Privacy-preserving Frequent Itemset Mining for Sparse and Dense Data

Peeter Laud[1] and Alisa Pankova[1,2,3]

[1] Cybernetica AS
[2] Software Technologies and Applications Competence Centre (STACC)
[3] University of Tartu
{peeter.laud|alisa.pankova}@cyber.ee

Abstract. Frequent itemset mining is a task that can in turn be used for other purposes such as associative rule mining. One problem is that the data may be sensitive, and its owner may refuse to give it for analysis in plaintext. There exist many privacy-preserving solutions for frequent itemset mining, but in any case enhancing the privacy inevitably spoils the efficiency. Leaking some less sensitive information such as data density might improve the efficiency. In this paper, we devise an approach that works better for sparse matrices and compare it to the related work that uses similar security requirements on similar secure multiparty computation platform.

1 Introduction

Frequent itemset mining (FIM) is a standard data mining task that can in turn be used to extract some more interesting knowledge such as association rules. Not only the task itself, but also its privacy-preserving variant has been well studied in many related works. The goal of this task is, given a collection of sets, find the subsets that are contained in sufficiently many of these sets. After finding which elements are more likely to occur together, one may search for the reason for that co-occurrence, and whether the existence of one item implies the existence of the other one.

In this paper, our goal is not to implement yet another efficient algorithm for FIM, but to see if we can gain more efficiency given some additional assumptions about the matrix density. We are going to present some algorithms that can be used for very sparse and very dense data matrices. Moreover, as the algorithms for FIM are iterative, even if data has not been sparse on the first iteration, it may become very sparse or very dense on later iterations, and hence we are not very strict about constraining our algorithms to sparse and dense data only.

2 Preliminaries

In this section, we define some notions and quantities that will be used further.

2.1 FIM

Input Data Traditionally, the sets themselves are called the *transactions*, and their elements are called *items*. This comes from one possible use case, where the items are some goods sold in the supermarket, and each transaction corresponds to the contents of one shopping cart that a client has bought. In this case, analyzing frequent itemsets tells which items are usually bought together, so that they could be deliberately put into opposite corners of the supermarket, and the clients would have to increase the length of their trajectories, which hopefully forces them to buy more goods than they initially wanted. In general, the shopping carts are nothing more than just sets T over some universal set U (e.g all the goods sold in the shop), and the task is to find the subsets $I \subseteq U$ that are encountered in sufficiently many sets T.

Frequency Definition First, one has to define a *frequent* itemset. Usually, the criterion of being frequent depends on the particular task, and hence initially one defines some threshold $t \geq 1$ (sets of size 0 are not very interesting) in such a way that an itemset I is considered frequent if the number of transactions that contain I is at least t.

Data Representation In general, the data table is represented by a bit matrix $B = (b_{ij})$. Each column of the matrix corresponds to an item, and each row to a transaction. The matrix is defined as $b_{ij} = 1$ iff the j-th item belongs to the i-th transaction.

2.2 Our Contribution

In a perfectly secure mining, nothing is leaked besides the final result that will be published anyway. This means that a dense matrix is in fact indistinguishable from a sparse matrix, since that information also remains private. Could we do something better if we agreed to leak the matrix density (the number of nonzero entries in the bit matrix)?

In this paper, we think on representing the matrix B as a set of pairs $C = (tid, item)$, such that $(i, j) \in C$ iff $b_{ij} = 1$. The cardinality $|C|$ clearly leaks information about the number of nonzero entries, but we find it acceptable. We see that unfortunately leaking just $|C|$ is not enough to achieve better efficiency than for bit matrices, and hence we additionally will have to leak an upper bound on the number of elements in each column. Leaking the precise column size allows to achieve even more efficient solutions.

2.3 Notation

Throughout this paper, we use the following quantities:

- m is the number of rows in the data table (transactions);
- n is the number of columns in the data table (items); the same notation is used to denote the number of columns on current iteration (itemsets);
- $\ell \leq mn$ is the total number of non-zero entries in the data table;
- $m_j \leq m$ is the number of non-zero entries in the j-th column;
- $m' \geq m_j (\forall j)$ is the upper bound on the number of non-zero entries in each column;
- t is the threshold of being a frequent set;

- k is the size of the currently generated itemsets;
- $\hat{k}$ is the maximal size of frequent itemsets that have to be generated.

Some FIM-specific notation:

- $\sigma(I)$ is the set of all transactions containing the itemset I (the support of I);
- $\Delta(I_1, I_2) := \sigma(I_1) \setminus \sigma(I_2)$ is the difference of the supports of I_1 and I_2.

The following more general shorthand operation notation will be used:

- protocol input length (if the input is a vector) n;
- number of bits of protocol input: k;
- secret shared value (additive or xor sharing) $\langle[a]\rangle$;
- additively shared value $[\![a]\!]$;
- xor shared value $\langle\langle a \rangle\rangle$;
- i-th element of a vector $\mathbf{a}$: a_i and $\mathbf{a}[i]$;
- i-th row of a matrix $\mathbf{A}$: $\mathbf{A}[i,*]$;
- rows $i_1, \ldots, i_k$ of $\mathbf{A}$: $\mathbf{A}[I,*]$ for $I = \{i_1, \ldots, i_n\}$;
- j-th column of a matrix $\mathbf{A}$: $\mathbf{A}[*,j]$;
- columns $j_1, \ldots, j_k$ of $\mathbf{A}$: $\mathbf{A}[*,J]$ for $J = \{j_1, \ldots, j_n\}$;
- (i,j)-th element of a matrix $\mathbf{A}$: (a_{ij}) and $\mathbf{A}[i,j]$;
- vector concatenation: $\mathbf{x} \| \mathbf{y}$;
- $[1]_m = \overbrace{[1, \ldots, 1]}^{m}$;
- zip of two (equal-length) vectors: $\mathbf{x} \bowtie \mathbf{y} = [(x_i, y_i) \mid i \leftarrow [1, \ldots, |\mathbf{x}|]]$.
- matrix columnwise multiplication: $\mathbf{A} \otimes \mathbf{B}$. Namely, if $\mathbf{A} = (\mathbf{a}_1 \| \ldots \| \mathbf{a}_{n_A})$ and $B = (\mathbf{b}_1 \| \ldots \| \mathbf{b}_{n_B})$, then $\mathbf{A} \otimes \mathbf{B} = (\mathbf{c}_{1,1}, \ldots, \mathbf{c}_{n_A,n_B})$ where $\mathbf{c}_{i,j}[k] = \mathbf{a}_i[k] \cdot \mathbf{b}_j[k]$;
- protocol composition:
  - $P1 \oplus P2$ execute P_1 and P_2 in parallel;
  - $P1 + P2$ execute P_1 and P_2 sequentially;
  - $n \odot P$ execute n instances of P in parallel;
  - $n \cdot P$ execute P sequentially n times.

2.4 General FIM Algorithms

There exist several variations for the standard FIM algorithms. In this section, we just give the intuition about how these algorithms work, without introducing particular algorithm descriptions, as the actual implementations may vary. We present some particular private versions of these algorithms in Sec. 3.

Apriori This algorithm sequentially constructs all the frequent itemsets of size 1, then of size 2, until all the frequent sets of size $\hat{k}$. Any infrequent itemsets are immediately discarded. The frequent sets of size k are constructed only for those sets whose all $k-1$ subsets have been frequent. The straightforward implementation of this algorithm does not keep in memory the lists of transactions that contain sets of size other than 1, and on each iteration, the sets are constructed from the initial database. The way in which these sets are constructed depends on the particular algorithm instance. One possible recursive implementation is given in Alg. 1 (although it is actually linear and can be written out into a for-cycle as well, we have written it recursively to make it comparable to the other algorithms).

Algorithm 1: Apriori

```
Data: $M$ all the frequent sets of size $k-1$
Result: Frequent itemsets of size at least $k$
1 $F \leftarrow \emptyset$ ;
2 foreach $X_i \in M$ do
3     foreach $X_j \in M,\ j > i$ do
4         $R \leftarrow X_i \cup X_j$ ;
5         if $|R| \geq t$ then
6             $F \leftarrow F \cup \{R\}$ ;
7 if $F \neq \emptyset$ then
8     $F' \leftarrow \mathsf{Apriori}(F)$ ;
9 return $F \cup F'$ ;
```

Algorithm 2: Eclat

```
Data: $[P]$ all the frequent sets of size $k-1$ with a prefix $P$
Result: Frequent itemsets of size at least $k$ with a prefix $P$
1 foreach $X_i \in [P]$ do
2     $F_i \leftarrow \emptyset$ ;
3     foreach $X_j \in [P],\ j > i$ do
4         $R = X_i \cup X_j$ ;
5         if $|R| \geq t$ then
6             $F_i = F_i \cup \{R\}$ ;
7     if $F_i \neq \emptyset$ then
8         $F_i' = \mathsf{Eclat}(F_i)$ ;
9 return $\bigcup_i F_i'$ ;
```

Eclat Similarly to Apriori, this algorithm constructs set of size k from sets of size $k-1$. The main difference from Apriori is that this algorithm uses depth-first search, considering on one step not all the possible subsets of size k, but rather constrains one step to the sets of size k with a common *prefix* P of length $k-1$ (these are all sets of the form $P \cup \{x\}$ for $x \notin P$). Let the support of P be denoted $\sigma(P)$. For each item x, all possible frequent sets with prefix $P' := P \cup \{x\}$ can be constructed as $\sigma(P \cup \{x\}) \cap \sigma(P \cup \{y\})$ for all other sets $(P \cup \{y\})$, $y \neq x$. This new longer prefix is then processed recursively. The description is given in Alg. 2.

Diffset If the matrix columns, are dense, then instead of keeping a set of transactions that *contain* the given dataset, one could try to keep a set of transactions that *do not contain* the given dataset. Actually, even something more clever can be done. Another FIM algorithm Diffset [11] is similar to Eclat, but instead of keeping the set of transactions in each itemset, it keeps just the *sizes of supports* of sets of size $k-1$, and the differences between a set of size k and its subsets of size $k-1$. In this way, even if the initial matrix is not dense, the algorithm may still give better efficiency on later iterations.

Algorithm 3: Diffset

```
Data: $[P]$ all the frequent sets of size $k-1$ with a prefix $P$
Result: Frequent itemsets of size at least $k$ with a prefix $P$
1  foreach $X_i \in [P]$ do
2      $F_i \leftarrow \emptyset$ ;
3      foreach $X_j \in [P],\ j > i$ do
4          $R \leftarrow X_i \cup X_j$ ;
5          $\Delta(P,R) \leftarrow \Delta(P,X_j) \setminus \Delta(P,X_i)$ ;
6          $|\sigma(R)| = |\sigma(P)| - |\Delta(P,R)|$ ;
7          if $|\sigma(R)| \geq t$ then
8              $F_i = F_i \cup \{R\}$ ;
9      if $F_i \neq \emptyset$ then
10         $F_i' = \mathsf{Diffset}(F_i)$ ;
11 return $\bigcup_i F_i'$ ;
```

Namely, suppose that the itemsets $P \cup \{x\}$ and $P \cup \{y\}$, are frequent. The question is whether the itemset $P \cup \{x\} \cup \{y\}$ is frequent. Let $\Delta(P, P \cup \{x\})$ be the difference in supports of the itemsets P and $P \cup \{x\}$. We can compute $\sigma(P \cup \{x\}) = \sigma(P) \setminus \Delta(P, P \cup \{x\})$, as shown in Alg. 3.

3 Privacy-preserving FIM

In privacy-preserving setting, the initial data table is partitioned amongst several parties. The partitioning can be horizontal (rowwise), vertical (columnwise), or just any arbitrary sharing. Since FIM can in turn be used for various purposes such as associative rule mining, preserving privacy may be very important in some cases. For example, several shops may want to make some statistics of the contents of shopping carts without revealing what exactly has been sold. Privacy is especially important in cases where the shopping cart is associated with the customer.

It is not so easy to implement more efficient complex algorithms in a privacy-preserving way. Some simpler algorithms such as Apriori and Eclat have been implemented and optimized in [2,5,9]. Implementing an algorithm such as FP-tree (that we have not described in Sec. 2.4 since it is irrelevant in this paper) is not suitable for all security settings since its structure leaks a lot of information. In [9] that makes use of FP-trees, the tree is constructed *after* the frequent itemsets have been found (using Apriori-based algorithm), and its goal is to introduce noise into the public output and make the task differentially private. Differential privacy has been considered in [5,12,9], and these methods are based on adding noise to the data. Similar distortion-based approach is also used in [10]. There are also some solutions designed for specific initial data sharing, such as vertical or horizontal partitioning [7].

In this work, we mainly extend the results of [2]. The efficiency results are based on the operation complexities of Sharemind [4]. We do not consider differential privacy here. We assume that the algorithm should work with any initial partitioning. The algo-

Operation Call	Returned Value
$\mathsf{Mult}(\langle\!\![x]\!\!\rangle, \langle\!\![y]\!\!\rangle)$	$\langle\!\![x \cdot y]\!\!\rangle$
$\mathsf{OuterProd}(\langle\!\![\mathbf{x}]\!\!\rangle, \langle\!\![\mathbf{y}]\!\!\rangle)$	$\langle\!\![\mathbf{Z}]\!\!\rangle$ where $z_{ij} = x_i \cdot y_j$
$\mathsf{ColSum}(\langle\!\![\mathbf{X}]\!\!\rangle)$	$\langle\!\![\mathbf{y}]\!\!\rangle$, $y_i = \sum_{j=1}^{m} x_{ji}$
$\mathsf{ShareConv}(\langle\!\![x]\!\!\rangle, k)$	$\langle\!\![y]\!\!\rangle$, $x \in \mathbb{Z}_2$, $y \in \mathbb{Z}_{2^k}$, $x = y$
$\mathsf{Shuffle}(\langle\!\![\mathbf{x}]\!\!\rangle)$	$\langle\!\![\mathbf{y}]\!\!\rangle$ a random reordering of $\mathbf{x}$
$\mathsf{Equal}(\langle\!\![x]\!\!\rangle, \langle\!\![y]\!\!\rangle)$	$\langle\!\![x == y]\!\!\rangle$
$\mathsf{LessThan}(\langle\!\![x]\!\!\rangle, \langle\!\![y]\!\!\rangle)$	$\langle\!\![x \leq y]\!\!\rangle$
$\mathsf{Declassify}(\langle\!\![x]\!\!\rangle)$	x

Table 1. Basic blackbox operations

rithms of [2] are based on bit matrix representation, and hence it works with the same efficiency for both sparse and dense matrices. We want a solution that works better with sparse matrices.

3.1 Basic Black-box Operations

The algorithms will use some blackbox operations that in general depend on the underlying SMC platform, and whose implementation is not the part of development of FIM algorithms. The notation used for basic operations is presented in Tab. 1. While notation $\mathsf{Protocol}(\mathbf{x})$ denotes the application of a protocol to the vector of inputs $\mathbf{x}$, we use $\overline{\mathsf{Protocol}}(n,k)$ for denoting the complexity of running the protocol on a k-bit vector of inputs $\mathbf{x}$ such that $|\mathbf{x}| = n$ (if the protocol has two input vectors $\mathbf{x}$ and $\mathbf{y}$, we take $n = |\mathbf{x}| + |\mathbf{y}|$, as it will be sufficient to estimate the complexity of our particular protocols). As a shorthand notation, we write just $\overline{\mathsf{Protocol}}(k)$ for the protocol whose inputs are not vectors, but single elements. We also sometimes use explicit notation $\widetilde{\mathsf{Protocol}}(n,k)$ for the number of rounds and $\overrightarrow{\mathsf{Protocol}}(n,k)$ for communication (in bits).

3.2 Bit Matrix Representation

First, we describe the existing implementations of [2] based on representing the data table as a secret shared bit matrix. Our own algorithms will be based on this work.

Initially, the n items and m transactions are ordered in some way. For each item $j \in \{1,\ldots,n\}$, a shared bit vector $[\![\mathbf{b}_i]\!]$ of length m is stored. It is defined as $[\![b_{ij}]\!] = 1$ iff the i-th transaction contains the i-th item. These vectors comprise a bit matrix $[\![\mathbf{B}]\!] = ([\![\mathbf{b}_1]\!] \mid \ldots \mid [\![\mathbf{b}_n]\!])$ of size $m \times n$.

We give the two complexity estimations: the number of *rounds*, and the number of *communicated bits* that are spent on computing one frequent itemset of size k. The reason why we take this metric is that it is good when comparing bit representation with the set representation (which we are going to present in this paper). If we parallelize the computation of several itemsets of size k, then it can be done in the bit representation as well as in the set representation.

Algorithm 4: Privacy Preserving Apriori

```
   Data: $[\![\mathbf{D}]\!]$ the initial data matrix
   Result: All the frequent itemsets
   // Compute support for all cover vectors
1  $[\![\mathbf{s}]\!] \leftarrow \mathsf{CountOnes}([\![\mathbf{D}]\!])$ ;
   // Declassify index vector of frequent columns
   $f_1 \leftarrow \mathsf{Declassify}([\![\mathbf{s}]\!] \geq [\![t]\!])$ ;
   // Gather frequent column data
2  $F_1 \leftarrow \{[\![\mathbf{D}]\!][i] \mid f_1[i] = 1\}$ ;
3  $[\![\mathbf{M}_1]\!] \leftarrow [\![\mathbf{D}]\!][*, F_1]$ ;
   // Validate candidate itemsets until size $\hat{k}$
   foreach $k \in \{2, \ldots, \hat{k}\}$ do
       // Generate candidates
4      $(C_k, I_1, I_2) \leftarrow GenCandidates(F_{k-1})$ ;
       // Compute covers for all candidate sets
5      $[\![\mathbf{M}_k]\!] \leftarrow [\![\mathbf{M}_{k-1}]\!][*, I_1] \otimes [\![\mathbf{M}_{k-1}]\!][*, I_2]$ ;
       // Compute support for all covers
6      $[\![\mathbf{s}]\!] \leftarrow \mathsf{CountOnes}([\![\mathbf{M}_k]\!])$ ;
7      $f_k \leftarrow \mathsf{Declassify}([\![\mathbf{s}]\!] \geq [\![t]\!])$ ;
       // Remember frequent sets
8      $F_k \leftarrow \{C_k[i] \mid f_k[i] = 1\}$ ;
9      $[\![\mathbf{M}_k]\!] \leftarrow [\![\mathbf{M}_k]\!][*, F_k]$ ;
10 return $\bigcup_k F_k$ ;
```

First, we present privacy-preserving implementations of some of the algorithms presented in Sec. 2.4, taken from [2]. They are entirely based on additive secret sharing. The descriptions of subalgorithms used in these algorithms are given in Tab. 1.

Apriori In Sec. 2.4, we omitted the precise description of how the sets of size k are constructed. In [2], two possible implementations of Apriori are proposed: *with* and *without* caching. In the first case, the sets of size k are constructed from the intersections of size $k-1$, what makes Apriori similar to Eclat. In the second case, the intersection of k vectors that correspond to sets of size 1 have to be found, and the algorithm becomes too inefficient in practice. The cached version of the algorithm is shown in Alg. 4.

The function *GenCandidates* filters out the possible pairs of $k-1$ itemsets for which it makes sense to find the intersection and obtain a frequent set of size k. For example, if AB and AC are frequent, but BC is not, then it makes no sense to construct ABC since it cannot be frequent in any case, and hence the pair (AB, BC) can be discarded. This function returns a set of itemsets of size k (the quantity C_k) and two sets of indices I_1 and I_2 that show how C_k is being constructed from M_k ($C_k[j] = I_1[j] \cup I_2[j]$ for all j) In particular, the Apriori algorithm of [2] is implemented similarly to Eclat, in such a way that, for each j, the itemsets $I_1[j]$ and $I_2[j]$ share the same prefix. The only way in which it differs from Eclat is the traversal order.

Algorithm 5: Privacy Preserving Eclat (PPEclat)

```
   Data: $[\![\mathbf{M}]\!]$ the columns corresponding to itemsets sharing the same prefix $P$
   Result: All the frequent sets with all the extensions of the prefix $P$
1  $[\![\mathbf{M}]\!] \leftarrow [\![\mathbf{M}]\!][*,X] \otimes [\![\mathbf{M}]\!][*,N]$ ;
   // Compute supports ;
2  $[\![\mathbf{s}]\!] \leftarrow \mathsf{CountOnes}([\![\mathbf{M}]\!])$ ;
3  $f \leftarrow \mathsf{Declassify}([\![s]\!] \geq [\![t]\!])$ ;
   // Construct new frequent item sets
4  $F \leftarrow \{X\}, F^* \leftarrow \{X \cup N[i] \mid f[i] = 1\}$ ;
   // If we have reached the target set size, return ;
   if $|X| + 1 \geq \hat{k}$ then
5      return $F^*$ ;
   // See how we could extend the current frequent sets
   foreach $Y \in F^*$ do
6      $N^* = \{Z \in F^* \mid Y \preceq Z\}$ // Recursively extend the frequent itemset candidate
7      $F \leftarrow F \cup \mathsf{PPEclat}(Y, N^*, [\![\mathbf{M}]\!][*,N^*])$ ;
8  return $F$ ;
```

PP Eclat This algorithm works works similarly to the memory cached Apriori. The communication and round complexities of computing all the frequent sets of size k are the same. The difference is in the way in which the itemsets are traversed, and how many intermediate computations should be kept in memory at once. Not all the itemsets are constantly stored in memory, but some of them are discarded and are recomputed again in the recursive Depth-First-Search process, while Apriori uses pure Breadth-First-Search. The recursive step of this algorithm is given in 5. The notation $Y \preceq Z$ denotes the itemsets that share the same prefix, and where the remaining element of Z has higher index than the remaining element of Y (this is needed just not to generate the same sets multiple times). An optimized version of Eclat does a certain amount of caching, giving hybrid Eclat-Apriori solutions, which have also been implemented in [2].

PP Diffset This algorithm has not been implemented in [2], but it could be analogous to Eclat. The reason why it has not been implemented is that it gives no advantage compared to Eclat. Namely, since information about the columns should remain private, and the difference between a set of size k and its subset of size $k-1$ can be up to $m-t$ (any frequent set is found in at most m and at least t transactions), we cannot make any significant advantage unless t is very large. Moreover, in the case of bit matrix implementation, we need m bits to represent an itemset anyway, and hence Diffset gives no advantage at all.

The main step of presented Privacy-Preserving FIM algorithms We now make a small summary of the privacy-preserving FIM algorithms proposed above. A similar property of these algorithms is that, on each step of each iteration, all they compute a frequent itemset of size k, based on the frequent itemsets of size $k-1$ (the difference of these algorithms is mainly in which of k-sets are computed in parallel). The basis of finding a k-set from $k-1$ subsets is private *set intersection* (for Apriori and Eclat), or *set difference* (for Diffset).

A straightforward approach to find a set intersection is to represent each set as a characteristic vector of the item universe, and then find their pointwise product. This method is used in the bit matrix representation.

Although all matrix elements are bits, at some moment the column sum has to be computed. In this way, in [2] the matrix elements are initially all at least $\log m$-bit, since the maximal value that the sum may take is m. For very sparse sets, such an encoding may be excessive due to large amount of zeroes that will not be needed anyway. In [2], finding an intersection of two itemsets i and j and checking its cardinality is implemented as:

1. multiply pointwise two $\log m$-bit vectors of length m;
2. sum the obtained m products up;
3. compare the obtained $\log m$-bit number with a $\log m$-bit number t.

In Sec. 3.5, we discuss whether it is reasonable to keep all the bits in $\log m$ format, or there is another more efficient way to find the sum.

Precise complexities of bit matrix approach Implementing the intersection straightforwardly (as in [2]), this is $m \odot \overline{\mathsf{Mult}}(\log m)$ for multiplying pointwise two $\log m$ bit vectors of length m. Another possibility to do the same thing is to keep all the bits in $\mathbb{Z}_2$, doing the share conversion *after* the multiplication. Now the multiplication of m bit pairs has complexity $m \odot \overline{\mathsf{Mult}}(1)$, and the share conversion $m \odot \overline{\mathsf{ShareConv}}(\log m)$, which can be useful if ShareConv is implemented more efficiently than Mult.

Note that, if we need to compute $\mathsf{Mult}(\langle\!\langle \mathbf{a}_i \rangle\!\rangle, \langle\!\langle \mathbf{b}_j \rangle\!\rangle)$ for all $i \in \{1,\dots,n_a\}$, $j \in \{1,\dots,n_b\}$, then we could apply $\mathsf{OuterProd}(\langle\!\langle \mathbf{a}_1 \rangle\!\rangle | \ \dots \| \langle\!\langle \mathbf{a}_{n_a} \rangle\!\rangle, \langle\!\langle \mathbf{b}_1 \rangle\!\rangle \| \dots \| \langle\!\langle \mathbf{b}_{n_b} \rangle\!\rangle)$ instead, which has the same operation complexity as $(n_a + n_b)\overline{\mathsf{Mult}}(1)$. Hence treating intersections independently may look like cheating. We must ensure that our protocol for sparse matrices achieves the same property. Namely, if some set participates in some intersection, then for several intersections we have to pay as much as for one intersection. Hence we need to think about something similar for the set based approach. We discuss it in Sec. 3.3.

3.3 Set Representation

Set representation makes sense for sparse and dense matrices. First, let us assume that the columns are sparse, i.e each column of the matrix contains at most m' entries for $m' \ll m$. We will now use an $m' \times n$ matrix for data table representation. Each column will now contain not the characteristic bit vector, but the indices of transactions straightforwardly. Encoding a number from $[1,\dots,m]$ requires $\log m$ bits. If the table contains at most nm' nonzero entries, then $nm' \cdot \log m$ bits are sufficient to encode it. If the size of some column is $m_j < m'$, then some $m' - m_j$ of its entries are set to 0. The order of values in a column does not matter.

Algorithm 6: CountOnes for sorted inputs

```
  Data: $\langle\langle \mathbf{a} \rangle\rangle$ is a *sorted* vector
  Result: $\langle\langle c \rangle\rangle$ is the number of non-zeroes in $\mathbf{a}$
1 $\langle\langle b_1 \rangle\rangle \leftarrow \langle\langle a_1 \rangle\rangle$ ;
2 foreach $i \in \{1, \dots, |\mathbf{a}|\}$ do
3     $\langle\langle b_i \rangle\rangle \leftarrow \langle\langle a_i \rangle\rangle \oplus \langle\langle a_{i-1} \rangle\rangle$ ;
4 $\langle\langle c \rangle\rangle \leftarrow \bigoplus_{i=1}^{|\mathbf{a}|} i \cdot \langle\langle b_i \rangle\rangle$ ;
5 return $\langle\langle c \rangle\rangle$ ;
```

Building Blocks First of all, we present some algorithms that will be later used as building blocks for set operations.

– **Radix Sort** (Rsort). We use the algorithm proposed in [3, Algorithm 3]. That paper uses notation $m \cdot \mathsf{Protocol}(n)$ to denote that Protocol have been run sequentially m times on n inputs. Although [3, Algorithm 3] does not include the ring in which the computation is made (this is implicit since it is always the same throughout their paper), the number of bits is actually very important in our settings. Hence we write $m \cdot \overline{\mathsf{Protocol}}(n,k)$ to denote that the protocol has been run m times on n inputs of k bits each (the details of protocol composition can be found in Sec. 2). The precise interpretation of protocols is given in Tab. 1.
The secure operation complexity of the radix sort of [3] is $k \cdot (n \odot \overline{\mathsf{Mult}}(k) + n \odot \overline{\mathsf{ShareConv}}(k) + \overline{\mathsf{Shuffle}}(n, 2k) + n \odot \overline{\mathsf{Declassify}}(k))$. More precisely, the protocol runs in k steps, on each of which the elements are sorted according to one bit, using counting sort (we further denote one iteration of Rsort as Csort). In our algorithms, we often need to sort the values just by one bit, and that makes Csort especially useful.

– **Quicksort** (Qsort). We use the algorithm proposed in [6, Protocol 1] to sort n elements of k bits each. More precise secure operation complexity of this algorithm is given in [3], and in average case this is $\overrightarrow{\mathsf{Shuffle}}(n,k) + \log n \cdot (n \odot \overrightarrow{\mathsf{LessThan}}(k) + n \odot \overrightarrow{\mathsf{Declassify}}(k))$. Since the case only depends on the random shuffle, it turns out that the worst case comes with negligible probability, and we may indeed believe that we get the average case in practice.

– **Counting the number of ones in a sorted bit list** (CountOnes)
This operation is used on each iteration in each of the three FIM algorithms, and it can be generalized to finding the number of non-zero entries by performing a comparison with 0 first. Let $\mathbf{a}$ be a sorted list for which we know that the zero elements come first. If the values are shared over a sufficiently large $\mathbb{Z}_n$, where $n \geq |\mathbf{a}|$, then clearly we can just sum up the entries of $\mathbf{a}$, which is a free operation in additive secret sharing. However, if we are using bitwise xor sharing, then we cannot sum up the elements of $\mathbb{Z}_2$ straightforwardly. We do not want to make a share conversion since it is expensive. However, we know that $\oplus$ (xor) operation is free in xor-shared data.
The algorithm is shown in Alg. 6. Since the list is already sorted, $\mathbf{b}$ is a vector such that its coordinates are 0 everywhere except the one coordinate whose index is equal

Algorithm 7: MultByBit for xor shared inputs

```
  Data: $\langle\langle \mathbf{a} \rangle\rangle$ and $\langle\langle b \rangle\rangle$
  Result: $\langle\langle \mathbf{c} \rangle\rangle$ such that $\mathbf{c} = \mathbf{a}$ iff $b == 1$
1 $\langle\langle \mathbf{C} \rangle\rangle = \mathsf{OuterProd}(\langle\langle \mathbf{a} \rangle\rangle, \langle\langle b \rangle\rangle)$ ;
  // since $|[b]| = 1$, there is just one column in $\mathbf{C}$
2 return $\mathbf{C}[*, 0]$ ;
```

to the number of non-zero entries in $\mathbf{a}$ (if all the entries are 0, then $\mathbf{b} = \mathbf{0}$). Hence c is equal to the number of non-zero entries in $\mathbf{a}$. Computing $i \cdot \langle\langle b_i \rangle\rangle$ is free since i is public and $b_i \in \{0, 1\}$, and the multiplication can be written out as $i \wedge [\langle\langle b_i \rangle\rangle]_{\log|\mathbf{a}|}$, which is free for xor shared data. The overall secure operation complexity is 0, and we conclude is that this operation is free for both additive and bitwise secret sharing.

- **Finding the Characteristic Vector** ($\mathsf{CharVec}$) Given a xor shared k-bit value i, the task is to find a bit vector $\mathbf{b}$ of length 2^k such that its i-th coordinate is 1 and the rest are 0. We use the algorithm proposed in [8] since it allows to reduce the complexity to the order $k\sqrt{2^k}$, and in our case the value of 2^k will be pretty large in practice. The precise complexity of this algorithm is at most $k \cdot \overline{\mathsf{OuterProd}}(\sqrt{2^k}, 1)$.
- **Multiplication by a bit for xor sharing (**$\mathsf{MultByBit}$**)** For xor shared data $\mathbf{a} = (a_1 \| \ldots \| a_k)$ and b, one may compute $\mathsf{MultByBit}(\mathbf{a}, b) = (a_1 \wedge b, \ldots, a_k \wedge b)$, which now reduces to $\mathsf{OuterProd}(a_1 \| \ldots \| a_k, b)$. This simple protocol is given in Alg. 7, and its complexity is $\overline{\mathsf{OuterProd}}(|\mathbf{a}| + 1, 1)$.
- **Outer Equality (**$\mathsf{OuterEq}$**)**. This protocol is in fact just an outer product where the multiplication is replaced by equality. This is defined as $\mathsf{OuterEq}(\langle\langle \mathbf{a} \rangle\rangle, \langle\langle \mathbf{b} \rangle\rangle) = \mathbf{C}$ where $c_{ij} = (a_i = b_j)$. We define the protocol for xor shared data only. Given two k-bit values a and b, for the i-th bits of a and b we can define $(a_i = b_i) \leftarrow (1 \oplus a_i \oplus b_i)$, and this computation is free. However, if $k > 1$, then we have $(a = b) \leftarrow \bigwedge_{i=1}^{k}(1 \oplus a_i \oplus b_i)$. For $k = 2$, these expression can be reduced to $\mathsf{OuterProd}$, but we will need this algorithm for larger k. Another solution is to convert a to its binary representation: a 2^k-bit vector $\mathbf{a}'$ such that $a'_a = 1$ and $a'_i = 0$ for all $i \neq a$. In this case, we can define $(a = b) \leftarrow \bigoplus_{i=1}^{k}(a'_i \wedge b'_i)$, where finding all possible $a'_i \wedge b'_i$ is now equivalent to $\mathsf{OuterProd}$. Although finding the characteristic vector is rather expensive, the overall complexity will be linear in $|\mathbf{a}| + |\mathbf{b}|$. The protocol is formally given on Alg. 8. Its total complexity is $(|\mathbf{a}| + |\mathbf{b}|) \odot \overline{\mathsf{CharVec}}(k) + \overline{\mathsf{OuterProd}}((|\mathbf{a}| + |\mathbf{b}|) \cdot 2^k, 1)$

The summary of building block definitions can be found in in Tab. 2, and their secure operation complexities in Tab. 3.

Set Operations We now present the algorithms for set intersection and set difference. We represent sets with arrays. Although leaking the set size would be too much, we can assume that there is a known upper bound m on the number of elements. If the set has less than m elements, then the entries that represent missing elements are set to 0. Since 0 is now reserved, we start element indexation with 1.

Algorithm 8: Pairwise equality $\mathsf{OuterEq}$

```
Data: $k$-bit vectors $\langle\langle\mathbf{a}\rangle\rangle$ and $\langle\langle\mathbf{b}\rangle\rangle$ such that $|\mathbf{a}| = |\mathbf{b}|$
Result: $\langle\langle\mathbf{C}\rangle\rangle$ such that $c_{ij} = \mathbf{a}$
1 foreach $i \in \{1, \dots, |\mathbf{a}|\}$ do
2     $\langle\langle\mathbf{A}[i,*]\rangle\rangle \leftarrow \mathsf{CharVec}(\langle\langle a_i\rangle\rangle, k)$ ;
3     $\langle\langle\mathbf{B}[i,*]\rangle\rangle \leftarrow \mathsf{CharVec}(\langle\langle b_i\rangle\rangle, k)$ ;
  // find the conjunctions for each bit separately
4 foreach $j \in \{1, \dots, 2^k\}$ do
5     $\langle\langle\mathbf{C}_k\rangle\rangle = \mathsf{OuterProd}(\mathbf{A}[*,j], \mathbf{B}[*,j])$ ;
  // xor all the conjunctions up
6 $\langle\langle\mathbf{C}\rangle\rangle \leftarrow \bigoplus_{i=1}^{k} \langle\langle\mathbf{C}_k\rangle\rangle$ ;
7 return $\langle\langle c\rangle\rangle$ ;
```

Algorithm Call	Returned Value
$\mathsf{Csort}(\langle[\mathbf{x}]\rangle \bowtie \langle[\mathbf{b}]\rangle)$	$\langle[\mathbf{y}]\rangle$ which is $\mathbf{x}$ sorted by $\langle[\mathbf{b}]\rangle$
$\mathsf{Rsort}(\langle[\mathbf{x}]\rangle)$	$\langle[\mathbf{y}]\rangle$ which is sorted $\mathbf{x}$
$\mathsf{Qsort}(\langle[\mathbf{x}]\rangle)$	$\langle[\mathbf{y}]\rangle$ which is sorted $\mathbf{x}$
$\mathsf{CountOnes}(\langle\langle\mathbf{x}\rangle\rangle)$	$\langle\langle c\rangle\rangle$ s.t $c = \sum_{i=1}^{n} x_i$, $x_i \in \mathbb{Z}_2$
$\mathsf{CharVec}(\langle\langle a\rangle\rangle, k)$	2^k-bit vector $\mathbf{b}$ s.t $b_i = 1$ iff $i = a$
$\mathsf{MultByBit}(\langle\langle\mathbf{a}\rangle\rangle, \langle\langle b\rangle\rangle)$	$\mathbf{a}$ if $b == 1$ and $\mathbf{0}$ otherwise
$\mathsf{OuterEq}(\langle\langle\mathbf{a}\rangle\rangle, \langle\langle\mathbf{b}\rangle\rangle)$	$\mathbf{C}$ s.t $c_{ij} = (a_i == b_j)$

Table 2. Building block operations

Algorithm	Secure Operation Complexity
$\overline{\mathsf{Csort}}(n,k)$	$n \odot \overline{\mathsf{Mult}}(k) + n \odot \overline{\mathsf{ShareConv}}(k) + \overline{\mathsf{Shuffle}}(n, 2k) + n \odot \overline{\mathsf{Declassify}}(k)$
$\overline{\mathsf{Rsort}}(n,k)$	$k \cdot (n \odot \overline{\mathsf{Mult}}(k) + n \odot \overline{\mathsf{ShareConv}}(k) + \overline{\mathsf{Shuffle}}(n, 2k) + n \odot \overline{\mathsf{Declassify}}(k))$
$\overline{\mathsf{Qsort}}(n,k)$	$\overline{\mathsf{Shuffle}}(n,k) + \log n \cdot (n \odot \overline{\mathsf{LessThan}}(k) + n \odot \overline{\mathsf{Declassify}}(k))$
$\overline{\mathsf{CountOnes}}(n,k)$	0
$\overline{\mathsf{CharVec}}(k)$	$k \cdot \overline{\mathsf{OuterProd}}(\sqrt{2^k}, 1)$
$\overline{\mathsf{MultByBit}}(k)$	$\overline{\mathsf{OuterProd}}(\lvert\mathbf{a}\rvert + 1, 1)$.
$\overline{\mathsf{OuterEq}}(n,k)$	$2n \odot \overline{\mathsf{CharVec}}(k) + k \odot \overline{\mathsf{OuterProd}}(2n, 2^k)$

Table 3. Complexity of building block operations

Algorithm 9: Set intersection $\mathsf{Set}_\cap$

```
Data: $\langle\!\![\mathbf{a}]\!\!\rangle$ and $\langle\!\![\mathbf{b}]\!\!\rangle$ where all elements except 0 are unique
Result: $\langle\!\![\mathbf{c}]\!\!\rangle = \langle\!\![\mathbf{a} \cap \mathbf{b}]\!\!\rangle$, $|\mathbf{c}| = \min(|\mathbf{a}|, |\mathbf{b}|)$
1 $\langle\!\![\mathbf{d}]\!\!\rangle \leftarrow \mathsf{Sort}(\langle\!\![\mathbf{a}]\!\!\rangle \| \langle\!\![\mathbf{b}]\!\!\rangle)$ ;
2 $\langle\!\![t_1]\!\!\rangle \leftarrow 0$ ;
3 $\langle\!\![s_1]\!\!\rangle \leftarrow 0$ ;
4 foreach $i \in \{1, \dots, |\mathbf{a}| + |\mathbf{b}| - 1\}$ do
5     $\langle\!\![t_i]\!\!\rangle = \mathsf{Equal}(\langle\!\![d_i]\!\!\rangle, \langle\!\![d_{i-1}]\!\!\rangle)$ ;
6     $\langle\!\![s_i]\!\!\rangle = \mathsf{MultByBit}(\langle\!\![t_i]\!\!\rangle, \langle\!\![d_i]\!\!\rangle)$ ;
7 $\langle\!\![\mathbf{c}]\!\!\rangle = \mathsf{Csort}(\langle\!\![\mathbf{t}]\!\!\rangle \bowtie \langle\!\![\mathbf{s}]\!\!\rangle)$ ;
8 return $\langle\!\![\mathbf{c}]\!\!\rangle[0 : \min(|\mathbf{a}|, |\mathbf{b}|), 1]$ ;
```

- **Set intersection of k-bit elements**. Let **a** and **b** be the two arrays that represent the sets whose intersection we are going to find. Let $|\mathbf{a}| = n_1$, $|\mathbf{b}| = n_2$, $n = n_1 + n_2$. The computation of $\mathbf{c} = \mathbf{a} \cap \mathbf{b}$ is given in Alg. 9.

  First, the algorithm sorts the straightforward concatenation $\langle\!\![\mathbf{a}]\!\!\rangle \| \langle\!\![\mathbf{b}]\!\!\rangle$ by value, so that if an element occurs in both sets, then these two elements appear together in the resulting sorted array. Hence if there are two sequential instances of the same element $\mathbf{d}_{i-1}$ and $\mathbf{d}_i$ in $\mathbf{d}$, then we chose $\mathbf{s}_i = \mathbf{d}_i$. Everywhere else $\mathbf{s}_i = 0$. In this way, we keep exactly those elements that are present in both **a** and **b**. We sort the elements once more according to the bits $\mathbf{t}_i$ in order to get all the zeroes into the end of the array, so that the excessive zeroes could be safely removed. The intersection contains at most $\min(n_1, n_2)$ elements.

  We have $|\mathbf{s}| = |\mathbf{c}| = |\mathbf{a}| + |\mathbf{b}| = n$. The iterations of the for-cycle do not depend on each other and hence are parallelizable. The number of used operations is $\overline{\mathsf{Sort}}(n,k) + n \odot (\overline{\mathsf{Equal}}(k) + \overline{\mathsf{MultByBit}}(k)) + \overline{\mathsf{Csort}}(n,k)$.

  We assume that Sort can be instantiated either to Rsort or Qsort, and one may be preferable to the other depending on the parameters and whether we want to win more in rounds or in communication. However, note that the second sort depends on one bit only. Hence Csort is preferable. Note that, if we agree to leak the precise number of nonzero entries, then Csort may be replaced with a Shuffle followed by Declassify. The reason is that, after shuffling, the positions of zeroes will be random. Declassifying **t** reveals at most the number of zeroes. All entries such that $t_i = 0$ can be discarded.
- **Set difference of k-bit numbers:** the algorithm is analogous to set intersection. The computation of $\mathbf{c} = \mathbf{a} \setminus \mathbf{b}$ is given in Alg. 10.

  The difference is that now we should leave exactly the elements that are in **a**, but not in **b**. In order to do this, we add a bit 1 to each element of **a** and a bit 0 to each element of **b**, so that now we sort pairs of elements. After sorting, if two elements are equal, then the bit 0 comes before 1. Now if two sequential elements are the same in **c**, we set $\mathbf{t}_i = 0$. Otherwise, we set $\mathbf{t}_i = 1$ unless the element comes from the second set (has the label 0).

  Adding a bit to **a**, **b** and removing it from **d** is free in any secret sharing scheme since we treat this concatenation just as a pairing. The number of used operations

Algorithm 10: Set difference $\mathsf{Set}_{\setminus}$

```
Data: $\langle\!\![\mathbf{a}]\!\!\rangle$ and $\langle\!\![\mathbf{b}]\!\!\rangle$ where all elements except 0 are unique
Result: $\langle\!\![\mathbf{c}]\!\!\rangle = \langle\!\![\mathbf{a} \setminus \mathbf{b}]\!\!\rangle$, $|\mathbf{c}| = |\mathbf{a}|$
1  $\langle\!\![\mathbf{a}']\!\!\rangle := \langle\!\![\mathbf{a}]\!\!\rangle \bowtie [1]_{|\mathbf{a}|}$ ;
2  $\langle\!\![\mathbf{b}']\!\!\rangle := \langle\!\![\mathbf{b}]\!\!\rangle \bowtie [0]_{|\mathbf{b}|}$ ;
3  $\langle\!\![\mathbf{d}]\!\!\rangle \leftarrow \mathsf{Sort}(\langle\!\![\mathbf{a}']\!\!\rangle \| \langle\!\![\mathbf{b}']\!\!\rangle)$ ;
4  $\langle\!\![t_1]\!\!\rangle \leftarrow 0$ ;
5  $\langle\!\![s_1]\!\!\rangle \leftarrow 0$ ;
6  foreach $i \in \{1, \ldots, |\mathbf{a}| + |\mathbf{b}| - 1\}$ do
7      $\langle\!\![t_i]\!\!\rangle = \mathsf{Equal}(\langle\!\![d_i]\!\!\rangle[0], \langle\!\![d_{i-1}]\!\!\rangle[0]) - \langle\!\![d_i]\!\!\rangle[1]$ ;
8      $\langle\!\![s_i]\!\!\rangle = \mathsf{MultByBit}(\langle\!\![t_i]\!\!\rangle, \langle\!\![d_i]\!\!\rangle[0])$ ;
9  $\langle\!\![\mathbf{c}]\!\!\rangle = \mathsf{Csort}(\langle\!\![\mathbf{t}]\!\!\rangle \bowtie \langle\!\![\mathbf{s}]\!\!\rangle)$ ;
10 return $\langle\!\![\mathbf{c}]\!\!\rangle[0 : |\mathbf{a}|, 1]$ ;
```

Algorithm Call	Returned Value	Secure Operation Complexity
$\mathsf{Set}_{\cap}(\langle\!\![\mathbf{a}]\!\!\rangle \| \langle\!\![\mathbf{b}]\!\!\rangle)$	$\langle\!\![\mathbf{c}]\!\!\rangle = \langle\!\![\mathbf{a} \cup \mathbf{b}]\!\!\rangle$	$\overline{\mathsf{Sort}}(n,k) + n \odot (\overline{\mathsf{Equal}}(k) + \overline{\mathsf{MultByBit}}(k)) + \overline{\mathsf{Csort}}(n,k)$
$\mathsf{Set}_{\setminus}(\langle\!\![\mathbf{a}]\!\!\rangle \| \langle\!\![\mathbf{b}]\!\!\rangle)$	$\langle\!\![\mathbf{c}]\!\!\rangle = \langle\!\![\mathbf{a} \setminus \mathbf{b}]\!\!\rangle$	$\overline{\mathsf{Sort}}(n,k+1) + n \odot (\overline{\mathsf{Equal}}(k) + \overline{\mathsf{MultByBit}}(k)) + \overline{\mathsf{Csort}}(n,k)$

Table 4. Set operations

is almost the same as for the set intersection, except one extra bit in $\overline{\mathsf{Sort}}(n, k+1)$ that adds a negligible complexity overhead compared to $\overline{\mathsf{Set}_{\cap}}$.

The summary of set operation complexities is given in Tab. 4

Parallelizing Set Intersections As we have shown in Sec. 3.2, in the bit matrix approach, for each set that participates in some intersection, one should pay as much as for one intersection only (thanks to OuterProd). We show that a similar property can be achieved by our set intersection algorithm.

Let the sparse matrix be represented by ℓ pairs $(tid, item)$. Let $A_i \cap B_j$, $i \in I_A$, $j \in I_B$, $|I_A| = n_A$, $|I_B| = n_B$ be the intersections that we need to find.

Algorithm The algorithm for finding all these intersections using set-based approach is shown in 11. First of all, on the Line 1, all the $(tid, item)$ pairs are sorted by tid, getting a vector $\mathbf{d}$ of size ℓ (in the algorithm, these pairs are already distributed to columns, but they could be just a sequence of pairs as well). Now the goal is to find all the transactions that belong to each possible pair A_i, B_j. That is, in the $\ell \times \ell$ intersection matrix, we need to locate all the entries where the same item is present in both the row and the column. Since the matrix is symmetric, and there are at most n instances of each tid, these entries are accumulated at distance at most n from the diagonal. Moreover, half of them can be removed due to symmetry. The upper bound on the number of such entries is $\frac{\ell n}{2}$ (start with an $n \times n$ square in the corner, move it along the diagonal to the opposite corner, and take $\frac{1}{2}$ of the covered entries).

```
Algorithm 11: Multiple set intersection $\mathsf{MSet}_\cap$
  Data: $\langle\!\langle \mathbf{a}_i \rangle\!\rangle$, $\langle\!\langle \mathbf{b}_j \rangle\!\rangle$, $i \in I_A$, $j \in I_B$, where in each vector all elements except 0 are unique
  Result: $\langle\!\langle \mathbf{c}_{ij} \rangle\!\rangle = \langle\!\langle \mathbf{a}_i \cap \mathbf{b}_j \rangle\!\rangle$, $|\mathbf{c}_{ij}| = |\mathbf{a}_i|$, $i \in I_A$, $j \in I_B$
1  $\langle\!\langle \mathbf{d} \rangle\!\rangle \leftarrow \mathsf{Sort}(\langle\!\langle \mathbf{a}_1 \bowtie 1 \rangle\!\rangle \| \cdots \| \langle\!\langle \mathbf{a}_{n_A} \bowtie n_A \rangle\!\rangle \| \langle\!\langle \mathbf{b}_1 \bowtie 1 \rangle\!\rangle \| \cdots \| \langle\!\langle \mathbf{b}_{n_B} \bowtie n_B \rangle\!\rangle)$ ;
2  foreach $i \in \{1, \ldots, |\mathbf{d}|\}$ do
3      foreach $j \in \{\max(0, i-n), \ldots, \max(0, i-n) + n\}$ do
4          $\langle\!\langle t_{ij} \rangle\!\rangle = \mathsf{Equal}(\langle\!\langle d_i \rangle\!\rangle[0], \langle\!\langle d_j \rangle\!\rangle[0])$ ;
5          $\langle\!\langle s_{ij} \rangle\!\rangle = \mathsf{MultByBit}(\langle\!\langle t_{ij} \rangle\!\rangle, \langle\!\langle d_i \rangle\!\rangle[0])$ ;
6          $\langle\!\langle \mathbf{d}'_{\mathbf{ij}} \rangle\!\rangle = (\langle\!\langle d_i \rangle\!\rangle[1] \| \langle\!\langle d_j \rangle\!\rangle[1], \langle\!\langle s_{ij} \rangle\!\rangle)$
7  $\langle\!\langle \mathbf{e} \rangle\!\rangle = \mathsf{Sort}(\langle\!\langle \mathbf{d}' \rangle\!\rangle)$ ;
8  foreach $i \in \{1, \ldots, n\}$ do
9      $\mathbf{c}_i \leftarrow \mathbf{e}[ni, \ldots, n(i+1)][1]$ ;
10 return $\langle\!\langle \mathbf{c}_{1,1} \rangle\!\rangle, \ldots, \langle\!\langle \mathbf{c}_{n_A,n_B} \rangle\!\rangle$ ;
```

For each of these $\frac{\ell n}{2}$ entries, we perform one comparison, one multiplication, and leave a pair $(A_i \| B_j, s_i)$ behind, where $A_j \| B_j$ will denote the set $A_i \cap B_j$, and $s_i = a_i$ if $a_i \in A_i \cap B_j$, and 0 otherwise.

If we do not leak the precise column density, but still know that the number of nonzero entries in each column is at most m', we have the same number m' for all the intersections we have obtained so far. Hence after we sort the elements according to the itemset $A_i \| B_j$, we know that each of them now takes exactly m' elements (some of which are 0).

Similarly to the ordinary two-set intersection, if we agree to leak the precise column size, then instead of sorting we may include t_k into $\mathbf{d}'$ (getting triples instead of pairs), and then apply Shuffle and leak all the shuffled t_k, leaving behind only the pairs for which $t_k = 1$. For the pairs left behind, we may in turn declassify $A_i \| A_j$, so that the set sizes are now clearly visible, and the infrequent ones may be eliminated.

The analogous algorithm for Diffset is given in Alg. 12

Complexity The complexity of Alg. 11 is $\overline{\mathsf{Sort}}(\ell, \log m + \log n)$ for the initial sorting, $\frac{\ell n}{2} \odot \overline{\mathsf{Equal}}(\log m)$ for the equality checks, and $\frac{\ell n}{2} \odot \overline{\mathsf{MultByBit}}(\log m)$ for multiplications. After that, apply one more sort of complexity $\overline{\mathsf{Sort}}(\frac{\ell n}{2}, \log m + \log n + 1)$ to the pairs. Since each block that corresponds to one $A_i \cap B_j$ is in turn sorted by s_l, CountOnes will be free. The overall operation complexity is $\overline{\mathsf{Sort}}(\ell, \log m + \log n) + \frac{\ell n}{2} \odot \overline{\mathsf{Equal}}(\log m) + \frac{\ell n}{2} \odot \overline{\mathsf{MultByBit}}(\log m) + \overline{\mathsf{Sort}}(\frac{\ell n}{2}, \log m + 2\log n)$. Here MultByBit may be reduced to OuterProd, as in each row we actually multiply every t_l by the same tid.

As a particular case, if we take $n = 2$ (just one intersection), we get $\overline{\mathsf{Sort}}(2m', \log m + 1) + 2m' \odot \overline{\mathsf{Equal}}(\log m) + 2m' \odot \overline{\mathsf{MultByBit}}(\log m) + \overline{\mathsf{Sort}}(2m', \log m + 1)$, getting the same complexity than the ordinary two-set intersection of Sec. 3.3 presented above, applied to two sets of length m' (the last sorting uses only one bit in sorting and hence is equivalent to Csort).

If we agree to leak the number of entries, then instead of the last sorting we have $\overline{\mathsf{Shuffle}}(\frac{\ell n}{2}, \log m + \log n + 1)$, and then $\overline{\mathsf{Declassify}}(\frac{\ell n}{2}, 1)$.

Algorithm 12: Multiple set difference $\mathsf{MSet}_{\setminus}$

```
Data: $\langle\!\llbracket \mathbf{a}_i \rrbracket\!\rangle$, $\langle\!\llbracket \mathbf{b}_j \rrbracket\!\rangle$, $i \in I_A$, $j \in I_B$, where in each vector all elements except 0 are unique
Result: $\langle\!\llbracket \mathbf{c}_{ij} \rrbracket\!\rangle = \langle\!\llbracket \mathbf{a}_i \setminus \mathbf{b}_j \rrbracket\!\rangle$, $|\mathbf{c}_{ij}| = |\mathbf{a}_i|$, $i \in I_A$, $j \in I_B$
1  $\langle\!\llbracket \mathbf{a}'_i \rrbracket\!\rangle \leftarrow \langle\!\llbracket \mathbf{a}_i \bowtie [1]_{|\mathbf{a}|} \rrbracket\!\rangle$ ;
2  $\langle\!\llbracket \mathbf{b}'_j \rrbracket\!\rangle \leftarrow \langle\!\llbracket \mathbf{b}_j \bowtie [0]_{|\mathbf{b}|} \rrbracket\!\rangle$ ;
3  $\langle\!\llbracket \mathbf{d} \rrbracket\!\rangle \leftarrow \mathsf{Sort}(\langle\!\llbracket \mathbf{a}'_1 \bowtie 1 \rrbracket\!\rangle \| \cdots \| \langle\!\llbracket \mathbf{a}'_{n_A} \bowtie n_A \rrbracket\!\rangle \| \langle\!\llbracket \mathbf{b}'_1 \bowtie 1 \rrbracket\!\rangle \| \cdots \| \langle\!\llbracket \mathbf{b}'_{n_B} \bowtie n_B \rrbracket\!\rangle)$ ;
4  foreach $i \in \{1, \ldots, |\mathbf{d}|\}$ do
5      foreach $i \in \{\max(0, i-n), \ldots, \max(0, i-n)+n\}$ do
6          $\langle\!\llbracket t_{ij} \rrbracket\!\rangle = \mathsf{Equal}(\langle\!\llbracket d_i \rrbracket\!\rangle[0], \langle\!\llbracket d_j \rrbracket\!\rangle[0]) - \langle\!\llbracket d_i \rrbracket\!\rangle[1]$ ;
7          $\langle\!\llbracket s_{ij} \rrbracket\!\rangle = \mathsf{MultByBit}(\langle\!\llbracket t_{ij} \rrbracket\!\rangle, \langle\!\llbracket d_i \rrbracket\!\rangle[0])$ ;
8          $\langle\!\llbracket \mathbf{d}'_{\mathbf{ij}} \rrbracket\!\rangle = (\langle\!\llbracket d_i \rrbracket\!\rangle[2] \| \langle\!\llbracket d_j \rrbracket\!\rangle[2], \langle\!\llbracket s_{ij} \rrbracket\!\rangle)$
9  $\langle\!\llbracket \mathbf{e} \rrbracket\!\rangle = \mathsf{Sort}(\langle\!\llbracket \mathbf{d}' \rrbracket\!\rangle)$ ;
10 foreach $i \in \{1, \ldots, n\}$ do
11     $\mathbf{c}_i \leftarrow \mathbf{e}[ni, \ldots, n(i+1)][1]$ ;
12 return $\langle\!\llbracket \mathbf{c}_{1,1} \rrbracket\!\rangle, \ldots, \langle\!\llbracket \mathbf{c}_{n_A,n_B} \rrbracket\!\rangle$ ;
```

Type	Operation	Operation Complexity
bit	$\mathsf{MSet}_{\cap}(n,k)$	$m \cdot \overline{\mathsf{Mult}}(1) + m \cdot \overline{\mathsf{ShareConv}}(\log m)$
	$\mathsf{MSet}_{\setminus}(n,k)$	$m \cdot \overline{\mathsf{Mult}}(1) + m \cdot \overline{\mathsf{ShareConv}}(\log m)$
set	$\mathsf{MSet}_{\cap}(n,k)$	$\overline{\mathsf{Sort}}(\ell, \log(mn)) + \frac{\ell n}{2} \odot \overline{\mathsf{Equal}}(\log m) + \overline{\mathsf{OuterProd}}(2\ell, \log m) + \overline{\mathsf{Sort}}(\frac{\ell n}{2}, \log(mn^2))$
	$\mathsf{MSet}_{\setminus}(n,k)$	$\overline{\mathsf{Sort}}(\ell, \log(mn)) + \frac{\ell n}{2} \odot \overline{\mathsf{Equal}}(\log m) + \overline{\mathsf{OuterProd}}(2\ell, \log m) + \overline{\mathsf{Sort}}(\frac{\ell n}{2}, \log(mn^2))$

Table 5. Multiple set algorithm complexities of Sharemind

Similarly to $\mathsf{OuterProd}$, this produces all possible intersections, including $A_i \cap A_j$ and $B_j \cap A_i$ that we probably did not need. Hence even if these intersections would be more efficiently computable with the bit vector approach, we do not have to do it again. If we decide to leak the number of zero entries, then these additional sets may leak information that we did not intend to (for example, the size of $A_i \cap A_j$ for which we have already known that it would not be frequent). In this case, revealing the $A_i \| A_j$ should be done *before* revealing $\mathbf{t}$, and the unnecessary sets discarded immediately.

Comparisons of the bit matrix and the set matrix based approaches is shown in Tab. 10.

3.4 Balancing Set and Bit Based Approaches

In practice, it may happen that a dataset contains both dense and sparse columns. What could we do if the most columns are sparse, but there are still some columns that are so dense that finding an set based intersection becomes too inefficient for them?

Let m' the number of nonzero entries per column that we need to achieve in order that the set based approach would be preferable.

1. The simplest solution is to split each column of size m into $\lceil \frac{m}{m'} \rceil$ columns of size m'. Instead of a set of transactions T^A that correspond to the itemset A, there are

now $\lceil \frac{m}{m'} \rceil$ sets T_i such that $T_1^A \cup \cdots \cup T^A_{\lceil \frac{m}{m'} \rceil} = T^A$. Since the radix sort is linear in the set size, such a splitting gives no difference for $\mathsf{RSet}_\cap$, but may indeed be useful for the superlinear $\mathsf{Qsort}_\cap$.
2. If we agree to additionally leak not only $|T^A| \geq t$, but also $|T_i^A| \geq t$, then we may dispose of a lot of columns if the matrix is sparse. This will however be leaking additional information about the size of T_A. Moreover, the number of comparisons increases up to $\lceil \frac{m}{m'} \rceil$ times.
3. If we agree to leak the number of entries per column, then we may just leave the dense columns as they are. Although finding the intersections that involve those dense columns are expensive, we may still win if their number is small, especially taking into account the fact that the dense columns become sparse after a couple of iterations. It is even sufficient to leak not the total number of nonzero entries in each column, but just whether this number exceeds the threshold m'.

One possible way to remedy the overhead caused by dense columns is is to maintain the set format for sparse columns and the bit format for dense columns. The main question is how to find the intersection of a set-represented and a bit-represented column. In Sec. 3.4, we discuss how the dense columns can be handled.

In this subsection, we still assume that we are using the standard Apriori, Eclat, and Diffset algorithms without modifying them in general. In our comparisons, we consider a particular iteration on which k-sets are being constructed from $k-1$-sets. The algorithm should now decide to which columns it applies the set-based approach, and to which the bit-based approach.

Since the matrix is now mixed, and the particular partitioning depends on the choice of m, we assume for simplicity that initially there are ℓ secret shared pairs of the form $(row(tid), column(item))$ that correspond to the non-zero entries. The table has size $m \times n$ as before. The algorithms convert this set of pairs to set and bit based columns based on the value of m'.

We will further assume that all the algorithms are based on xor sharing, as the subalgorithms that we will use are significantly less efficient for additive sharing.

Auxiliary Building Block Algorithms We define some auxiliary subalgorithms that we will use.

– **Converting the set of** $(row, column)$ **pairs to a bit matrix** (Pairs2Bits) The task is easy if the sizes of columns are leaked so let us assume that it is not the case. The idea is to first map each (i, j) pair into an $m \times n$ matrix M such that $M[i, j] = 1$ and all the other entries are 0 otherwise. Then it is easy to find the final result by summing these matrices up, which is free. This is shown in Alg. 13.
The complexity of this algorithm consists of finding ℓ characteristic vectors of $\log m$ bits, ℓ vectors of $\log n$ bits, ℓ outer products of $\log n \times \log m$ vectors (each outer product is a matrix B_k such that $B_k[i, j] = 1$, and all the other entries are 0), and summing the obtained matrices B_k up. The complexity is

$$\ell \odot ((\overline{\mathsf{CharVec}}(\log m) \oplus \overline{\mathsf{CharVec}}(\log n)) + \overline{\mathsf{OuterProd}}(n+m, 1)) \enspace .$$

Algorithm 13: Pairs to a bit matrix Pairs2Bits

```
Data: A list of $(row, column)$ pairs $\langle\langle \mathbf{m} \rangle\rangle$
Result: A bit matrix $\langle\langle \mathbf{M} \rangle\rangle$ that corresponds to $\langle\langle \mathbf{m} \rangle\rangle$
1 foreach $(\langle\langle i \rangle\rangle, \langle\langle j \rangle\rangle) \in \langle\langle \mathbf{m} \rangle\rangle$ do
2     $\langle\langle \mathbf{i} \rangle\rangle = \mathsf{CharVec}(\langle\langle i \rangle\rangle)$ ;
3     $\langle\langle \mathbf{j} \rangle\rangle = \mathsf{CharVec}(\langle\langle j \rangle\rangle)$ ;
4     $\langle\langle \mathbf{B}_k \rangle\rangle = \mathsf{OuterProd}(\langle\langle \mathbf{i} \rangle\rangle \| \langle\langle \mathbf{j} \rangle\rangle)$ ;
5 $\langle\langle \mathbf{B} \rangle\rangle = \bigoplus_k \langle\langle \mathbf{B}_k \rangle\rangle$ ;
6 return $\langle\langle \mathbf{B} \rangle\rangle$ ;
```

Algorithm 14: Bit vector to a set Bits2Set

```
Data: A xor shared bit vector $\langle\langle \mathbf{b} \rangle\rangle$ of length $m$ with at most $m'$ nonzero entries
Result: A xor shared set representation $\langle\langle \mathbf{c} \rangle\rangle$ of $\langle\langle \mathbf{b} \rangle\rangle$
1 foreach $i \in \{1, \dots, m\}$ do
2     $\langle\langle \mathbf{c}_i \rangle\rangle = \langle\langle \mathbf{b}_i \rangle\rangle \cdot i$ ;
3 $\langle\langle \mathbf{d} \rangle\rangle = \mathsf{CSort}(\langle\langle \mathbf{b} \rangle\rangle \bowtie \langle\langle \mathbf{c} \rangle\rangle)$ ;
4 return $\langle\langle \mathbf{d} \rangle\rangle[0 : m', 1]$ ;
```

– **Converting a bit matrix column to a set matrix column** (Bits2Set) This algorithm transforms a column of a bit matrix to a column of xor shared row identifiers of length m' where m' is a known upper bound on the number of nonzero entries. This is shown in Alg. 14
  Computing the multiplications is free since we are multiplying by a public value j, and $\mathbf{b}_i \in \{0, 1\}$. The only thing that remains is Csort. This transformation is itself already more expensive than multiplying bit vectors, and hence it should be used only if the set representation will be reused afterwards.
– **Converting a set matrix column to a bit matrix column** (Set2Bits) This algorithm is based on finding the characteristic vector of each set element and summing them up. This is shown in Alg. 15.
  The complexity of this algorithm is $m' \odot \overline{\mathsf{CharVec}}(k)$.

The summary of the auxiliary protocols is given in Tab. 6.

Leaking Column Density

Algorithm 15: Set to a bit vector Set2Bits

```
Data: A xor shared set representation $\langle\langle \mathbf{c} \rangle\rangle$ of length $m'$ with over $m$ elements
Result: A xor shared bit vector representation $\langle\langle \mathbf{b} \rangle\rangle$ of $\langle\langle \mathbf{c} \rangle\rangle$
1 foreach $i \in \{1, \dots, m'\}$ do
2     $\langle\langle d_i \rangle\rangle = \mathsf{CharVec}(\langle\langle c_i \rangle\rangle, m)$ ;
3 $\langle\langle \mathbf{b} \rangle\rangle = \bigoplus_{i=1}^{m} \langle\langle d_i \rangle\rangle$ ;
4 return $\langle\langle \mathbf{b} \rangle\rangle$ ;
```

Type	Operation	Secure Operation Complexity
xor	$\mathsf{Pairs2Bits}(\ell,m,n,k)$	$\ell \odot (\overline{\mathsf{CharVec}}(m) + \overline{\mathsf{CharVec}}(n) + \overline{\mathsf{OuterProd}}(n+m,1))$
	$\mathsf{Sets2Bits}(m',k)$	$m' \odot \overline{\mathsf{CharVec}}(k)$
	$\mathsf{Bits2Set}(m,k)$	$\overline{\mathsf{Csort}}(m,k)$

Table 6. Complexities of Protocols of Sec. 3.4

Algorithm 16: Transforming sparse bit columns to set columns Partition

```
Data: Bit vector columns $(\langle\langle \mathbf{c}_1 \rangle\rangle, \ldots, \langle\langle \mathbf{c}_n \rangle\rangle)$, threshold $m'$
Result: Partitioning to bit columns of density more than $m'$ and set columns of density at
        most $m'$.
1 foreach $i \in \{1, \ldots, n\}$ do
2     $s = \mathsf{CountOnes}(\langle\langle \mathbf{c}_i \rangle\rangle)$ ;
3     if $s \leq m'$ then
4         $\langle\langle \mathbf{d}_i \rangle\rangle = \mathsf{Bits2Set}(\langle\langle \mathbf{c_i} \rangle\rangle)$ ;
5     else
6         $\langle\langle \mathbf{d}_i \rangle\rangle = \mathbf{c_i}$ ;
7 return $\langle\langle \mathbf{d} \rangle\rangle$ ;
```

Algorithm Let m' be the bound for which set based approach is applicable. The two main modifications to the previous standard FIM are the following.

1. *Initially* all the columns are represented by bit vectors. If the input comes in sparse form, it can be done using Pairs2Bits (just calling that protocol on the initial set of pairs).
2. *On each iteration* the columns that have at most m' elements are converted to set columns using Bits2Set protocol. This is described in Alg. 16.
3. *Each intersection* is based on the types of columns: whether both are set based, both are bit based, or the representation is different. The algorithm for finding one intersection is shown in Alg. 17. We may either transform the bit column into a set, or the set column into a bit vector. The parties may decide which approach is more efficient dynamically, estimating both complexities based on public information (Line 10).

Complexity The complexity is a bit difficult to estimate since the size of columns may vary, and the exact complexity depends on each column density on each iteration. In the worst case, there would be two representations maintained in parallel for each column: the bit-based an the set-based one. The worst case complexity would be the following.

1. *Initially* The complexity of this phase is up to $n \odot \overline{\mathsf{Bits2Set}}(m, \log m)$.
2. *On each iteration* In the worst case, each bit column is transformed to a set column. The complexity is $n \odot \overline{\mathsf{Bits2Set}}(m, \log m)$.
3. *For each intersection* There are now several cases.
   – *Both are bit vectors:* $m \odot \overline{\mathsf{Mult}}(1) + m \odot \overline{\mathsf{ShareConv}}(\log m)$.

Algorithm 17: Mixed column set intersection $\mathsf{MSet}_\cap$

```
    Data: Two columns $\langle[\mathbf{a}]\rangle$ and $\langle[\mathbf{b}]\rangle$
    Result: $\langle[\mathbf{c}]\rangle = \langle[\mathbf{a} \cap \mathbf{b}]\rangle$.
1   if a and b are bit vectors then
2       $\langle[\mathbf{c}']\rangle \leftarrow \mathsf{Mult}(\langle[\mathbf{a}]\rangle \| \langle[\mathbf{b}]\rangle)$ ;
3       $\langle[\mathbf{c}]\rangle \leftarrow \mathsf{ShareConv}(\langle[\mathbf{c}']\rangle, \log m)$ ;
4   else if a and b are sets then
5       $\langle[\mathbf{c}]\rangle \leftarrow \mathsf{Set}_\cap(\langle[\mathbf{a}]\rangle \| \langle[\mathbf{b}]\rangle)$ ;
6   else if a is a set and and b is a bit vector then
7       $c_1 \leftarrow \overrightarrow{\mathsf{Bits2Set}}(m, \log m) + \overrightarrow{\mathsf{Set}_\cap}(|\mathbf{a}| + |\mathbf{b}|, \log m)$ ;
8       $c_2 \leftarrow \overrightarrow{\mathsf{Set2Bits}}(|\mathbf{a}|, \log m) + m \odot \overrightarrow{\mathsf{Mult}}(1)$ ;
9       $c_3 \leftarrow \overrightarrow{\mathsf{ShareConv}}(m, \log m)$ ;
10      if $c_1 \leq c_2 + c_3$ then
11          $\langle[\mathbf{b}']\rangle \leftarrow \mathsf{Bits2Set}(\langle[\mathbf{b}]\rangle)$ ; //if it does not exist yet
12          $\langle[\mathbf{c}]\rangle \leftarrow \mathsf{Set}_\cap(\langle[\mathbf{a}]\rangle \| \langle[\mathbf{b}']\rangle)$ ;
13      else
14          $\langle[\mathbf{a}']\rangle \leftarrow \mathsf{Set2Bits}(\langle[\mathbf{a}]\rangle)$ ; //if it does not exist yet
15          $\langle[\mathbf{c}']\rangle \leftarrow \mathsf{Mult}(\langle[\mathbf{a}']\rangle \| \langle[\mathbf{b}]\rangle)$ ;
16          $\langle[\mathbf{c}]\rangle \leftarrow \mathsf{ShareConv}(\langle[\mathbf{c}']\rangle, \log m)$ ;
    else
        // analogous to the previous case
17  return $\langle[\mathbf{c}]\rangle$ ;
```

- *Both are sets:* $\overline{\mathsf{Set}_\cap}(|\mathbf{a}| + |\mathbf{b}|, \log m)$.
- *Different Representations:* $n \odot \min(\overline{\mathsf{Set}_\cap}(|\mathbf{a}| + |\mathbf{b}|, \log m), m \odot \overline{\mathsf{Mult}}(1) + m \odot \overline{\mathsf{ShareConv}}(\log m))$.

It will be more easy to estimate the complexity of an analogous algorithm in the next subsubsection, where all the columns have the same length m', as then the quantity $|\mathbf{a}| + |\mathbf{b}|$ can be estimated.

Complexity using Parallel Set Intersection As we have discussed in Sec. 3.3, it would be unfair to treat set intersections independently. In practice, the number of such parallelizable set intersections is unpredictable. Since the candidate sets are public, we could decide dynamically whether it still makes sense to apply the set intersection. Let $A_i \cap B_j$, $i \in I_A$, $j \in I_B$ be the intersections that we need to find. Let $n_A = |I_A|$, $n_B = |I_B|$, $n' = n_A + n_B$. We can immediately see that the intersections of A_i and B_j that participate in some intersection with *some* bit column are now actually free for *all* bit columns. Hence let A_i and B_j be such that at least one of them participates in set intersections only. Even in this case, if some A_i in participating in numerous intersections, it may be more efficient to compute all the intersections of A_i in bit format.

Using Eclat and Diffset, the parallelization of intersections is not important, as on each iteration all the current k-subsets are intersected with just one particular element

Sharing	Operation	Rounds	Communication
additive	$\mathsf{LessThan}(k)$	$\log k+3$	$27(\log k)(k-1)+64k-27$
	$\mathsf{Equal}(k)$	$\log k+1$	$14k-9$
	$\mathsf{Mult}(k)$	1	$15k$
	$\mathsf{OuterProd}(n,k)$	1	$9nk$
xor	$\mathsf{LessThan}(k)$	$\log k$	$30k$
	$\mathsf{Equal}(k)$	$\log k$	$12k-9$
	$\mathsf{OuterProd}(n,k)$	1	$3n(2k+1)$
both	$\mathsf{OuterProd}(n,1)$	1	$9n$
	$\mathsf{ShareConv}(k)$	2	$5k+4$
	$\mathsf{Shuffle}(n,k)$	3	$6nk$
	$\mathsf{Declassify}(k)$	1	$6k$

Table 7. Basic operation complexities of Sharemind

Sharing	Operation	Rounds	Communication
additive	$\mathsf{Qsort}(n,k)$	$3+\log n\log k$	$6nk+36k\cdot n\log n$
xor	$\mathsf{Csort}(n,k)$	7	$n(32k+10)$
	$\mathsf{Rsort}(n,k)$	$7k$	$nk(32k+10)$
	$\mathsf{QSort}(n,k)$	$3+\log n\log k$	$6nk+30k\cdot n\log n+6n\log n$
	$\mathsf{CharVec}(k)$	k	$15k\sqrt{2^k}$
	$\mathsf{MultByBit}(k)$	1	$9k+6$
	$\mathsf{OuterEq}(n,k)$	1	$n\cdot(15k\sqrt{2^k})+15n2^k$

Table 8. Auxiliary algorithm complexities of Sharemind

(if we do not attempt to parallelize several distinct branches, as it was done in [2]). Using Apriori, we could split the n columns into equivalence classes C_i of size n_{C_i}, such that no intersections need to be found for columns of different classes. Namely, first compute the transitive closure for intersection relation, and then split according to obtained equivalence classes. For each n_{C_i}, decide whether the bit or the set approach is more efficient. Since using parallel column intersection makes more sense if we share bits over $\mathbb{Z}_m$, all the conversions between bit and set columns become more expensive, and deciding which variant is optimal may be not so easy.

3.5 Comparing Bit Matrix and Set Based Approaches for a Particular Platform

So far it has been completely unclear in which cases the set based approach can be preferable to the bit based approach. Let us now see which communication and round complexity we get if we instantiate the algorithms to some particular platform. For this, we have chosen Sharemind [4], as it uses both additive and xor sharing, and also contains all the necessary black box algorithms needed in the computation. The complexities of basic operations are given in Tab. 7. On the basis of this table, we estimated the complexities of the algorithms introduced in Sec. 3.3. They are presented in Tab. 8 and Tab. 9.

Sharing	Operation	Rounds	Communication
xor	$\mathsf{RSet}_{\cap}(n,k)$, $\mathsf{RSet}_{\setminus}(n,k)$	$7k+\log k+8$	$32nk^2+63nk+7n$
	$\mathsf{QSet}_{\cap}(n,k)$, $\mathsf{QSet}_{\setminus}(n,k)$	$\log n \log k+\log k+11$	$30nk\log n+6n\log\log n+59nk+7n$

Table 9. Set algorithm complexities of Sharemind

We are now ready to compare the bit matrix and the set based approaches. The main structure of the algorithms remains unchanged. The only difference is that the bit vector products are substituted with the set intersection and set difference defined in Sec. 3.3. The size of the sets is m', and the number of bits is $\log m$, where the numbers m' and n are defined as in the beginning of Sec. 3.3.

One set intersection First of all, let us for simplicity estimate the complexity of one set intersection, based on the operation complexities of Tab. 7. We compare it to the complexity of the bit matrix approach.

1. **Using radix sort:**
   - **Round advantage:** $7\log m+\log\log m+8$ instead of 3. This is an obvious disadvantage, but we hope to win in communication.
   - **Communication:** $32(2m')(\log m)^2+63(2m')\log m+7(2m')=m'(64(\log m)^2+126\log m+14)$ instead of $5m\log m+19m$. The advantage is nonnegative iff

$$m' \leq \frac{5m\log m+19m}{64(\log m)^2+126\log m+14} \quad .$$

2. **Using quicksort:**
   - **Round advantage:** $\log(2m')\log\log m+\log\log m+11$ instead of 3. Again, an obvious disadvantage, but we hope to win in communication.
   - **Communication:** $30(2m')\log(m)\log(2m')+6(2m')\log(2m')+59(2m')\log(m)+7(2m')=m'(178\log m+60\log m\log m'+12\log m'+26)$. The advantage is nonnegative iff

$$m' \leq \frac{5m\log m+19m}{178\log m+60\log m\log m'+12\log m'+26} \quad .$$

   Knowing that $m' \leq m$, we may get rid of $\log m'$ on the right hand side, getting

$$m' \leq \frac{5m\log m+19m}{60(\log m)^2+190\log m+26} \quad .$$

   Since in practice the bound is even larger, using quicksort seems to have better efficiency. Nevertheless, since the number of rounds may be larger for quicksort, both algorithms should still be considered.

What if not all Columns are Sparse In general, even for a sparse dataset we cannot assume that all the columns will be sparse enough, so that the upper bounds for m' are satisfied.

Example: A standard Retail testing set (which is available at the Frequent Itemset Mining Dataset Repository [1]) has $m = 88163$ rows, $n = 16470$ columns, and $\ell = 908576$ nonzero entries in total. If we want to get advantage in communication, for radix sort we need $m' \leq 460$ (the estimated bound for quicksort is 500, and actually it is even around 700 in this case). In the case of Retail set, the most frequent item has 50675 occurrences, and so we definitely do not reach advantage in communication. This dataset is bad example since although the matrix is sparse, its columns are not all sparse. There are only 211 items out of 16470 that exceed 460 occurrences, but we cannot just round them down to 460 since we would have to decide which transaction id-s exactly should be removed, and that choice may affect the final result significantly. In this case, we will need to use the mixed approach of Sec. 3.4.

It is easier to estimate the complexity of Alg. 17 of Sec. 3.4 if we leak not the precise value of $\mathsf{CountOnes}(\mathbf{c})$ for each column $\mathbf{c}$, but just whether $\mathsf{CountOnes}(\mathbf{c}) \leq m'$. If a column has at most m' entries, it is considered a set column, and a bit column otherwise. Let us now for simplicity take $m' = \left(\frac{\sqrt{m}}{3} + \frac{19\sqrt{m}}{15\log m}\right)$ (in this case, estimating the complexities will be easier). Such an m' satisfies the bounds of 3.5. We use the same algorithms as in the previous section. Since all the sparse columns are of size m' now, and the dense columns of size m, the choice of Line 10 of Alg. 17 now always evaluates to the one that transforms the set column to a bit column. Namely, $\overrightarrow{\mathsf{Set}_\cap}(|\mathbf{a}|+|\mathbf{b}|, \log m) = \overrightarrow{\mathsf{Set}_\cap}(m+m', \log m) \geq \overrightarrow{\mathsf{Set}_\cap}(m, \log m) = 32m\log m^2 + 63m\log m + 7m$, which is larger than $\overrightarrow{\mathsf{Set2Bits}}(|\mathbf{a}|, \log m) + m \odot \overrightarrow{\mathsf{Mult}}(m) + \overrightarrow{\mathsf{ShareConv}}(m, \log m) = \overrightarrow{\mathsf{Set2Bits}}(m', \log m) + m \odot \overrightarrow{\mathsf{Mult}}(1) + \overrightarrow{\mathsf{ShareConv}}(m, \log m) = \left(\frac{\sqrt{m}}{3} + \frac{19\sqrt{m}}{15\log m}\right) \cdot 15\sqrt{m} \cdot \log m + 5m\log m + 19m = 2(5m\log m + 19m)$. This means that if we want to find the intersection of different columns, then converting the set column to a bit column is always preferable. Note that, for this choice of m', the transformation itself has complexity $2(5m\log m + 19m)$, and in overall the intersection is 3 times as expensive as it would be if the set column had already been a bit column.

We may now estimate the complexity of Alg. 17 more precisely.

1. *Initially* The complexity of this phase is up to $n \odot \overline{\mathsf{Bits2Set}(m, \log m)}$ as before.
2. *On each iteration* As before, in the worst case the complexity is $n \odot \overline{\mathsf{Bits2Set}(m, \log m)}$, which is 7 rounds and $n \odot m(32\log m + 10)$ communication.
3. *For each intersection* Again, there are now several cases.
   - *Both are bit vectors:* The complexity estimation is $\overline{\mathsf{Mult}}(m,1) + \overline{\mathsf{ShareConv}}(m, \log m)$, which is 3 rounds and $5m\log m + 19m$ communication.
   - *Both are sets:* The complexity is $\overline{\mathsf{Set}_\cap}(2m', \log m)$, which is $7\log m + \log\log m + 8$ rounds and $2m'(32 \cdot (\log m)^2 + 63\log m + 7) = 2\left(\frac{\sqrt{m}}{3} + \frac{19\sqrt{m}}{15\log m}\right)(32 \cdot (\log m)^2 + 63 \cdot \log m + 7) < 2\sqrt{m}(32\log m^2 + 63\log m + 7)$ communication.
   - *Different Representations:* As we have discussed above, we always choose the set to bit branch on Line 10, and the complexity is $7\log m + \log\log m + 11$ rounds and $2(5m\log m + 19m)$ communication.

Type	Operation	Rounds	Communication
bit	$\mathsf{MSet}_{\cap}(n,k)$	3	$9mn\log m$
	$\mathsf{MSet}_{\setminus}(n,k)$	3	$9mn\log m$
set	$\mathsf{MSet}_{\cap}(n,k)$	$\approx 14\log(mn) + 7\log n$	$\approx \ell[(\frac{n}{2}+1)\log m(32\log(mn^2)+10) + \frac{n}{2}(12\log m - 9)]$
	$\mathsf{MSet}_{\setminus}(n,k)$	$\approx 14\log(mn) + 7\log n$	$\approx \ell[(\frac{n}{2}+1)\log m(32\log(mn^2)+10) + \frac{n}{2}(12\log m - 9)]$

Table 10. Multiple set algorithm complexities of Sharemind

In this way, using mixed representation gives us additional overhead if the columns are of different type (three times as much as would be if they we bit matrices initially). Another overhead is caused by converting bit matrices to set matrices when they become sparse. Nevertheless, since the intersections of sparse columns become much more efficient for small m', they may compensate that overhead.

3.6 Complexity using Parallel Set Intersection

The particular complexities of Mset protocols are presented in table Tab. 10.

Example: Consider again $m' = \left(\frac{\sqrt{m}}{3} + \frac{19\sqrt{m}}{15\log m}\right)$ using Retail dataset. For $m = 88163$, we get $m' \approx 121$. Let us now suppose that we use Apriori algorithm and apply it If we substitute this number into $\mathsf{MSet}_{\cap}$, we get the complexity that is still too large due to the n^2 factor. It would be better than the bit column approach for small n, if we take only 80 columns out of 16470 (while there are actually 14986 columns that have at most m' entries). Taking smaller m' allows to increase the number of columns. For example, $m' = 50$ allows to increase n to 500.

We see that even if the matrix is sparse, we are not going to apply the set intersection to most of the columns, but only to some of them, preferring to chose those that do not repeat several times in different intersections. Using Eclat and Diffset, we find intersection of the form $A \cap B_1, \ldots, A \cap B_{n_B}$ on each iteration, and hence deciding on which approach should be used is straightforward, as $n \cdot 1 = n$, and the OuterProd can at most reduce the computation 2 times compared to straightforward multiplication, so the chosen m' should just take this factor of 2 into account.

3.7 Estimating the Number of Sparse Columns

Our last question is whether our algorithms are indeed suitable for arbitrary sparse matrices. Namely, they give advantage if the number of sparse columns in the matrix is large. How is the total number of nonzero entries ℓ related to possible choices of m'?

If the total number of non-zero entries is ℓ and the number of columns is n, then there can be at most $\lfloor \frac{n \cdot m'}{\ell} \rfloor$ columns that have at least m' non-zero entries. Taking m' below the upper bound for which set based approach is reasonable, one may check if using set-based approach gives any advantage. It may happen that initially the matrix is quite dense and bit based approach is preferable, but the columns may become sparser on further steps. A proposed value for m' is $\left(\frac{\sqrt{m}}{3} + \frac{19\sqrt{m}}{15\log m}\right)$, which makes it easier to decide dynamically during the computation which approach is preferable, using the bounds presented in this paper.

4 Conclusions

We have presented two basic FIM algorithm for sparse datasets, an Eclat/Apriori based one, and a Diffset based one, where Diffset may be useful also for non-sparse matrices. The main problem of these algorithms is that they are not as linearizable as the bit vector algorithms are. Nevertheless, since our protocols can be easily integrated into the bit based approach, we may choose to apply them only on those steps where they indeed give advantage.

References

1. Frequent itemset mining dataset repository. `http://fimi.ua.ac.be/data/`. Last accessed 2015-07-07.
2. D. Bogdanov, R. Jagomägis, and S. Laur. A universal toolkit for cryptographically secure privacy-preserving data mining. In *Proceedings of the 2012 Pacific Asia Conference on Intelligence and Security Informatics*, PAISI'12, pages 112–126, Berlin, Heidelberg, 2012. Springer-Verlag.
3. D. Bogdanov, S. Laur, and R. Talviste. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In K. Bernsmed and S. Fischer-Hübner, editors, *Secure IT Systems - 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings*, volume 8788 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2014.
4. D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.
5. X. Cheng, S. Su, S. Xu, and Z. Li. Dp-apriori: A differentially private frequent itemset mining algorithm based on transaction splitting. *Computers & Security*, 50:74–90, 2015.
6. K. Hamada, R. Kikuchi, D. Ikarashi, K. Chida, and K. Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In T. Kwon, M.-K. Lee, and D. Kwon, editors, *ICISC*, volume 7839 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2012.
7. M. Kantarcioglu and C. Clifton. Privacy-preserving distributed mining of association rules on horizontally partitioned data. *IEEE Trans. on Knowl. and Data Eng.*, 16(9):1026–1037, Sept. 2004.
8. J. Launchbury, I. S. Diatchki, T. DuBuisson, and A. Adams-Moran. Efficient lookup-table protocol in secure multiparty computation. In P. Thiemann and R. B. Findler, editors, *ICFP*, pages 189–200. ACM, 2012.
9. J. Lee and C. W. Clifton. Top-k frequent itemsets via differentially private fp-trees. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 931–940, New York, NY, USA, 2014. ACM.
10. C. Sun, Y. Fu, J. Zhou, and H. Gao. Personalized privacy-preserving frequent itemset mining using randomized response. *The Scientific World Journal*, 2014, 2014.
11. M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 326–335, New York, NY, USA, 2003. ACM.
12. C. Zeng, J. F. Naughton, and J.-Y. Cai. On differentially private frequent itemset mining. *Proc. VLDB Endow.*, 6(1):25–36, Nov. 2012.