Privacy-preserving Frequent Itemset Mining for Sparse and Dense Data

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

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

Abstract. Frequent itemset mining is a data mining task that can in turn be used for other purposes such as associative rule mining. The data may be sensitive. There exist multiple privacy-preserving solutions for frequent itemset mining, which should consider the tradeoff between efficiency and privacy. Leaking some less sensitive information such as density of the datatable may improve the efficiency. In this paper, we consider secure multiparty computation setting, where the final output (the frequent itemsets) is public, and no other information should be inferred by the adversary that corrupts some of the computing parties. We devise privacy-preserving algorithms that have advantage when applied to very sparse and very dense matrices. We compare them to related work that has similar security requirements, estimating the efficiency of our new solution on a similar secure multiparty computation platform.

Keywords: Secure multiparty computation, frequent itemset mining

1 Introduction

Frequent itemset mining (FIM) is a standard data mining task. Given a collection of sets, the goal is to find the subsets that are contained in sufficiently many of these sets. After finding out 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, extracting some more interesting knowledge such as association rules. Not only the task itself, but also its privacy-preserving variants have been well studied in related work.

In this paper, we consider the security model where the final output is public, and the adversary, corrupting some of the parties, should be unable to infer any other information in addition to these public outputs. Our goal is to see if we can gain more efficiency given some additional assumptions about the matrix density. This allows us to make use of FIM algorithms whose efficiency depends on data density, that would not give advantage in privacy-preserving settings otherwise. Since the standard 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. We propose algorithms that allow to combine dense and sparse columns in the same computation.

2 Preliminaries

Let the sets be called *transactions*, and their elements *items*. These names come from one possible use case of FIM, where the items are goods sold in the supermarket, and each transaction corresponds to contents of a shopping cart. The task of FIM is to find out which subsets of items occur together in sufficiently many transactions. 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 of items $I \subseteq U$ that are encountered in sufficiently many sets T. A subset I is considered frequent iff $|\{T \mid I \subseteq T\}| \geq t$, where $t \geq 1$ is some threshold that is given as a parameter.

2.1 Secure Multiparty Computation

In this paper, we solve FIM using secure multiparty computation. We adjust our algorithms to a specific platform Sharemind [3], which is based on secret sharing. The main security domain of Sharemind supports 3 *computing* parties (P_1, P_2, P_3), tolerating at most one passively corrupted party. The number of *input* parties who provide the inputs by secret-sharing them among computing parties is unbounded. No party should learn any private data besides the final output that we consider public. There is some less sensitive information that we agree to leak, such as the total number of items and transactions.

Sharemind uses mainly *additive* and *bitwise* secret sharing. The costs of their standard operations are different. In the *additive* sharing, the value is shared as $a = a_1 + a_2 + a_3$ over some ring (in Sharemind $\mathbb{Z}_{2^m}$), where a_i is the share that belongs to the party P_i, and linear combinations can be computed without communication. In the *bitwise* sharing, the value is shared bitwise as $a = a_1 \oplus a_2 \oplus a_3$, and bitwise linear combinations of bitvectors can be computed without communication. Also, bitwise sharing is more efficient for comparison operations.

This paper builds algorithms from standard operations of Sharemind that it uses as black boxes. The algorithms would work for any platform in which these basic operations are universally composable. However, the particular costs that we get in this paper are very related to particular Sharemind protocols. On some other platform, the new methods could give even more advantage, but they also could be too inefficient to make sparse representation reasonable.

2.2 Notation

Throughout this paper, we use the following quantities:

- capital letters X denote sets, and calligraphic font $\mathcal{X}$ denotes a set of sets;
- $\sigma(I)$ is the support of I, i.e. the set of transactions that contain itemset I;
- $\Delta(I_1, I_2) := \sigma(I_1) \setminus \sigma(I_2)$ is the difference of supports of I_1 and I_2;
- secret shared value (either additive or bitwise sharing) $\langle\!\langle a \rangle\!\rangle$;
- additively shared value $[\![a]\!]$; $a = a_1 + \cdots + a_n$;
- bitwise shared value $\langle\!\langle a \rangle\!\rangle$; $a = a_1 \oplus \cdots \oplus a_n$;

- i-th element of a vector $\boldsymbol{a}$: $\boldsymbol{a}[i]$;
- element of a matrix $\mathbf{A}$ in the i-th row and j-th column: $\mathbf{A}[i, j]$;
- i-th row and j-th column of a matrix $\mathbf{A}$: $\mathbf{A}[i, :]$, $\mathbf{A}[:, j]$;
- vector concatenation: $\boldsymbol{x} \| \boldsymbol{y}$;

Protocol cost. We measure the number of rounds as well as the total number of bits communicated through the network. Formally, we define a type $Cost = \mathbb{N} \times \mathbb{N}$, where the first component is the number of communicated bits, and the second component is the number of rounds. We define the operations $\otimes : Cost \times Cost \to Cost$ (parallel composition) and $\oplus : Cost \times Cost \to Cost$ (sequential composition) as follows:

- $(a, b) \otimes (c, d) = (a + c, \max(b, d))$;
- $(a, b) \oplus (c, d) = (a + c, b + d)$.

We will use the shorthand $(a, b)^{\otimes n}$ to denote $(a, b) \otimes \cdots \otimes (a, b)$, and $(a, b)^{\oplus n}$ to denote $(a, b) \oplus \cdots \oplus (a, b)$, where (a, b) occurs n times. Let the operation $\otimes$ have higher priority than $\oplus$.

For a protocol Prot, we write Prot_k^n to denote the cost of application of Prot to a vector of length n of k-bit values. The number n is omitted if the protocol is applied to a single input, and $\mathsf{Prot}_k^{n_1,\ldots,n_m}$ denotes applying the protocol to several inputs of different lengths.

2.3 General FIM Algorithms

There exist several variations of the standard (not privacy-preserving) FIM algorithms. We give some examples in this section. A similar property of these algorithms is that, on each iteration, all they compute a frequent itemset of size k, based on the frequent itemsets of size $k - 1$ found so far. The basis of finding a k-set from $k - 1$ subsets is *set intersection*, or *set difference*. We will turn special attention to these set operations when we use these algorithms in a privacy-preserving setting.

Apriori. This algorithm sequentially constructs all the frequent itemsets of size 1, then of size 2, until all the frequent sets are obtained in this way. Any infrequent itemsets are immediately discarded. The frequent sets of size k are constructed only for those sets whose all subsets of size $k - 1$ have been frequent. The way in which these subsets are constructed depends on the particular algorithm instance. One possible implementation of this method is given in Alg. 1.

Eclat. Similarly to $\mathsf{Apriori}$, this algorithm constructs a set of size k from sets of size $k - 1$. The main difference from $\mathsf{Apriori}$ is that Eclat 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 $k - 1$ (sets of the form $P \cup \{x\}$ for $x \notin P$). Let $\sigma(P)$ be the support of P. For each item x, all possible frequent sets with prefix $P' := P \cup \{x\}$ can be constructed as $\sigma(P \cup \{x\}) \cap \sigma(P \cup \{y\})$ from the sets $P \cup \{y\}$ such that $y \neq x$. The new prefix is then processed recursively. The description of this method is given in Alg. 2.

Algorithm 1: Apriori

Data: $\mathcal{M}$: frequent itemsets of size $k-1$ found so far
Result: Frequent itemsets of size at least k

```
F ← ∅ ;
foreach X_i ∈ M do
    foreach X_j ∈ M, j > i do
        R ← X_i ∪ X_j ;
        if |σ(R)| ≥ t then
            F ← F ∪ {R} ;
if F ≠ ∅ then
    F' ← Apriori(F) ;
return F ∪ F' ;
```

Algorithm 2: Eclat

Data: $\mathcal{P}$ all the frequent sets of size $k-1$ with a prefix P
Result: Frequent itemsets of size at least k with a prefix P

```
foreach X_i ∈ P do
    F_i ← ∅ ;
    foreach X_j ∈ P, j > i do
        R ← X_i ∪ X_j ;
        σ(R) ← σ(X_i) ∩ σ(X_j) ;
        if |σ(R)| ≥ t then
            F_i ← F_i ∪ {R} ;
    if F_i ≠ ∅ then
        F'_i ← Eclat(F_i) ;
return ⋃_i F'_i ;
```

Diffset. If the matrix columns are very dense, then instead of keeping transactions that *contain* the given dataset, one could try to keep transactions that *do not contain* the given dataset. Actually, even something more interesting can be done. Another FIM algorithm Diffset [14] is similar to Eclat, but instead of keeping the set of transactions in each itemset, it keeps 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 set differences that this algorithm keeps may become very small on later iterations.

Let the itemsets $P \cup \{x\}$ and $P \cup \{y\}$ be frequent. We want to know 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 the support as $\sigma(P \cup \{x\}) = \sigma(P) \setminus \Delta(P, P \cup \{x\})$, and the difference as $\Delta(P, P \cup \{x\} \cup \{y\}) = \Delta(P, P \cup \{x\}) \setminus \Delta(P, P \cup \{y\})$. The description of this method is given in Alg. 3.

Algorithm 3: Diffset

Data: $\mathcal{P}$ all the frequent sets of size $k-1$ with a prefix P
Result: Frequent itemsets of size at least k with a prefix P

```
foreach X_i ∈ P do
    F_i ← ∅ ;
    foreach X_j ∈ P, j > i do
        R ← X_i ∪ X_j ;
        Δ(P,R) ← Δ(P,X_j) \ Δ(P,X_i) ;
        |σ(R)| ← |σ(P)| − |Δ(P,R)| ;
        if |σ(R)| ≥ t then
            F_i ← F_i ∪ {R} ;
    if F_i ≠ ∅ then
        F'_i ← Diffset(F_i) ;
return ⋃_i F'_i ;
```

3 Privacy-preserving FIM

Since FIM can in turn be used for various purposes such as associative rule mining, preserving privacy may be very important. 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.

Privacy-preserving versions of Apriori and Eclat have been implemented and optimized in [1, 5, 11]. There are also some solutions designed for specific initial data sharing, such as vertical or horizontal partitioning [9]. Implementing an algorithm such as FP-tree is not suitable for our security model since its structure leaks more information than the frequent itemsets themselves. In [11], the FP-tree is constructed *after* the frequent itemsets have been found (using Apriori-based algorithm), and the goal is to introduce noise into the public output.

Besides making the computation secure, it may be also important to consider how much the public output leaks by itself. Differential privacy guarantees that the adversary will not learn too much from the public output, providing statistical privacy for each individual record in the dataset. There exist systems such as PINQ [12], PDDP [4], RAPPOR [6], that allow to make statistical computations that achieve this property. In the context of FIM, differential privacy has been considered in [5, 11, 15]. Similar distortion-based approach is also used in [13]. In this work, we do not consider differential privacy. To achieve it, we could add noise to the initial data, so that the final output (that is, all the frequent itemsets up to certain size) would provide privacy for each individual record. We would need to define what exactly an individual record is (a column or a row), while in this paper the initial distribution of the records is not important.

In this work, we mainly extend the results of [1], where Eclat and Apriori algorithms (but not Diffset), as well as hybrid solutions, have been implemented. The algorithms of [1] are based on bit matrix representation. Their efficiency

does not depend on matrix sparsity, and they could not use the advantages of Diffset. We give a solution that works better with sparse matrices. Our algorithms use some blackbox operations that in general depend on the underlying secure multiparty computation platform, and whose implementation is not the part of FIM algorithms. The cost estimations for our algorithms are based on the blackbox operation costs of Sharemind [3].

Bit Matrix Representation. First, we describe the existing implementations of [1] based on representing the data table as a secret shared bit matrix. Initially, the n items and m transactions are assigned unique indices $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$ respectively. A matrix $\mathbf{B}$ is defined, such that $\mathbf{B}[i, j] = 1$ iff the i-th transaction contains the j-th item. On further iterations of FIM algorithms, the columns correspond not to single items, but to itemsets.

Although all matrix elements are bits, in order to determine whether an itemset is frequent, at some moment the sum of column elements has to be computed. Therefore, at least before the addition, the bits should be converted to at least $(\log m)$-bit values to avoid addition overflow, since the maximal value that the sum may take is m. In [1], the matrix elements are permanently stored as $\log m$-bit values to avoid conversion overheads. For very sparse sets, such an encoding may consume excessive space due to large amount of zeroes that will not be needed anyway. In Sec. 3.2, we discuss whether it is better to keep the bits in $\log m$ format, or to convert 1-bit values to $\log m$-bit values on demand.

Finding an intersection of two itemsets i and j and checking its cardinality is implemented as follows:

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 threshold t.

Sparse Set Representation. In this paper, we propose another way to represent transactions that contain the given itemset. This representation makes sense for sparse matrices, i.e. when 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. The order of indices in a column does not matter. Encoding a number from $\{1, \ldots, m\}$ 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. Since 0 now has special meaning, we start indexation with 1. Alternatively, we could leave exactly m_j elements in the column, but that would leak too much additional information about the data.

3.1 Algorithms for Privacy Preserving FIM

Existing Building Blocks. We describe the building block algorithms that we use for our constructions. Some very basic operations that do not require additional description are given in Tab. 1. We now describe some more complicated algorithms. The summary of the costs of used building blocks is given in Tab. 2.

Operation Call	Returned Value
$\mathsf{Sum}(\langle\!\langle \boldsymbol{x} \rangle\!\rangle)$	$\langle\!\langle \sum_{j=1}^{m} \boldsymbol{x}[j] \rangle\!\rangle$
$\mathsf{Mult}(\langle\!\langle x \rangle\!\rangle, \langle\!\langle y \rangle\!\rangle)$	$\langle\!\langle x \cdot y \rangle\!\rangle$
$\mathsf{OuterProd}(\langle\!\langle \boldsymbol{x} \rangle\!\rangle, \langle\!\langle \boldsymbol{y} \rangle\!\rangle)$	$\langle\!\langle \mathbf{Z} \rangle\!\rangle$, where $\mathbf{Z}[i,j] = \boldsymbol{x}[i] \cdot \boldsymbol{y}[j]$
$\mathsf{ShareConv}(\langle\!\langle x \rangle\!\rangle, k)$	$\langle\!\langle y \rangle\!\rangle$, where $x \in \mathbb{Z}_2$, $y \in \mathbb{Z}_{2^k}$, $x = y$
$\mathsf{Shuffle}(\langle\!\langle \boldsymbol{x} \rangle\!\rangle)$	$\langle\!\langle \boldsymbol{y} \rangle\!\rangle$, where $\boldsymbol{y}$ is a random reordering of $\boldsymbol{x}$
$\mathsf{UnShuffle}(\langle\!\langle \boldsymbol{y} \rangle\!\rangle)$	$\langle\!\langle \boldsymbol{x} \rangle\!\rangle$, where $\boldsymbol{x}$ is a restored previously shuffled vector
$\mathsf{Equal}(\langle\!\langle x \rangle\!\rangle, \langle\!\langle y \rangle\!\rangle)$	$\langle\!\langle x == y \rangle\!\rangle \in \mathbb{Z}_2$
$\mathsf{LessThan}(\langle\!\langle x \rangle\!\rangle, \langle\!\langle y \rangle\!\rangle)$	$\langle\!\langle x \leq y \rangle\!\rangle \in \mathbb{Z}_2$
$\mathsf{Declassify}(\langle\!\langle x \rangle\!\rangle)$	x

Table 1. Basic blackbox operations

Protocol Call	Returned Value	Cost
$\mathsf{Csort}(\langle\!\langle \boldsymbol{x} \rangle\!\rangle, \langle\!\langle \boldsymbol{b} \rangle\!\rangle)$	$\langle\!\langle \boldsymbol{x} \rangle\!\rangle$ sorted by $\langle\!\langle \boldsymbol{b} \rangle\!\rangle$	$\mathsf{ShareConv}_k^n \oplus \mathsf{Mult}_k^n \oplus \mathsf{Shuffle}_{2k}^n \oplus \mathsf{Declassify}_k^n$
$\mathsf{Rsort}(\langle\!\langle \boldsymbol{x} \rangle\!\rangle)$	sorted $\boldsymbol{x}$	$(\mathsf{ShareConv}_k^n \oplus \mathsf{Mult}_k^n \oplus \mathsf{Shuffle}_{2k}^n \oplus \mathsf{Declassify}_k^n)^{\oplus k}$
$\mathsf{Qsort}(\langle\!\langle \boldsymbol{x} \rangle\!\rangle)$	sorted $\boldsymbol{x}$	$\mathsf{Shuffle}_k^n \oplus (\mathsf{LessThan}_k^n \oplus \mathsf{Declassify}_1^n)^{\oplus \log n}$
$\mathsf{PQsort}(\langle\!\langle \boldsymbol{x} \rangle\!\rangle)$	sorted $\boldsymbol{x}$	$(\mathsf{LessThan}_k^n \oplus \mathsf{Declassify}_1^n)^{\oplus \log n}$

Table 2. Building block operations (k-bit elements, $|\boldsymbol{x}| = |\boldsymbol{b}| = n$)

Radix Sort (Rsort*) and counting sort (*Csort*)* . We use the algorithms from [2, Alg.3]. Let k be the number of bits needed to encode each value. The sorting protocol runs in k iterations. On each iteration, the elements are sorted according to one bit using counting sort (we denote it by Csort), starting from the least significant bit. In our algorithms, we will use CSort when we need to sort values just by one bit. Using numbers of [2], the cost of a single counting sort, applied to a vector of length n of k-bit values, is $\mathsf{Csort}_k^n = \mathsf{ShareConv}_k^n \oplus \mathsf{Mult}_k^n \oplus \mathsf{Shuffle}_{2k}^n \oplus \mathsf{Declassify}_k^n$ (the description of used subprotocols is given in Tab. 1). The cost of the entire radix sort is $\mathsf{Rsort}_k^n = (\mathsf{Csort}_k^n)^{\oplus k}$.

Quicksort (Qsort*)* . We use the algorithm from [8, Protocol 1] to sort n elements of k bits each. First of all, the array is shuffled, and then ordinary quicksort algorithm is run, declassifying only the outcomes of comparisons to decide where the element should be placed. As far as all elements are distinct, this does not cause any privacy breach. A Sharemind version of this algorithm is described in [2], and its average complexity is $\mathsf{Shuffle}_k^n \oplus (\mathsf{LessThan}_k^n \oplus \mathsf{Declassify}_1^n)^{\oplus \log n}$. Because of the random shuffle, the worst case comes with negligible probability, and we may indeed expect the average cost in practice.

In some cases, we apply quicksort to arrays that have already been shuffled. In this case, we denote the sorting itself by PQsort (*plain quicksort*), and its cost is $\mathsf{PQsort}_k^n = (\mathsf{LessThan}_k^n \oplus \mathsf{Declassify}_1^n)^{\oplus \log n}$. A good property of this sort is, that the sorting permutation is public.

Algorithm 4: Pointwise product of two sparse vectors

Data: Shared sparse vectors $\langle\!\langle \boldsymbol{u} \rangle\!\rangle$, $\langle\!\langle \boldsymbol{v} \rangle\!\rangle$
Data: n — the number of non-zero elements in a vector
Result: The vector $\langle\!\langle \boldsymbol{d} \rangle\!\rangle$ such that $\boldsymbol{d}[i] = \boldsymbol{u}[i] \cdot \boldsymbol{v}[i]$
$\langle\!\langle \boldsymbol{w} \rangle\!\rangle \leftarrow \langle\!\langle \boldsymbol{u} \rangle\!\rangle \| \langle\!\langle \boldsymbol{v} \rangle\!\rangle$;
$\langle\!\langle \boldsymbol{w} \rangle\!\rangle \leftarrow \mathsf{Shuffle}(\langle\!\langle \boldsymbol{w} \rangle\!\rangle)$;
$\langle\!\langle \boldsymbol{w} \rangle\!\rangle \leftarrow \mathsf{PQsort}(\langle\!\langle \boldsymbol{w} \rangle\!\rangle, \langle\!\langle \boldsymbol{w} \rangle\!\rangle.\mathsf{idx})$;
Let σ be the public sorting permutation of PQsort ;
foreach $i \in \{1, \ldots, 2n-1\}$ **do**
  $\langle\!\langle \boldsymbol{b}[i] \rangle\!\rangle \leftarrow \mathsf{ShareConv}(\mathsf{Equal}(\langle\!\langle \boldsymbol{w}[i].\mathsf{idx} \rangle\!\rangle, \langle\!\langle \boldsymbol{w}[i+1].\mathsf{idx} \rangle\!\rangle), k)$
  $\langle\!\langle \boldsymbol{d}[i] \rangle\!\rangle \leftarrow \mathsf{Mult}(\langle\!\langle \boldsymbol{b}[i] \rangle\!\rangle, \mathsf{Mult}(\langle\!\langle \boldsymbol{w}[i].\mathsf{val} \rangle\!\rangle, \langle\!\langle \boldsymbol{w}[i+1].\mathsf{val} \rangle\!\rangle))$
$[\![\boldsymbol{d}]\!] \leftarrow \sigma^{-1}([\![\boldsymbol{d}]\!])$
$[\![\boldsymbol{d}]\!] \leftarrow \mathsf{UnShuffle}([\![\boldsymbol{d}]\!])$
return $\langle\!\langle \boldsymbol{d}[1] \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{d}[n] \rangle\!\rangle$

Set Intersection and Difference. We describe the algorithms that we use for set intersection and set difference. The sets are represented as arrays of elements. We do not want to leak the precise set cardinality, and we assume that there is a known upper bound n on the number of elements. If the set has less than n elements, then the entries that represent missing elements are set to 0.

Let two sparse vectors be represented as sequences of index-value pairs, where the indices are encoded by ℓ bits, and values are encoded by k bits. We give an algorithm for a bit more general task, that allows to compute a pointwise product of values these vectors, matching their entries by indices. We then show how to instantiate it to set intersection and set difference.

Let $\boldsymbol{u}$ be a vector of length n. For each $\boldsymbol{u}[i]$, let $\boldsymbol{u}[i].\mathsf{idx}$ and $\boldsymbol{u}[i].\mathsf{val}$ denote the index and value component respectively. The pointwise product algorithm is given in Alg. 4. It concatenates the vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, obtaining a vector $\boldsymbol{w}$. It then sorts the obtained vector $\boldsymbol{w}$ by indices, so that similar indices are now consequent. It then computes the products $\boldsymbol{w}[i].\mathsf{val} \cdot \boldsymbol{w}[i+1].\mathsf{val}$ and leaves only those $\boldsymbol{w}[i]$ for which $\boldsymbol{w}[i].\mathsf{idx} = \boldsymbol{w}[i+1].\mathsf{idx}$ holds. In other words, it leaves exactly the products $\boldsymbol{v}[i].\mathsf{val} \cdot \boldsymbol{u}[j].\mathsf{val}$ such that $\boldsymbol{v}[i].\mathsf{idx} = \boldsymbol{u}[j].\mathsf{idx}$. In the end, the algorithm sorts the entries back to their initial positions (applying the sorting permutation inverse σ^{-1} and $\mathsf{UnShuffle}$), so that the second half of the resulting vector (the entries that are 0 anyway) can be discarded. Counting the number of all used subprotocols of this algorithm, we get the cost

$$\mathsf{Shuffle}^{2n}_{\ell+k} \oplus \mathsf{PQsort}^{2n}_{\ell} \oplus (\mathsf{Equal}_{\ell} \oplus \mathsf{ShareConv}_{k} \oplus \mathsf{Mult}^{2}_{k})^{\otimes 2n-1} \oplus \mathsf{UnShuffle}^{2n}_{k} \ .$$

This algorithm can be easily adjusted to set intersection and set difference. The summary of costs of these set operations is given in Tab. 3. Let the set elements be the indices of $\boldsymbol{u}$ and $\boldsymbol{v}$. We show how to assign the values.

Set intersection. For the set intersection task, we set $\boldsymbol{u}[i].\mathsf{val} = \boldsymbol{u}[i].\mathsf{idx}$ and $\boldsymbol{v}[i].\mathsf{val} = 1$ for all $i \in \{1, \ldots, n\}$. As the result, Alg. 4 returns exactly those indices of $\langle\!\langle \boldsymbol{u} \rangle\!\rangle$ that are present in $\langle\!\langle \boldsymbol{v} \rangle\!\rangle$.

Protocol Call	Returned Value	Cost
$\mathsf{Set}_\cap(\langle\!\langle \boldsymbol{a}\rangle\!\rangle, \langle\!\langle \boldsymbol{b}\rangle\!\rangle)$	$\langle\!\langle \boldsymbol{c}\rangle\!\rangle = \langle\!\langle \boldsymbol{a} \cap \boldsymbol{b}\rangle\!\rangle$	$\mathsf{Shuffle}_{2k}^{2n} \oplus \mathsf{PQsort}_k^{2n}$
$\mathsf{Set}_\setminus(\langle\!\langle \boldsymbol{a}\rangle\!\rangle, \langle\!\langle \boldsymbol{b}\rangle\!\rangle)$	$\langle\!\langle \boldsymbol{c}\rangle\!\rangle = \langle\!\langle \boldsymbol{a} \setminus \boldsymbol{b}\rangle\!\rangle$	$\oplus(\mathsf{Equal}_k \oplus \mathsf{ShareConv}_k \oplus \mathsf{Mult}_k^2)^{\otimes 2n-1} \oplus \mathsf{UnShuffle}_k^{2n}$

Table 3. Set operations (k-bit elements, $|\boldsymbol{a}| = |\boldsymbol{b}| = n$)

Sharing	Operation	Rounds	Communication
additive	Sum_k^n	0	0
	Mult_k	1	$6k$
	$\mathsf{OuterProd}_k^{n,m}$	1	$3(n+m)k$
	$\mathsf{ShareConv}_k$	2	$5k+4$
bitwise	$\mathsf{LessThan}_k$	$\log k$	$30k$
	Equal_k	$\log k$	$12k-9$
both	$\mathsf{Shuffle}_k^n$, $\mathsf{UnShuffle}_k^n$	3	$6nk$
	$\mathsf{Declassify}_k$	1	$6k$

Table 4. Basic operation costs of Sharemind

Set difference. To compute the difference between two sets, we need to keep exactly those elements of $\langle\!\langle \boldsymbol{u}\rangle\!\rangle$ that are *not* present in $\langle\!\langle \boldsymbol{v}\rangle\!\rangle$. If we flip the bit $\langle\!\langle \boldsymbol{b}[i]\rangle\!\rangle = \mathsf{Equal}(\langle\!\langle \boldsymbol{w}[i].\mathsf{idx}\rangle\!\rangle, \langle\!\langle \boldsymbol{w}[i+1].\mathsf{idx}\rangle\!\rangle)$ and keep only those elements $\langle\!\langle \boldsymbol{w}[i].\mathsf{idx}\rangle\!\rangle$ for which $\langle\!\langle \boldsymbol{b}[i]\rangle\!\rangle = 0$, we will also get elements of $\langle\!\langle \boldsymbol{v}\rangle\!\rangle$ that are not present in $\langle\!\langle \boldsymbol{u}\rangle\!\rangle$, and we do not need them. To get rid of these elements, for all $i \in \{1, \ldots, n\}$, we may initially set $\boldsymbol{u}[i].\mathsf{val} = \boldsymbol{u}[i].\mathsf{idx}$ and $\boldsymbol{v}[i].\mathsf{val} = 0$, and take $\langle\!\langle \boldsymbol{d}[i]\rangle\!\rangle \leftarrow \mathsf{Mult}(1 - \langle\!\langle \boldsymbol{b}[i]\rangle\!\rangle, \langle\!\langle \boldsymbol{w}[i].\mathsf{val}\rangle\!\rangle))$ as the final result. Only those entries $\langle\!\langle \boldsymbol{d}[i]\rangle\!\rangle$ that correspond to $\boldsymbol{u}$ can now be nonzero.

Algorithm Costs on Sharemind. We assume that the algorithms are run on secure multiparty computation system Sharemind [3]. In Tab. 4 we present the costs of basic operations that are used in our algorithms. The numbers are taken mainly from [3,10]. We take the cost of set intersection (and set difference) from Tab. 3, and substitute the costs of basic operations with values from Tab. 4. The summary of protocol costs on Sharemind platform is presented in Tab. 5.

Counting sort: The sorting assumes that the secret-shared input bits, according to which the sorting is done, are already given. The cost of counting sort is $\mathsf{ShareConv}_k^n \oplus \mathsf{Mult}_k^n \oplus \mathsf{Shuffle}_{2k}^n \oplus \mathsf{Declassify}_k^n$. The total communication is $n \cdot (5k+4) + n \cdot 6k + 6n \cdot 2k + n \cdot 6k = n(29k+4) \approx 30nk$ bits. The total number of rounds is $2+1+3+1 = 7$.

Quicksort: The cost of PQsort is $(\mathsf{LessThan}_k^n \oplus \mathsf{Declassify}_1^n)^{\oplus \log n}$. Since $\mathsf{LessThan}$ is more efficient using bitwise sharing, the entire PQsort is also more efficient using bitwise sharing. For PQsort, the total number of rounds is $(\log k + 1) \cdot \log n$, and the total communication is $\log n \cdot (30nk + 6n)$ bits.

Sharing	Protocol	Rounds	Communication
bitwise	PQsort_k^n	$\log n(\log k+1)$	$\log n(30nk+6n)$
	$\mathsf{Set}_{\cap k}^n$, $\mathsf{Set}_{\setminus k}^n$	$9+(\log 2n+1)(\log k+1)$	$60nk\log n+154nk$
both	Csort_k^n	7	$30nk$

Table 5. Auxiliary algorithm costs of Sharemind

Set intersection and difference: let us assume that the values are bitwise shared, since the comparisons and the quicksort are faster in this case. The number of bits for a single instance of intersection is $6\cdot 2n\cdot 2k+(30k+6)2n\log 2n+(12k-9+5k+4+12k)\cdot(2n-1)+6\cdot 2n\cdot k=24nk+(60nk+12n)(\log n+1)+(29k-5)(2n-1)+12nk=154nk+60nk\log n+12n\log n+2n-29k+5$ bits, which is ca $60nk\log n+154nk$. The number of rounds is $3+\log 2n(\log k+1)+(\log k+2+2)+3=9+(\log 2n+1)(\log k+1)$.

3.2 Comparing Bit Matrix and Set Based Approaches

We will now compare the bit representation and the sparse representation for FIM task. As mentioned in Sec. 3, for the bit representation, the intersections are found by multiplying two bit vectors pointwise, and the set difference can be computing analogously, by taking the negation of the bit vector that is being subtracted. For the sparse representation, the set operations can be found by using the algorithms defined in Sec. 3.1. The size of the sparse sets is m', and the set elements are encoded with $\log m$ bits. The numbers m' and n are defined as in the beginning of Sec. 3.

Cost of intersection for bit representation. First of all, we estimate the rounds and communication of the bit matrix intersections, based on the operation costs of Tab. 4.

According to the multiplication protocol [10] of Sharemind, this is $6m\log m$ 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 cost $\mathsf{Mult}_1^m=6m$, and the share conversion $\mathsf{ShareConv}_{\log m}^m=m\cdot(5\log m+4)$, which is in total $5m\log m+10m$. This approach is more efficient for $m\geq 2^{10}$.

Note that, if we need to compute $\mathsf{Mult}(\langle\!\langle \boldsymbol{a}_i\rangle\!\rangle,\langle\!\langle \boldsymbol{b}_j\rangle\!\rangle)$ for all $i\in\{1,\ldots,n_a\}$, $j\in\{1,\ldots,n_b\}$, then we could apply $\mathsf{OuterProd}(\langle\!\langle \boldsymbol{a}_1\rangle\!\rangle|\ \ldots\|\langle\!\langle \boldsymbol{a}_{n_a}\rangle\!\rangle,\langle\!\langle \boldsymbol{b}_1\rangle\!\rangle\|\ldots\|\langle\!\langle \boldsymbol{b}_{n_b}\rangle\!\rangle)$ instead, which has the same operation cost as $\mathsf{Mult}_1^{n_a+n_b}$. However, the share conversion would still have to be applied to all products, having cost $\mathsf{ShareConv}_{\log m}^{n_a\cdot n_b}$, and it does not scale well with n_a and n_b. Hence, we use only the first approach in this paper. The number of rounds in the first approach is 1, compared to the 3 rounds of the second approach.

Cost of intersection for sparse representation. Now we estimate the rounds and communication of the sparse intersections, based on the operation costs of Tab. 4. We compare them to the analogous cost metrics of bit matrix approach.

In the sparse representation, we have m' elements in the sets, each encoded using $\log m$ bits. Suppose that we are going to find the intersections of some n_a sets with some n_b sets.

Round advantage: An intersection takes $9+(\log 2m'+1)(\log\log m+1)$ rounds instead of 1 round of bit representation. This is an obvious disadvantage, but we hope to win in memory consumption and communication.

Communication: One set intersection requires $60m' \log m \log m' + 154m' \log m$ bits of communication, compared to $6m \log m$ of the bit based approach. Hence, for a single intersection, the advantage is non-negative iff $m \geq 10m' \log m' + 26m'$. However, while the total cost of all intersections is $(n_a + n_b) \cdot 6m \log m$ for the bit approach, it is $n_a \cdot n_b \cdot (60m' \log m \log m' + 154m' \log m)$, so the sparse approach scales badly. The advantage of set intersection is non-negative iff

$$m \geq \frac{n_a \cdot n_b}{n_a + n_b}(10m' \log m' + 26m') \ .$$

Comparisons of the bit and the set representations is given in Tab. 6.

Caveats of Sparse Representation. The best choice of m' depends on the values of n_a and n_b, which in turn depends on particular input data and the parameters, and these values are in general not known beforehand. In general, we would like to fix m' already in the beginning, since making m' dependent on data may leak more about it. On the other hand, we can make some further intersections worse if we underestimate the values of n_a and n_b.

Another problem is that, even if the data is sparse, they may be some single columns that have too many elements to make sparse approach applicable. We cannot just remove excessive transactions since we would have to decide which transactions exactly should be removed, and that choice may affect the final result significantly. On the other hand, finding an intersection between a dense column and a set of sparse columns is even worse than if sparse columns were treated as bit columns, regardless of the advantage that intersections of sparse columns give themselves.

If we agree to leak whether the number of nonzero entries has become at most m' after finding the intersection of two dense columns, then we may turn the resulting column into sparse. We convert bit columns of some branch of Diffset and Eclat to set columns only after *all* columns of that branch become sparse, and only if conversion still makes sense for the number of intersections that is going to be done on the next step.

Converting a bit matrix column to a set matrix column. The protocol Bits2Set transforms a column of a bit matrix to m' bitwise shared row identifiers, where m' is a known upper bound on the number of nonzero entries of a sparse

Algorithm 5: Bit vector to a set Bits2Set

Data: A bit vector $[\![\boldsymbol{b}]\!]$ of length m with at most m' nonzero entries
Result: A bitwise shared set representation $\langle\langle \boldsymbol{c} \rangle\rangle$ of $\langle\langle \boldsymbol{b} \rangle\rangle$

$\langle\langle \boldsymbol{b} \rangle\rangle = [\![\boldsymbol{b}]\!] \bmod 2$;
foreach $i \in \{1, \ldots, m\}$ **do**
  $\langle\langle \boldsymbol{c}[i] \rangle\rangle = \langle\langle \boldsymbol{b}[i] \rangle\rangle \cdot i$;
$\langle\langle \boldsymbol{d} \rangle\rangle = \mathsf{CSort}(\langle\langle \boldsymbol{c} \rangle\rangle, \langle\langle \boldsymbol{b} \rangle\rangle)$;
return $\langle\langle \boldsymbol{d}[1] \rangle\rangle, \ldots, \langle\langle \boldsymbol{d}[m'] \rangle\rangle$;

Type	Bit Communication Cost
bit representation	$(n_a + n_b) \cdot 6m \log m$
set representation	$(n_a \cdot n_b) \cdot 6m' \log m \cdot (10 \log m' + 26)$

Table 6. Multiple set algorithm costs of Sharemind

column. This protocol is given in Alg. 5. Assuming that m is a power of 2, even though the input bit vector is additively shared in $\mathbb{Z}_m$, it is easy to convert it to a bit vector shared in $\mathbb{Z}_2$ by locally truncating each entry up to the least significant bit. In practice, at least using Sharemind system, the entries should be shared over $\mathbb{Z}_{2^{\lceil \log m \rceil}}$ anyway.

Computing the multiplications is a local operation since we are multiplying by a public value j. The bit $\boldsymbol{b}[i] \in \{0, 1\}$ should be multiplied (in $\mathbb{Z}_2$) with each bit of j, and this is a local operation. The cost of Csort is 7 rounds and $30mk$ communication. For fixed m and m', the total cost of the protocol is denoted $\mathsf{Bits2Set}_{m,m'}$, which is 7 rounds and $30m' \log m$ communication.

3.3 Combining Dense and Sparse Representations

We still assume that we are using the standard Eclat and Diffset algorithms without modifying them in general. The algorithms should now additionally decide, which columns should be represented as sets, and which columns as bit vectors. As an example, we describe the new privacy preserving Eclat algorithm, and Diffset would be analogous, just using set difference instead of set intersection. Let m' be the bound for which set based approach is applicable. Each iteration of privacy preserving Eclat (depicted in Alg. 6) works as follows.

The itemsets that are are found to be frequent are public: similarly to [1], they will be declassified in the end anyway and hence do not leak any additional information. Let the itemsets of the current iteration of Eclat be represented by $\mathcal{P}$, as in Alg. 2. The invariant is that, the secret shared supports of the itemsets $\mathcal{P}$ are either all in bit representation, or are all in set representation. It is not possible that some supports of the same prefix are bit columns, while others are set columns, since computing intersections between columns of different representations would be too inefficient.

The representation determines the algorithm used to compute the supports of the next iteration, which are the intersections of current supports. The resulting bit columns that have at most m' elements are converted to set columns using Bits2Set protocol. To keep the invariant, this is being done only if *all* columns of the current prefix have at most m' elements. It is important that, even if a column is sparse enough, it make sense to convert bit columns to set columns only if it indeed gives advantage on the next step. This is done by estimating the cost of both approaches and comparing them.

Let n' be the current number of columns for the given prefix. The cost of one conversion is $\mathsf{Bits2Set}^{n'}_{m,m'}$. In addition, set representation makes counting ones a bit harder, requiring more comparisons of cost $\mathsf{Equal}^{m'n'}_{\log m}$. This overhead should be added to the cost of sparse intersections $(\mathsf{Set}^{m'}_{\cap \log m})^{(n'^2-n')/2}$. The resulting cost $cost_{set}$ is compared to the cost that we would have without converting bit columns to sparse columns, which is $cost_{bit} = (\mathsf{OuterProd}^{n',n'}_{\log m})^m$. The bit columns are converted to set columns iff $cost_{set} \leq cost_{bit}$.

We note that comparison with m' can be done only if all other conditions are satisfied, and it is not needed for the bit representation. Hence, the cost $\mathsf{LessThan}^{n'}_{\log m} \oplus \mathsf{Declassify}^{n'}_{1}$ of the comparison itself can be added to $cost_{set}$ as well, but its contribution is very small.

4 Benchmarks

We have implemented our algorithms in Sharemind and tested them on some public datasets that are available e.g. in [7]. We tested Diffset on the denser dataset Chess (3196 rows, 75 columns, 49,3% density), the medium density dataset Mushroom (8124 rows, 119 columns, 19,3% density), and we tested Eclat on the sparse dataset Retail (88162 rows, 16470 columns, 0.06% density). Since Retail is a very large set, and we had to take a very small threshold t to get use of sparse columns, we have taken only the first 5500 of its rows for our tests. We ran the FIM task them with different thresholds t and different upper bounds m' on sparse column size. If $m' = 0$, then only the bit representation was considered. The results are given in Tab. 7. In addition to time, we measured the memory usage and the bit communication. For these two metrics, we have three columns "sparse", "dense" and "total", denoting how much overhead was coming from the set operations of sparse columns, the dense columns, and in total. The communication cost of converting a bit column to a set column is treated as the cost of sparse representation.

We see that the advantage of sparse representation is very small. The reason is that there are too few columns for which set representation was more efficient. Sometimes, the results are even slightly worse than for pure bit representation. One reason for that is the data types of Sharemind are of fixed size, and it is not possible to encode value in k bits for an arbitrary k. Moreover, the set representation increases the number of rounds, and even some local computations, which still affect the efficiency, even though they are less significant than the communication.

Algorithm 6: Privacy Preserving Eclat

Data: $\mathcal{X}$ is a set of n itemsets of size $k-1$ with the same prefix
Data: $\langle\!\langle \mathbf{M} \rangle\!\rangle$ is the $m \times n$ matrix of supports of the itemsets
Data: threshold t
Result: Frequent itemsets of size at least k with the same prefix

if $\langle\!\langle \mathbf{M} \rangle\!\rangle$ *has a bit representation* **then**
  $\langle\!\langle \mathbf{C} \rangle\!\rangle \leftarrow \mathsf{OuterProd}(\langle\!\langle \mathbf{M} \rangle\!\rangle, \langle\!\langle \mathbf{M} \rangle\!\rangle)$;
  $\langle\!\langle \boldsymbol{s} \rangle\!\rangle \leftarrow \mathsf{Sum}(\langle\!\langle \mathbf{C} \rangle\!\rangle)$;
else
  $\langle\!\langle \mathbf{C} \rangle\!\rangle \leftarrow \mathsf{Set}_{\cap}(\langle\!\langle \mathbf{M} \rangle\!\rangle, \langle\!\langle \mathbf{M} \rangle\!\rangle)$;
  $\langle\!\langle \boldsymbol{s} \rangle\!\rangle \leftarrow \mathsf{Sum}(1 - \mathsf{Equal}(\langle\!\langle \mathbf{C} \rangle\!\rangle, 0))$;
$\boldsymbol{b} \leftarrow \mathsf{Declassify}(\langle\!\langle \boldsymbol{s} \rangle\!\rangle \geq t)$;
foreach $i \in \{1, \ldots, n\}$ **do**
  $F_i \leftarrow \emptyset$; $\langle\!\langle \mathbf{M_i} \rangle\!\rangle \leftarrow [\,]$;
  foreach $j \in \{i+1, \ldots, n\}$ **do**
    $R = X_i \cup X_j$;
    if $b[i \cdot n + j] \geq t$ **then**
      $F_i = F_i \cup \{R\}$;
      $\langle\!\langle \mathbf{M_i} \rangle\!\rangle = \langle\!\langle \mathbf{M_i} \rangle\!\rangle \| \langle\!\langle \mathbf{C} \rangle\!\rangle[i \cdot n + j]$;
  if $F_i \neq \emptyset$ **then**
    $n' = |F_i|$; //number of all columns
    $n'' = \mathsf{Sum}(\langle\!\langle \boldsymbol{s} \rangle\!\rangle \leq m')$; //number of sparse columns
    $cost_{bit} = (\mathsf{OuterProduct}^{n',n'}_{\log m})^{\otimes m}$;
    $cost_{set} = (\mathsf{Bits2Set}_{m,m'} \oplus \mathsf{Equal}^{m'}_{\log m})^{\otimes n'} \oplus \mathsf{Set}_{\cap}^{(n'^2-n')/2}_{\log m}$;
    if $(n' \neq n'')$ ***or*** $(cost_{bit} < cost_{set})$ **then**
      $F'_i = \mathsf{Eclat}(F_i, \langle\!\langle \mathbf{M_i} \rangle\!\rangle, t)$;
    else
      $F'_i = \mathsf{Eclat}(F_i, \mathsf{Bits2Set}(\langle\!\langle \mathbf{M_i} \rangle\!\rangle, m), t)$;
return $\bigcup_i F'_i$;

5 Conclusion

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 algorithms turn out to be not as efficient as we wanted. The main challenge is that the algorithms for sparse representation 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. Also, while sparse representation has not improved efficiency for the benchmarked tables, it allowed to reduce the local memory usage, which may be important for large datasets.

References

1. 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. Springer-Verlag, 2012.
2. D. Bogdanov, S. Laur, and R. Talviste. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In *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.
3. 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.
4. R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards statistical queries over distributed private user data. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 169–182. USENIX Association, 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. Ú. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1054–1067. ACM, 2014.
7. Frequent itemset mining dataset repository. `http://fimi.ua.ac.be/data/`. Last accessed 2017-09-16.
8. K. Hamada, R. Kikuchi, D. Ikarashi, K. Chida, and K. Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *ICISC*, volume 7839 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2012.
9. 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.
10. L. Kerik, P. Laud, and J. Randmets. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In *Proceedings of WAHC'16 - 4th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, 2016.
11. 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. ACM, 2014.
12. F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. *Commun. ACM*, 53(9):89–97, 2010.
13. C. Sun, Y. Fu, J. Zhou, and H. Gao. Personalized privacy-preserving frequent itemset mining using randomized response. *The Scientific World Journal*, 2014.
14. 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. ACM, 2003.
15. C. Zeng, J. F. Naughton, and J.-Y. Cai. On differentially private frequent itemset mining. *Proc. VLDB Endow.*, 6(1):25–36, Nov. 2012.

dataset, FIM alg.	t	m'	memory			communication			time
			sparse	dense	total	sparse	dense	total	(s)
chess, Diffset	3000	36	288 B	968 KB	968 KB	22 KB	16.5 MB	16.6 MB	2.0 s
		18	360 B	929 KB	930 KB	22.9 KB	16.4 MB	16.4 MB	2.5 s
		9	72 B	968 KB	968 KB	15.2 KB	16.5 MB	16.6 MB	3.0 s
		4	32 B	965 KB	965 KB	14 KB	16.5 MB	16.5 MB	2.5 s
		0	0	991 KB	991 KB	0	16.5 MB	16.5 MB	2.6 s
	2800	36	1.4 KB	8.5 MB	8.5 MB	154 KB	140 MB	141 MB	18 s
		18	396 B	8.6 MB	8.6 MB	122 KB	140 MB	140 MB	19 s
		9	144 B	8.6 MB	8.6 MB	115 KB	140 MB	140 MB	19 s
		4	64 B	8.6 MB	8.6 MB	113 KB	140 MB	140 MB	19 s
		0	0	8.6 MB	8.6 MB	0	140 MB	140 MB	18 s
	2600	36	8 KB	38.6 MB	38.6 MB	710 KB	595 MB	596 MB	83 s
		18	2.1 KB	38.9 MB	38.9 MB	528 KB	595 MB	595 MB	83 s
		9	576 B	39.1 MB	39.1 MB	487 KB	595 MB	596 MB	80 s
		4	208 B	39 MB	39 MB	476 KB	595 MB	594 MB	79 s
		0	0	39 MB	39 MB	0	594 MB	594 MB	77 s
	2400	36	220 KB	130 MB	130 MB	2.2 MB	2.0 GB	2.0 GB	265 s
		18	857 B	130 MB	130 MB	1.8 MB	2.0 GB	2.0 GB	267 s
		9	738 B	132 MB	132 MB	1.57 MB	2.0 GB	2.0 GB	255 s
		4	328 B	131 MB	131 MB	1.55 MB	1.95 GB	1.95 GB	254 s
		0	0	132 MB	132 MB	1.5 MB	1.95 GB	1.95 GB	253 s
mushroom, Diffset	2600	92	180 B	30.4 MB	30.4 MB	701 KB	516 MB	517 MB	83 s
		46	136 B	29.4 MB	29.4 MB	546 KB	510 MB	511 MB	81 s
		23	7.64 KB	29 MB	29 MB	641 KB	507 MB	508 MB	80 s
		11	3.43 KB	29.2 MB	29.2 MB	430 KB	506 MB	507 MB	80 s
		0	0	31.7 MB	31.7 MB	0	512 MB	512 MB	64 s
	2400	92	35 KB	45.5 MB	45.5 MB	1.29 MB	769 MB	769 MB	117 s
		46	25.3 KB	43.9 MB	43.9 MB	958 KB	757 MB	758 MB	124 s
		23	13.9 KB	43.5 MB	43.5 MB	952 KB	754 MB	755 MB	123 s
		11	6.67 KB	43.4 MB	43.4 MB	556 KB	753 MB	753 MB	123 s
		0	0	48.2 MB	48.2 MB	0	761 MB	761 MB	97 s
	2200	92	53.5 KB	62.7 MB	62.7 MB	1.94 MB	1.10 GB	1.10 GB	183 s
		46	38 KB	60.3 MB	60.4 MB	1.43 MB	1.08 GB	1.08 GB	172 s
		23	36.5 KB	59.9 MB	59.9 MB	7.97 MB	1.07 GB	1.08 GB	180 s
		11	18 KB	59.3 MB	59.4 MB	3.70 MB	1.07 GB	1.07 GB	179 s
		0	0	66.9 MB	66.9 MB	0	1.08 GB	1.08 GB	151 s
	2000	92	82 KB	101 MB	101 MB	3.00 MB	1.73 GB	1.73 GB	279 s
		46	74 KB	94.8 MB	94.9 MB	2.59 MB	1.70 GB	1.70 GB	260 s
		23	76.5 KB	92.2 MB	92.2 MB	16.8 MB	1.67 GB	1.69 GB	289 s
		11	92.5 KB	88.4 MB	88.5 MB	28.4 MB	1.63 GB	1.66 GB	300 s
		0	0	108 MB	108 MB	0	1.72 GB	1.72 GB	214 s
retail (trimmed), Eclat	15	62	744 B	25.0 MB	25.0 MB	22.9 MB	49.8 GB	49.8 GB	2660 s
		31	1.05 KB	24.9 MB	24.9 MB	22.9 MB	49.8 GB	49.8 GB	2640 s
		15	0	25.0 MB	25.0 MB	0	49.7 GB	49.7 GB	2690 s
		7	0	25.0 MB	25.0 MB	0	49.6 GB	49.6 GB	2690 s
		0	0	25.0 MB	25.0 MB	0	49.6 GB	49.6 GB	2620 s

Table 7. Benchmarks on Sharemind