View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

A Performance Study of Three Disk-based Structures for Indexing and Querying Frequent Itemsets

Guimei Liu
School of Computing
National University of Singapore
liugm@comp.nus.edu.sg

Andre Suchitra
School of Computing
National University of Singapore
suchitra@comp.nus.edu.sg

Limsoon Wong
School of Computing
National University of Singapore
wongls@comp.nus.edu.sg

ABSTRACT

Frequent itemset mining is an important problem in the data mining area. Extensive efforts have been devoted to developing efficient algorithms for mining frequent itemsets. However, not much attention is paid on managing the large collection of frequent itemsets produced by these algorithms for subsequent analysis and for user exploration. In this paper, we study three structures for indexing and querying frequent itemsets: inverted files, signature files and CFP-tree. The first two structures have been widely used for indexing general set-valued data. We make some modifications to make them more suitable for indexing frequent itemsets. The CFP-tree structure is specially designed for storing frequent itemsets. We add a pruning technique based on length-2 frequent itemsets to make it more efficient for processing superset queries. We study the performance of the three structures in supporting five types of containment queries: exact match, subset/superset search and immediate subset/superset search. Our results show that no structure can outperform other structures for all the five types of queries on all the datasets. CFP-tree shows better overall performance than the other two structures.

Categories and Subject Descriptors

H.2.8 [**DATABASE MANAGEMENT**]: Database Applications—*Data Mining*

Keywords

Indexing frequent itemsets, inverted files, signature files, CFP-tree

1. INTRODUCTION

The frequent itemset mining problem was first proposed by Agrawal et al. [2] in 1993 and it has become an important problem in the data mining area since then. Let $D = \{t_1, t_2, \cdots, t_N\}$ be a transaction database and $I = \{a_1, a_2, \cdots, a_n\}$ be the set of items appearing in D, where t_i $(i \in [1, N])$ is a transaction and $t_i \subseteq I$. Every subset of I is called an *itemset*. If an itemset contains k items, then it is called a *length-k itemset*. The support of itemset X in database D is defined as the number of transactions in D containing X, that is, $supp_D(X) = |\{t|t \in D \text{ and } X \subseteq t\}|$. Given a user-specified minimum support threshold min_sup, if $supp_D(X) \geq min_sup$, then X is called *a frequent itemset* in D. The task of frequent itemset mining is to find all the frequent itemsets with respect to min_sup from a given transaction dataset.

Frequent itemset mining is often interactive and iterative. Users may not know exactly what they want at the beginning, and they may need to try different parameters and iteratively examine the set of frequent itemsets from different points of view to find things that are interesting to them. For example, in [12, 25], OLAP operations, such as slicing, dicing, drilling down and rolling up, are used to explore rules to systemically discover useful knowledge from phone usage data. It is very costly if a fresh mining is performed every time a user makes a request because frequent itemset mining is often very time-consuming. An alternative approach is to pre-compute frequent itemsets with a relatively low minimum support threshold and then answer user queries using the pre-computed frequent itemsets. The number of frequent itemsets in a dataset can be undesirably large, especially on dense datasets. It is therefore important to store and index frequent itemsets properly so that user queries can be answered promptly.

Supporting efficient retrieval of frequent itemsets is also very important for problems or algorithms that are built on top of frequent itemset mining, such as association rule mining [2], associative classification [11], frequent itemset based clustering [5] and exploratory hypothesis testing [13]. In these problems, the set of frequent itemsets is repeatedly queried with different constraints. For example, association rule mining aims to find rules of the form $X \Rightarrow Y$, where X and Y are two disjoint itemsets. We can generate association rules in two different ways. One way is to find all the supersets of each itemset X, and then for each superset Y of X, form a tentative association rule $X \Rightarrow (Y - X)$. The other way is to find all the subsets of X, and then for each subset Y of X, form an tentative association rule $Y \Rightarrow (X - Y)$. Either way, we need a proper structure to store and index frequent itemsets so that rules can be generated efficiently.

Indexing and querying frequent itemsets has not drawn much attention in the data mining community. Only a few papers address this problem [16, 1, 20, 15], and the tech-

niques proposed in these papers have not been comprehensively compared. In this paper, we study the performance of three structures for indexing and querying frequent itemsets. Two index structures, inverted files and signature files, are two classical structures for indexing set-valued data. They are employed for indexing frequent itemsets in [16, 20]. We make some modifications to the two structures to make them more suitable for indexing frequent itemsets. The third structure called CFP-tree [15] is specially designed for storing frequent itemsets compactly. During our study, we observed that CFP-tree is less efficient than the other two structures for processing superset queries when the number of frequent items is large. We use a pruning technique based on frequent length-2 itemsets to make it more efficient.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 defines the problem. The three structures are described in Section 4, 5 and 6 respectively. Experiment results are reported in Section 7. Finally, Section 8 concludes the paper.

2. RELATED WORK

Different approaches have been proposed to support frequent itemset queries. A simple approach is to pre-compute frequent itemsets with a sufficiently low threshold and then store and index them properly to support efficient retrieval. A more sophisticated approach is to cache and reuse the results of previous queries. In [17], if a query cannot be answered using previous results stored in a knowledge cache, then the original database is accessed. Another approach is to reduce frequent itemset number by considering representative itemsets only [24]. The above approaches are complementary to each other. The third approach can provide a quick and approximate answer. If users need accurate information, they can retrieve the information from the cache or from pre-computed itemsets on disk, or mine from the original database. The focus of this paper is on indexing and querying pre-computed frequent itemsets on disk.

Inverted files [8] and signature files [4] are two classical indexing structures for set-valued data, and both of them have many variations. These two structures have been used for indexing frequent itemsets and association rules. Morzy et al. [16] use a group bitmap index structure for retrieving association rules, which is essentially a sequential signature file. Winarko and Roddick [23] use signature files to index temporal patterns. Tuzhilin et al. [20] use B+ trees to index the support and the confidence of the rules and use inverted files to index patterns. In Opportunity Map [12], a hash-tree is used to store classification rules.

There are a couple of structures that are specially designed for storing and querying frequent itemsets. Aggarwal et al. [1] use a graph structure, called adjacency lattice, to store frequent itemsets. This structure contains many pointers, which not only take up a lot of space but also make the structure not disk-friendly. CFP-tree [14, 15] avoids this problem by using a prefix-tree structure to store patterns. It keeps only one child pointer for each itemset. CFP-tree also effectively utilizes the sharing and redundancy in frequent itemsets to save space.

Inverted files and signature files have been compared in various contexts. Zobel et al. [26] compare them on text databases and they conclude that inverted files are more superior. Helmer and Moerkotte [9] study their performance for supporting containment queries on set-valued attributes of low cardinality. Their results also show that inverted files exhibit better overall performance. Here we study the performance of the two structures and CFP-tree in the context of frequent itemsets. We believe our study will be useful for building frequent itemset mining based systems.

Table 1: An example dataset

TID	Transactions
1	a, c, e, f, m, p
2	b, e, v
3	a, b, f, m, p
4	d, e, f, h, p
5	a, c, d, m, v
6	a, c, h, m, s
7	a, f, m, p, u
8	a, b, d, f, g

Table 2: Frequent itemsets (*min_sup*=3)

ID	Itemsets	ID	itemsets	ID	itemsets
1	a:6	9	ac:3	17	acm:3
2	b:3	10	af:4	18	afm:3
3	c:3	11	am:5	19	afp:3
4	d:3	12	ap:3	20	amp:3
5	e:3	13	cm:3	21	fmp:3
6	f:5	14	fm:3	22	afmp:3
7	m:5	15	fp:4		
8	p:4	16	mp:3		

3. PRELIMINARIES

The frequent itemset mining problem has been briefly described at the beginning of Section 1. Here we give an example. This example is also used in the next three sections to illustrate the three structures. Table 1 shows a transaction dataset. Let *min_sup*=3. Frequent itemsets are shown in Table 2. To save space, a frequent itemset $\{i_1, i_2, \cdots, i_m\}$ with support n is represented as $i_1 i_2 \cdots i_m : n$.

Given two itemsets X and Y, if $\forall i \in X$, we have $i \in Y$, then X is called a *subset* of Y and Y is called a *superset* of X, denoted as $X \subseteq Y$ or $Y \supseteq X$. If $X \subseteq Y$ and $|X| = |Y| - 1$, then X is a called an *immediate subset* of Y and Y is called an *immediate superset* of X.

Let $\mathcal{F}$ be a set of frequent itemsets. We are interested in the following containment queries on $\mathcal{F}$:

- Exact match: find the support of itemset X.
- Subset/superset search: find all the subsets or supersets of X and their support.
- Immediate subset/superset search: find all the immediate subsets/supersets of X and their support.

If an itemset is in the answer set of a query, we call it *a hit*.

The above queries are common in interactive frequent itemset mining and in frequent itemset based applications. Exact match is the simplest point query. Subset search and superset search can be used to retrieve itemsets that contain or are contained in a particular set of items. They can also be used in association rule generation. In associative classification, subset search can be used to find the set of itemsets that are contained in an instance. Immediate subset/superset search can be used to explore frequent itemsets.

In the next three sections, we use the complete set of frequent itemsets to describe the three structures. It is not difficult to extend the three structures and their associated algorithms for indexing and querying other types of itemsets, such as frequent closed itemsets and generators [18].

4. INVERTED FILES

4.1 The inverted file index

An inverted file [8] consists of a directory and a number of inverted lists. The directory contains all the items in I, so we also call it *the item directory*. Each item in the directory has a pointer pointing to its inverted list. The inverted list of an item i contains the references to the itemsets containing i. Figure 1 shows the inverted file built on the frequent itemsets in Table 2.

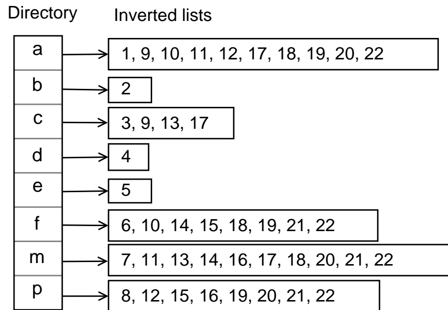

Figure 1: Inverted file constructed on frequent itemsets in Table 2

The inverted lists of some items can be very long, and only part of them are needed for answering queries. For exact match, only the itemsets that are of the same length as the query itemset can possibly be a hit. For superset (subset) search, only the itemsets that are no shorter (longer) than the query itemset can be in the answer set. To reduce query processing cost, we split the inverted list of an item i into k sub-lists based on itemset length, where k is the maximal length of the itemsets containing i. We add a directory called *length directory* to store the references to the k sub inverted lists. An entry in a length directory has two fields: the reference to the corresponding sub inverted list and the size of the sub inverted list. Figure 2 shows how the inverted list of item a is split into 4 sub-lists based on itemset length. The length directory is colored in grey. The numbers in the length directory are the number of itemset references in the respective sub inverted lists.

The benefits of using length directories are two-fold. First, they reduce I/O cost and computation cost considerably because now we need to read and process portions of inverted lists instead of entire inverted lists. Secondly, lengths of frequent itemsets are needed for processing queries except for superset queries, and length directories store them in a compact way. The space overhead for storing length directories is $O(\sum_{i \in I} k_i)$, where k_i is the maximal length of the frequent itemsets containing item i, and it is usually less than 20. An alternative strategy is to store lengths of itemsets together with references to itemsets within inverted lists as in [9]. The length of an itemset X is stored $|X|$ times, so the space overhead is $O(\sum_{X \in \mathcal{F}} |X|)$, which is usually orders of magnitude larger than that of length directories.

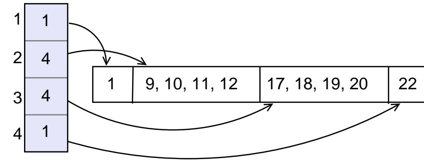

Figure 2: Length directory of item a.

4.2 Implementation details

In our implementation, we use consecutive integers to represent items, and use an array to store the item directory. The i-th entry in the item directory contains the reference to the length directory of item i. The length directories are also stored using arrays.

As in [9], we use lightweight compression techniques [22] to reduce the size of inverted lists. The references in inverted lists occupy one or more bytes depending on their values. To reduce the number of bytes needed for storing references, we sort the references in ascending order and store the differences between consecutive references instead. The same compression is done for itemsets.

All components are stored on pages of 4096 bytes. Inverted lists that are larger than one page are stored on consecutive pages. Inverted lists that are no larger than one page are not allowed to span two pages. The same applies to frequent itemsets. Frequent itemsets are mined using FP-growth [7], and they are stored in their output order.

4.3 Query processing using inverted files

To find the supersets of a query itemset Q and their support, we retrieve the inverted lists of the items in Q and then intersect them. We use a couple of optimization techniques to reduce query processing cost.

- We utilize length directories to avoid retrieving portions of inverted lists that do not contain references to hit itemsets. For example, for superset search, we access only sub inverted lists that contain references to itemsets no shorter than Q.
- We fetch inverted lists from the shortest to the longest. If at some point, the number of possible hits is smaller than the number of pages occupied by the next inverted list to be fetched, then we stop fetching subsequent inverted lists, and we retrieve itemsets directly. This technique is called *early termination*.

Algorithm 1 shows the pseudo-codes of superset search. We first fetch the length directories of the items in Q (line 1). The size of the length directory of item i is the maximal length of the itemsets containing item i. The supersets of Q must contain all the items in Q, so the maximal length of the supersets of Q, denoted as L, cannot be larger than the minimal size of the length directories (line 2). Next, for each length l between $|Q|$ and L, we get the references to the length-l supersets of Q by intersecting $IVT(i)[l]$s, where $i \in Q$ and $IVT(i)[l]$ is the sub inverted list of item i which contain references to length-l itemsets. We fetch the sub inverted lists from the shortest to the longest (line 5-9). We terminate the retrieval of inverted lists if the number of possible hits is no larger than the number of pages occupied by the next sub inverted list (line 10-11). The final set of references may contain itemsets that are not supersets of

Algorithm 1 InvFile_Superset_Search Algorithm

Input:
Q is the query itemset;

Description:
1: Retrieve the length directories of the items in Q;
2: L=the minimal size of the length directories retrieved;
3: A={};
4: **for all** l=$|Q|$ to L **do**
5: Sort items in Q in ascending order of $|IVT(i)[l]|$;
6: Let $Q = \{i_1, i_2, \cdots, i_k\}$ after sorting, where $k = |Q|$;
7: **if** $IVT(i_1)[l]$ is not empty **then**
8: A_l=$IVT(i_1)[l]$;
9: **for all** j=2 to k **do**
10: **if** $|A_l| \leq |IVT(i_j)[l]|/PAGE_SIZE$ **then**
11: **break**; //Early termination;
12: **else**
13: A_l=$A_l \bigcap IVT(i_j)[l]$;
14: $A = A \bigcup A_l$;
15: Retrieve itemsets using the references in A;

Algorithm 2 InvFile_Subset_Search Algorithm

Input:
Q is the query itemset;

Description:
1: Retrieve the length directories of the items in Q;
2: A={};
3: **for all** l=1 to $|Q|$ **do**
4: Sort items in Q in ascending order of $|IVT(i)[l]|$;
5: Let $Q = \{i_1, i_2, \cdots, i_k\}$ after sorting, where $k = |Q|$;
6: A_l=$IVT(i_1)[l]$;
7: **for all** t=1 to $|A_l|$ **do**
8: $A_l[t].miss = 0$;
9: **for all** j=2 to k **do**
10: **if** $|A_l| \leq |IVT(i_j)[l]|/PAGE_SIZE$ AND $l+j-1 > |Q|$ **then**
11: **break**; //Early termination;
12: **else**
13: A'_l=$\{r|r \in (A_l \bigcup IVT(i_j)[l]), l + r.miss \leq |Q|\}$;
14: A_l=A'_l;
15: $A = A \bigcup A_l$;
16: Retrieve itemsets using the references in A;

Q because of early termination. We need to do a checking before we output the itemsets.

At line 4 of Algorithm 1, if we allow l to take only one value $|Q|$, then the algorithm can be used to process exact match queries. If we allow l to take only one value $|Q| + 1$, then it can be used to process immediate superset queries.

It is more complicated to use inverted files to do subset search. The subsets of Q cannot contain items outside Q. We check this based on the following Lemma.

LEMMA 1. *Let Q be the query itemset and Q' be a subset of Q. If an itemset X is a subset of Q, then we must have $|X| + |Q' - X| \leq |Q|$.*

PROOF. Both X and Q' are subsets of Q, so $|X| + |Q' - X| \leq |X| + |Q - X| = |Q|$. ☐

Algorithm 2 shows the pseudo-codes for subset search. Only itemsets of length between 1 and $|Q|$ are considered(line 3). For every length $l \in [1, |Q|]$, we fetch the sub inverted lists from the shortest to the longest (line 4). Initially, all the references in $IVT(i_1)[l]$ are put into the answer set A_l (line 6). For every reference r in A_l, we record the number of inverted lists that do not contain r, denoted as $r.miss$. Initially, $r.miss$ is set to 0 (line 7-8). When we join A_l with a subsequent sub inverted list $IVT(i_j)[l]$ (line 13), there are three cases:

1. r occurs in both A_l and $IVT(i_j)[l]$. In this case, r is put into the new answer set A'_l and $r.miss$ is unchanged.

2. r occurs in A_l but not in $IVT(i_j)[l]$. In this case, $r.miss$ is increased by one. If $l + r.miss$ is still no larger than $|Q|$, then r is put into the new answer set A'_l. Otherwise, it is discarded based on Lemma 1.

3. r occurs in $IVT(i_j)[l]$ but not in A_l. In this case, $r.miss$ is set to $j - 1$ since r does not appear in the previous $j - 1$ inverted lists. If $l + r.miss = l + j - 1 \leq |Q|$, then r is put into the new answer set A'_l. Otherwise, r is discarded.

The early termination technique can still be used in subset search, but under a more restricted condition. In subset search, the size of A_l may increase when more inverted lists are joined. We observe that the only situation that the size of A_l increases is when the third case occurs. For the third case, we have $l+j-1 \leq |Q|$. If $l+j-1 > |Q|$, the size of A_l cannot increase any more, and we can safely apply the early termination technique if A_l is small enough (line 10-11).

In Algorithm 2, if we allow l to take only one value $|Q|-1$ at line 3, then it finds the immediate subsets of Q.

I/O cost analysis. Let H_Q be the number of hits of a query Q. In the worst case, the inverted lists of all the items in Q are retrieved. Hence, the number of page accessed in the worst-case is $\sum_{l=l_s}^{l_e} \sum_{i \in Q} |IVT(i)[l]|/PAGE_SIZE + H_Q$. For exact match, $l_s = l_e = |Q|$; for immediate subset search, $l_s = l_e = |Q| - 1$; for immediate superset search, $l_s = l_e = |Q| + 1$; for subset search, $l_s = 1$, and $l_e = |Q|$; for superset search, $l_s = |Q|$ and $l_e = L$, where L is the minimal size of the length directories of the items in Q.

5. SIGNATURE FILES

The signature file index maps itemsets to fixed-length signatures, and then uses bit operations on signatures to eliminate most of the itemsets that are not hits of a query. Bit operations are much more efficient than set comparison operations, hence the signature file index can reduce the query processing cost greatly. Signature files have many variations. We combine the advantages of signature trees and extendible signature hashing to create a new variation called extendible signature tree.

5.1 The extendible signature tree structure

A signature is a length-L bitmap. Given a single item i, we use $sig(i)$ to denote its signature. We map i to $sig(i)$ as follows. Initially, all the bits of $sig(i)$ is set to 0. We use a random number generator, which uses i as the seed, to generate k random numbers between 1 and L. For every generated random number n, the n-th bit of $sig(i)$ is set to 1. The signature of an itemset $X = \{i_1, i_2, \cdots, i_t\}$, denoted as $sig(X)$, is the superimposition of the signatures of the items in X. That is, $sig(X)$=$sig(i_1)|sig(i_2)|\cdots|sig(i_t)$, where "|" is the bitwise or operator.

Given two signatures sig_1 and sig_2, if every bit in sig_1 that is 1 is also 1 in sig_2, then we say sig_1 *is contained* in sig_2, denoted as $sig_1 \subseteq sig_2$. We use $sig[j_s : j_e]$ to denote a segment of signature sig starting from position j_s and

ending at position j_e, where $1 \leq j_s \leq j_e \leq L$. We call $sig[1:j]$ the length-j *prefix* of signature sig. Signatures have the following properties.

PROPERTY 1. *Given two itemsets X and Y, if $X = Y$, then $sig(X) = sig(Y)$.*

PROPERTY 2. *Given two itemsets X and Y, if $X \subseteq Y$, then $sig(X) \subseteq sig(Y)$.*

PROPERTY 3. *Given two signatures sig_1 and sig_2, if $sig_1 \subseteq sig_2$, then $\forall j_s, j_e \in [1, L]$ and $j_s \leq j_e$, we have $sig_1[j_s : j_e] \subseteq sig_2[j_s, : j_e]$.*

Based on the above properties, we compare the signature of an itemset with the signature of the query itemset first. Only if the two signatures satisfy the containment condition, we then proceed to compare the two itemsets. Note that different itemsets may be mapped to the same signature, so using signatures can produce false hits. We eliminate false hits by checking the itemsets before outputting them.

If the number of frequent itemsets is large, the number of signatures can also be very large. It is still costly to compare every signature with that of the query itemset. In [9], Helmer and Moerkotte show that using an extendible signature hashing index (ESH) reduces the query cost considerably. An ESH consists of a directory and a number of buckets. The directory contains 2^d entries, where d is called the depth of the directory. The length-d prefix of a signature determines which entry the signature is mapped to. A directory entry points to a bucket and the signatures that are mapped to the entry are put into the bucket. If a bucket overflows, we need to split the bucket by increasing the depth of the directory if necessary. The size of the directory cannot be increased beyond a certain threshold, as further increase leads to a too large directory. The threshold is usually set to around 20.

The main problem with ESH is that the number of signatures that are mapped to directory entries are often not evenly distributed. On the frequent itemsets generated on the datasets in the FIMI repository (`http://fimi.ua.ac.be/data/`), we have observed that it is often the case that the majority of the entries do not match any signatures, while a small portion of entries match thousands of or more signatures. To solve this problem, we use an extendible directory tree to replace the flat directory, and we call our structure extendible signature tree (EST).

Figure 3 shows the EST constructed on the itemsets in Table 2. The length of signatures L is 6. The number of bits per item k is 1. The signatures of individual items are as follows:

a: 100000, b: 000001, c: 010000, d: 000001

e: 010000, f: 001000, m: 000100, p: 000010

The size of a directory is controlled by its depth d. The value of d cannot exceed a threshold d_{max}. In Figure 3, d_{max} is set to 2, so the maximum directory size is 4. A directory entry points to either a bucket or a child directory. As in [9], a bucket contains the signatures that are mapped to the entry. If a directory has a child directory, then it is called an internal directory. Otherwise, it is called a leaf directory.

Given a signature sig, its first d_0 bits determines which entry it is mapped to in the root directory, where d_0 is the depth of the root directory. If the entry points to a child directory with depth d_1, then the next d_1 bits of sig decide

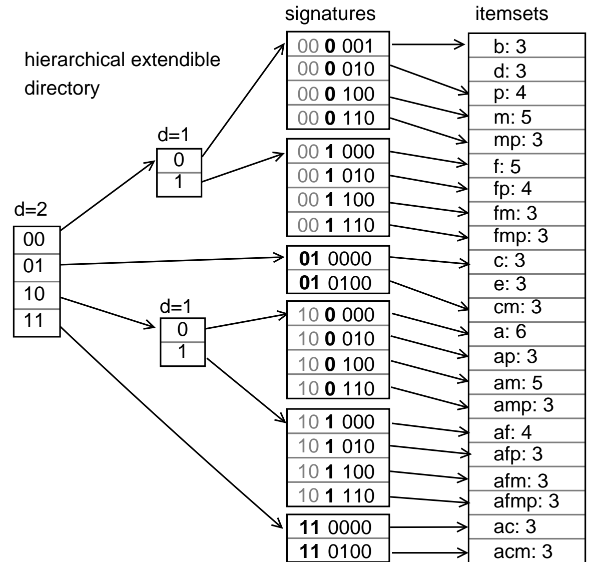

Figure 3: Extendible signature tree constructed on frequent itemsets in Table 2. L=6, k=1, d_{max}=2, $c_{max} = 4$.

which entry sig is mapped to in the child directory, and so on. When the bucket of an entry E overflows, we increase the depth of the directory containing E first to split the bucket. If the depth of the directory already reaches d_{max}, then we create a new directory and let E point to the new directory. The bucket is split accordingly. This way, the depth of an internal directory is always d_{max}, and the depth of a leaf directory ranges from 1 to d_{max}. In Figure 3, we set the maximum capacity of a bucket to be 4 signatures.

An extendible directory tree often requires less space than the corresponding flat directory. If we use the flat directory structure in Figure 3 and keep the maximum capacity of a bucket at 4, then the depth of the directory is 3 and the number of entries is 8. If we insert one more signature "000101", then the bucket pointed by entry "000" need to be split. The depth of the directory need to be increased by 1, and the total number of entries becomes 16. For the extendible directory tree, we only need to increase the depth of the child directory pointed by entry "00". The total number of entries is 10.

5.2 Implementation details

The maximal depth of the directories, denoted as d_{max}, should neither be too large nor too small. If d_{max} is too large, then many entries in the directories are wasted as in a flat directory. If it is too small, then the overhead is too high. In our implementation, we set d_{max} to 8. The size of internal directories is always 256 ($=2^8$) and the size of leaf directories ranges from 2 to 256.

All components are stored on pages of 4096 bytes. We use a clustered strategy to store signatures and itemsets. Itemsets that are mapped to the same signature are stored side by side on disk. Hence each signature is stored only once and it has a pointer pointing to the set of itemsets that are mapped to it. Signatures that are mapped to the same directory entry are stored together as well.

5.3 Query processing using EST

Algorithm 3 shows the pseudo-codes for processing exact match on an EST structure. Initially, D is set to be the root directory D_0 and its depth is d_0, and j=0. We use the first d_0 bits of $sig(Q)$ to locate the corresponding entry in directory D_0, which is $D_0[sig(Q)[1 : d_0]]$. If this entry points to a directory D' with depth d' (line 2), we then check the next d' bits of $sig(Q)$ to see whether entry $D'[sig(Q)[d_0+1 : d_0+d']]$ points to a directory or a bucket recursively (line 3). This process is repeated until we reach an entry pointing to a bucket. We then compare $sig(Q)$ with the signatures stored in the bucket. If a signature sig matches $sig(Q)$ (line 4), we then fetch the itemsets that are mapped to sig and compare them with Q (line 5).

Algorithm 3 EST_Exact_Search Algorithm

Input:
Q is the query itemset;
D is the current directory;
j is number of bits in $sig(Q)$ that have been checked;
Description:
1: d = the depth of directory D;
2: **if** $D[sig(Q)[j + 1 : j + d]]$ points to a directory D' **then**
3: EST_Exact_Search(Q, D', $j + d$);
4: **else if** $sig(Q)$ occurs in the bucket pointed by $D[sig(Q)[j+1 : j + d]]$ **then**
5: Retrieve and validate the itemsets that are mapped to $sig(Q)$;

Algorithm 4 shows the pseudo-codes for processing subset queries on EST. Initially, D is the root directory D_0 and its depth is d_0, and j=0. To find subsets of Q, we first enumerate all the length-d_0 bitmaps that are contained in $sig(Q)[1 : d_0]$ using the algorithm by Vance and Maier [21]. Then for each of these bitmaps B (line 2), if the corresponding entry $D_0[B]$ points to a directory D' with depth d' (line 3), we look at the next d' bits of $sig(Q)$ by calling Algorithm 4 recursively (line 4). This process is repeated until we reach a bucket. For each of the buckets reached, we compare $sig(Q)$ with the signatures stored in the bucket to get those signatures that are contained in $sig(Q)$. Then for each $sig \subseteq sig(Q)$ (line 6), we fetch its itemsets to see whether they are subsets of Q (line 7).

The other three types of queries are processed similarly.

Algorithm 4 EST_Subsets_Search Algorithm

Input:
Q is the query itemset;
D is the current directory;
j is number of bits in $sig(Q)$ that have been checked;
Description:
1: d = the depth of directory D;
2: **for all** length-d bitmap B such that $B \subseteq sig(Q)[j+1 : j+d]$ **do**
3: **if** $D[B]$ points to a directory D' **then**
4: EST_Subsets_Search(Q, D', $j + d$);
5: **else**
6: **for all** sig in the bucket pointed by $D[B]$ such that $sig \subseteq sig(Q)$ **do**
7: Retrieve and validate the itemsets that are mapped to sig;

I/O cost analysis. Let x be the number of signatures that need to be accessed for a query Q. Given a signature sig, we need to access at most L/d_{max} directories to find the directory entry pointing to sig, where d_{max} is the maximum depth of a directory. The directory I/O cost is thus bounded by $min\{P_D, x \cdot L/d_{max}\}$, where P_D is the number of pages needed for storing the whole directory tree. The signature I/O cost is bounded by $x \cdot C$, where C is the maximum number of pages that a bucket pointed by a directory entry can have. The pattern I/O cost is bounded by $x \cdot P_s$, where P_s is the maximal number of pages needed for storing the patterns that are mapped to the same signature. The total I/O cost is thus upper bounded by $min\{P_D, x \cdot L/d_{max}\} + x \cdot C + x \cdot P_s$. For exact match, x=1; for immediate subset search, $x = \sum_{t=0}^{k} \binom{l_Q}{t}$; for immediate superset search, $x = \sum_{t=0}^{k} \binom{L-l_Q}{t}$; for subset search, $x = 2^{|l_Q|} - 1$; for superset search, $x = 2^{L-|l_Q|}$, where k is number of bits per item and l_Q is number of 1s in $sig(Q)$.

6. CFP-TREE

6.1 The CFP-tree structure

The CFP-tree structure is specially designed for storing and querying frequent itemsets [15]. It resembles a set-enumeration tree [19]. The CFP-tree storing the frequent itemsets in Table 2 is shown in Figure 4. Each node in a CFP-tree is a variable-length array. If a node contains multiple entries, then each entry contains exactly one item. If a node has only one entry, then it is called a *singleton node*. Singleton nodes can contain more than one item. For example, node 2 in Figure 4 is a singleton node with two items m and a. Every entry in a CFP-tree represents one or more itemsets with the same support, and these itemsets contain the items on the path from the root to the entry. For instance, node 2 represents 3 itemsets: $\{c, m\}$, $\{c, a\}$ and $\{c, m, a\}$, and these three itemsets have the same support of 3. Let E be an entry and X be an itemset represented by E. Entry E stores three pieces of information: (1) m items ($m \geq 1$), (2) the support of X, and (3) a pointer pointing to the child node of E. In the rest of this paper, we use $E.items$, $E.support$ and $E.child$ to denote them.

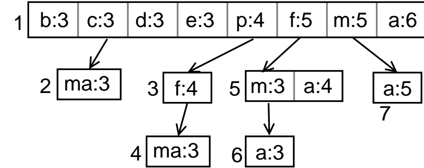

Figure 4: CFP-tree constructed on the frequent itemsets in Table 2

For every entry E in a multiple-entry node, only the items after E can appear in the subtree pointed by E and these items are called *candidate extensions of E*. If a candidate extension of E actually occurs in the subtree pointed by E, then it is called a *frequent extension* of E. In the root node of Figure 4, items d, e, p, f, m and a are candidate extensions of entry c, while item b is not. Items m and a are also frequent extensions of entry c.

The CFP-tree structure allows different itemsets to share the storage of their prefixes as well as suffixes, which makes it a very compact structure for storing frequent itemsets. Prefix sharing is easy to understand. For example, itemsets $\{c, m\}$ and $\{c, a\}$ share the same prefix $\{c\}$ in Figure 4. Suffix sharing occurs when a candidate extension

of an entry E occurs in the same set of transactions as the itemset represented by E. Let i be a candidate extension of E and X be an itemset represented by E. If $supp(X) = supp(X \cup \{i\})$, then for any itemset Z, we must have $supp(X \cup Z) = supp(X \cup \{i\} \cup Z)$. In other words, X and $X \cup \{i\}$ have the same extensions. A singleton node containing item i is created to enable the sharing between X and $X \cup \{i\}$. In Figure 4, itemset $\{p\}$ and $\{p, f\}$ have the same support, so a singleton node containing item f is created, which is node 3, to allow $\{p\}$ and $\{p, f\}$ to share the same subtree.

An entry E may represent multiple itemsets. These itemsets are enumerated by combining three set of items: X_m, X_s and $E.items$, where X_m is the set of items in the multiple-entry nodes and X_s is the set of items in the singleton nodes on the path from the root to the parent of E respectively. The items in X_m decide the unique path from the root to the parent of E, hence they must be included in the itemsets represented by E. The items in X_s are optional. Including or excluding them leads to different itemsets with the same support. The itemsets represented by E should contain at least one item in E. As a result, the number of itemsets represented by entry E is $2^{|X_s|} \cdot (2^{|E.items|} - 1)$. Let us look at an example. For node 4, we have $X_m = \{p\}$, $X_s = \{f\}$ and $E.items = \{m, a\}$. Hence node 4 represents $2^1 \cdot (2^2 - 1)$=6 itemsets: $\{p, m\}$, $\{p, a\}$, $\{p, m, a\}$, $\{p, f, m\}$, $\{p, f, a\}$ and $\{p, f, m, a\}$. This example also shows that if there are more than one singleton node on a path, the space saved by suffix sharing multiplies.

6.2 Implementation details

We sort the entries in a multiple-entry node in ascending order of their support as in [15]. Using this order, the constructed CFP-tree is more balanced. The reason is as follows. The first item in the node has the largest number of candidate extensions, but it also has the lowest support, so few of its candidate extensions can be frequent. Subsequent items becomes more and more frequent but the number of their candidate extensions becomes smaller and smaller, so their subtrees cannot be too large either. The last item has the highest support, but it has no candidate extensions.

The CFP-tree nodes are stored on pages of 4096 bytes in depth-first order. A node in stored before its child nodes. If the size of a node is larger than page size, then it is stored on several consecutive pages. Nodes that are no larger than one page are not allowed to span two pages.

6.3 Query processing using CFP-tree

Algorithm 5 shows the pseudo-codes for processing exact match queries on a CFP-tree. When it is first called, *cnode* is the root node and $Q' = Q$. To find the support of an itemset Q, we simply match Q against the CFP-tree. When matching Q against a singleton node, if all the items in Q' are contained in $E.items$ (line 2), then E is the entry representing Q and we output the support of E (line 3). Otherwise, if E has a child node (line 4), then the search is performed on the child node recursively (line 5). If E does not have a child node (line 6), it means the CFP-tree does not contain Q (line 7). When matching Q against a multiple-entry node *cnode*, we first check whether *cnode* contains all the items in Q' (line 9). If some item i in Q' does not appear in *cnode*, then there is no need to visit the subtrees pointed by entries in *cnode* because they do not contain item i either. Otherwise, we find the first entry E in *cnode* such that $E.items$ is in Q' (line 10). The search on E is the same as that on the entry in a singleton node (line 11-16).

Algorithm 5 CFP-tree_Exact_Match Algorithm

```
Input:
    cnode is a CFP-tree node;
    Q' is the set of items in Q that have not be covered yet;
Description:
 1: if cnode contains only one entry E then
 2:   if Q' − E.items = ∅ then
 3:     output E.support;
 4:   else if E.child ≠ NULL then
 5:     CFP-tree_Exact_Match(E.child, Q' − E.items);
 6:   else
 7:     Output "not found";
 8: else if cnode contains multiple entries then
 9:   if all items in Q' occur in cnode then
10:     E = the first entry in cnode such that E.items ∈ Q';
11:     if Q' − E.items = ∅ then
12:       Output E.support;
13:     else if E.child ≠ NULL then
14:       CFP-tree_Exact_Match(E.child, Q' − E.items);
15:     else
16:       Output "not found";
```

To search for the subsets of an itemset Q, we also match Q against the CFP-tree as in exact match, but the paths visited do not need to contain all the items in Q. The pseudo-codes are shown in Algorithm 6, where itemsets X_m and X_s are used to enumerate the itemsets represented by an entry as described previously. When Algorithm 6 is first called, *cnode* is the root node, $Q' = Q$, $X_m = X_s = \emptyset$. When matching Q against a singleton node, the items of the singleton node that do not appear in Q are discarded, and those that are contained in Q are put into X_s (line 5). When matching Q against a node with multiple entries, all the subtrees that are pointed by entries containing an item in Q are visited and only these subtrees are visited (line 7-8).

The immediate subsets of an itemset Q contain exactly one less item than Q. To find the immediate subsets of Q, we need to add one more parameter n_{miss} to Algorithm 6, which records the number of items in Q that does not occur on the current path. Initially n_{miss}=0. If n_{miss} is increased to 1, then we switch to exact match by calling Algorithm 5.

Algorithm 6 CFP-tree_Subset_Search Algorithm

```
Input:
    cnode is a CFP-tree node;
    Q' is the set of items in Q that have not be covered yet;
    X_m is the set of items in multiple-entry nodes;
    X_s is the set of items in singleton nodes;
Description:
 1: if cnode contains only one entry E then
 2:   if E.items ⋂ Q' ≠ ∅ then
 3:     output itemsets by combining X_m, X_l and
        E.items ⋂ Q';
 4:   if E.child ≠ NULL AND Q' − E.items ≠ ∅ then
 5:     CFP-tree_Subset_Search(E.child, Q' − E.items, X_m,
        X_s ⋃(E.items ⋂ Q'));
 6: else if cnode contains multiple entries then
 7:   for all entry E ∈ cnode do
 8:     if E.items ∈ Q' then
 9:       output itemsets by combining X_m, X_l, E.items;
10:       if E.child ≠ NULL AND Q' − E.items ≠ ∅ then
11:         CFP-tree_Subset_Search(E.child, Q' − E.items,
            X_m ⋃ E.items, X_s);
```

Algorithm 7 CFP-tree_Superset_Search Algorithm

Input:
cnode is a CFP-tree node;
Q' is the set of items in Q that have not be covered yet;
X_m is the set of items in multiple-entry nodes;
X_s is the set of items in singleton nodes;

Description:
```
1: if cnode contains only one entry E then
2:   if (Q' − E.items) = ∅ then
3:     output itemsets by combining X_m, X_l, E.items;
4:   if E.child ≠ NULL then
5:     CFP-tree_Superset_Search(E.child, Q' − E.items, X_m,
       X_s ⋃ E.items);
6: else if cnode contains multiple entries then
7:   if all items in Q' occur in cnode then
8:     if Q' ≠ ∅ then
9:       E' = the first entry in cnode whose item is in Q';
10:    else
11:      E' = the last entry of cnode;
12:    for all entry E ∈ cnode, E before E' or E=E' do
13:      if (Q' − E.items) = ∅ then
14:        Output itemsets by combining X_m, X_s and
           E.items;
15:      if E.child ≠ NULL then
16:        CFP-tree_Superset_Search(E.child, Q' − E.items,
           X_m ⋃ E.items, X_s);
```

Algorithm 7 shows the pseudo-codes for superset search. As in exact match, a multiple-entry node must contain all the items in Q' (line 7). Let E' be the first entry in *cnode* such that $E'.items \in Q'$. We then visit only the subtrees pointed by entries before E' and E' itself (line 12). The subtrees pointed by entries after E' do not contain $E'.items \in Q'$, so there is no need to visit them. In the root node of Figure 4, to search for the supersets of $\{p, m, a\}$, we need to access the subtrees pointed by b, c, d, e and p only.

If the number of entries in a CFP-tree node is large, then superset search can be very costly because the number of entries before E' can be large and all of their subtrees need to be visited. Many of them do not contain all the items in Q'. In [15], a 32-bit bitmap is stored in each entry to avoid unnecessary traversal. If an item i appears in the subtree pointed by an entry E, then item i is hashed to a number k ($k \in [0, 31]$), and the k-th bit of the bitmap of E is set to 1. The subtree of an entry E is visited only if for every item in Q', the bit that the item is hashed to is 1 in the bitmap of E. During our study, we found that this pruning technique is not effective. The reason is as follows: if the number of items in a subtree is large, then almost all the bits in the bitmap is 1 and the subtree cannot be pruned; if the number of items in a subtree is small, then the subtree is small too and the cost saved is minimal even if the subtree can be pruned. Furthermore, the bitmaps increase the size of the CFP-tree considerably which may make CFP-tree perform worse than that without the bitmaps in some cases.

We have observed that most of the unnecessary traversal happens at the child nodes of the root node because the number and the size of these nodes are relatively large. We use a pruning technique based on length-2 frequent itemsets (denoted as F_2) to reduce unnecessary traversal. It is based on the observation that when the number of distinct items in a dataset is large, usually the dataset is sparse and the probability that an item is frequent with another item is relatively low. Consequently, the probability that all the items in the query itemset Q are frequent with a particular item is low too. In the root node, for every entry E before E', before we visit the child node of E (line 15), we first check whether all the items in Q are frequent extensions of E. To facilitate this checking, we build a list called inverted frequent extension list for each frequent item i, and this list contains all the items i' in the root node such that item i is a frequent extension of i'. We denote this list as *INV_FE(i)* and we use a hash table to store it. To check whether all the items in Q are frequent with E, we simply look for the item of E in *INV_FE(i)* for all $i \in Q$. This step takes $O(|Q|)$ time on average, and it is generally less costly than visiting the child node of E, which may contain hundreds of entries. The subtree pointed by E is visited only if the item of E is found in *INV_FE(i)* for all $i \in Q$. This way, most of the subtrees pointed by entries before E' can be pruned.

To find the immediate supersets of Q, we modify Algorithm 7 by adding one more parameter n_{extra}, which is the number of items outside Q on the current path. Initially, n_{extra}=0. If n_{extra} is increased to 1, we switch to exact match by calling Algorithm 5. The above pruning technique based on F_2 is applicable for immediate superset search too.

I/O cost analysis. The largest node in a CFP-tree is the root node. If the root node can be accommodated in one page, then all other nodes can be stored within one page as well. If the root node takes many pages, it means the number of distinct items in the dataset is very large and the dataset must be sparse. As a result, the child nodes of the root node must be much smaller than the root node and usually they can be stored within one page. In the following analysis, we assume all the nodes can be stored within one page except the root node, and we analyze the number of nodes that a query needs to access. We use $f(|Q|)$ to denote the number of nodes to be accessed for a query Q in the worst case. For exact match, $f(|Q|) = |Q|$. Let E be the leftmost entry in the root node containing an item in Q. Searching for the subsets of Q can be decomposed into searching for the subsets of Q that contain $E.item$ in the subtree of E and searching for the subsets of $(Q - \{E.item\})$ on the right of E. Hence we have $f(|Q|) = 2f(|Q|-1)$ and $f(1)$=1. Solving the recursive formula, we have $f(|Q|) = 2^{|Q|-1}$. Similarly, searching for the immediate subsets of Q can be decomposed into searching for the immediate subsets of Q that contain $E.item$ in the subtree of E and searching for $(Q - E.item)$. We have $f(|Q|) = f(|Q|-1) + |Q| - 1$ and $f(1)$=0. Solving the recursive formula, we get $f(|Q|) = |Q|(|Q|-1)/2$. Let n be the number of distinct items in a CFP-tree. Q has $n - |Q|$ possible immediate supersets. Hence for immediate superset search, $f(|Q|) = (n - |Q|) \cdot |Q| + 1$. The number of supersets of Q is upper bounded by $\sum_{l=|Q|}^{L} \binom{n-|Q|}{l-|Q|}$, where L is the maximal length of frequent itemsets. Hence for superset search, $f(|Q|)$ is upper bounded by $\sum_{l=|Q|}^{L}(l \cdot \binom{n-|Q|}{L-|Q|})$.

7. A PERFORMANCE STUDY

Our experiments are conducted on a Dell Optiplex 990 with 3.40GHz CPU, 8GB memory, 500GB SATA hard drive and 64-bit Windows 7. All the three structures were implemented using C++ and complied using Microsoft Visual Studio 2010.

7.1 Experiment settings

We use both synthetic and real datasets in our experiments. On each dataset, we mine frequent itemsets with re-

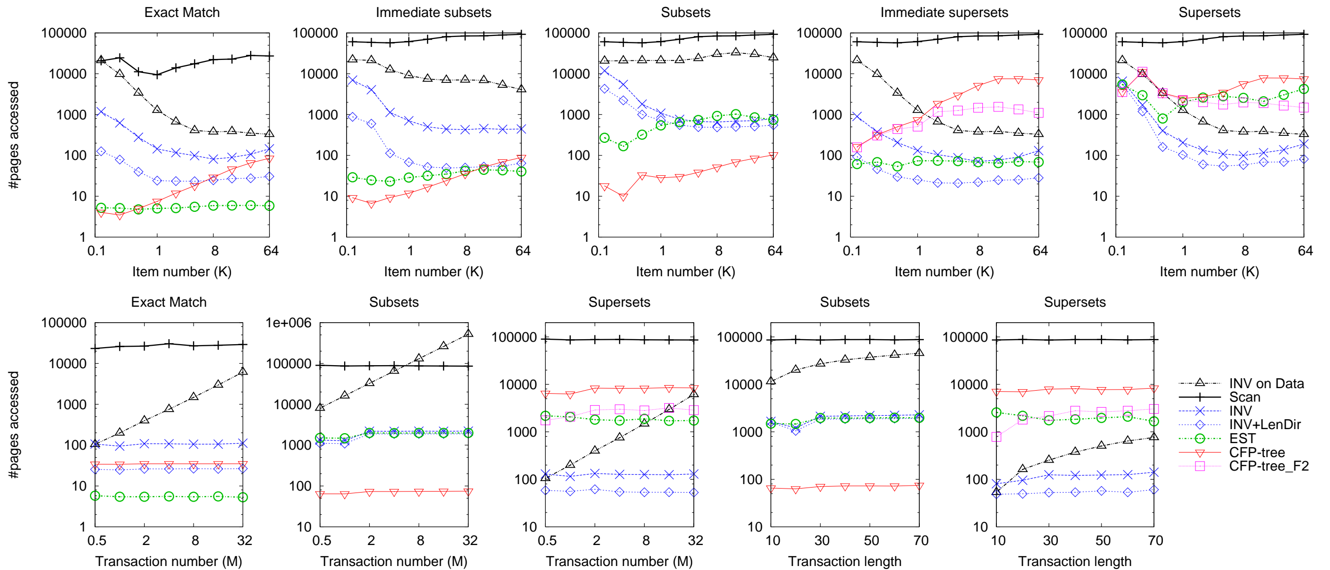

Figure 5: Impact of item number, transaction number and transaction length

spect to a minimum support threshold, and then randomly pick 1000 frequent itemsets as queries. Let L_{max} be the maximal length of frequent itemsets. Queries are selected from itemsets with length between 2 and L_{max}-1. This way, all of the queries have at least one hit except for a few long immediate superset queries. Queries of different lengths are uniformly distributed. In the results shown below, we report the average number of pages accessed for the 1000 queries by each structure, which includes the number of pages accessed for retrieving the indexes and frequent itemsets, but does not include the output cost. The output cost is the same for all the methods.

Parameter tuning for EST. The extendible signature tree (EST) structure has three parameters: k, L and C, where k is the number of bits per item, L is the length of signatures, and C is the maximum number of pages that the bucket pointed by a directory entry can have. We studied the impact of the three parameters on the performance of EST. We found that with the increase of C, the query processing time of EST increases very gradually. EST is sensitive to k and L. The best value for k is 1. When k=1 and L is between 32 and 64, EST has near-optimal performance. In the experiments below, the following parameters are used: k=1, C=2 and L=32.

We implemented two versions of CFP-tree, one uses the pruning technique based on F_2 (denoted as "CFP-tree_F2"), the other one does not (denoted as "CFP-tree"). For inverted files, we also implemented two versions, one uses length directories (denoted as "INV+LenDir") and the other one stores lengths of itemsets explicitly in inverted lists as in [9] (denoted as "INV"). We also build inverted files on the original dataset and answer queries using the dataset directly. For subset and superset search, we use FP-growth [6] to mine frequent itemsets once relevant transactions are retrieved. This method is denoted as "INV on Data". We also include the method that sequentially scans frequent itemsets, which are compressed using the light-weight compression technique, as the baseline (denoted as "Scan").

7.2 Effect of data characteristics

We use synthetic datasets to study the effect of item number, transaction number and transaction length, and we use IBM QUEST synthetic data generator [3] to generate datasets. When one parameter is varied, other parameters are fixed as follows: transaction number is fixed at 2M, transaction length is fixed at 40 and item number is fixed at 20K. The number of frequent itemsets is fixed at 20M.

Figure 5 shows the impact of the three parameters. EST is relatively stable with respect to item number. The cost of the three inverted file based methods decreases with the increase of item number because inverted lists become shorter. The cost of CFP-tree increases with the increase of item number because the root node becomes larger.

The performance of the structures built on frequent itemsets mainly depends on the size of the frequent itemsets, not on the size of the original transaction dataset. Hence transaction number and transaction length have little impact on the performance of the several structures except for "INV on Data". The cost of "INV on data" increases linearly with respect to the two parameters. Note that here the I/O cost of "INV on Data" includes only the cost for retrieving relevant transactions. For subset/superset search, the final itemsets are generated from the retrieved transactions using FPgrowth [6], which is often computationally expensive. Hence, even though "INV on Data" has lower I/O cost than EST and CFP-tree for superset search when transaction number is smaller than frequent itemset number, the overall cost of "INV on Data" for superset search is much higher than that of EST and CFP-tree.

All the three types of structures are at least an order of magnitude better than sequentially scanning frequent itemsets. EST has the best performance for exact match, and relatively good performance for immediate subset/superset search. CFP-tree has the best performance for (immediate) subset search, and has the worst performance for (immediate) superset search especially when item number is large. "CFP-tree_F2" is able to reduce the cost for (immediate)

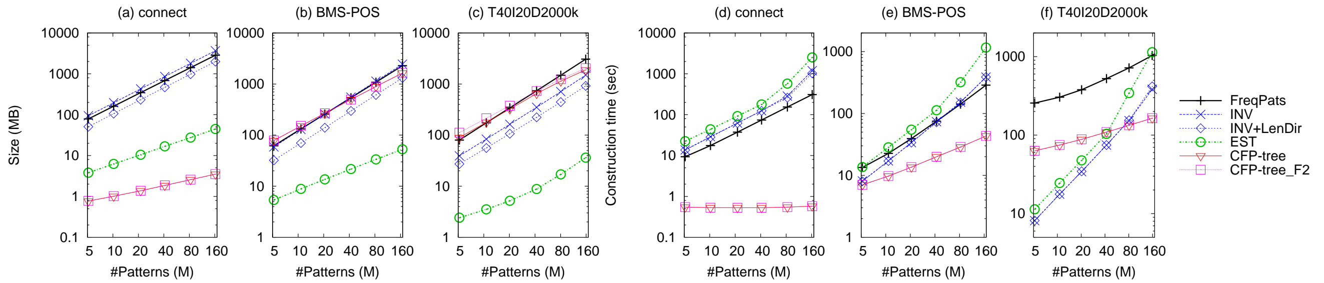

Figure 6: Size and construction time of the several structures.

Datasets	#transactions	#items	avg_tlen	max_tlen	size
connect	67,557	129	43	43	0.34MB
BMS-POS	515,597	1657	6.5	164	11.1MB
T40I20D2000k	2,000,000	12,228	40.0	78	427.9MB

Table 3: Datasets

superset search by using the pruning technique based on F_2 when item number is large. "INV+LenDir" has the best performance for (immediate) superset search, but it performs the worst for (immediate) subset search. "INV+LenDir" is several times better than "INV" for exact match and immediate subset/superset search, which indicates that length directories are very effective in reducing query cost.

7.3 Scalability

We use two real datasets and one synthetic dataset to study the scalability of the several structures with respect to the number of frequent itemsets. Table 3 shows some statistics of the datasets. The 4th and 5th columns are the average and maximal transaction length respectively. Dataset *connect* is obtained from `http://fimi.ua.ac.be/data/`, and it is very dense. Dataset *BMS-POS* is contributed by Blue Martini Software for KDD cup 2000 [10], and it is very sparse. The last dataset is generated using IBM QUEST synthetic data generator [3] with the following parameters: tlen=40, nitems=20 and ntrans=2000. The number of distinct items in the three datasets is in the order of 100, 1000 and 10000 respectively. We intentionally use such three datasets as item number has a significant impact on the performance of CFP-tree and inverted files.

Figure 6 shows the size and construction time of the several structures when the number of frequent itemsets is increased from 5M to 160M. The size and construction time of EST and inverted files exclude the size and mining time of frequent itemsets. We use a separate curve to show the time for mining frequent itemsets and the space for storing them, denoted as "FreqPats". FPgrowth [6] is used to mine frequent itemsets. The light-wight compression technique is used to reduce the space for storing frequent itemsets.

EST is dozens of times smaller than frequent itemsets. The size of inverted files is close to that of frequent itemsets. On all the three datasets, using length directories reduces the size of inverted files. On *connect*, CFP-tree is hundreds of times smaller than frequent itemsets stored in a flat format, and it is even several times smaller than the EST index. This makes CFP-tree extremely efficient on *connect*. On the other two datasets, the size of CFP-tree is similar to that of frequent itemsets. The inverted frequent extension lists generated for F_2 pruning take little space.

CFP-tree is constructed directly from datasets, so its construction cost is no more than frequent itemset mining cost. Figure 6 shows that the construction time of CFP-tree is less than frequent itemset mining time on all the three datasets, partly because of its smaller output size. Inverted files and EST require additional cost to build the index on top of frequent itemsets, and their construction cost is nearly linear to the size of frequent itemsets. EST takes slightly longer time to build than inverted files.

Figure 7 shows the query processing cost of the several structures on the three datasets. Queries are generated as described in Section 7.1. The two inverted file based indexes show linear scalability for all the five types of queries as the size of inverted lists increases linearly with respect to the number of frequent itemsets. EST scales linearly for subset/superset queries, and sub-linearly for the other three types of queries. CFP-tree shows sub-linear scalability for all the five types of queries.

On *connect*, CFP-tree is extremely efficient due to the compactness of the structure as shown in Figure 6 (a). It performs the best for all the five types of queries. For subset/superset search, it is hundreds of times better than EST and inverted files. We observed the same result on other very dense datasets like *mushroom*, *chess* and *pumsb*. On *BMS-POS* and *T40I20D2000k*, EST is the best for exact match and nearly the best for immediate subset/superset search, CFP-tree is the best for (immediate) subset search, and "INV+LenDir" is the best for superset search. This is consistent with Figure 5. It is obvious that "CFP-tree_F2 has the best overall performance on *connect* and *BMS-POS*. If we take the sum of the cost of the five types of queries as the overall cost, "CFP-tree_F2" also has the best overall performance on *T40I20D2000k*.

In the above experiments, frequent itemsets are picked as queries and almost every query has at least one hit. Next we study the scalability of the several structures for processing queries without hits. We randomly pick items that occur in the original dataset to form queries. Nearly all the queries generated in this way have a few hits for subset search and have no hit for the other four types of queries. We again generate 1000 queries, and the distribution of queries of different lengths is the same as in the previous experiment. Figure 8 shows that the I/O cost of all the three structures drops significantly. "INV+LenDir" has the biggest drop while EST has the smallest drop. On *connect*, CFP-tree still has the best performance, and "INV+LenDir" still performs worse than CFP-tree and EST. On *BMS-POS*, CFP-tree and

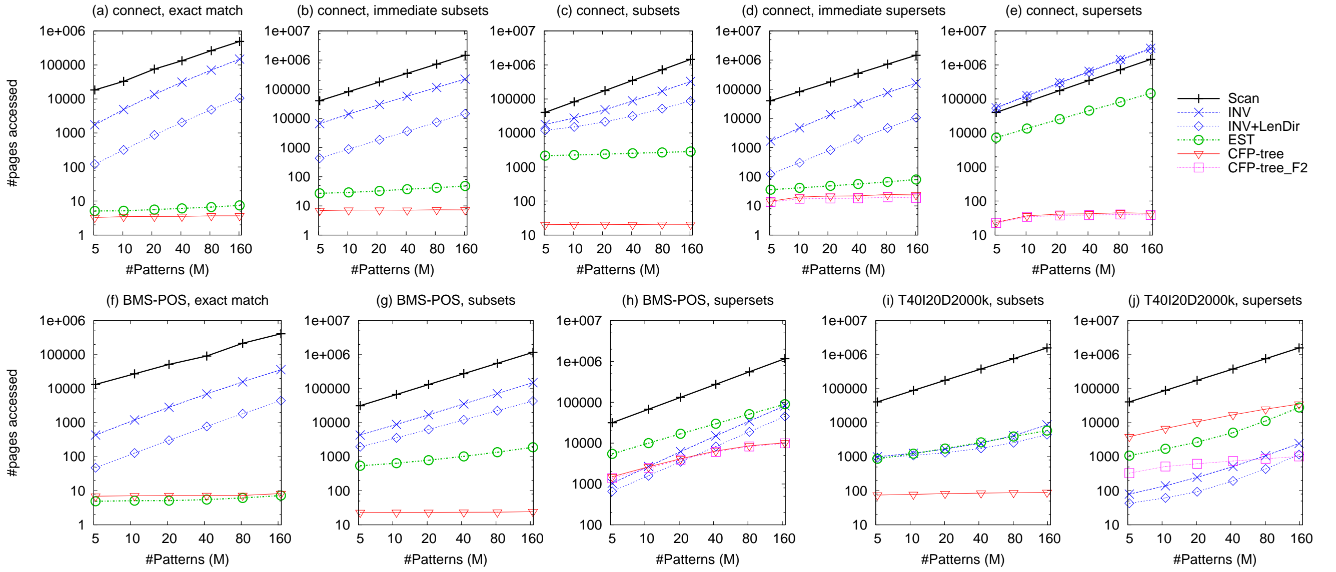

Figure 7: Scalability of the several structures for processing queries with hits.

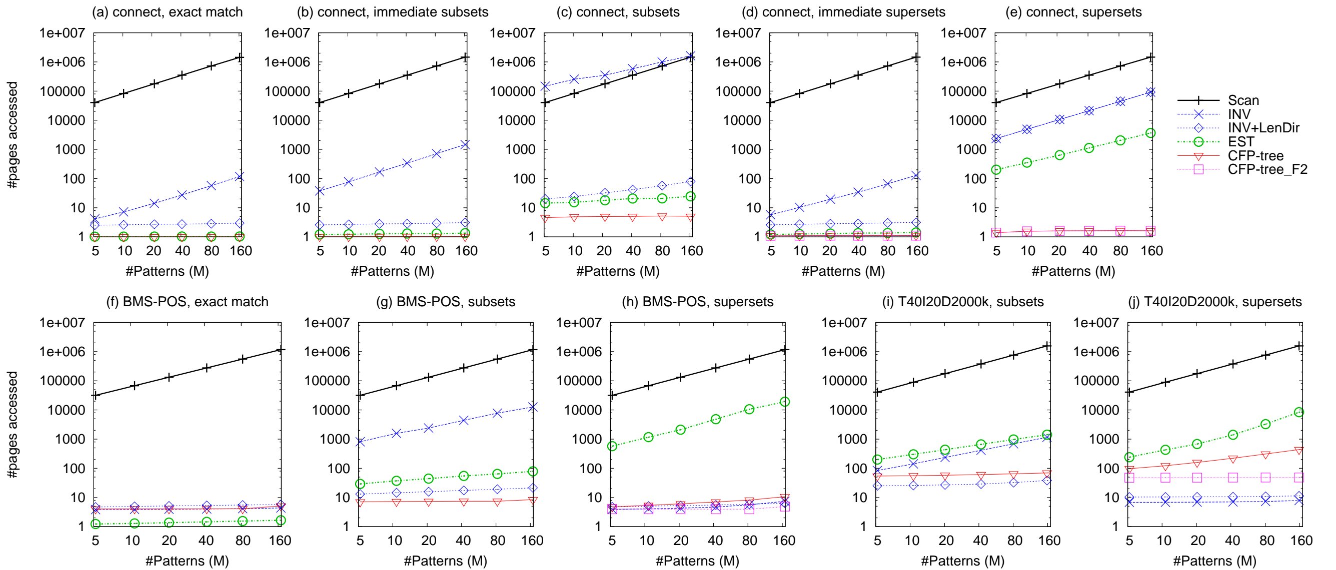

Figure 8: Scalability of the several structures for processing queries without hits.

"INV+LenDir" have similar performance and they are better than EST. On *T40I20D2000k*, "INV+LenDir" has the best performance.

8. DISCUSSION AND CONCLUSION

In this paper, we studied the performance of three structures, EST, CFP-tree and inverted files, for indexing and querying frequent itemsets. We have proposed techniques to improve their performance. For CFP-tree, we use a pruning technique based on length-2 frequent itemsets to speed-up (immediate) superset search. For inverted files, we use length directories to reduce the cost for retrieving inverted lists. Our experiment results show that these techniques are very effective in reducing query processing cost.

We considered five types of queries: exact match, subset/superset search and immediate subset/superset search. Our results show that all the three structures have reasonable performance, and no structure can outperform other structures for all the five types of queries on all the datasets. Our findings can be summarized as follows:

- On dense datasets with a small number of items, such as *connect*, *mushroom* and *pumsb*, CFP-tree shows dominant performance due to its compactness. Inverted files show very poor performance on such datasets because inverted lists are very long on such datasets.
- On other datasets, for exact match, EST performs the best; for subset search, CFP-tree performs the best; for

superset search, inverted files with length directories often have the best performance; for immediate subset search, EST and CFP-tree have similar performance and both of them are better than inverted files; for immediate superset search, EST and inverted files are similar, and both of them are better than CFP-tree.

- Subset/superset queries are the most expensive. EST is not good at either of them, so it often has worse overall performance.
- EST is not sensitive to data characteristics. Both inverted files and CFP-tree are sensitive to item number. With the increase of item number, the relative performance of CFP-tree worsens while the relative performance of inverted files improves.
- CFP-tree shows the best scalability among the three structures. It scales sub-linearly for queries with hits and its cost is almost constant for queries without hits.

Among the three structures, CFP-tree requires the least construction and maintenance cost because it can be constructed and maintained using frequent itemset mining techniques[15]. Hence its cost is the same as frequent itemset mining. The other two require additional cost to build the index on top of the generated frequent itemsets. The construction cost is linear to the size of frequent itemsets.

Acknowledgments

This work is supported in part by Singapore Agency for Science, Technology and Research grant SERC 102 101 0030.

9. REFERENCES

[1] C. C. Aggarwal and P. S. Yu. Online generation of association rules. In *Proc. of the 14th IEEE ICDE Conference*, pages 402–411, 1998.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Proc. of the 1993 ACM SIGMOD Conference*, pages 207–216, 1993.

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

[4] C. Faloutsos. Signature files. In W. B. Frakes and R. Baeza-Yates, editors, *Information retrieval: Data Structures and Algorithms*, pages 44–65, Upper Saddle River, NJ, USA, 1992. Prentice-Hall, Inc.

[5] B. C. M. Fung, K. Wang, and M. Ester. Hierarchical document clustering using frequent itemsets. In *Proc. of SIAM International Conference on Data Mining*, 2003.

[6] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, 2003.

[7] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of the 2000 ACM SIGMOD Conference*, pages 1–12, 2000.

[8] D. Harman, E. Fox, R. Baeza-Yates, and W. Lee. Inverted files. In W. B. Frakes and R. Baeza-Yates, editors, *Information retrieval: Data Structures and Algorithms*, pages 28–43, Upper Saddle River, NJ, USA, 1992. Prentice-Hall, Inc.

[9] S. Helmer and G. Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *The VLDB Journal*, 12(3):244–261, 2003.

[10] R. Kohavi, C. E. Brodley, B. Frasca, L. Mason, and Z. Zheng. Kdd-cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000.

[11] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proc. of the 4th SIGKDD Conference*, pages 80–86, 1998.

[12] B. Liu, K. Zhao, J. Benkler, and W. Xiao. Rule interestingness analysis using olap operations. In *SIGKDD*, pages 297–306, 2006.

[13] G. Liu, M. Feng, Y. Wang, L. Wong, S.-K. Ng, T. L. Mah, and E. J. D. Lee. Towards exploratory hypothesis testing and analysis. In *Proc. of the 27th ICDE Conference*, pages 745–756, 2011.

[14] G. Liu, H. Lu, W. Lou, and J. X. Yu. On computing, storing and querying frequent patterns. In *Proc. of the 9th ACM SIGKDD Conference*, pages 607–612, 2003.

[15] G. Liu, H. Lu, and J. X. Yu. Cfp-tree: A compact disk-based structure for storing and querying frequent itemsets. *Information Systems*, 32(2):295–319, 2007.

[16] T. Morzy and M. Zakrzewicz. Group bitmap index: A structure for association rules retrieval. In *Proc. of the 4th ACM SIGKDD Conference*, pages 284–288, 1998.

[17] B. Nag, P. M. Deshpande, and D. J. DeWitt. Using a knowledge cache for interactive discovery of association rules. In *Proc. of the 5th ACM SIGKDD Conference*, pages 244–253, 1999.

[18] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. of the 7th ICDT Conference*, pages 398–416, 1999.

[19] R. Raymon. Search through systematic set enumeration. In *Proc. of the Internation Conference on Principles of Knowledge Representation and Reasoning*, 1992.

[20] A. Tuzhilin and B. Liu. Querying multiple sets of discovered rules. In *Proc. of the 8th ACM SIGKDD Conference*, pages 52–60, 2002.

[21] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proc. of the 1996 SIGMOD Conference*, pages 35–46, 1996.

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

[23] E. Winarko and J. F. Roddick. A signature-based indexing method for efficient content-based retrieval of relative temporal patterns. *IEEE Transactions on Knowledge and Data Engineering*, 20(6):825–835, 2008.

[24] D. Xin, J. Han, X. Yan, and H. Cheng. Mining compressed frequent-pattern sets. In *VLDB*, pages 709–720, 2005.

[25] K. Zhao, B. Liu, J. Benkler, and W. Xiao. Opportunity map: identifying causes of failure—a deployed data mining system. In *SIGKDD*, pages 892–901, 2006.

[26] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *TODS*, 23(4):453–490, 1998.