Memory-Efficient Frequent-Itemset Mining

Benjamin Schlegel
Technische Universität Dresden
Dresden, Germany

Rainer Gemulla
Max-Planck-Institut
Saarbrücken, Germany

Wolfgang Lehner
Technische Universität Dresden
Dresden, Germany

ABSTRACT

Efficient discovery of frequent itemsets in large datasets is a key component of many data mining tasks. In-core algorithms—which operate entirely in main memory and avoid expensive disk accesses—and in particular the prefix tree-based algorithm FP-growth are generally among the most efficient of the available algorithms. Unfortunately, their excessive memory requirements render them inapplicable for large datasets with many distinct items and/or itemsets of high cardinality. To overcome this limitation, we propose two novel data structures—the CFP-tree and the CFP-array—, which reduce memory consumption by about an order of magnitude. This allows us to process significantly larger datasets in main memory than previously possible. Our data structures are based on structural modifications of the prefix tree that increase compressability, an optimized physical representation, lightweight compression techniques, and intelligent node ordering and indexing. Experiments with both real-world and synthetic datasets show the effectiveness of our approach.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data mining*

General Terms

Algorithms, Performance

1. INTRODUCTION

The goal of frequent-itemset mining is to discover sets of items that frequently co-occur in the data. The problem is non-trivial because datasets can be very large, consist of many distinct items, and contain interesting itemsets of high cardinality. Frequent-itemset mining is a key component of many data mining tasks and has applications in areas such as bioinformatics [30], market basket analysis [13], and Web usage mining [22]. For example, frequent-itemset mining is an increasingly popular tool to support web-shop customers in finding products of interest ("customers who bought this item also bought ...").

The need for finding frequent itemsets in ever-growing datasets has driven a wealth of research in efficient algorithms and data structures [1, 32, 14, 3]. Existing approaches can be classified into three major categories. First, *bottom-up algorithms*—such as the well-known Apriori algorithm [1, 3]—repeatedly scan the database to build itemsets of increasing cardinality. They exploit monotonicity properties between frequent itemsets of different cardinalities and are simple to implement, but they suffer from a large number of expensive database scans as well as costly generation and storage of "candidate itemsets". Second, *top-down algorithms*—such as TopDown [32]—proceed the other way around: The largest frequent itemset is built first and itemsets of smaller cardinality are constructed afterwards, again using repeated scans over the database. Finally, *prefix-tree algorithms* operate in two phases. In the first phase, the database is transformed into a prefix tree designed for efficient mining. The second phase extracts the frequent itemsets from this tree without further base data access. Algorithms of this class require only a fixed number of database scans, but may require large amounts of memory.

There is no "best" algorithm for frequent-itemset mining in general, but the prefix-tree algorithm FP-growth [14] is usually considered as one of the fastest available algorithms [11]. The key advantage of FP-growth is that it requires only two passes over the database; it is thus very I/O efficient. The two passes are used to build an FP-tree, which can be viewed as a compressed representation of the frequent items and their co-occurrence in the data. Based on the initial FP-tree, FP-growth recursively builds smaller FP-trees that are eventually used to obtain the actual frequent itemsets. Many optimizations to FP-growth have been proposed; see Section 5 for a brief overview.

The main disadvantage of FP-growth is its excessive memory requirement. Even for moderately sized datasets, the FP-tree may easily grow to billions of nodes. Recent research has shown that *out-of-core* processing techniques, which make use of disk-resident data structures, can handle such large datasets in principle. For example, Buehrer [6] proposes an approach that uses approximate hash sorting to initially partition the database, and builds the complete FP-tree piece by piece from these partitions. However, due to expensive disk accesses, out-of-core processing is generally about an order of magnitude slower than in-core processing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2011, March 22–24, 2011, Uppsala, Sweden.
Copyright 2011 ACM 978-1-4503-0528-0/11/0003 ...$10.00

In this paper, we show how to reduce the memory requirements of FP-growth by about an order of magnitude. This allows us to avoid out-of-core processing in many cases where it is needed with existing approaches, and reduces the cost of out-of-core processing in the remaining cases due to favorable access patterns. Our *in-core approach* does not deteriorate the runtime of the FP-growth algorithm when the dataset is small, but leads to significant improvements when the data is large. The key to memory efficiency are two data structures, one for each of the phases of FP-growth. The *CFP-tree* is optimized for the build phase and is based on structural changes to the FP-tree, a highly tuned physical representation by means of a ternary tree, and various lightweight compression techniques. The CFP-tree provides a high compression ratio with reasonably small compression and decompression cost. The additional cost of (de)compression is largely offset by better usage of memory bandwidth. After the initial build phase, the CFP-tree is transformed into a different data structure called the *CFP-array*;[1] the cost of this transformation constitutes only a negligible fraction of the overall FP-growth runtime. Since different access paths are required in the mining phase, the CFP-array uses an array-based physical representation of the FP-tree, and employs intelligent node ordering, indexing, and compression. During the mining phase, the process recurses: many more smaller "conditional" CFP-trees will be built and transformed into CFP-arrays; this makes memory efficiency the more crucial.

We conducted an extensive evaluation of our data structures with various datasets. We observed an order-of-magnitude decrease in memory consumption on all the data sets we experimented with, whether real world or synthetic. When applied to very large datasets, the reduced memory consumption leads to multiple order-of-magnitude performance improvements when compared to plain FP-growth. On large datasets, our approach outperforms the best (and highly optimized) frequent-itemset mining algorithms of the well-known FIMI repository [8].

The remainder of this paper is structured as follows. Section 2 describes preliminaries: the FP-growth algorithm, ternary-tree representations of the FP-tree, and lightweight compression techniques. Section 3 introduces the CFP-tree and the CFP-array. In Section 4, we report the results of our experimental evaluation. Section 5 gives an overview of related work, and Section 6 concludes the paper.

2. PREREQUISITES

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 *support* of an *itemset* $I \subseteq \mathcal{I}$ is the number of transactions that contain I; an itemset is *frequent* if its support exceeds some minimum support ξ. The goal of frequent-itemset mining is to find all itemsets that are frequent. In this section, we briefly review the FP-growth algorithm for frequent-itemset mining, its basic data structure, the FP-tree, as well as a physical representation of the FP-tree by means of a ternary tree. We also discuss lightweight compression techniques that will serve as the basis for our CFP-tree.

[1]As discussed in Section 5, the CFP-array is a different data structure than the similarly-named FP-array of [16].

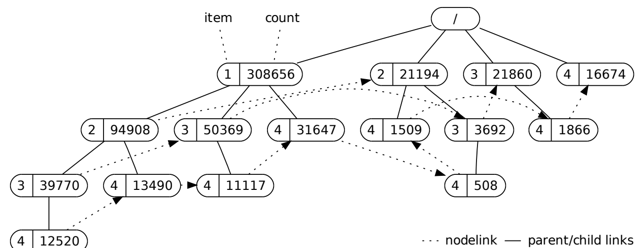


Figure 1: An FP-tree

2.1 FP-Growth and the FP-Tree

FP-growth is a divide-and-conquer algorithm that consists of two recurring phases: build and mine. The *build phase* transforms the database into a representation that is well-suited for mining: the *FP-tree* (for frequent-pattern tree). Figure 1 shows an FP-tree built from a FIMI dataset [8] with minimum support of $\xi = 80{,}000$. The FP-tree is constructed in two passes over the database. The first pass counts the support of each individual item; only frequent items and their support (count values) are retained. In the example, items 1, 2, 3, and 4 are frequent and have counts $f_1 = 308{,}656$, $f_2 = 116{,}102$, $f_3 = 115{,}691$, and $f_4 = 89{,}331$, respectively. The database is then scanned a second time and the FP-tree is built. The items in each transaction are sorted in descending order of their support; the FP-tree is simply a prefix tree on the sorted transactions enriched with some additional information. Apart from the item itself, this information includes the number of times each prefix has been encountered (`count`) and links for efficient navigation. For example, leaf node $(4 \mid 13490)$ represents prefix $(1, 2, 4)$; this prefix occurred 13,490 times in the database.

To obtain the support of a specific itemset I, we add up the counts of the prefixes that both contain I and end with the least frequent item in I. In order to do this efficiently, all occurrences of a (least frequent) item are connected using so-called `nodelink`s (dotted lines in Figure 1) and each node has a link to its `parent` (solid lines). For example, the support for itemset $\{ 3, 4 \}$ is given by summing up the frequencies of prefixes $(1, 2, 3, 4)$, $(1, 3, 4)$, $(2, 3, 4)$, and $(3, 4)$ (this gives 26,011). The *mine phase* of FP-growth proceeds more intelligently than just described: It repeatedly (1) picks the least frequent item a in the tree (until empty), (2) recursively runs FP-growth on the "conditional" FP-tree built from the prefixes that contain a, (3) uses the result to construct all frequent itemsets that contain a, and (4) removes a from the FP-tree. More details about this conditioning process can be found in [14]. For the purpose of this paper, note that both parent links and nodelinks are only traversed in the mine phase.

2.2 Physical Design

An important part of all prefix tree-based algorithms is the design and implementation of the physical representation of the tree. During the build phase, one wants to quickly find (or create) the prefix corresponding to the current transaction. In what follows, we refer to the children of a node in the FP-tree as the *direct suffixes* of the node. Thus, to insert the transaction $\{ 2, 4 \}$ into the FP-tree shown in Figure 1, we have to find the node of item 2 among the direct suffixes of the root, and then item 4 among the direct suffixes of

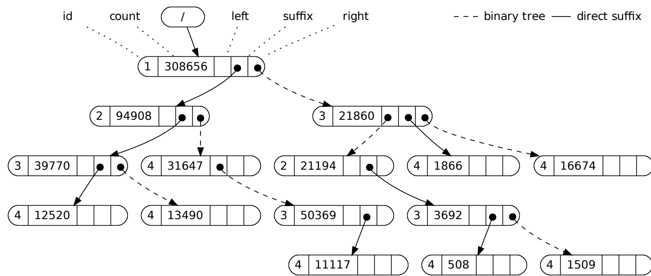


Figure 2: A ternary FP-tree

the selected node. The counts of both nodes are increased by one. Unless implemented carefully, a significant part of the overall runtime is spent on the search for the next node among the direct suffixes.

In this paper, we use a physical representation based on a ternary tree [25].[2] The main idea of a ternary tree is to arrange direct suffixes in a binary search tree; this requires two additional *left* and *right* pointers for each node. A *suffix* pointer connects each node with the binary tree of its direct suffixes. Thus, a node within the ternary tree has fields `id`, `count`, `parent`, `nodelink`, `left`, `right`, and `suffix`. Figure 2 shows a ternary search tree for the FP-tree of Figure 1. For readability, parent pointers and nodelinks have been omitted. Note that left and right pointers connect nodes within a level of the FP-tree; suffix pointers move down one level.

Search in a balanced binary search tree has logarithmic complexity, but knowledge of count values can be used to construct near optimal search trees. Since each node has the same size, the ternary tree also facilitates simple and efficient memory management without fragmentation. Its major drawback, however, is the high memory consumption: the five pointers alone require 40 bytes per node on a 64-bit systems. On trees with billions of nodes, this high memory requirement forces out-of-core computation and thus severely slows down computation. It is the impetus of this paper to drive down this memory requirement while maintaining fast execution times.

2.3 Lightweight Compression

It will become evident later on that the FP-tree has a huge compression potential. The challenge, of course, is to select compression techniques that balance compression ratio and runtime overhead accordingly. We found that entropy-based compression as well as all bit-level compression techniques have too high runtime overhead. This overhead is magnified because the FP-tree is traversed many times. For this reason, we focus on byte-level static encodings like null suppression or variable byte encoding.

The basic idea of null suppression is to substitute null values or successive zeros with a description of how many zeros are omitted. Perhaps the simplest form of null suppression is used to tag missing fields in a database record using *presence bits* stored up front [24]. Another version of null suppression—proposed by Westmann [31] to compress "small integers"—removes leading zero bytes of an integer value and instead stores the number of eliminated bytes. There are two variants of this *leading zero-bytes suppression*. For 32-bit integers, the first variant uses 3 bits to encode the number of suppressed zero bytes (0–4), followed by the remaining (non-zero) bytes. For example, hexadecimal value 00000090 will be encoded as binary value 01110010000. The second variant omits the first bit of the compression mask and stores the least significant byte even when zero; it is preferable when 0 values are encountered infrequently.

Variable byte encoding (also known as varint128 or 7-bit encoding) works by splitting an n-bit integer into a sequence of $\lceil n/7 \rceil$ 7-bit blocks. This sequence is stored in $\lceil n/7 \rceil$ successive bytes, in which the lower 7 bits are used to store the block, and the highest bit is a continuation bit which indicates whether or not an additional block follows (i.e., it is set to 0 if the sequence ends). For example, the hexadecimal value 00000090 is encoded using the following 2 bytes: 1000000100010000. Compared to leading zero-byte suppression, variable byte encoding achieves better compression ratios for small values (< 128) and avoids a separate compression mask for storing the number of suppressed zeros. However, variable byte encoding has usually higher decompression costs for large values (≥ 128), and lookup of the length $\lceil n/7 \rceil$ of a compressed value is not possible without decompression.

Every bit or byte saved when storing each node of the FP-tree has significant impact on the overall memory consumption on large trees with billions of nodes. Since the various static encodings differ in compression ratio and runtime cost, we opt to employ different compression techniques for different parts of the CFP-tree and the CFP-array. In fact, each of the above techniques will play an important role.

3. COMPRESSED PREFIX TREES

As mentioned previously, the *CFP-tree* is a variant of the FP-tree tailored to the build phase of FP-growth. By design, most of the fields stored in a CFP-tree node have small values so that lightweight compression is very effective. We exploit this compression potential in an optimized ternary representation, the *ternary CFP-tree*. After being built, the CFP-tree is converted to an *CFP-array*, which are tailored to the mine phase. The CFP-array draws from intelligent node ordering, indexing, and variable byte suppression.

3.1 Compression Potential

To get some intuition about the size of an uncompressed FP-tree and its compression potential, we exemplarily analyze the ternary-tree representation of an FP-tree built for the Webdocs dataset of the FIMI repository [19]. With a minimum support of 10%, the FP-tree is built from 1,661,662 transactions with an average length of 46.78 distinct items. The tree comprises 50,407,635 nodes, each consisting of seven 4-byte fields (we used 32-bit pointers), and requires 1.4GB of memory. Thus even a relatively small dataset and a high value of minimum support may lead to substantial memory consumption.

The in-memory representation of our example FP-tree is summarized in Table 1. As can be seen, most of the space is spent for storing zero bytes. We make the following observations:

- Roughly 53% of the total memory consists of zero bytes. In most cases, four out of the seven fields in

[2]Most of our techniques can be applied to other physical designs, such as the binary tree representation of [12].

Leading 0s →	0	1	2	3	4
`item`	0%	0%	2%	**98%**	0%
`count`	0%	<1%	<1%	>**99%**	0%
`nodelink`	**67%**	**33%**	<1%	0%	0%
`parent`	**66%**	**34%**	<1%	0%	0%
`suffix`	**65%**	**32%**	<1%	0%	3%
`left`	<1%	<1%	0%	0%	**99%**
`right`	<1%	<1%	0%	0%	**99%**

Table 1: Fields and number of leading zero bytes in an FP-tree for the Webdocs data

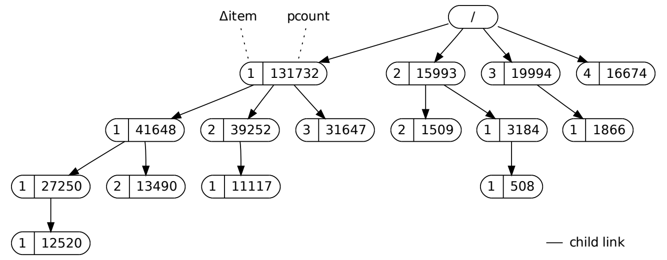


Figure 3: A CFP-tree

each node start with 3 or 4 leading zero bytes; the remaining three fields have none or only one leading zero byte.

- The `item` field, which stores identifiers rather than actual items, ranges from 1 to 262 so that two of the field's four bytes are always zero; only 2% of the nodes actually require two bytes to store the item.
- Almost all nodes require only one byte for the `count` field (but the maximum value is 1,429,525).
- Only one third of the forward-traversal pointers `left`, `right`, and `suffix` are non-zero. This is because every ternary tree with n nodes has $2n+1$ zero pointers and $n-1$ non-zero pointers.[3] In contrast, the backwards-traversal pointer `parent` and the sidewards-traversal pointer `nodelink` are rarely zero.
- About two-thirds of all non-zero pointers require 4 bytes of memory; most of the remaining pointers require 3 bytes. This is true even though we used a virtual address space so that pointer values start at zero and are tight.

We observed similar patterns in our experiments with other datasets. As argued below, the CFP-tree improves the compression potential further and is guaranteed to be effective under mild assumptions on the data.

3.2 The CFP-tree

This section introduces the CFP-tree; its physical representation is discussed in the next section. The main goal of the CFP-tree is to *make values as small as possible* so that the effectiveness of static encoding is maximized. The structure of the CFP-tree is identical to the structure of the corresponding FP-tree, but the information stored in its nodes is different. We illustrate our discussion using the CFP-tree shown in Figure 3; it corresponds to the FP-tree of Figure 1.

In what follows, we only describe differences between the FP-tree and CFP-tree. The first such difference concerns the link structure. An analysis of FP-growth shows that the build and mine phases require disjoint sets of links. In fact, to build an FP-tree, only *child links* are necessary because traversal during insertion is from top (root) to bottom (inner node or leaf). Since the CFP-tree is designed for only the build phase, we can safely omit parent links or nodelinks. As shown in Figure 3, this leads to a significant reduction of the number of links. Note that omitting these links is not beneficial for the FP-tree itself because they are needed during the mine phase. In contrast, the CFP-array handles parent and nodelinks in a different fashion; they are reconstructed (but not stored) during the transition from the CFP-tree to the CFP-array.

The remaining differences concern the data stored in each node. Since an item may occur many times in the FP-tree, both the FP-tree and the CFP-tree make use of item identifiers instead of working with the actual items. A supplementary dictionary is used to store the mapping between identifiers to actual items. We assign identifiers in increasing order of item frequency: The k-th most frequent item gets identifier k. With this choice of item identifiers, frequent items require less bits to store; e.g., each of the 256 most frequent identifiers can be stored in just one byte. Under the assumption that item frequencies are skewed (which makes frequent itemset mining useful), the space savings are significant. Moreover, the value of the identifier stored in each node is expected to be close to the identifiers stored in its children; cf. Figure 1. This property is exploited in the CFP-tree, which replaces the `item` field of the FP-tree by a Δ`item` field. The Δ`item` field stores only the difference of the item identifier from the respective parent node. To obtain the actual identifier, simply accumulate the values of all Δ`item` fields on the path from the root node to the node of interest. For example, node (4 | 13490) of the FP-tree of Figure 1 has item identifier 4. The corresponding node in the CFP-tree of Figure 3 is labeled (2 | 13490); its item identifier can be reconstructed by adding the Δ`item` fields of nodes (1 | 131732), (1 | 41648), and (2 | 13490).

Perhaps surprisingly, delta encoding is not beneficial for count values. Although the counts along the path from the root to a leaf form a non-increasing integer sequence, the count of a node is often much smaller than, and very rarely equal to, the count of its parent; Figure 1 illustrates this property. In fact, the delta to the parent is often greater than the actual count value. Moreover, storage of delta counts would lead to decreased efficiency when updating the tree: The delta counts of a large number of nodes would have to be modified to incorporate an insertion or update.

Instead of using deltas, the CFP-tree employs *non-cumulative count values*. It replaces the `count` field of the FP-tree by a `pcount` field (for partial count). Recall that the insertion of a prefix into the FP-tree increases all `counts` along the path corresponding to the inserted prefix by 1 (because all these paths are contained in the transaction). Instead, insertion into the CFP-tree increases only the `pcount`

[3] Proof: A node added to a ternary tree transforms a zero pointer into a non-zero pointer, but also adds three new zero pointers. The smallest ternary search tree with 1 node has three zero pointers.

Leading 0s →	0	1	2	3	4
Δ`item`	0%	0%	0%	**100%**	0%
`pcount`	0%	0%	<1%	3%	**97%**

Table 2: Fields and number of leading zero bytes in the CFP-tree for the Webdocs data

field of the *final node* in the path. One can show that the value of the `count` field of a node in an FP-tree is equal to the sum of the `pcount` fields of the corresponding CFP-tree node *and all its children*. Thus, for every corresponding pairs of nodes, we have `count`≥`pcount`. For example, consider node (1 | 41648) in Figure 3. To obtain the `count` of the node, take its `pcount` value add the `pcount` values of nodes (1 | 27250), (1 | 12520), and (2 | 13490). This gives 94,908, which equals the count of the corresponding node in Figure 1. Note that the sum of all non-cumulative count values of a CFP-tree is equal to the number of transactions that generated the tree. Often, the CFP-tree has many more nodes than transactions so that the average value of the non-cumulative count is less than 1; this makes lightweight compression extremely effective.

Table 2 shows the distribution of leading zeros for the fields in the CFP-tree built on the Webdocs dataset. When compared to Table 1, one observes that the `pcount` field is almost always zero, while `count` is never zero. This means that there is a huge number of nodes that do not correspond to a full transaction in the database (these nodes correspond to subsets of transactions). We also observe that delta coding of item identifiers shows only a minor improvement on this particular dataset. This is because the number of frequent items in this dataset is very low (just 262); significant improvements can be observed for smaller support values or datasets with more distinct items.

3.3 The Ternary CFP-tree

The ternary CFP-tree is a highly-compressed physical representation of the CFP-tree. It exploits the fact that the CFP-tree contains many small values. Each node in the ternary tree consists of compressed versions of the data fields from the CFP-tree (Δ`item`, `pcount`) and, in most cases, the relevant pointer fields of the ternary tree (`left`, `right`, `suffix`). We use different compression techniques for the various fields so as to minimize the overall memory consumption. Moreover, we reduce the number of pointers by embedding small leaf nodes into their parents and representing "chains" in the tree as arrays.

Since the data fields Δ`item` and `pcount` usually have very small values, we compress them using leading zero byte suppression. Recall that leading zero byte suppression makes use of a compression mask that indicates how many zero bytes have been omitted. The length of this mask is either 3-bit (0–4 bytes omitted) or 2-bit (0–3 bytes omitted). In all our experiments, the `pcount` value was equal to zero for the majority of the nodes in the CFP-tree; cf. Table 2. In contrast, the Δ`item` field is arguably never 0. Thus, a 3-bit compression mask pays off for `pcount`, while a 2-bit compression mask is sufficient for Δ`item`. Note that apart from increased compressibility, usage of non-cumulative counts also reduces the (de)compression overhead because only a single `pcount` field has to be updated when inserting a prefix that already exists in the tree.

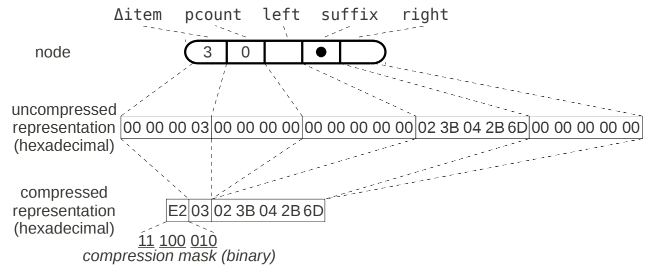


Figure 4: A CFP-tree node and its in-memory representation

We reduced the size of each of the three pointer fields `left`, `right`, and `suffix` from 64 bits to 40 bits, which is sufficient to address 1TB of main memory. Furthermore, all three pointer are compressed using null suppression, i.e., we use a presence bit to indicate whether or not a pointer is null. This amounts to 3 presence bits in total. The pointer fields of a ternary CFP-tree with n nodes compress to a fraction of $\frac{n-1}{3n}$ of the size of the respective fields in the ternary FP-tree (since there are $n-1$ non-zero pointers). For large n, the size of the pointers in the ternary CFP-tree averages to 43 bits, while the size of the pointer in the ternary FP-tree averages to 120 bits (also assuming 40-bit pointers).

To summarize, a *standard node* of the ternary CFP-tree is stored as follows. The first byte holds the compression mask for all fields ($2+3+3=8$ bits). The subsequent bytes hold Δ`item` and `pcount` field without leading zero bytes, followed by 0–15 bytes containing the non-zero pointers. Figure 4 gives an example of ternary CFP-tree compression as discussed thus far. The first two bits of the compression mask are set to 11 and indicate that Δ`item`=3 has three leading zero bytes. The next three bits of the compression mask are set to 100 because `pcount`=0. The remaining bits are set to 010 and indicate zero values for the `left` and `right` pointers and a non-zero value for the `suffix` pointer. The compressed node requires 7 bytes in total.

In addition to standard nodes, the CFP-tree comprises *embedded leaf nodes* and *chain nodes*. Embedded leaf nodes are based on the observation that the smallest possible standard node requires just 3 bytes: one byte for each the compression mask, the Δ`item` field, and the `pcount` field, and zero bytes for the pointers (i.e., the node is a leaf). Since this is less memory than required for storing a pointer, we embed such small leaf nodes within the respective pointer field of their parents. The first byte of the pointer field is used to distinguish between a pointer (≠255) and an embedded leaf node (=255); our memory manager never uses memory addresses that start with 255. All leaves with 0 ≤Δ`item`< 256 and `pcount`< 16,777,216 are stored in the parent. Thus an embedded leaf consists of a marker byte, one byte for the value of Δ`item`, and three bytes for the value of `pcount`. We found that in particular for datasets with small average transaction length (i.e., datasets with a CFP-tree of low height), node embedding achieves significant memory reduction.

Interestingly, experiments on several datasets with low minimum support suggest that often more than 90% of the nodes in a CFP-tree follow exactly the node layout of the example of Figure 4: a Δ`item` value lower than 256, a zero `pcount` value, zero `left` and `right` pointers, and a `suffix`

pointer. Note that the suffix pointer consumes most of the space in the node. This gives rise to a further optimization technique in the form of *chain nodes*, which were originally proposed for Patricia tries [15]. The key idea is as follows: Whenever there is a chain of multiple nodes of the pattern described above, we store just the item identifiers of the nodes in the chain. To indicate the presence of such a chain node, we set all three bits of the `pcount` compression mask to one. This reduces the number of possible states of `pcount` in a standard node, but forms no problem in practice[4]. The remaining 5 bits of the compression mask store the number of nodes in the chain, followed by that many bytes each containing one Δ`item` value. The chain node is terminated by a `suffix` pointer that connects it to the child of the last node in the chain. In total, a chain node requires $m + 6$ bytes for a chain of length m. For example, if six FP-tree nodes are merged into a single chain node, then the space consumption of each node averages to only 2 bytes.

3.4 The CFP-array

The CFP-array is a data structure optimized for the mine phase of FP-growth. Recall that the mine phase performs both sidewards and backward traversal and requires quick access to the value of `item` and `count`. To provide the navigational and data information, the CFP-array consists of an array of compressed triples and a small index. One of the key observations behind this layout is that the CFP-array is static, i.e., built once and never changed. We can thus safely ignore update costs. In what follows, we describe how to build the CFP-array from a standard FP-tree. The conversion process from a ternary CFP-tree is slightly more involved and postponed to Section 3.5.

Consider a representation of the FP-tree as an array of quadruples of form (`item`, `count`, `parent`, `nodelink`). An FP-tree with n nodes corresponds to an array with exactly n elements. Observe that the order of the FP-tree nodes in the array does not matter, but that nevertheless each node has a certain *position* in the array. The CFP-array exploits this fact by picking a node-to-position mapping that allows us to omit the `nodelink` field.

We assign positions to nodes such that for any pair of nodes `u` and `v` with `u.item<v.item`, the position of `u` is smaller then the position of `v`. This property ensures that the `item` field is non-decreasing and that all nodes corresponding to a particular item are clustered together; they form consecutive subarrays within the CFP-array. Since the purpose of the `nodelink` field is to connect nodes that correspond to the same item, and since we gave all those items consecutive positions and thus made them easy to find, the `nodelink` field is redundant and can be omitted. Instead, we maintain a small *item index* (also in the form of an array) that maps each distinct item to its first occurence in the CFP-array; we call this position the *starting position*. Sidewards traversal is performed by processing the nodes from the starting position onwards. The traversal ends when a node with a different item is encountered. Observe that the item index also contains all the information that is required to determine the item of a node at some position i: It is the item that with the largest starting position less than or

[4]The remaining seven states allow `pcount` values taking 0–6 bytes, which is sufficient for all datasets in which the most frequent item occurs less than 2^{44} times.

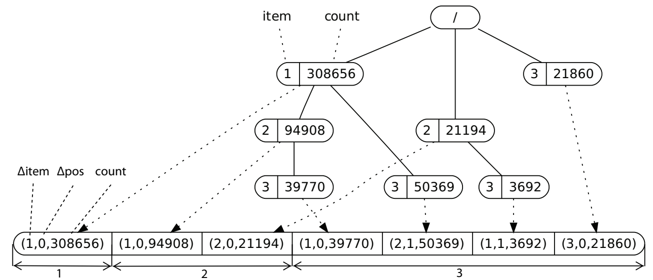


Figure 5: Conversion of an FP-tree to a CFP-array

equal to i. Thus, the `item` field could potentially be omitted as well, but we chose not to do so for performance reasons.[5].

Just as done in the CFP-tree, we increase the compression potential of static encodings by making the fields of the CFP-array as small as possible. As before, we replace the `item` field by the difference Δ`item` to the item identifier of the parent. To obtain the actual identifier of a node during a backward traversal, subtract all Δ`item` values on the path from the leaf node to the node of interest (excluding its Δ`item`). The identifier of the leaf node itself is obtained via the item index. To encode the `parent` field succinctly, we do not store the parents global position in the CFP-array. Define the *local position* of a node as its position within its subarray, i.e., the subarray that contains all the nodes with the same item. We replace the `parent` field by a Δ`pos` field, which contains the delta between the local position of the child and the local position of its parent. The conversion of the CFP-tree into the CFP-array is performed such that this delta encoding is very effective. The final field of each triple is the `count` value. In contrast to the CFP-tree, we do not store partial counts because we do not have direct access to the descendants of a node, and thus cannot efficiently reconstruct counts from partial counts.

An example FP-tree and the corresponding CFP-array are shown in Figure 5. There are three subarrays, one for each of the three distinct items. Node (3 | 3692) is stored in the subarray for item 3 at local position 2 (we use 0-based local positions). Its parent is (2 | 21194), which is stored at position 1 in its subarray. Thus, we have Δ`item` $= 3 - 2 = 1$ and Δ`pos` $= 2 - 1 = 1$. Note that node (2 | 21194) does not have a parent; this is indicated by setting Δ`item` = `item` = 2.

We compress each of the three fields (Δ`item`, Δ`pos`, `count`) using variable byte encoding in the order specified. We prefer this technique because it requires no separate compression mask, and because both Δ`item` and `count` require almost always just one byte (c.f. Tables 1 and 2). The field order allows us to perform backwards traversal without decoding the `count` field; we thus avoid the expensive length lookups associated with variable byte encodings. Note that since each array element is stored using a variable number of bytes, we use local positions that refer to the byte offset of a node within its subarray in the physical representation. This allows us to directly access a node given its local position.

[5]If we omit the `item` field, we have to reconstruct it for each backward traversal via binary search in the item index.

3.5 Conversion

Conversion of the ternary CFP-tree to the CFP-array is required in between every build phase and its subsequent mine phase. We perform this transformation in two passes over the ternary CFP-tree. The first pass is required to determine the size of the CFP-array, while the second pass is used for the actual transformation. The transformation is not performed in-place so that the main memory has to be large enough to hold both data structures simultaneously. Fortunately, this does not increase the peak memory consumption because (1) FP-growth will produce many more FP-trees in the mine phase, and (2) the CFP-tree can be discarded immediately after the conversion, i.e., before these FP-trees are created. Therefore, the memory that stores the CFP-tree is reused by FP-growth during the mine phase. Furthermore—as shown in our experiments—even in cases where the main memory is not large enough to hold both data structures simultaneously, the performance deterioration due to thrashing is low due to favorable memory access patterns during conversion. Our experiments indicate that the conversion consumes only a small fraction of the overall execution time of FP-growth.

The conversion process is illustrated in Figure 5. The size and starting position of each subarray in the CFP-array can be determined via a recursive depth-first traversal of the CFP-tree. For each node, we compute the space consumption of the variable byte encodings of the `count`, Δ`pos`, and Δ`item` fields and accumulate it individually for each item. The transformation itself also performs a recursive depth-first traversal, and writes each node directly into its final position within the CFP-array. This position is easily determined from the information gathered in the first pass. The Δ`pos` fields are obtained via a stack that contains the path taken from the root to the current node during the traversal.

The transformation has a good memory access behavior because (1) each triple is written only once into the CFP-array and, more importantly, (2) triples are written sequentially within each of the n subarrays. For this reason, the operating system must hold only n physical pages for the CFP-array so that trashing is avoided.

4. EXPERIMENTS

We ran experiments with different algorithms for frequent-itemset mining, and different data structures for FP-growth. We will refer to the combination of FP-growth with the CFP-tree and the CFP-array as *CFP-growth*. Our experiments suggest that CFP-growth requires significantly less main memory than plain FP-growth. The positive effect on runtime is minor when the dataset is small, but significant for large data. Compared to existing algorithms, we found that our data structures make CFP-growth the algorithm of choice for in-core processing. When the dataset is large, CFP-growth is multiple times faster than the fastest previous algorithms.

4.1 Experimental Setup

All experiments were run on a machine with an Intel Core i7-920 processor (2.67GHz), 6GB of physical memory, and a 64-bit Linux operation system. We implemented the ternary CFP-tree, the CFP-array and the complete CFP-growth algorithm in C++. We used the GCC compiler with standard optimization flags. Like all efficient frequent-itemset mining algorithms, we make use of a simple memory manager (see Appendix A), which (1) avoids expensive `malloc` calls when creating new nodes, (2) reduces the pointer size, and (3) provides unpadded chunks of memory. For CFP-growth, the memory manager allocates two separate chunks of 8GB virtual memory; thrashing occurs when the actual memory usage exceeds 6GB. The first chunk is used for storing ternary CFP-trees (one at a time), the second chunk is used for CFP-arrays (multiple at a time).

Name	# TA	Avg. card.	Distinct items	Size
Quest1	25M	100	20k	13GB (33GB)
Quest2	50M	100	20k	26GB (65GB)

Table 3: Summary of datasets

We restrict the maximum length of a single chain node to 15; longer chains are broken into multiple chain nodes. Although longer chains may have lower memory requirements, we found that both runtime overhead as well as memory fragmentation increase. We perform construction of new chain nodes only when a new leaf is inserted; we do not merge existing chain nodes. Note that chain nodes may be split when subsequent transactions are inserted.

To evaluate the compression ratio of CFP-growth, we used all of the real-world datasets in the FIMI repository [8]. The FIMI repository has been created for the purpose of evaluating different frequent-itemset mining algorithms; it contains both data and implementations. Provided datasets include retail [5], connect [8], webdocs [19], and accidents [9]. The size of the raw data varies from 335kB to 1.4GB.

For our performance experiments, we generated two synthetic datasets generated with the IBM Quest Dataset Generator [2]. All datasets were stored in plain text files, which follow the standard FIMI format. Each line consists of a list of items that constitute a single transaction. The average storage space per occurrence of each item is below 6 bytes. Table 3 summarizes the characteristics of the synthetic datasets. The size in brackets denotes the size of the IBM Quest output before conversion to the FIMI file format.

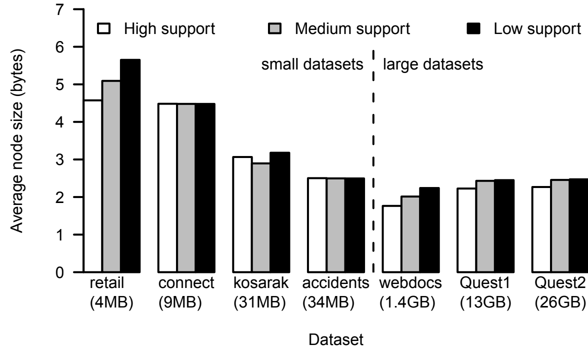
An important part of the frequent-itemset mining is data input. We implemented asynchronous double buffering, i.e., we work with two input buffers: one that is being processed and one that is being loaded from disk. We found that the build phase of the *initial* ternary CFP-tree is still I/O bound.[6] This would even be true if we replaced the text-based input files by binary files, which corresponds to a file size reduction of roughly 40%.

4.2 Compression Ratio

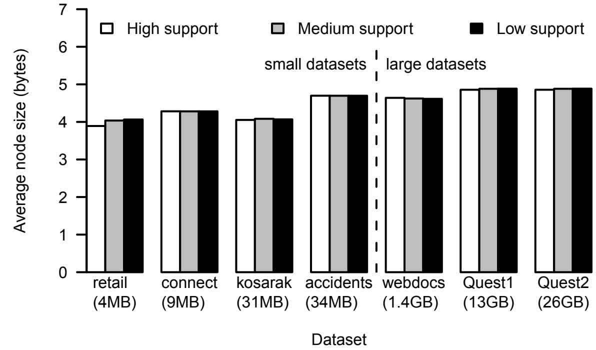
The purpose of this set of experiments is to evaluate the compression ratio of our data structures on a large variety of real world datasets.

In the first set of experiments, we measured the average node size in bytes for both the ternary CFP-tree and the CFP-array. The average is taken over the number of nodes in the corresponding FP-tree; in the ternary CFP-tree, the number of nodes may be less because multiple FP-tree nodes may be merged into a single chain node.

[6] Our hard disk has a maximum bandwidth of 108MB/s (measured using hdparm).



(a) Average CFP-tree node size



(b) Average CFP-array node size

Figure 6: Average node sizes for different datasets

We determined the node size of state-of-the-art implementations of FP-growth by investigating source code. We found that each node requires 40 bytes of main memory in most implementations, including [12]. During compilation, the compiler performs alignment optimization which increases node size to 48 bytes; this behavior can be avoided with appropriate modifications of the source code. We assume such a modification, and take 40 bytes per node as the baseline.

Figure 6(a) shows the average node size of a ternary CFP-tree built from the retail, connect, kosarak, accidents, webdocs, Quest1, and Quest2 datasets with various choices of minimum support ($\xi_{high} = 0.31\%$, $\xi_{medium} = 0.07\%$, $\xi_{low} = 0.01\%$). We omit results for the other FIMI datasets, which have similar characteristics. The ternary CFP-tree is most effective for the webdocs dataset, which benefits heavily from node chaining. On this dataset, the space consumption per node averages to about 2 bytes. The lowest observed average node size of 1.56 bytes is obtained with a minimum support of $\xi = 2.5\%$ (not shown). This corresponds to a 25x size reduction compared to the standard FP-tree. When the minimum support decreases, the tree size increases and the ternary CFP-tree branches out more. Therefore, the tree has less or shorter chains so that the average node size increases slightly. This behavior can also be observed on the retail dataset, which exhibits the highest average node size of 5.7 bytes. This corresponds to a 7x reduction of memory consumption.

Results for the CFP-array are given in Figure 6(b). Here, the average node size is dominated by the Δpos field. This field requires the most space in the webdocs, Quest1, and Quest2 datasets. For all datasets, the average node size is below 5 bytes; it is close to 4 bytes in the best cases. These node sizes correspond to an 8x–10x reduction of memory consumption.

4.3 Build Phase and Conversion

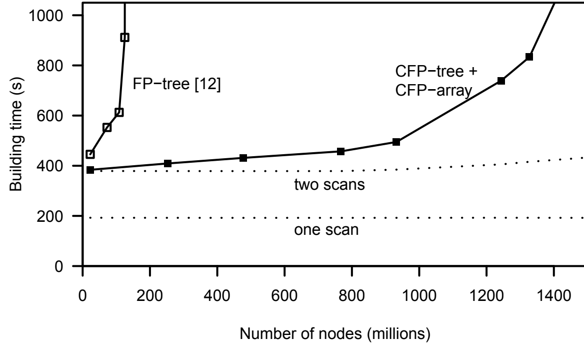
We next investigated the build phase of CFP-growth, including the conversion of the ternary CFP-tree to the CFP-array. For this set of experiments, we use the Quest1 dataset; the datasets from the FIMI repository are so small that no memory bottleneck arises. We compare our algorithm against the best-performing FP-growth implementation—proposed by Grahne and Zhu [12]—of the FIMI repository. Other implementations and algorithms are discussed in Section 4.5.

Figure 7(a) shows the time to build the ternary CFP-tree and convert it to the CFP-array (for CFP-growth) or just the time to build an FP-tree (for FP-growth). We varied the minimum support ξ during the experiments, but show the corresponding number of FP-tree nodes on the x-axis. This presentation highlights the behavior of the algorithms under memory pressure. We also plot the time required for scanning the data. Recall that FP-growth performs two scans: The first scan counts the frequency of each item, while the second scan builds the prefix tree. We found that FP-growth and CFP-growth perform similarly for small prefix trees. The slight advantage of CFP-growth may be explained by our usage of double buffering; adding double buffering to the FP-growth implementation may also speed it up slightly. Note that the extra runtime of CFP-growth for conversion to the CFP-array is relatively small. For example, the conversion of a ternary CFP-tree with 150M nodes requires 22s; a single scan of the data alone takes 186s.

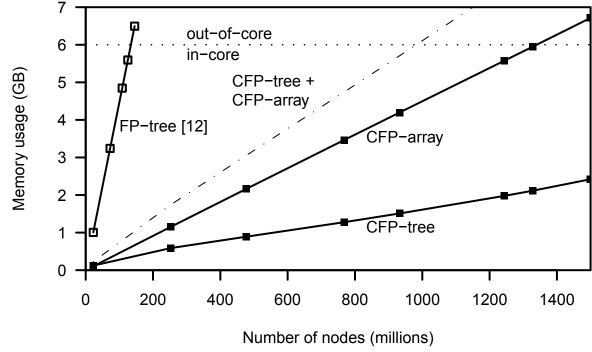
As the tree size increases, the build time of the FP-tree explodes. This is because the tree size quickly exceeds the available main memory (6GB) so that thrashing occurs; see Figure 7(b). In fact, for a tree size of about 145M nodes, the FP-tree required 6.5GB of main memory and took roughly 8.5h to build. In contrast, CFP-growth can process much larger trees without thrashing. The CFP-tree and the CFP-array together hit the main memory limit for a tree size of roughly 950M nodes. As we go beyond this size, the runtime of CFP-growth starts to increase more quickly but remains acceptable. For example, a prefix tree of 1,300M nodes corresponds to a 2.0GB CFP-tree and a 6.0GB CFP-array. Still CFP-growth requires only about 800s to build and convert the tree. To see why, observe that the CFP-tree fits entirely into main memory so that thrashing only occurs during conversion. Memory access during conversion is almost sequential so that the amount of thrashing is expected to be quite low. As we go to even larger tree sizes, thrashing increases heavily. For example, to build and convert a tree of 1.75B nodes, CFP-growth requires 4,000s.

4.4 Overall Performance

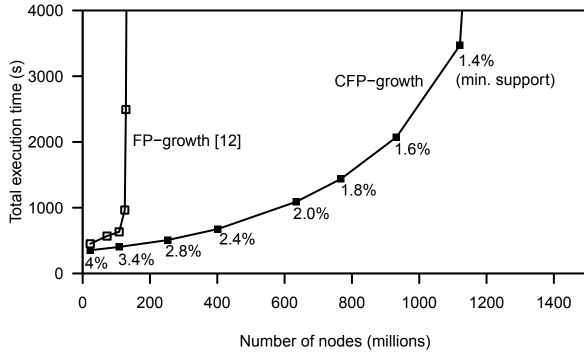
We move on to experiments that cover the overall performance of FP-growth and CFP-growth. As before, we compare our CFP-growth implementation with the best-performing FP-growth implementation of the FIMI repository; see the next section for a comparison with other algo-



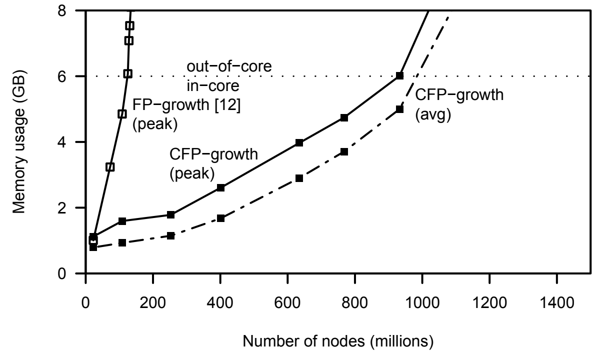
(a) Tree build time



(b) Tree memory requirements



(c) Total execution time



(d) Total memory requirements

Figure 7: Tree build phase and Full Frequent-Itemset Mining on Quest1

rithms. In contrast to the FP-growth implementation, our CFP-growth implementation does not perform any particular optimization of the mining step. We chose this approach because our focus is mainly on overall memory consumption. Note that we implemented the mining phase in a sequential fashion; it can be easily parallelized via sharing of its data structures (CFP-tree or CFP-array) across multiple processors.

The total execution time and memory consumption for various values of minimum support are shown in Figures 7(c) and 7(d), respectively. As before, the x-axis shows the tree sizes of the *initial* FP-tree corresponding to the choice of minimum support. For FP-growth we give the peak memory consumption. For CFP-growth, we instrumented the source code to also obtain the average memory consumption. As can be seen, CFP-growth outperforms FP-growth for all problem sizes. As mentioned previously, the performance advantage of CFP-growth for small tree sizes may be a consequence of our use of double buffering when building the initial tree. As tree sizes increase, FP-growth exceeds the available main memory and its overall execution time increases significantly. For example, at 135M nodes ($\xi = 3.24\%$), FP-growth has a peak memory consumption of 8.1GB and requires 8,696s to complete. For the same minimum support, CFP-growth has a peak memory consumption of 1.6GB and requires 438s to complete. This corresponds to a speed-up of 20x. For larger tree sizes (smaller support), the speed-up increases further.

CFP-growth scales well up to an initial tree size of 930M nodes. At this point, the peak memory consumption hits the main memory size of 6GB. Compared to FP-growth, which reaches this size at 130M nodes, CFP-growth can perform pure in-core processing for a 7.5x larger tree. Note that the non-linear increase of runtime and memory consumption of CFP-growth results from the fact that, as the minimum support decreases, more and more itemsets become frequent. To find these itemsets, more work has to be done and more conditional CFP-trees have to be created and stored in main memory.

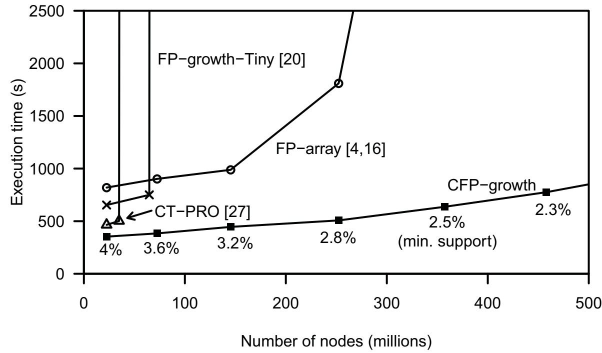
The overall behavior of both FP-growth and CFP-growth can be summarized as follows:

1. Best performance is achieved when all data structures fit in main memory.

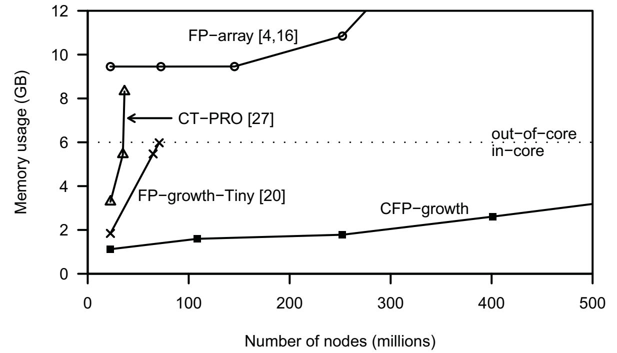
2. The performance degrades but still remains acceptable when only the working set of the data structures fit in main memory and the remaining data structures are swapped to disk.

3. A strong performance degradation occurs when the working set does not fit in memory.

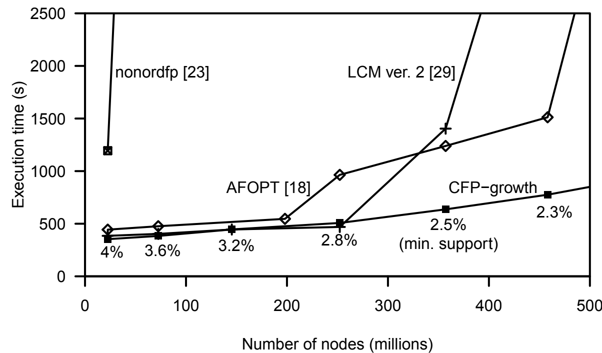
The main advantage of CFP-growth is that the transitioning from 1 to 3 occurs at a much slower pace.



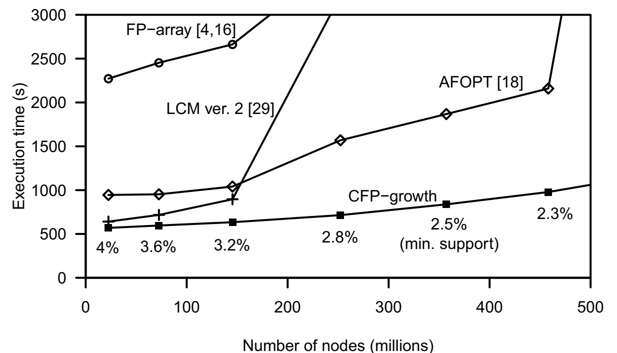
(a) Comparison to FP-growth-based algorithms on Quest1



(b) Peak memory consumption of FP-growth-based algorithms on Quest1



(c) Comparison to FIMI algorithms on Quest1



(d) Comparison with most competitive algorithms on Quest2

Figure 8: Comparison to other algorithms

4.5 Comparison With Other Algorithms

We compared CFP-growth to several alternative algorithms. Fast implementations of these algorithms have been taken from the FIMI repository [8] as well as the PARSEC benchmark suite [4]. We used both the Quest1 and Quest2 datasets. Some algorithms had very high execution times even with the highest minimum support used in our experiments ($\xi = 4.0\%$); for example, we stopped PatriciaMine [21] after 2 hours of working at the highest support level (it used only 1.3GB of main memory at this time). The remaining algorithms have been used in our experiments.

In a first set of experiments, we compared CFP-growth with the algorithms CT-pro [27], FP-growth-Tiny [20], and FP-array [16],[7] each of which improves FP-growth with certain optimizations. The execution time and peak memory consumption on the Quest1 dataset are shown in Figures 8(a) and 8(b), respectively. CFP-growth consistently outperforms all three FP-growth variants across all choices of minimum support. Both FP-growth-Tiny and CT-pro consume the entire available main memory (6GB) at a minimum support of $\xi = 3.6\%$ ($\approx$ 80M nodes), so that expensive out-of-core computation is necessary even when the minimum support is large. For example, FP-growth-Tiny required about 38 hours at this point. The FP-array implementation always requires more than the available main memory because it loads the complete dataset into main memory during the first scan. During the second (in-memory) scan, the FP-tree is built using the memory space of already processed input data. Our experiments indicate that the FP-array has high costs for the first two scans as well as the FP-tree to FP-array conversion, but it scales better than FP-growth-Tiny and CT-pro. FP-array's mining procedure goes out-of-core at a minimum support of $\xi = 2.6\%$. At this point, CFP-growth is already an order of magnitude faster.

Figure 8(c) shows the execution time of CFP-growth and the best-performing FIMI algorithms—nonordfp [23], LCM (ver. 2) [29] and AFOPT [18]—on the Quest1 dataset. The nonordfp implementation required about 1,200s at the highest minimum support of $\xi = 4.0\%$. At the slightly smaller value of $\xi = 3.6\%$, the time increases drastically to 12,000s. This is caused by the high memory requirements of nonordfp, which forces early out-of-core computation. LCM as well as AFOPT scale to larger trees. AFOPT is the slowest of the remaining algorithms. LCM and CFP-growth perform similar (with LCM being slightly faster) up to $\xi = 2.8\%$. From this point on, both LCM as well as AFOPT go out-of-core and their performance degrades accordingly. CFP-growth performs in-core computation up to $\xi = 1.6\%$ (c.f. Figure 7(d)). At that level, the CFP-tree contains 900M nodes; this is 3x larger than the out-of-core threshold for LCM and AFOPT.

[7]The FP-array implementation has been taken from the PARSEC benchmark suite [4].

We observe different trends in behavior on the larger Quest2 dataset, which has twice as many transactions. The results are shown in Figure 8(d). As before, the FP-array implementation has high execution time even for the large minimum support values; similar to Quest1, its mining procedure goes out-of-core at a minimum support of $\xi = 2.6\%$. Since the memory requirement of LCM depends directly on the number of transactions, the algorithm breaks down much earlier ($\xi = 3.2\%$). This dependency on input size renders LCM problematic for datasets with many transactions. In contrast, AFOPT shows similar performance as in the previous experiments, but CFP-growth shows better scalability. Comparing performance on Quest1 and Quest2 at $\xi = 3.2\%$, we find that AFOPT runtime increases by 50%, while the runtime of CFP-growth increases by only 25%. This behavior results from the fact that the cost of the mine phase are similar on both datasets, but the build phase is more expensive on Quest2. As before, AFOPT goes out-of-core for support values larger than $\xi = 2.5\%$. At this point, CFP-growth is significantly more efficient than both AFOPT and LCM.

5. RELATED WORK

Recent work on prefix-tree based frequent-itemset mining can be divided into four major areas: (1) memory reduction, (2) CPU reduction, (3) out-of-core processing, and (4) distributed FP-growth. Many of the approaches are orthogonal to our work; combinations may thus provide further improvement.

Research in (1) focuses on the reduction of the memory requirements of FP-growth. This paper belongs to this class. Özkural et al. propose FP-growth-Tiny [20], which does not create conditional FP-trees and performs all work on the initial big FP-tree. On large data, this initial tree is too large to fit in main memory. Sucahyo et al. propose two algorithms named CT-ITL [26] and CT-PRO [27], which both work with a compressed FP-tree structure that avoids repeated storage of similar subtrees. Its compression ratio is less than that of CFP-growth. Note that each of the aforementioned approaches achieves memory reduction by removing nodes, subtrees, or even complete FP-trees. In contrast, CFP-growth modifies the *physical design* of the tree. Pietracaprina et al. [21] use a Patricia trie to represent the base data. Our chain nodes are based on this idea, but we merge nodes slightly differently and use compression techniques. The CFP-array is inspired by the core data structure of nonordfp [23], which in the mine phase stores the `count` and `parent` fields of all nodes in two separate arrays. The CFP-array can be seen as refinement with better navigational properties and lower space consumption. nonordfp does not reduce memory in the build phase.

Approaches in class (2) adapt FP-growth to the characteristics of modern hardware. These approaches focus on small problem sizes, where memory bottlenecks are not an issue. They all use two different data structures for the build and mine phase of FP-growth; the separation of CFP-tree and CFP-array was inspired by this idea. In contrast to our approach, however, the CPU-optimized algorithms unroll the paths from each leaf node to the root. Therefore, the nodes of each path are stored continuously in memory. This procedure improves cache locality but *increases* the size of the FP-tree. Examples of such approaches include the CC-tree [10] and the FP-array [16]. Although these optimization significantly speed up the mining process, the CC-tree requires twice as much memory as the FP-tree [16], while the FP-array requires roughly the same amount of memory as regular FP-growth.

Research in out-of-core mining—class (3)—focuses on the case where the FP-tree does not fit into main memory. One approach is to use sampling [28]; this leads to approximate results so that some frequent itemsets may not be found. Buehrer [6] classifies out-of-core algorithms and proposes I/O-conscious optimizations that reduce the overhead of out-of-core processing. In contrast, we try to avoid out-of-core processing to the extent possible.

Research in class (4) adapts FP-growth to parallel or distributed processing. Li et al. [17] propose a distributed FP-growth implementation based on MapReduce. They divide the data into sets of "group-dependent transactions," which form independent FP-trees. Depending on the dataset, such a partitioning may or may not be effective. Buehrer et. al [7] propose a parallel implementation of FP-growth that minimizes I/O and communication overhead based on an efficient tree pruning method.

6. CONCLUSION

The discovery of frequent itemsets in large datasets is a fundamental task in data mining. Prefix-tree algorithms belong to the fastest of the available algorithms but have excessive memory consumption. This may force out-of-core processing, which significantly increases runtime. To address this problem, we introduce two highly-tuned data structures: the CFP-tree and the CFP-array. Both data structures exploit a combination of structural changes to the FP-tree and lightweight compression techniques. The resulting reduction of memory consumption—a factor of 7x–25x in our experiments—allows in-core processing for significantly larger datasets. No significant overhead is imposed when mining small datasets. Our algorithm thus constitutes an efficient solution for in-core frequent-itemset mining on a wide range of datasets.

7. REFERENCES

[1] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216, New York, NY, USA, 1993. ACM.

[2] R. Agrawal and R. Srikant. Quest synthetic data generator. *IBM Almaden Research Center*.

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

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, October 2008.

[5] 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.

[6] G. Buehrer, S. Parthasarathy, and A. Ghoting. Out-of-core frequent pattern mining on a commodity pc. In *SIGKDD*, pages 86–95, New York, NY, USA, 2006. ACM.

[7] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz. Toward terabyte pattern mining: an

architecture-conscious solution. In *PPoPP*, pages 2–12, New York, NY, USA, 2007. ACM.

[8] Frequent Itemset Mining Implementations Repository: http://fimi.cs.helsinki.fi/.

[9] K. Geurts, G. Wets, T. Brijs, and K. Vanhoof. Profiling high frequency accident locations using association rules. In *Proceedings of the 82nd Annual Transportation Research Board, Washington DC. (USA), January 12-16*, page 18pp, 2003.

[10] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *VLDB*, pages 577–588, 2005.

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

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

[13] J. Han. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[14] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1–12, New York, NY, USA, 2000. ACM.

[15] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[16] E. Li and L. Liu. Optimization of frequent itemset mining on multiple-core processor. In *VLDB*, pages 1275–1285, 2007.

[17] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang. Pfp: parallel fp-growth for query recommendation. In *RecSys*, pages 107–114, New York, NY, USA, 2008. ACM.

[18] G. Liu, H. Lu, and J. X. Yu. Afopt: An efficient implementation of pattern growth approach. In *In Proceedings of the ICDM workshop*, 2003.

[19] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Webdocs: a real-life huge transactional dataset. In *FIMI*, 2004.

[20] E. Özkural and C. Aykanat. A space optimization for fp-growth. In *FIMI*, 2004.

[21] A. Pietracaprina and D. Zandolin. Mining frequent itemsets using patricia tries. In *FIMI*, 2003.

[22] J. R. Punin, M. S. Krishnamoorthy, and M. J. Zaki. Web usage mining - languages and algorithms. In *In Studies in Classification, Data Analysis, and Knowledge Organization*, pages 88–112. Springer-Verlag, 2001.

[23] B. Rácz. nonordfp: An fp-growth variation without rebuilding the fp-tree. In *FIMI*, 2004.

[24] H. K. Reghbati. An overview of data compression techniques. *IEEE Computer*, 14(4):71–75, 1981.

[25] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[26] Y. G. Sucahyo and R. P. Gopalan. Ct-itl : Efficient frequent item set mining using a compressed prefix tree with pattern growth. In *ADC*, pages 95–104, 2003.

[27] Y. G. Sucahyo and R. P. Gopalan. Ct-pro: A bottom-up non recursive frequent itemset mining algorithm using compressed fp-tree data structure. In *FIMI*, 2004.

[28] H. Toivonen. Sampling large databases for association rules. In *VLDB*, pages 134–145, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[29] T. Uno, M. Kiyomi, and H. Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *FIMI*, 2004.

[30] J. T.-L. Wang, M. J. Zaki, H. Toivonen, and D. Shasha, editors. *Data Mining in Bioinformatics.* Springer, 2005.

[31] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

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

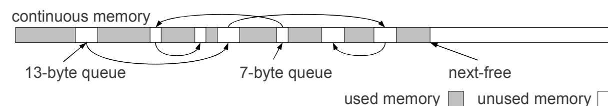


Figure 9: Illustration of our memory manager

APPENDIX

A. MEMORY MANAGEMENT

We implemented a simple but effective memory manager for CFP-growth. The memory manager is aware of the fact that (1) our compression scheme leads to different node sizes and that (2) node sizes may change as new transactions are processed (e.g., count values may grow, references to new nodes may be added). We exploit the fact that the memory footprint of a compressed node is within a fixed range, i.e., from 7 up to 24 bytes (three 40-bit pointers; 9 byte for Δ`item`, `pcount`, and compression mask); smaller nodes are stored as embedded nodes.

Figure 9 illustrates our memory manager. The memory is divided into two continuous parts: used memory (consisting of nodes and free memory chunks) and unused memory. A `next-free` pointer separates these two parts. In used memory, we connect all free memory chunks *of the same size* with a LIFO queue. The elements of the queue are stored directly in the memory chunks being managed. This is possible because the queue actually contains the locations of these free memory chunks, and only 5 bytes are needed to store such a location. Whenever a node grows/shrinks from b_1 bytes to b_2 bytes, we dequeue a b_2-byte chunk from the b_2-byte queue. If no such chunk exists, we create a new b_2-byte chunk at the `next-free` pointer, which we subsequently increase by b_2 bytes. We then store the updated node in the so-obtained chunk, and enqueue the node's old b_1-byte chunk to the b_1-byte queue (thereby freeing its memory).

Our memory manager avoids fragmentation but imposes overhead whenever a node shrinks or grows. For this reason, we made use of compression techniques that avoid fluctuations of the node sizes to the extent possible.