Optimizing Frequency Queries for Data Mining Applications

Hassan H. Malik and John R. Kender
Department of Computer Science
Columbia University
New York, NY 10027
{hhm2104, jrk}@cs.columbia.edu

Abstract

Data mining algorithms use various Trie and bitmap-based representations to optimize the support (i.e., frequency) counting performance. In this paper, we compare the memory requirements and support counting performance of FP Tree, and Compressed Patricia Trie against several novel variants of vertical bit vectors. First, borrowing ideas from the VLDB domain, we compress vertical bit vectors using WAH encoding. Second, we evaluate the Gray code rank-based transaction reordering scheme, and show that in practice, simple lexicographic ordering, obtained by applying LSB Radix sort, outperforms this scheme.

Led by these results, we propose HDO, a novel Hamming-distance-based greedy transaction reordering scheme, and aHDO, a linear-time approximation to HDO. We present results of experiments performed on 15 common datasets with varying degrees of sparseness, and show that HDO-reordered, WAH encoded bit vectors can take as little as 5% of the uncompressed space, while aHDO achieves similar compression on sparse datasets. Finally, with results from over a billion database and data mining style frequency query executions, we show that bitmap-based approaches result in up to hundreds of times faster support counting, and HDO-WAH encoded bitmaps offer the best space-time tradeoff.

1. Introduction and Related Work

Calculating itemset support (or frequency counting) is a fundamental operation that directly impacts space and time requirements of many widely used data mining algorithms. Some data mining algorithms (i.e., frequent itemset mining [1]) are only concerned with identifying the support of a given query itemset, while others (i.e., pattern-based clustering algorithms [13,14,15]) must in addition identify the transactions that contain the query itemset.

1.1. Trie-based representations

First generation data mining algorithms used the Trie data structure to improve the itemset support counting performance. In the following years, a number of improvements like [16,17] were proposed to further optimize support counting using Trie. These approaches, however, did not address the major drawback of overwhelming (possibly exponential in depth [10]) space requirements.

Table 1. A transaction database as running example, assuming minimum support = 2

TID	*Items*	*Frequent items ordered w.r.t. decreasing supports*	*Bitmaps representing each transaction*
T1	{1, 2}	{2, 1}	11000
T2	{1, 3, 4, 5}	{3, 4, 5, 1}	10111
T3	{2, 3, 4}	{3, 2, 4}	01110
T4	{2, 3, 4, 5}	{3, 2, 4, 5}	01111
T5	{2, 3, 4}	{3, 2, 4}	01110
T6	{1, 2, 3, 5}	{3, 2, 5, 1}	11101
T7	{2, 3}	{3, 2}	01100
T8	{3, 4}	{3, 4}	00110
T9	{5}	{5}	00001
T10	{3}	{3}	00100

Han et. al. [2] addressed this issue by introducing FP Tree, a Trie-inspired data structure that reduces the space requirements of the original Trie data structure by eliminating the need to insert each transaction into all paths corresponding to the subsets of the transaction. The FP Tree is generated by identifying frequent 1-items in one pass over the dataset, These items are sorted in descending order of their supports, and inserted into the *FList*. A second pass is made to construct the FP Tree in which items are considered in the order of the *FList*. The first node corresponding to each item is pointed from a header table and each FP Tree node contains a link to the next node corresponding to the same item.

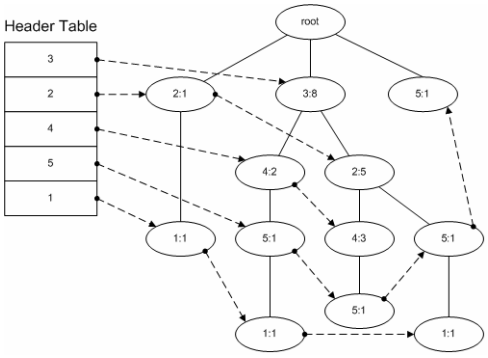

Figure 1. The FP Tree of dataset in Table 1, each node contains an item:frequency pair, and dotted arrows represent node links

***Example 1*:** Considering the transaction database in Table 1, the *FList* contains items in the order (3, 2, 4, 5, 1). Column 3 of Table 1 presents items in each transaction ordered according to the *FList*, and Figure 1 presents the corresponding FP Tree.

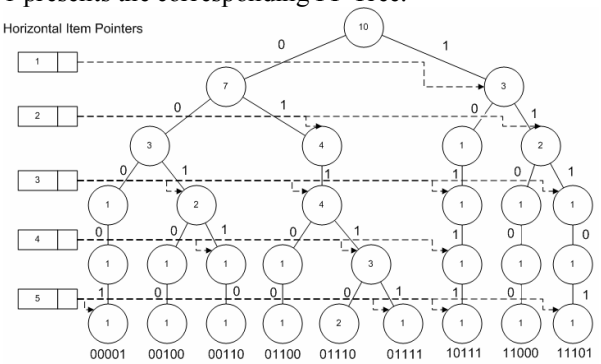

Figure 2. A Binary Trie, nodes contain the count of transactions with the same prefix, and dotted arrows represent pointers from the horizontal lists

In another approach, Yang et. al. [10] reduced the space requirements of the Trie data structure by limiting the branching factor to 2. This is achieved by generating a Binary Trie which considers presence or absence of all items in the transaction, rather than only considering items that exist in the transaction. For each item, a global list of horizontal pointers containing pointers to all nodes that represent the item is maintained. This list enables efficient support counting. Note that the Binary Trie may contain a large number of single-child nodes, especially on sparse datasets. This observation is used to merge these degree-1 nodes with their children, while maintaining the corresponding horizontal pointer lists. The resulting data structure is called a Compressed Patricia Trie.

***Example 2*:** Column 4 of Table 1 contains a binary representation (i.e., presence or absence of all features) for each transaction. The corresponding Binary Trie is presented in Figure 2, and the Patricia Trie obtained by compressing the Binary Trie is presented in Figure 3. Note that the Binary Trie presented in [10] contains additional horizontal pointers to represent absence of items. We eliminate these pointers as they are not relevant for support counting purposes.

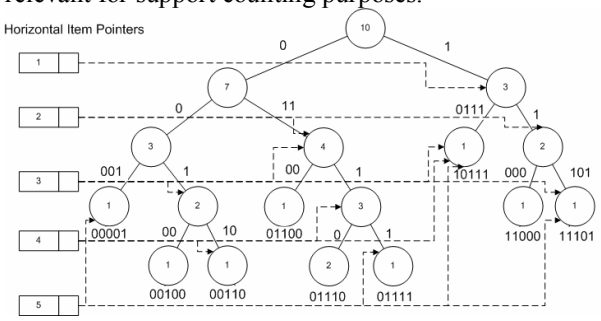

Figure 3. A Compressed Patricia Trie

1.2. Bitmap-based representations

Trie-based representations are suitable for algorithms that are not concerned with the actual transactions that contain the query itemset, but they fell short when these transactions must also be identified. One solution, used by Hu et. al. in a hierarchical clustering algorithm [14] is to store a list containing the applicable transaction IDs at each node of the Trie. This approach may work for small datasets but is impractical for large datasets because of its significant space requirements. In the worst case (i.e., where each transaction contains each item), IDs of all transactions are replicated at each node. Another possible, but very inefficient solution is to first find the support count using the Trie-based data structure and then scan the dataset once to find the applicable transactions.

Considering these issues, a number of recent approaches [11,12,15,18,19,20] adapted uncompressed bitmap-based representations (i.e., vertical bit vectors). In these approaches, a bitmap is generated for each item in the dataset, where each bit represents presence or absence of the item in a transaction. Some of these approaches [15] also reduce the number of bitmaps by eliminating non-frequent 1-itemsets as a preprocessing step. Support is calculated by ANDing (i.e., intersecting) bitmaps of all items in the itemset, and counting the number of one-bits in the resulting bitmap. Note that in typical data mining algorithms (i.e., itemset mining), the number of bitmaps ANDed to find support of an itemset of size k (where $k >= 2$) is exactly 2, as the previous step would have already generated and preserved bitmaps of all large k-1 sized subsets of the query itemset (i.e., guaranteed by the downward closure property). Unlike Trie-based approaches, no additional processing is needed to find the transactions containing the query itemset, as these

transactions are readily available in the resulting bitmap.

Table 2. Vertical bit vectors and corresponding WAH compressed bitmaps for the dataset in Table 1, assuming 4-bit words for WAH encoding

Item	*Vertical bit vector*	*WAH compressed bitmap*
1	1100010000	**0**110 **0**001 **1**001 **0**000
2	1011111000	**0**101 **1**101 **0**100 **0**000
3	0111111101	**0**011 **1**101 **0**110 **0**100
4	0111100100	**0**011 **0**110 **0**010 **0**000
5	0101010010	**0**010 **0**101 **0**001 **0**000

***Example 3*:** Table 2 presents vertical bit vectors for the dataset in Table 1 (section 2.1 provide details on the third column). Considering a query itemset {2, 5}, we obtain support of the query itemset by ANDing 1011111000 with 0101010010. This results in a new vertical bit vector 0001010000. Counting the number of 1-bits, we obtain the itemset support count of 2. The resulting bit vector also identifies the transactions (i.e., 4, 6) that contain the query itemset.

The most significant disadvantage of this approach is that for a dataset containing *n* transactions and *m* frequent 1-items, the amount of space needed for these bitmaps is always *m* x *n* bits, regardless of the characteristics of the underlying dataset. In reality, many data mining datasets are sparse, which would result in bitmaps with a lot more zero-bits than one-bits. Moonesinghe et. al. [12] addressed this problem by first generating a prefix graph that contains a node for each item, and then storing a separate set of variable-sized horizontal bitmaps along with each node. This approach facilitates fast support counting, and frequent itemset mining but does not automatically identify corresponding transactions.

1.3. Contributions

In an attempt to find a space and time efficient dataset representation for fast support counting, that also identifies corresponding transactions, we first identified similarities between the support counting problem and the problem of performing logical operations on equality coded index bitmaps in the very large databases (VLDB) domain. We then considered various compressed bitmap representations of database indices, and limit ourselves to schemes that allow efficient logical operations directly on two compressed bitmaps, resulting in a compressed bitmap, without decompressing any of the operand bitmaps. We evaluated the best of these representations (i.e., WAH compressed bitmaps [5,6,7], with two different word sizes) against FP Tree, Patricia Trie, and uncompressed vertical bit vectors, both in terms of space requirements, and the query processing performance on more than a billion database and data mining style frequency queries. We then evaluated the effectiveness of recently proposed [8] pre-compression step of applying Gray code sorting to re-order transactions, on bitmaps representing 15 widely used datasets and found that this reordering scheme does not result in an optimal solution on real-life datasets, because of the large number of empty cells (section 3.2). We observe that in practice, even simple lexicographic ordering, obtained by applying Least Significant Bit Radix sort on transaction bitmaps, may outperform this scheme.

As a replacement, we propose two novel, Hamming-distance-based transaction reordering schemes (sections 3.3 and 3.4) with different space and time characteristics, and show (section 4.1) that these schemes increase the compressibility of bitmaps.

2. Compressing Vertical Bit Vectors

We observe that when vertical bit vectors are used, the itemset support counting problem is a specialization (i.e., subset) of the problem of processing bitmap indices to find all rows from a database table, that satisfy the given criteria. Column values in a database table can be both equality and range coded, and the criteria can contain a variety of logical operations (i.e., the 'where' clause in SQL, ignoring joins and other cross-table operations). Whereas in case of support counting, the values are equality coded (i.e., presence or absence of an item in a transaction) and the problem is to find all rows that contain all items in the given criteria (i.e., the query itemset). Considering this observation, existing techniques to optimize the performance of bitwise logical operations on equality coded index bitmaps from the very large databases (VLDB) domain can be directly applied on the vertical bit vectors used for itemset support calculation.

There exists a tradeoff between the degree of compression, and the amount of time needed to perform logical operations on compressed bitmaps. Studies [5,9] show that many well known lossless compression schemes such as LZ coding, B-W text compression and Huffman coding are very effective in compressing bit sequences, but require decompressing operand bitmaps to perform logical operations. Even though these schemes may achieve a higher compression ratio, the computational cost of performing logical operations makes them impractical for query intensive and real-time applications. Considering these issues, a number of schemes that mix run-length encoding and direct storage were proposed. These schemes allow logical operations directly on two compressed bitmaps, resulting in a

compressed bitmap. Some of these schemes like BBC, PackBits and PBM are byte-based, while other schemes like HRL, WAH, PWC and WBC are word-based. Studies show that word-based schemes like WAH offer the best space-time tradeoff for performing logical operations. For the reason of space, we do not compare these schemes here, and refer the reader to [5,6,7].

2.1. WAH compressed bitmaps

Word-Aligned Hybrid code (WAH) [5] is a simple linear-time compression scheme that reads a bit sequence one bit at a time, and produces a word aligned compressed bitmap, where the word size W is configurable. Each word in the resulting compressed bitmap represents either a literal run or a fill run. Literal runs contain uncompressed data while the fill runs contain a word-aligned sequence (i.e., fill) of consecutive zero or one bits. The first bit in each word identifies the run type (i.e., 0 = literal run, and 1 = fill run). In case of a literal run, the rest of the W - 1 bits in the word contain a direct sequence of bits, whereas in the case of a fill run, the second bit in the word identifies the fill bit b, and the remaining W - 2 bits contain a count c which represents a sequence of c * (W - 1), b-bits. Note that for an input bitmap with n bits, the size of WAH compressed bitmap is upper bounded by $\lceil \frac{n}{W-1} \rceil \times W$ bits, or O(n). The worst case occurs when there are no fill runs in the resulting WAH compressed bitmap. Furthermore, the absolute value for the worst case (and the overhead) decreases as W increases.

***Example 4*:** Column 3 of Table 2 presents WAH compressed bitmaps for vertical bit vectors in column 2. Bold bits are run identifiers, and the highlighted bits are fill bits. Note that many of the resulting WAH compressed bitmaps demonstrate the worst case space scenario, because we used an artificially small value for W, and a very small transaction database for simplicity sake. We show in section 4.1 that WAH encoded bitmaps do not use more space than the corresponding uncompressed bitmaps on real-life datasets.

2.2. Counting Support using WAH compressed bitmaps

Similar to vertical bit vectors, support of a query itemset is obtained by ANDing the corresponding WAH compressed bitmaps, and counting one-bits in the resulting bitmap. Two WAH compressed bitmaps are ANDed by iteratively decoding words from each of the operand bitmaps, and applying the AND operation on the decoded words [5]. The outcome is then added to the output bitmap. If both operand words represent literal runs, the outcome is determined by simply ANDing the two words. If one of the operand words represent a zero-fill, the same number of zeros is added to the output, and an equal number of bits are skipped from the other operand bitmap. Finally, if one of the operand bitmaps represent a one-fill, number of bits equal to the fill size is added from the other bitmap. Since processing fill runs can result in left over bits from either operand word, some bookkeeping is needed to track these leftover bits. Also, when adding a fill run to the output bitmap, the previous word in the output bitmap is checked for the presence of a matching fill, and the existing fill count is incremented by the new fill count, in case of a match. For more details, see [5].

***Example 5*:** Considering WAH compressed bitmaps in the third column of Table 2, and a query itemset {2, 5}, we first decode the first word in the first operand bitmap (i.e., **0**101) and identify it as a literal run. We then decode the first word in the second bitmap (i.e., **0**010), and identify that it is a literal run as well. Therefore, we AND 101 and 010 to obtain 000, which results in adding a zero-fill-run with count = 1 (i.e., **1**001) to the output bitmap. Similarly, processing the next words (i.e., **1**101 and **0**101), we add a literal fill **0**101 to the output bitmap (since the first operand is a 1-fill of size 1). Processing the next words (i.e., **0**100 and **0**001), we add a new zero-fill **1**001 to the output bitmap. Finally, we process the last two words (i.e., **0**000 and **0**000), and add a new zero fill with count = 1. Since the previous word in the output bitmap was a zero-fill, we just increment it and the final output bitmap becomes **1**001 **0**101 **1**010. Next, we count one-bits in the output bitmap to determine itemset support. Decoding the first word (i.e., **1**001), we find a zero-fill and continue to the next word. Decoding the second word (i.e., **0**101), which is a literal fill, we add 2 (i.e., the number of 1-bits) to the support count. Finally, the last word is decoded and ignored as it is a zero-fill and we obtain the itemset support count of 2.

Note that Support has an interesting property that the support of an itemset of size k is less than or equal to the support of all of its k-1 size subset-itemsets. In practice, a large number of itemsets have supports that are less than their subset-itemsets. This results in an important side effect of smaller and smaller WAH compressed bitmaps as the itemset size increases. As an example, to calculate support of the itemset {1, 3, 4}, we AND the compressed bitmap for item {1} with the compressed bitmap for itemset {3, 4} (above), and obtain the output bitmap **0**010 **1**011, with only two words. Consequently, this side effect makes WAH compressed vertical bit vectors even more feasible

(i.e., space efficient) for algorithms that store interim results.

3. Increasing Bitmap Compressibility by Reordering Transactions

The amount of compression achieved by run-length-based compression schemes such as WAH encoding depends heavily on the availability of long sequences of 0 or 1 bits in the input bitmap. The best compression is achieved when the transactions are organized in a way that minimizes the total number of bit shifts across all columns. As an example, the first column of Table 3 presents a small transaction dataset. The original order of the rows causes three bit shifts in the first column, three bit shifts in the second column and four bit shifts in the third column, adding to a total of 10 bit shifts. In contrast, the transaction ordering in the second column requires only two bit shifts in each column, adding to a total of six bit shifts for the transaction dataset, which represents a 40% reduction.

Table 3. A transaction dataset in original order, an optimal ordering, and reordered using two schemes

	Original order	***Optimal order***	***Gray code sorted***	***Radix sorted***
	T1: 101 T2: 110 T3: 001 T4: 100	T3: 001 T1: 101 T4: 100 T2: 110	T3: 001 T2: 110 T1: 101 T4: 100	T3: 001 T4: 100 T1: 101 T2: 110
Bit changes in each column	3, 3, 4	2, 2, 2	2, 3, 4	2, 2, 4
Total bit changes	10	**6**	9	8

Unfortunately, reorganizing transactions to achieve such an optimal ordering in general is same as the consecutive block minimization problem (or CBMP) which was proven NP-complete in 70's by Kou [3]. More recently, even a fairly restricted version of this problem which limits the number of 1's in each row to 2, called 2CBMP [4], was also proven NP-hard.

3.1. Reordering rows using Gray code sorting

Pinar et. al. [8] named this problem as the "Tuple Reordering Problem", and proven it NP-Complete by providing a reduction from the Traveling Salesman Problem. They proposed a linear in time and space transaction reordering scheme that is based on Gray code ranks, and showed that the reordered bitmaps achieve better WAH compression. As an example, the Gray code rank-based reordering reduces the total number of bit shifts from 10 to 9, on dataset in Table 3.

3.2. Reordering rows using LSB Radix Sort

It is important to note that Gray code rank-based transaction reordering results in an optimal solution only if all cells are "full" [8]. This means that for a transaction dataset with c columns, an optimal solution is obtained when there is at-least one transaction covering each of the 2^c possible combinations, which is not realistic. Therefore, even on our toy dataset (Table 3), applying Gray code rank-based reordering resulted in a small improvement.

As an alternate, simple linear-time Least Significant Bit (LSB) Radix sort [21], with one bin for zero-bits and one bin for one-bits, can be used which results in a lexicographic ordering of transactions. We show in section 4.1 that lexicographic ordering outperforms the Gray code scheme on many real-life datasets. Column 4 of Table 3 presents such an example, where lexicographic ordering results in 8 bit shifts, which is better than the Gray code rank-based solution.

3.3. HDO, a Greedy, Hamming-distance-based transaction reordering scheme

From Table 3, we observe that both Gray code rank-based, as well as lexicographic reordering may not result in close to optimal solutions on transaction datasets. We propose HDO, a greedy algorithm that reorders transactions in a way that ensures a high degree of similarity between neighboring transactions (i.e., minimizes Hamming-distance), hoping that this greedy choice results in a near-optimal solution. In other words, for each position i, HDO finds a transaction t that is closest to the transaction at position i-1. If there is more than one such candidate, it selects the transaction that results in least impact on the number of existing fill runs.

***Definition 1 (inter-transaction distance)*:** Let t_i be a transaction at position i and t_j be a transaction at position j, distance between t_i and t_j is defined as

$$tDist(t_i, t_j) = countOneBits(bitmap_{t_i} XOR\ bitmap_{t_j})$$

The function countOneBits(*bitmap*) returns the number of 1-bits in *bitmap*. Furthermore, the smaller is the value of *tDist* between t_i and t_j, the closer are t_i and t_j to each other. If *tDist* = 0, bitmaps for t_i and t_j are exactly the same.

***Example 6*:** Considering transactions T1 and T2 in Table 3, tDist(T1, T2) = countOneBits(101 XOR 110) = countOneBits(011) = 2.

***Definition 2 (set of least-distant transactions)*:** Let S be a set of transactions and t be a transaction in S. Let S' is a subset of S that does not include t and some other transactions. The set CL_t of transactions that are closest (i.e., least-distant) to t is obtained by:

Step 1: For each transaction x in S', calculate tDist(t, x) and store the outcome in list L. Additionally, track the minimum Distance value *MIN*.

Step 2: For each transaction x in S, add x to CL_t iff tDist(t, x) = *MIN*.

***Definition 3 (HDO)*:** Let S be a set of transactions, assume that transactions S_1 to S_{i-1} are already in HDO. Let $S' = \{S\} - \{S_{1..i-1}\}$, the next transaction S_i is *HDOrdered* by:

Step 1: Using t = S_{i-1}, and S', obtain the set of least-distant transactions CL_t using the method above.

Step 2: If $|CL_t| = 1$, swap the unique transaction with the transaction at S_i. Otherwise, call break-ties(S, i, CL_t) in Figure 5, and swap the resulting transaction with the transaction at S_i. We explain this heuristic peephole (i.e., window) optimization below.

To apply HDO on a transaction dataset with n transactions, we first swap the first transaction in the dataset with a transaction with minimum number of columns, and then iteratively call HDO on transactions 2 to n-1, using the method above. As an example, Figure 4 demonstrates applying HDO on the dataset in Table 3. We can see that the final reordered bitmap achieves a total bit count of 6, which is same as the optimal ordering in this case. Note that our HDO algorithm is an in-place algorithm and works linear in terms of space. However, it has a time complexity of O ($|rows|^2$ x $|cols|$), which is worst than both Gray code rank-based and Radix sort-based reordering schemes, since these schemes has a time complexity linear to the number of bits in the dataset (i.e., O ($|rows|$ x $|cols|$)). We address this issues in section 3.4.

Step 1: Find a transaction t with minimum number of 1-bits	*Step 2: Swap row 1 with t (i.e., row 3)*	*Step 3: Calculate difference bitmaps and counts for rows 2 to 4, against row 1*
1: T1: 101	**1: T3: 001**	1: 001
2: T2: 110	2: T2: 110	2: 110 = 111 = 3
3: T3: 001	**3: T1: 101**	**3: 101 = 100 = 1**
4: T4: 100	4: T4: 100	4: 100 = 101 = 2
Step 4: Swap rows 2 and 3	*Step 5: Calculate difference bitmaps and counts for rows 3 & 4*	*Step 6: Swap rows 3 and 4*
1: T3: 001	1: 001	1: T3: 001
2: T1: 101	2: 101	2: T1: 101
3: T2: 110	3: 110 = 011 = 2	**3: T4: 100**
4: T4: 100	**4: 100 = 001 = 1**	4: T2: 110

Figure 4. Applying HDO

Breaking the ties: If $|CL_t| > 1$ (i.e., there is more than one least-distant transaction to t), we break the ties by selecting the candidate that minimizes the bit changes among the three transactions (i.e., the transaction t, the transaction prior to t, and the candidate itself). In other words, we select the candidate with maximum overlap in difference bits against transaction t and its prior transaction, as these bits are part of literal runs started in t. Selecting other bits may break existing fill runs and impact the overall compressibility of the transaction dataset.

```
1)  break-ties(S, i, candidates)
2)      d = bitmap of S_{i-1} XOR bitmap of S_{i-2}
3)      L = Φ
4)      for i = 1 to |candidates| do begin
5)          temp = bitmap of candidates[i] XOR
6)                 bitmap of S_{i-1}
7)          L[i] = countOneBits(temp XOR d)
8)      end
9)      M = {minimum value in L}
10)     C = {index of first candidate with M in L}
11)     return candidates[C]
12) end
```

Figure 5. Method break-ties

***Example 7*:** Consider t = 1001 and the transaction prior to t = 1101. Let us assume that there are two candidate transactions in set CL_t, i.e., *c1* = 1100 and *c2* = 1010, such that tDist(t, *c1*) = tDist(t, *c2*) = 2. We first compute the difference bitmap between t and its prior transaction, i.e., d = 1001 XOR 1101 = 0100. Considering *c1*, we calculate the difference bitmap *dc1* between *c1* and t (i.e., *dc1* = 1100 XOR 1001 = 0101), and find the number of different bits *ndc1* between d and *dc1*, i.e., 0100 XOR 0101 = 0001 = 1. Candidate *c2* is processed in a similar fashion, i.e., *dc2* = 1010 XOR 1001 = 0011, and *ndc2* = 0100 XOR 0011 = 0111 = 3. Since *ndc1* < *ndc2*, we select *c1*.

3.4. A linear-time approximation to HDO

Because of its high worst-case computational cost, HDO might not be suitable for very large, frequently-updated transaction datasets. We propose aHDO, an approximation to HDO that has a time complexity linear to the number of bits in the dataset. Even so, it achieves close results, especially on sparse datasets.

Figure 6 presents the aHDO algorithm. The algorithm accepts the transaction dataset S, and a constant k, which is used to select k positions in S at uniform intervals, for the inter-loop processing. Hamming-distances of transactions at positions i + 1 to $|S|$ are calculated against each of the selected transaction t_i, and Counting Sort [21] is then applied to reorder these transactions, according to their Hamming-distances against t_i. Note that the linear-time Counting Sort is applicable in this case because the worst case range of Hamming-distances, for a dataset with c columns is already known (i.e., 0..c). Next, we calculate distances between all consecutive rows (lines 14-16), and make another (up to) k passes over S. In each pass, pairs of consecutive transactions are evaluated, and transactions in the pair are swapped if it

reduces the overall number of bit shifts in the solution. Considering four rows at positions $j - 1$, j, $j + 1$ and $j + 2$, distances between consecutive row pairs $(j - 1, j)$, $(j, j + 1)$ and $(j + 1, j + 2)$ are already available. Rows at positions j and $j + 1$ are swapped only if tDist$(j - 1, j)$ is greater than $(j - 1, j + 1)$ or tDist$(j + 1, j + 2)$ is greater than tDist$(j, j + 2)$, and neither of them results in a difference greater than the current order of the four transactions. This guarantees that swapping a row pair results in reducing the total number of bit changes by at-least 1. Note that reducing the total number of bit changes does not guarantee that the overall size of the compressed transaction dataset will also reduce (i.e., it may replace a long, existing fill run with two small fill runs), as providing such a guarantee would require checking a number of additional conditions, against all other bits and transactions in worst case, resulting in an exponential-time algorithm. For the reason of space, we do not demonstrate applying aHDO here, and note that setting k in the range of 50 to 1,000, i.e., a small proportion to the number of transactions, worked well on datasets used in our experiments.

```
1)  aHDO (S, k)
2)          {find a row M with minimum number of columns}
3)          {swap rows 1 and M}
4)          interval = ⌊ |S| / k ⌋
5)          for i = 0 to k - 1 do begin
6)                      L = Φ
7)                      for j = (i * interval) + 2 to |S| do begin
8)                              L[j] =  tDist(S(i*interval)+1, Sj)
9)                      end
10)                     {Using values in L, apply counting sort
11)                     to order transactions S(i * interval)+2 to S|S|}
12)         end
13)         L = Φ
14)         for i = 2 to |S| do begin
15)                     L[i] = tDist(Si, Si-1)
16)         end
17)         for i = 2 to k do begin
18)                     numberOfSwaps = 0
19)                     for j = 2 to |S| - 1
20)                             distj-1And j+1 = tDist(Sj-1, Sj+1)
21)                             distjAnd j+2 = tDist(Sj, Sj+2)
22)                             d1 = L[j] - distj-1And j+1
23)                             d2 = L[j+2] - distjAnd j+2
24)                             if (d1 > 0 OR d2 > 0) AND
25)                                 (d1>=0 AND d2>=0) then
26)                                         numberOfSwaps++
27)                                         {swap rows at j, j+1}
28)                                         L[j] = distj-1And j+1
29)                                         L[j+1] = distjAnd j+2
30)                             end
31)                     end
32)                     if numberOfSwaps = 0 then break
33)         end
34) end
```

Figure 6. Algorithm aHDO

4. Experimental Results

We evaluated the data structures and transaction reordering schemes discussed in this paper in terms of memory requirements, and run-time performance of the support counting operation on fifteen widely used datasets (Table 4), with varying degrees of sparseness.

Table 4. Datasets used in our experiments, #entries correspond to the total number of 1-bits (i.e., columns with non-zero values), Sp = sparseness as the average number of 0's for each 1, rounded to nearest integer

Dataset	*Source*	*#rows*	*#cols*	*#entries*	*Sp*
Flare	UCI ML	1,389	30	13,890	2
Mushroom	UCI ML	8,124	88	176,248	3
Pima	UCI ML	768	36	6,144	4
Anneal	UCI ML	898	66	11,949	4
Adult	UCI ML	48,842	95	677,323	6
FBIS	TREC	2,463	2,000	393,386	12
TR 23	TREC	204	5,832	78,609	14
Hitech	SJMN (TREC)	2,301	22,498	346,881	148
Reviews	SJMN (TREC)	4,069	36,746	781,635	190
LA12	LA Times	6,279	30,125	939,407	200
Sports	SJMN (TREC)	8,580	27,673	1,107,980	213
Reuters	Reuters-21578	10,787	19,127	465,959	442
Ohsumed	Ohsumed-233445	34,389	36,250	2,018,254	617
20NG	20 Newsgroups	9,840	57,675	871,808	650
Classic4	SMART	7,094	41,681	223,839	1320

4.1. Space comparison of various structures

Table 5 compares the memory used by the Trie-based structures on our datasets. On our test (64-bit) system, each FP Tree node used 24 bytes of memory (i.e., 32-bits for frequency, 32-bits for the item ID, 64-bits for the parent pointer, and 64-bits for the node link), and the header table used 64-bits for each item. On the other hand, each node in the Patricia Trie used 12 bytes (i.e., 32-bits for frequency, and 64-bits for the parent pointer), and each pointer in the horizontal list used 64-bits. Interestingly enough, if the features in each transaction are ordered with respect to FP Tree's *FList*, the corresponding Patricia Trie contains exactly the same number of horizontal pointers as the number of nodes in the FP Tree (trivial proof omitted).

On our test datasets, Compressed Patricia Tries resulted in space savings between 36% and 67%, with greater savings realized on sparser datasets (i.e., higher percentage of degree-one nodes). It is important to note that in order to generate a Compressed Patricia Trie, Binary Trie generation appears to be a necessary interim step [10], which can be very expensive. Unlike FP Tree, where we only generate nodes for items that are present in a transaction, Binary Tries consider both the presence and absence of items, resulting in a significantly higher number of nodes. On our test

datasets, Patricia Trie generation needed between two and twenty times more computational time as compare to FP Tree, with higher times observed on sparse datasets. Furthermore, it was not always possible to generate the Binary Trie in memory. As an example, Binary Trie generation exhausted the available memory on our test system (i.e., about 4GB) on LA12, Sports, Ohsumed, and 20NG datasets when the total number of nodes in the Binary Trie reached around 175 million.

Table 5. Space comparison of Trie-based structures

Dataset	*FP Tree*		*Compressed Patricie Trie*		
	#nodes	*Size (KB)*	*#nodes*	*#ptrs*	*Size (KB)*
Flare	1,361	32.13	599	1,361	17.65
Mushroom	20,799	488.16	10,073	20,799	280.54
Pima	389	9.40	228	389	5.71
Anneal	1,399	33.30	730	1,399	19.48
Adult	21,877	513.48	13,464	21,877	328.70
FBIS	367,553	8,630.15	3,911	367,553	2,917.34
TR 23	75,797	1,822.05	329	75,797	596.02
Hitech	337,474	8,085.31	3,316	337,474	2,675.38
Reviews	760,265	18,105.79	5,949	760,265	6,009.29
LA12	873,862	20,716.49	N/A	N/A	N/A
Sports	1,050,754	24,843.24	N/A	N/A	N/A
Reuters	399,439	9,511.28	15,359	399,439	3,300.61
Ohsumed	1,860,347	43,885.09	N/A	N/A	N/A
20NG	792,123	19,015.97	N/A	N/A	N/A
Classic4	208,414	5,210.34	8,454	208,414	1,727.30

Table 6 compares the space used by uncompressed bit vectors, WAH encoded bitmaps in the original order, and after applying various reordering schemes. Our experiments included both 32 and 64-bit words for WAH encoding, but we only report the 32-bit results here for the reason of space, and note that 64-bit WAH encoded bitmaps used between 4 and 71 percent more space as compare to the corresponding 32-bit bitmaps, because with 64-bit words, uniform bit sequences smaller than 126 bits result in no space savings (i.e., fill count = 2), while 32-bit words realize space savings on shorter (i.e., >=62-bit) uniform bit sequences.

We observe that the uncompressed vertical bit vectors used less space as compare to both Trie-based representations on dense datasets (i.e., Mushroom) but used significantly more space on highly sparse datasets (i.e., Classic). WAH encoding resulted in significant space savings, especially on sparse datasets. Also, lexicographic ordering outperformed Gray code rank-based reordering scheme on 12/15 datasets. Furthermore, HDO-WAH encoded bitmaps outperformed all other reordering schemes on 14/15 datasets and resulted in the most significant overall space savings. HDO even worked well on Hitech, Reviews, and Sports datasets, where both Gray code and lexicographic schemes negatively impacted the compression achieved on the original-ordered bitmap. Finally, aHDO resulted in compression very close to HDO, especially on sparse datasets. The Classic dataset exhibits an interesting behavior, where all reordering schemes negatively impacted the WAH compression achieved on the original-ordered bitmap, while HDO still outperformed other reordering schemes.

For the reason of space, we do not report the times needed to apply various reordering schemes here, and note that Gray code sorting, LSB Radix sort and aHDO takes comparable amount of time while HDO takes the most amount of time. Counter-intuitively, for small values of k (i.e., 50), we observe that aHDO may take less time than the other linear-time schemes because it calculates the inter-transaction distances by XORing whole words (i.e., 64 bits), while other schemes needs to decode and evaluate each bit, requiring more operations. Furthermore, we trivially optimized the second most frequent operation in aHDO (i.e., counting 1-bits in a word) by caching bit-patterns.

4.2. Performance of frequency queries

Database style frequency queries: We first compared the performance of various structures by generating 25 million random frequency queries for each dataset, with 5 million queries for each of the max query sizes 1-5 (i.e., for max query size = 2, there would be about 2.5 million size-1 queries, and an equal number of size-2 queries). This adds to a total of 375 million queries on all datasets, with each query executed on all available structures, adding to many billion query executions. We assumed no prior knowledge about the query itemsets, which means that for a query itemset of size k, all k-bitmaps were used for frequency calculation. This setting is close to real-life database usage where variable-size, random query are common, with a higher percentage of short queries. For the reason of space, we only report the query execution results on eight datasets in Figure 7.

We observe that bitmap structures resulted in an orders of magnitude faster frequency counting as compare to Trie structures on short queries. The performance difference minimized as the query size increased, because the number of bitmaps ANDed linearly increase with the number of items in the query, whereas the number of upward paths considered in a Trie remains constant, and more paths can be quickly pruned for longer, randomly generated queries (i.e., decreasing number of co-occurring items). Note that regardless of (potentially) better frequency counting performance on long queries, Trie structures are

Table 6. Compression achieved by various reordering schemes. Best results highlighted, WAH compression uses a word size of 32-bit, IF = Improvement Factor as in [8], and all values rounded to 2 decimal places

	Size of the uncompressed bit vectors (Kbytes)	*WAH, original order*		*WAH, Gray code reordered*		*WAH, LSB Radix sorted*		*WAH, HDO*		*WAH, aHDO*	
		Size (KBytes)	*% of original*	*Size (KBytes)*	*IF*	*Size (KBytes)*	*IF*	*Size (KBytes)*	*IF*	*Size (KBytes)*	*IF*
Flare	5.16	5.04	97.73	3	1.68	2.91	1.73	**2.6**	**1.94**	2.67	1.89
Mushroom	87.31	70.26	80.47	22.74	3.09	20.4	3.44	**20.24**	**3.47**	21.55	3.26
Pima	3.38	2.61	77.31	1.22	2.13	1.23	2.13	**0.96**	**2.70**	1.02	2.55
Anneal	7.73	5.02	64.85	3.82	1.31	3.51	1.43	**3.33**	**1.51**	3.58	1.40
Adult	567.03	292.53	51.59	70.24	4.16	**68.3**	**4.28**	71.21	4.11	82.08	3.56
FBIS	609.38	551.57	90.51	456.15	1.21	455.74	1.21	**433.8**	**1.27**	434.64	1.27
TR 23	182.25	173.58	95.24	154.08	1.13	153.67	1.13	**144.11**	**1.20**	144.66	1.20
Hitech	6,327.56	1,222.46	19.32	1,244.64	0.98	1,244.68	0.98	**1,155.45**	**1.06**	1,174.3	1.04
Reviews	18,373	2,689.43	14.64	2,807.85	0.96	2,806.11	0.96	**2,571.97**	**1.05**	2,592.99	1.04
LA12	23,299.80	3,410.82	14.64	3,143.75	1.08	3,144.02	1.08	**2,807.72**	**1.21**	2,875.78	1.19
Sports	29,186.37	3,103.27	10.63	3,445.02	0.90	3,441.88	0.90	**2,949.74**	**1.05**	3,010.04	1.03
Reuters	25,253.62	1,826.09	7.23	1,552.38	1.18	1,549.45	1.18	**1,277.50**	**1.43**	1,359.62	1.34
Ohsumed	152,363.28	7,594.60	4.98	7,119.27	1.07	7,118.09	1.07	**6,502.91**	**1.17**	6,657.52	1.14
20NG	69,390.23	4,121.75	5.94	3,705.17	1.11	3,701.30	1.11	**2,955.54**	**1.39**	3,364.05	1.23
Classic4	36,145.24	**1,280.65**	3.54	1,387.30	0.92	1,386.88	0.92	1,317.56	0.97	1,336.81	0.96

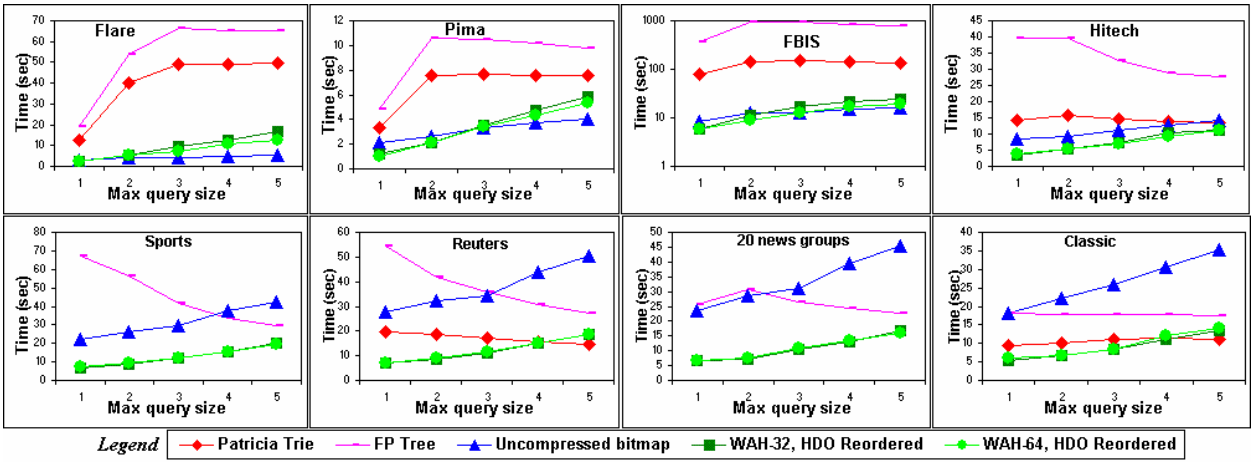

Figure 7. Performance comparison of various structures on 200 million random, variable-sized frequency queries

practically unusable for database style queries because most database style queries (except COUNT) must also identify the corresponding transactions. We also observe that Compressed Patricia Tries outperformed FP Trees, (i.e., a smaller number of nodes traversed). Furthermore, uncompressed vertical bit vectors resulted in shortest query execution times on dense datasets, and HDO-WAH encoded vertical bit vectors outperformed uncompressed vertical bit vectors as the sparseness increased. Finally, we observe that Tries performed poorly on datasets that do not have many transactions that share common prefixes (i.e., more upward paths to consider), while the performance of bitmap structures remained un-impacted. For example, on FBIS dataset with 393,386 non-zero entries, the corresponding FP Tree contained 367,553 nodes. Consequently, it took 3,854 seconds to execute 25 million queries using the FP Tree, as compare to only 63 seconds using the HDO-WAH encoded bitmaps, a significant difference!

Data mining style frequency queries: To evaluate the performance of data mining style frequency queries, we applied APRIORI [1] to mine frequent itemsets of sizes 1-5, on datasets in Figure 7. Unlike the previous test (i.e., no prior knowledge), we stored the bitmaps of large itemsets found at each step. Consequently, support calculation was performed by ANDing only two bitmaps (section 1.2). An advanced nanosecond timer was used to record individual query execution times, and the total times are reported in Table 7.

We observe that bitmap structures significantly outperformed both Tries. Furthermore, unlike the previous test, the performance gap did not minimize with increasing query sizes for at-least two reasons (graphs omitted for the reason of space). First, the

number of bitmaps ANDed remained constant (i.e., 2), and second, the percentage of upward paths pruned in Tries may actually decrease because unlike the random test, where up to k-1 items in a query of size k can be non-existent in an upward path, all k-1 sized subsets of each query are guaranteed to meet minimum support.

Table 7. Itemset mining performance

Dataset	*min supp*	*#itemsets (size 1-5)*	*Time (seconds)*				
			Patricia Trie	*FP Tree*	*Ucomp*	*WAH 32*	*WAH 64*
Flare	2	21,063	0.36	0.13	0.06	0.07	0.08
Pima	2	3,860	0.05	0.02	0.03	0.03	0.03
FBIS	250	654,525	172.87	361.65	1.07	2.29	1.43
Hitech	50	2,859,310	211.33	478.55	4.00	8.02	5.55
Sports	300	1,297,271	N/A	1543.58	5.04	10.50	7.03
Reuters	100	996,097	269.19	375.08	4.80	6.74	4.72
20NG	200	643,537	N/A	249.44	3.10	5.24	3.37
Classic4	10	5,800,199	212.23	278.21	20.22	15.25	12.77

Finally, we note that the runtime performance of bitmap-based schemes depend on the program structure, and the underlying system architecture, in addition to the total number of operations involved. As an example, in spite of their significantly higher space usage (which translates to more instructions needed to AND bitmaps), uncompressed bitmaps may outperform compressed bitmaps in time. This happens because two uncompressed bitmaps can be ANDed in a simple loop, with no inter-iteration dependencies. This simple structure allows exploiting maximum instruction level parallelism, and enables compilers to apply techniques like loop unrolling. On the other hand, the decoding logic of compressed bitmaps do not allow exploiting the same level of ILP. Similarly 64-bit WAH compressed bitmaps used more space, but outperformed 32-bit bitmaps on our 64-bit test system, because the system processed twice as much data in each cycle. We conclude that HDO-WAH encoded bitmaps offer the best space-time tradeoff for data mining style queries. For example, performance was comparable to uncompressed bit vectors on Reuters and 20NG, while consuming 20 times less space.

5. Conclusions

We compared Trie and bitmap-based structures in this paper, and conclude that Trie structures are viable for applications that mostly execute long, random queries, as long as we are not concerned with identifying the actual transactions. We proposed HDO, a Hamming-distance-based greedy transaction reordering scheme, and showed that it results in better compression, and outperforms other structures on short database style frequency queries. We also showed that aHDO can serve as a practical alternate to HDO on sparse datasets. Finally, we showed that uncompressed bitmaps can be a good choice for data mining applications that are not concerned with high space requirements, while HDO-WAH encoded bitmaps provide the best space-time tradeoff.

6. References

[1] R. Agrawal, and R. Srikant, "Fast Algorithms for Mining Association Rules", In *Proc. VLDB 1994*.

[2] J. Han, J. Pei, Y. Yin and R. Mao, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach", *Data Mining & Knowledge Disc.*, 2004.

[3] L.T. Kou, "Polynomial complete consecutive information retrieval problems", *SIAM Jnl. on Computing*, Vol. 6, 1977.

[4] S. Haddadi, "A note on the NP-hardness of the consecutive block minimization problem", *International Transactions in Operational Research*, Vol. 9, No. 6, 2002.

[5] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg, "Notes on design and implementation of compressed bit vectors", *Tech. rep. LBNL/PUB-3161*, Berkeley, CA.

[6] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression", ACM *TODS*, 2006.

[7] K. Wu, E. J. Otoo, and A. Shoshani, "On the Performance of Bitmap Indices for High Cardinality Attributes", In *Proc. VLDB 2004*, pp. 24-35.

[8] A. Pinar, T. Tao and H. Ferhatosmanoglu, "Compressing Bitmap Indices by Data Reorganization", In *Proc. ICDE '05*.

[9] T. Johnson, "Performance Measurements of Compressed Bitmap Indices", In *Proc. VLDB 1999*.

[10] D.-Y. Yang, A. Johar, A. Grama, and W. Szpankowski, "Summary structures for frequency queries on large transaction sets", In *Proc. DCC 2000,* pp. 420-429.

[11] X. Hu, T.Y. Lin, and E. Louie, "Bitmap techniques for optimizing decision support queries and association rule algorithms", In *Proc. DB Engg. & Aps Symposium*, 2003.

[12] H. D. K. Moonesinghe, et. al., "Frequent Closed Itemset Mining Using Prefix Graphs with an Efficient Flow-Based Pruning Strategy", In *Proc. ICDM 2006*, pp. 75-86.

[13] F. Beil, M. Ester, and X. Xu, "Frequent term-based text clustering", In *Proc. ACM SIGKDD 2002*, pp. 436-442.

[14] H. Yu, D. Searsmith, X. Li and J. Han, "Scalable Construction of Topic Directory with Nonparametric Closed Termset Mining", In *Proc. ICDM'04*, pp. 563-566.

[15] H. H. Malik, and J. R. Kender, "High Quality, Efficient Hierarchical Document Clustering Using Closed Interesting Itemsets", In *Proc. ICDM 2006*, pp. 991-996.

[16] F. Bodon, "A fast apriori implementation." In *Proc. IEEE ICDM Workshop FIM Implementations*, 2003.

[17] A. Amir, R. Feldman and R. Kashi. "A New and Versatile Method for Association Generation", *Information Systems*, 1997, Vo. 22, No. 6, pp. 333-347

[18] F. Verhein, and S. Chawla, "Geometrically Inspired Itemset Mining", In *Proc. ICDM 2006*, pp. 655-666.

[19] D. Burdick, M. Calimlim, and J. Gehrke, "MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases", In *Proc. of ICDE*, 2001.

[20] C. Lucchese, et. al., "Fast and Memory Efficient Mining of Frequent Closed Itemsets", *TKDE*, 2006.

[21] T. Cormen et. al., "Introduction to Algorithms, 2nd Edition", *McGraw Hill / MIT Press*, ISBN: 0-07-013151-1.