use direct counting in combinations with a prefix tree to obtain the frequent items in the later iterations ($k > 2$). Finally, Lucchese et al. [19] uses the direct count technique directly in the third iteration ($k = 3$). Clearly, this works only when the number of frequent items is rather small.

Optimizations for the transaction representation mainly target at improving support counting. Some algorithms [22, 23] employ pruning techniques that remove unnecessary transactions and items in the database to reduce the effort of later scans. Orlando et al. [21] use *dynamic data type selection* to reduce the size of the physical representation of the database. If the database consists of only 256 different frequent items, a single byte is sufficient to store an item. Hence, storing a transaction with five items requires only 6 bytes; one byte for the length information and 5 bytes for the items. Finally, Borgelt [6] and Bodon [5] represent the transaction database using prefix trees. These trees *summarize* all database transactions. Mining is employed directly on the tree by counting the subsets in all tree paths. Depending on the dataset, the tree might be smaller than the transaction database, however, building the tree is more expensive than converting the dataset into the regular array-based representation.

The number of parallel Apriori version for multiprocessor systems is rather low. Zaki et al. [27] discussed several parallel versions. The algorithm CCPD uses a shared candidate hash tree, which is synchronized using locks at the leaf nodes, and partitions the dataset in equal-sized chunks. Clearly, the lock-based shared hash tree does not scale to a large number of threads. The algorithm PCCD partitions the candidates as in pcApriori, however, each thread scans the full transaction database. Hence, much redundant work is performed by each thread so that Zaki et al. did not consider this variant as useful. Cheung et al. [9] proposed APM, which uses a shared prefix tree for all threads. As for CCPD, synchronization is required for this tree and thus hinders scalability. Finally, Jin et al. [16] examined different variants for parallel support counting. They proposed four different lock-based tree methods, which however all scale worse than the replicated tree counting method used in this paper for comparison. Hence, none of the existing parallel Apriori implementations scales to a large number of threads.

Mining algorithms that are not based on Apriori employ different core data structures and strategies for obtaining the frequent items. As HooshSadat et al. [15] observed, all mining algorithms have their sweet spot in which they are usually superior over other mining algorithms.

Eclat [28] converts the transaction database into the *vertical layout* where each frequent item has an assigned set of transactions ids that denotes in which transactions the item occurs. Intersecting two of such sets results in a new set, which contains all transactions in which both items occur. Hence, frequent itemsets are obtained by intersecting the sets of items or itemsets. Unlike Apriori, Eclat works well for mining long transactions or when the result set contains very long frequent itemsets. For short transactions, however, Apriori performs much better than Eclat.

FP-growth [14] represents the transaction database using frequent-pattern trees. Such trees resemble prefix trees and are often smaller than the transaction database. During mining, the frequent-pattern trees are repeatedly traversed to build smaller ones of them and thereby obtaining the frequent itemsets. FP-growth usually outperforms Apriori when many long itemsets are found. In the remaining cases, Apriori performs better because building the prefix trees then does not payoff.

6. CONCLUSIONS

Frequent-itemset mining has many application areas and is often performed on very large datasets. To cope with such datasets, (1) the right algorithm has to be chosen for mining and (2) it must exploit the large degree of parallelism provided by current systems. Apriori is usually the algorithm of choice for datasets that have many short transactions. Existing parallel versions of it, however, do not scale to a large number of threads because they rely on either expensive synchronization primitives or replicated data

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 20, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484879

structures. In this paper, we introduce PCAPRIORI, which is a highly scalable version of APRIORI. It incorporates an efficient data layout, fast subset counting, and efficient but simple database pruning techniques. Parallel subset counting is performed using partitioned candidates. The communication between the threads required for it is performed using a variant of the producer–consumer processing model where each thread is both, producer and consumer. The candidate partitioning and employed processing model allow a scaling to a large number of threads. In various experiments with large datasets, we achieve a near-linear scaling on our test system that has 32 cores. We believe that our processing model can also be employed to obtain highly scalable versions of algorithms [2, 26, 18] that mine other popular pattern types (e.g., rare itemsets, sequences) and are based on APRIORI.

Acknowledgements

This work is partly funded by the European Regional Development Fund (EFRE) and the Free State of Saxony under the grant 100067363 ("cool iBit computing") and the German Research Foundation (DFG) in the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing' and the grant LE 1416/22-1.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, pages 3–14, 1995.

[3] R. Agrawal and R. Srikant. Quest synthetic data generator, 1997.

[4] F. Bodon. A fast apriori implementation. In *FIMI*, 2003.

[5] F. Bodon. Surprising results of trie-based fim algorithms. In *FIMI*, 2004.

[6] C. Borgelt. Efficient implementations of apriori and eclat. In *FIMI*, 2003.

[7] C. Borgelt. Recursion pruning for the apriori algorithm. In *FIMI*, 2004.

[8] 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.

[9] D. Cheung, K. Hu, and S. Xia. Asynchronous parallel algorithm for mining association rules on a shared-memory multi-processors. In *In 10th ACM Symp. Parallel Algorithms and Architectures*, pages 279–288, 1998.

[10] E. G. Coffman, Jr. and J. Eve. File structures using hashing functions. *Commun. ACM*, 13:427–432, July 1970.

[11] FIMI. Frequent itemset mining implementations repository. http://fimi.cs.helsinki.fi/, November 2004.

[12] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: report on fimi'03. *SIGKDD Explorations*, 6(1):109–117, 2004.

[13] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI*, 2003.

[14] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[15] M. HooshSadat, H. W. Samuel, S. Patel, and O. R. Zaïane. Fastest association rule mining algorithm predictor (farm-ap). In *C3S2E*, pages 43–50, 2011.

[16] R. Jin, G. Yang, and G. Agrawal. Shared memory parallelization of data mining algorithms: techniques, programming interface, and performance. *Knowledge and Data Engineering*, 17(1):71 – 89, 2005.

[17] V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[18] B. Liu, W. Hsu, and Y. Ma. Mining association rules with multiple minimum supports. In *SIGKDD*, pages 337–341, 1999.

[19] C. Lucchese, S. Orlando, and R. Perego. kdci: on using direct count up to the third iteration. In *FIMI*, 2004.

[20] A. Mueller. Fast sequential and parallel algorithms for association rule mining: a comparison. Technical report, 1995.

[21] S. Orlando, C. Lucchese, P. Palmerini, R. Perego, and F. Silvestri. kdci: a multi-strategy algorithm for mining frequent sets. In *FIMI*, 2003.

[22] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *SIGMOD*, pages 175–186, 1995.

[23] R. Perego, S. Orlando, and P. Palmerini. Enhancing the apriori algorithm for frequent set counting. In *DaWaK*, pages 71–82. 2001.

[24] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[25] A. Veloso, B. Rocha, M. d. Carvalho, and W. Meira, Jr. Real world association rule mining. In *BNCOD*, pages 77–89, 2002.

[26] T. Washio and H. Motoda. State of the art of graph-based data mining. *SIGKDD Explor. Newsl.*, 5(1):59–68, July 2003.

[27] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. In *Supercomputing*, 1996.

[28] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. Technical report, 1997.

[29] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *SIGKDD*, pages 401–406, 2001.

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden