Pasteur: Scaling Privacy-aware Data Synthesis

Antheas Kapenekakis[1], Daniele Dell'Aglio[1], Martin Bøgsted[2], Minos Garofalakis[3], and Katja Hose[4]

[1] Aalborg University, Aalborg, Denmark
[2] Aalborg University Hospital, Aalborg, Denmark
[3] Athena Research Center, Athens, Greece
[4] TU Wien, Vienna, Austria

Abstract. Privacy-aware data synthesis is a field aiming to liberate data access through the generation of synthetic data which mirrors the original without resulting in privacy exposure. State-of-the-art algorithms for structured data perform well in datasets with tables of a few million rows but result in prohibitive runtimes when scaling to hundreds of millions of rows. In addition, due to the sensitive nature of data, practitioners are often limited to a single server environment. In this paper, we present the framework Pasteur, which aims to scale privacy-aware data synthesis linearly under a single server environment. Pasteur achieves this through a parallelization approach tailored for synthesis, optimized memory representations, and an accelerated marginal calculation algorithm (bottleneck in a class of privacy-aware algorithms). We show Pasteur performing pre-processing, synthesis, and evaluation of a tabular dataset with 1 billion rows (200 GB) in 1 hour on a 16 core CPU server.

1 Introduction

Data synthesis aims to promote responsible data sharing through anonymization [11]. However, state-of-the-art algorithms have focused on structured data of limited size and complexity, i.e., tables with up to 5 million rows (less than 100 MB) [3,4,13,14,20,23]. Hence, these datasets are small enough to fit conveniently in main memory and can be processed swiftly without the need for parallel or efficient processing. However, real-world datasets may contain dozens of tables with a larger degree of complexity and billions of rows. This is exacerbated with sensitive data, as GDPR resources are limited, often to a small percentage of a cluster's resources and to a single machine, due to regulatory requirements.

Therefore, in this paper we attempt to address the challenge of scaling synthesis linearly to larger-than-memory datasets when confined to a single server environment. To solve this challenge, we present Pasteur, a system for scalable privacy-aware data synthesis for datasets far larger than main memory. We focus on scaling privacy-aware Probabilistic Graphical Model (PGM) algorithms [3, 12, 13, 22], since recent benchmarks [18] show them performing well in terms of privacy versus utility.

In summary, the main contributions of this paper are: 1. Improved memory usage by information pruning and optimized memory formats. 2. A parallelization approach that maintains a constant scaling factor equal to the number of cores, with stable memory use regardless of dataset size. 3. A theoretically optimal Single Input Multiple Data (SIMD) marginal calculation algorithm for PGMs offering a 150x speedup over existing implementations in single core containing multiple parallelization strategies to ensure it scales linearly across all CPU cores for tables from 1M to 1B rows and beyond.

The remainder of this paper is structured as follows: Section 2 provides preliminaries and the problem definition, Section 3 presents the Pasteur system with its improved memory usage and parallelization. Afterwards, Section 4 presents the marginal calculation algorithm; the first part covers the algorithm and the second part its parallelization. Our evaluation results are discussed in Section 5. Finally, Section 6 covers related work and Section 7 concludes the paper.

2 Problem Definition and Analysis

Let D be a sensitive relational dataset consisting of multiple tables $T \in D$ with foreign key dependencies. Each row in T describes a protected entity such as patients. The goal of privacy-aware data synthesis is to produce a dataset S that approximates D without incurring privacy exposure of the entities of D. For privacy-aware data synthesis, recent work has focused on Probabilistic Graphical Model (PGM) algorithms [3,12,13,22] as they have compelling performance [18] when privacy is required, so in this work we optimize those. When working with sensitive data, analysts are often limited to a single-machine environment, typically a server or laptop, with a predefined number w of CPU cores and amount M of main memory. The problem we aim at studying is:

Problem 1. How can we maximize the usage of w available CPU cores to efficiently perform PGM-based data synthesis with datasets D, where the size of a dataset D in memory $|D|$ is up to $|D| >> M$.

To solve this problem, we identify three challenges. Firstly, the memory footprint of data needs to be optimized, as a smaller footprint allows for higher throughput, leading to faster synthesis. Therefore, the first challenge is *identifying and removing unnecessary information* in the data and optimizing data encodings to compress the data both in main memory and disk. Once the data is in an optimal format, the next challenge is to optimize parallelization. That means, we aim *to maximize the usage of w cores*, while *memory usage remains stable under M*, but close to it to minimize swapping. Finally, we aim to optimize how synthesis algorithms handle data themselves. We focus on PGM algorithms, which only access input data through a single operation: marginal calculation. Therefore, the third challenge lies in *how to optimize marginal calculation.* By achieving these challenges, the PGM algorithms we surveyed scale to larger-than-memory datasets without any changes.

Here, it is worth to present an overview of marginal calculation to complement the complexity analysis in Section 4. Let $T \in D$ be a table with N rows and C columns associated to discrete variables $T_1, \ldots, T_C$. Let $supp(T_j)$ be the support of T_j, i.e., the possible values that T_j can take. Given a subset of the column indices $\{j_1, \ldots, j_r\} \subseteq \{1, \ldots, C\}$, we define an r-dimensional marginal of T as a set of r columns $T' = \{T_{j_1}, \ldots, T_{j_r}\} \subseteq T$, where $E' = supp(T_{j_1}) \times \cdots \times supp(T_{j_r})$ is the set of possible values of the rows in T'. We define the marginal frequency distribution of T' as:

$$f(e_1, \ldots, e_r) = \frac{1}{N} \sum_{i=1}^{N} 1[t_{i,j_1} = e_1, \ldots, t_{i,t_r} = e_r], \quad \forall (e_1, \ldots, e_r) \in E',$$

where $1[\cdot]$ denotes the indicator function. The goal of marginal calculation is computing the marginal frequency distribution of T'. This is performed in three steps. Firstly, we linearize the column values into a unique value per combination:

$$s_i = t_{i,j_1} + \sum_{k=2}^{r} t_{i,j_k} \prod_{l=1}^{k-1} |supp(T_{j_l})|, \forall i \in \{1, \ldots, N\}.$$

Secondly, we create the marginal frequency distribution of the linearized values:

$$h_s = \sum_{i=1}^{N} 1(s_i = s),\ \forall s \in \{1, \ldots, |E'|\}.$$

Thirdly, we reshape h_s to the r-dimensional distribution $T' = [T_{j_1}, \ldots, T_{j_r}]$.

For a collection of R marginals, we obtain the following complexity analysis. Linearization is an arithmetic operation that scales with the number of marginals R and, the number of columns per marginal r, and the number of rows N in the table the marginal is calculated for: $O(R \cdot N \cdot r)$. The second step accesses one vector of size N per marginal: $O(R \cdot N)$. Finally, post-processing depends on the marginal domain size $|E'|$ and the number of marginals R: $O(R \cdot |E'|)$. $|E'|$ is in the thousands for all marginals, as that is required for them to be statistically significant, which results in $N \gg |E'|$. We may treat r as a constant, as it is a small single digit number for all marginals. Given the above, the dominant term $O(R \cdot N)$. Since $O(R \cdot N)$ is a very large term, if the synthesis algorithm requests a large R, it is important this operation is performed efficiently. In addition, the size of a table T affects its ideal placement in memory (single copy, multiple copies, or swapping), necessitating varying memory placement depending on the magnitude on N.

3 Pasteur System and Principles

In this section, we cover a solution to the two first challenges, defining the architecture system: *identifying and removing unnecessary information* in the data and optimizing data encodings to compress the data both in main memory and disk, and *optimizing for parallelization* in a single node with memory pressure.

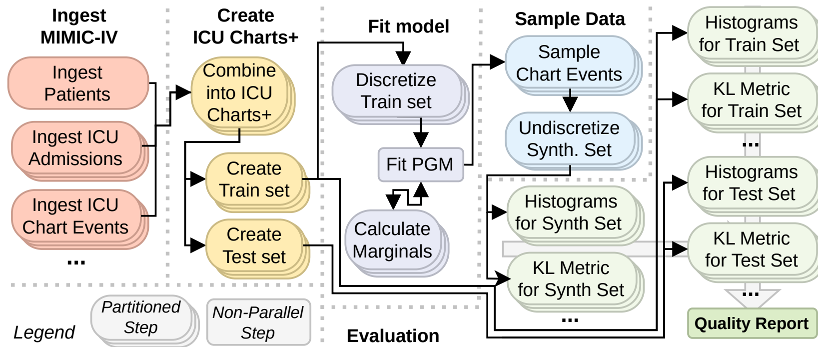

Fig. 1: Example Pipeline for the ICU Charts Dataset

3.1 Data Representation and Preprocessing

Improved memory density acts as a multiplier for the efficiency of data synthesis. This is for two reasons: first, it allows holding more data in memory at a time, minimizing swapping, and second, denser representations enable Single Instruction Multiple Data (SIMD) to handle more data per CPU instruction. Following, we walk through the steps of optimizing a dataset by using as an example the dataset MIMIC-IV [9], a large medical dataset (110 GB uncompressed) sourced from a US hospital and provided as a set of 27 tables $T \in D$ in CSV files. In it, we find text columns containing medicine, dosages, doctor's notes, dates in US format, and numerical columns with various ranges.

Pruning Columns. We distinguish columns in two types: learnable and non-learnable. Learnable columns store the information that should be learned and synthesized, such as age and diagnosis. Non-learnable columns are personal information with little or no value in the context of data synthesis, such as patient name and email. We prune non-learnable columns during ingestion, to avoid overhead, and when one is used as a join attribute, we replace it with an integer.

Encoding. Many datasets come with data in string format. For example, MIMIC-IV is available as a CSV file, where dates and categorical values are encoded as strings. We transform these values to reduce the memory footprint and improve performance. Specifically, dates are converted to Unix timestamps (int64). Categorical columns are converted to integers through dictionaries to store the original value. For integers, when they represent categorical values, the smallest bit width possible is chosen. Finally, numerical columns are converted to floating point values, with a preference for float32. The end result is a compacted dataset that can be used directly for synthesis.

Partitioning. Despite the optimized dataset being smaller, it might still not fit in main memory. For such cases, we designed a partitioning strategy for efficient swapping between disk and main memory. Here, we leverage the properties of synthesis: it consists of two types of operations: processing steps (e.g., discretizing, encoding) and model fitting (e.g., sampling marginals) on a set of entities (e.g., patients). In these operations, synthesis entities (e.g., patients, customers)

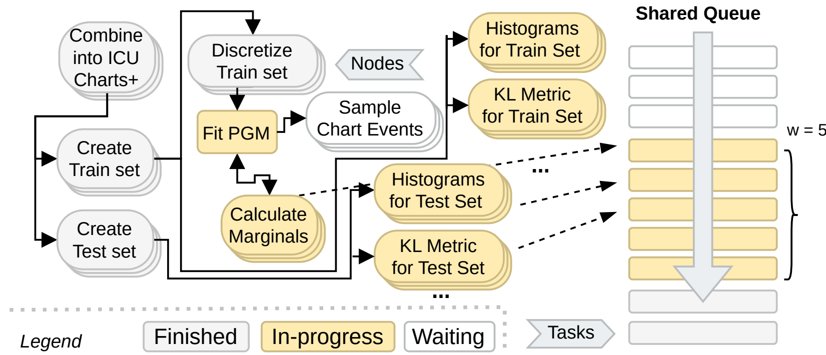

Fig. 2: The architecture applied to the example in Figure 1

are treated independently (e.g., during preprocessing each patient is encoded independently, in fitting models batches may be used). Therefore, it is possible to shard the input dataset using synthesis entities as a key.

Conclusion. Pruning, partitioning, and encoding form a foundation for processing arbitrarily large datasets with maximal throughput. Following, we build upon it with a parallelization approach tailored to running on single nodes.

3.2 Parallelization Approach

Data synthesis experiments are processing heavy: dataset is re-ingested every time to account for hyper-parameters, output dataset has to be processed and evaluated. We observe that this results in many independent processing steps that can be executed simultaneously (evaluation, preprocessing model fitting; inter-task parallelism). In addition, each step is parallelizable, as the data is partitioned (intra-task parallelism).

Therefore, in the architecture of Pasteur, we represent synthesis pipelines as Directed Acyclic Graphs (DAGs) G where nodes correspond to processing steps and edges to artifact dependencies. Figure 1 shows a Pasteur's synthesis pipeline with the MIMIC-ICU dataset. The parallelization challenge can then be reformulated into parallelizing a synthesis DAG G with a constant parallelization factor, close to w, while memory use is under M.

We note that due to the memory constraint, naive parallelization with fixed factors for inter-task (w_1) and intra-task (w_2) would result in a varying parallelization factor $w_{\max} = w_1 \cdot w_2$ and memory use, which is not optimal on a single server (either average CPU use is low or there is a risk of memory overflow). Therefore, we propose a new parallelization approach where each synthesis step schedules sub-steps, which act as the atoms of computation, where each sub-step operates on a partition of the dataset.

In this architecture, there is a shared queue which is bound to process up to w sub-steps at a time. Steps in G launch as light threads that schedule sub-steps to run per partition on the queue. Each sub-step loads, processes, and saves a

partition to disk, avoiding Inter Process Communication (IPC). Since each sub-step operates on a similar sized partition, they have similar memory footprint proportional to that. Therefore, given a predetermined partition size, w parallel tasks can be bound to a memory footprint under M. This architecture is shown in Figure 2. The benefit of this approach is that it allows processing multiple heterogeneous and arbitrary operations simultaneously, while remaining agnostic to them, with a stable factor close to w and bounded memory use under M, for a wide range of w and $|D|$ values.

Partition Sizing. The final part is determining the number of partitions N_P to split the dataset to. Assuming each partition has size $|P|$, we have $|D| = N_P \cdot |P|$. Each sub-step execution uses a similar amount of memory $M_{\text{sub-step}}$, where $w \cdot M_{\text{worker}} < M$. In addition, there is certain overhead for sub-step operations, therefore we define c as a constant representing memory headroom (e.g., 2-5): $M_{\text{worker}} = c \cdot |P|$. From combining the previous we get $w \cdot c \cdot |P| < M$. Finally, we derive the partition number needs to meet the following $N_P > \frac{|D|}{M} \cdot c \cdot w$.

This is a lower bound. For the optimal number, we insert two additional constraints. First, since sub-steps are scheduled in groups of size N_P (one for each partition) and have similar execution times (similar operation and size), there is a preference for making N_P a multiple of w. Then, a smaller N_P minimizes scheduling overhead by maximizing memory use. Applying both constraints results in the following: $N_P = \left\lceil \frac{|D|}{M} \cdot c \right\rceil \cdot w$, where $\lceil \cdot \rceil$ is the ceiling operator.

4 Accelerated Marginal Calculation

As we analyzed in the third challenge of Section 2, marginal calculation is a crucial operation in PGM algorithms, and the only function which affects their scaling related to dataset size. Various PGM works [3, 13] use the NumPy function "histogramdd" for marginal calculation. This is sufficient for small datasets but features a large overhead due to not being specialized for marginal calculation, as it is a floating point multidimensional histogram function (e.g., unnecessary binning, 32 bit widths). Therefore, in this section, we optimize this operation in two steps: first, we derive an optimized SIMD variant of the marginal calculation algorithm (Section 4.1) under a single core in-memory. Then, we integrate this approach to the parallelization architecture of the system (Section 4.2), to provide linear scaling across a wide range of dataset sizes.

4.1 Optimizing the Computation of Histograms

To optimize linearization, we begin with a vectorized approach. Then, we further optimize it with a custom Operator Fusing SIMD algorithm, which applies both SIMD and operator fusing optimizations.

Vectorized Approach. First, we apply the marginal calculation steps using vectorized functions, resulting in the following complexity: when linearizing the column values of an r-dimensional marginal, we perform one multiplication for each column apart from the first one ($r-1$ reads, $r-1$ writes) and then sum

the results (r reads, $r-1$ writes). For N rows, we have a memory access complexity of $O(N(4 \cdot r - 3))$. Finally, counting is performed by iterating over the final index (1 read) and increasing a marginal vector (1 read, 1 write) for $O(3N)$. The total memory access complexity is $O(4 \cdot N \cdot r)$. In addition, through using vectorization we benefit from SIMD. SIMD allows processing multiple rows at a time which, when combined with the efficient memory structure introduced in Section 3.1, results in a considerable theoretical speedup (16-32x for 8-16bit values over 256bit SIMD). However, due to the way the linearization and counting steps interact (loup dependencies, mixed integer widths), operator fusing, another crucial optimization is not applied automatically.

Operator Fusing SIMD Approach. Linearization creates multiple intermediary multiplication results for calculating the vector s_i, which is then used for the marginal frequency distribution calculation. Both s_i and other intermediary multiplication results are not used after the calculation, and saving them to memory consumes main memory bandwidth. Ideally, we would use the concept of operator fusing, where all intermediary outputs are kept in registers, and only h_s is written to memory. Since h_s has a cardinality up to thousands (see Section 2), this results in it residing in the CPU cache, eliminating all memory writes.

While operator fusing and Single Input Multiple Data (SIMD) are widespread techniques, marginal calculations break automatic optimizations that apply in compilers in two ways: by mixing integer widths during linearizing (columns use different widths depending on cardinality) and introducing a loop-driven dependency during the creation of the frequency vector. To embed both SIMD and Operator Fusing at the same time, we create a custom algorithm, shown in Algorithm 1. We use 256 bit SIMD instructions as a base. Since the majority of marginals have a size smaller than 65535, we present a version for 16 bit marginals in the paper, which can be trivially extended to 32 bits.

In an initial step, the algorithm separates variables based on their cardinality into unsigned 8 bit and 16 bit groups and calculate a multiplier for each based on cardinality. Then, a partially unrolled loop is used to handle 16 (16 byte) rows at a time. 8 bit variable groupings are first expanded to 16 bit. Then, the columns are linearized using 16 bit vectorized instructions (1 read per variable and row), by multiplying the values with a precalculated multiplier. The final result is used to increment the histogram with individual instructions, as SIMD does not apply due to potentially incrementing the same memory location. We compile different versions of the algorithm for combinations of n_8 and n_{16}, which removes the inner for loops that would be used for linearization.

Therefore, during execution, the algorithm runs a single for loop that processes 16 rows per iteration, with no inner-for loops. It executes a set of predefined SIMD operations back-to-back which result in creating the linearized vector, then 16 individual writes increment the histogram residing in CPU cache. Incrementing and SIMD use different ALU units, causing the out-of-order execution of the CPU to partially overlaps them. Since the resulting marginal frequency distribution fits in CPU cache, and all operands are stored in regis-

Algorithm 1 256 bit marginal calculation for 16 bit marginals.

```
1: procedure MARGINAL_16BIT_256BIT(N, out, n_8, mul_8, arr_8, n_16, mul_16, arr_16)
2:     for i ← 1 to N − 16 step 16 do
3:         r_{idx,256} = 0
4:         for j ← 1 to n_8 do                                  ▷ Unrolled loop
5:             r_128 ← arr_8[j][i : i + 16]                      ▷ 16 ∗ 8 = 128
6:             r_256 ← EXPAND(r_128)
7:             r_256 = mul_8[j] ∗ r_256
8:             r_{idx,256} = r_{idx,256} + r_256
9:         end for
10:        for j ← 1 to n_16 do                                 ▷ Unrolled loop
11:            r_256 ← arr_16[j][i : i + 16]
12:            r_256 = mul_16[j] ∗ r_256
13:            r_{idx,256} = r_{idx,256} + r_256
14:        end for
15:        for j ← 1 to 16 do                                   ▷ Unrolled loop
16:            INCREMENT(out[r_{idx,256}[j]])
17:        end for
18:    end for
19:    ...                                          ▷ Handle last N mod 16 rows.
20: end procedure
```

ters, the algorithm performs $O(rN)$ memory accesses, which is 4 times lower than the vectorized approach. Moreover, if the dataset is smaller than the processor's cache (e.g., a couple thousand rows), we remove all memory accesses after calculating the first marginal as the dataset is cached.

4.2 Parallelizing Marginals

The prior approach calculates a single marginal on a single core in memory. Given a collection of marginals to compute, we need to parallelize and scale it to w cores, for a wide range of dataset sizes. Here, memory bandwidth becomes a concern. We have three memory tiers ordered by bandwidth: CPU Cache, RAM, and disk. Given a certain dataset size, we want to remain to the fastest tier possible, and ensure we saturate its bandwidth. Therefore, we propose three memory strategies, where each is optimal for a specific dataset size range, as we investigate in Section 5.

Using Shared Memory. In this approach, we place the whole dataset in a shared memory pool and let cores share it during marginal calculation. Then, we evenly distribute the marginal collection to each core. If the dataset is small enough to fit in the CPU L3 cache (e.g., up to 100k rows), this removes most memory accesses with both the marginal vector and the dataset staying in it. If it is larger, however, cache invalidation and limited memory channel bandwidth harm performance, making this approach non-optimal.

Using Unique Copy. Here, we give each CPU performing marginal calculations its own copy of the data and a subset of marginals to calculate. The random

placement ensures that the w cores utilize all DRAM channels and their access patterns do not cross, which would cause cache retrieval invalidation. The downside of this approach is that for w cores, the system memory used is $(w+1) \cdot |D|$, so it is only applicable if $(w+1) \cdot |D| < M$.

Using Unique Partition. Finally, for datasets that do not fit in memory, we partition the dataset as in Challenge 2, and give each core a partition and the whole marginal batch, having each core calculate sub-marginals for its partition. In the end, the sub-marginals are merged to create the final dataset marginal. If $N_P > w$, after finishing, each core receives another partition. This approach is only viable if there are enough rows in each partition, otherwise it results in slower performance. The overall computation scales with $O(R \cdot |D|)$ where merging sub-marginals scales with $O(R \cdot |r| \cdot N_P)$. If $|D| \gg r \cdot N_P$, which is equivalent to $|P| \gg r$, the overhead of merging sub-marginals is theoretically amortized. This approach scales infinitely, as cores can load partitions from disk. Moreover, if $|D| < M$, storing the partitions in memory can remove disk overhead for subsequent marginal batches.

5 Evaluation

In this section, we present the results of our evaluation of Pasteur; we evaluate scalability, with special focus on memory usage, parallelization, computing marginals, and current state of the art. For these experiments, we use the dataset Adult [6] and two datasets derived from MIMIC-IV [9]. Adult is a 55k row table containing US census data. Then from MIMIC-IV, we derive an Admissions dataset, which is the combination of MIMIC's Admissions and Patients tables, and the ICU Charts dataset, which combines the ICU chartevents, ICU stays and Patients tables. To obtain a challenging large dataset with ca. 1B rows, we duplicate the ICU Charts dataset rows three times and shuffle. After preparing the datasets, we obtain the statistics sketched in Table 1.

State-of-the-art PGM algorithms for data synthesis typically require input data in a single table with categorical attributes only. Therefore, we then transform numerical attributes and dates into categorical ones (shown under Categorical), by binning the numerical attributes and extracting information from dates (e.g., season, time of day). Scripts for creating these datasets from the original artifacts are available with the Pasteur source code online at `https://github.com/pasteur-dev/pasteur`.

Regarding the experimental environment, we run the in-memory marginal experiment on an AMD EPYC VM with a T4 GPU (as JAX requires a GPU), using a single core. We run the parallelization comparison with Dask on the same machine, using 20 virtual cores and 64GB, to study the behavior of the systems under increased memory pressure. For the rest of the experiments, we use an AMD EPYC 16c/32t 256 GB RAM bare-metal server with a SATA SSD. Both machines use Ubuntu 20.04.

Table 1: Datasets used in experiments

Dataset	Rows	Original Form: Date	Original Form: Number	Original Form: Discrete	Categorical Form: Discrete
Adult	50k	0	5	10	15
MIMIC IV Admissions	450k	4	0	10	19
MIMIC IV ICU Charts	$\leq$ 1B	4	1	5	15

5.1 Memory Representation

To evaluate memory consumption (Challenge 1), we use ICU Charts before (Original Form) and after discretizing all attributes (Categorical Form). For an in-memory representation, we use Python/Pandas because of its widespread use by most implementations of state-of-the-art synthesis algorithms. Since there is a need for swapping to disk, we compare popular file formats.

In Table 3, we see that using Pandas to load the data into main memory without providing specific type information (Naive) results in a very high memory consumption (777GB and 136GB respectively) because suboptimal formats are used (strings, float64, and int64). Both numbers are extrapolated from 15M sample. When applying the optimizations from Section 3.1 (Optimized, i.e., pruning non-learnable columns, dictionary encoding, shrinking the integer width), we considerably reduce memory footprint to 49GB and 24GB, respectively.

Then, we compare popular file formats for disk storage: CSV, CSV with GZIP compression, and Parquet. We report storage space on disk as well as the time to create those files (write time). We see that the most space-optimal storage format is Parquet, which requires only 19GB (12GB) instead of 197GB (41GB) in CSV format. CSV with GZIP achieves a similar compression ratio, with a consumption of 31GB (14GB). However, unlike Parquet, the time required for compressing and writing to disk is very high (3h and 2h respectively). Therefore, according to our measurements, Parquet files combine an efficient compression ratio with efficient disk writes. Moreover, when we partition the data, we can even parallelize the write process, achieving 1m to write 19GB with 60 partitions and 16 hyper-threaded cores (w is 32).

5.2 Parallelization Approach

To evaluate parallelization, we compare Pasteur against an in-memory baseline with Pandas (no parallelization) and Dask [5], a state-of-the-art distributed replacement for Pandas, shown in Figure 3.

We use the transformation that loads the ICU Charts dataset, converts it from the Original format to Categorical, and saves it. Our approach is intertwined with our system, so we create a micro-benchmark with Pandas that does a similar transformation, such that at 1M rows it has a similar footprint, where we have a 10 % slowdown over the Pandas baseline while using Pandas, for fairness. Then, as Dask is a replacement for Pandas, we swap it out.

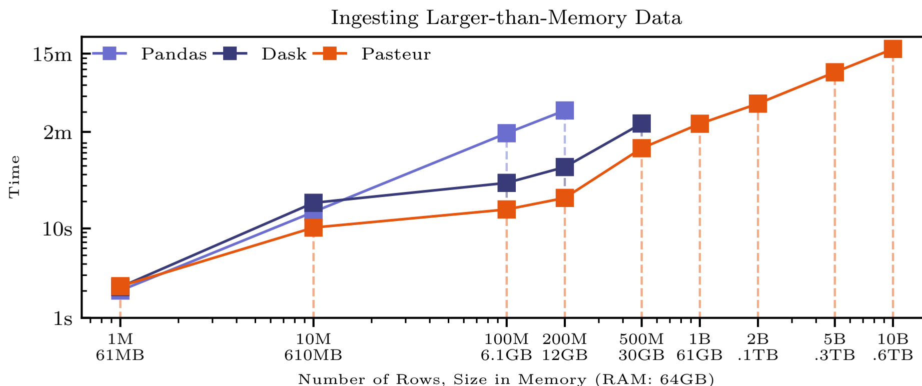

Fig. 3: Comparisons of Parallelization Approaches

Results. Pandas is a single-core in-memory baseline that scales linearly up to 200M rows (12 GB) in memory and then runs out of memory, as it requires a certain overhead for data processing. Dask performs better and can run partially on larger-than-memory data, as processing 500M rows (30 GB) required more than 64 GB of RAM for Pandas. However, when the dataset is larger than memory, Dask creates memory spikes that cause it to crash. Moreover, its eager spill-to-disk query architecture introduces overhead over Pandas (at 10M rows, it is 1.5x slower) and it only starts to parallelize effectively only above 10M rows.

Our approach uses a custom partition size and begins to parallelize before 10M rows. From 10M-200M rows, there is not enough work to parallelize across 40 workers which causes execution time to flatten out. Since our approach does not introduce overhead over Pandas, it is 1.5x faster than Dask from 10M to 500M rows. Following, it continues to parallelize linearly regardless of the dataset size. We stop the experiment after 10B rows, which have a memory size of 600 GB, a magnitude larger than system memory.

5.3 Marginal Calculation

To evaluate our marginal approach, we use a collection of marginals collected while running PGMs on the ICU Charts dataset. We examine the influence of the number of rows and the marginal domain size $|E'|$ on runtime. In Table 2, we report the average time to calculate a marginal with one core in memory across several implementations. We consider the following baselines. Two baselines are based on popular frameworks: one with Pandas' value_counts [15] and one with NumPy's histogramdd [3, 12, 13]. We built baselines that exploit the vectorized approach (as presented in Section 4) using two standard libraries: NumPy and JAX [2] – both environments lack support of operator fusing. For JAX, we report results of two versions: CPU and GPU. We also implement a baseline for marginal calculation in C using a standard for loop without SIMD. The last columns present Pasteur, which includes the Operator Fusing SIMD approach presented in Section 4.1 in full, using the AVX2 instruction set, both for 16 bit

Table 2: Marginal Calculation Times

		Arbitrary (SOTA)		Vectorized				Pasteur	
		Pandas	**NumPy**		JAX		C-based	**AVX2 (CPU)**	
Rows	$\|E'\|$	vc[1]	**hdd**[2]	Custom	CPU	GPU		uint32	**uint16**
20k	2^{12}	2.39 ms	2.99 ms	99 µs	211 µs	312 µs	118 µs	27.9 µs	**18.7 µs**
500k	2^{12}	51.9 ms	76.2 ms	1.65 ms	4.13 ms	538 µs	2.76 ms	613 µs	**383 µs**
1M	2^{12}	87.0 ms	144 ms	4.02 ms	8.33 ms	736 µs	5.47 ms	1.20 ms	**742 µs**
2M	2^{12}	167 ms	291 ms	10.3 ms	16.8 ms	1.07 ms	10.7 ms	2.40 ms	**1.55 ms**
5M	2^{12}	424 ms	716 ms	28.9 ms	48.6 ms	1.37 ms	26.9 ms	6.15 ms	**4.03 ms**
10M	2^{12}	785 ms	1.44 s	66 ms	94.2 ms	3.77 ms	53.8 ms	12.2 ms	**7.77 ms**
5M	2^{8}	313 ms	449 ms	20.8 ms	47.4 ms	2.68 ms	19.5 ms	5.11 ms	**3.29 ms**
5M	2^{12}	412 ms	727 ms	47.6 ms	6.19 ms	3.91 ms	26.9 ms	6.19 ms	**3.91 ms**
5M	2^{16}	556 ms	956 ms	40.2 ms	47.5 ms	2.21 ms	32.3 ms	11.3 ms	**8.15 ms**
5M	2^{20}	618 ms	1.24 s	71.9 ms	41.9 ms	2.2 ms	38.1 ms	**20.9 ms**	-
5M	2^{24}	642 ms	OVF[3]	79.1 ms	OVF[1]	OVF[1]	42.5 ms	**21.8 ms**	-
5M	2^{28}	966 ms	OVF[3]	127 ms	OVF[1]	OVF[1]	107 ms	**61.2 ms**	-

[1]Pandas value_counts function, [2]NumPy histogramdd function

[3]Memory Overflow (allocates 300GB-1.3TB of heap)

and 32 bit marginal domain sizes $|E'|$. The best CPU results are bolded and the best GPU results are underlined.

The results show that our proposed algorithm in 16 bits achieves the best performance, with up to 150x speedup in comparison to state-of-the art baseline implementations. In addition, by extending it to 32 bits, given a 1.5x slowdown, it can handle occurances with $|E'|$ larger than 16 bits. The vectorization approach shows with a speed-up of 20x on average in comparison to histogramdd, due to lower overhead. However, without operator fusing, it does not achieve the results of Pasteur. The JAX implementation using a GPU shows the fastest runtime. However, in the context of using a full GPU and being limited to its VRAM, compared to using a single CPU core, it has a higher cost.

Parallelization. We evaluate parallelizing the Operator Fusing SIMD approach using the three memory strategies (Shared Memory, Unique Copy, Unique Partition) proposed in Section 4.2, under 32 logical cores (i.e. 16 hyper-threaded cores). Table 4 reports the results for varying numbers of input rows; showing marginal throughput per second as well as relative speed-up when running on 32 logical cores versus a single core. If it is not feasible to use one of the approaches, we put "-" in the table. Specifically, Unique Partition is not applicable for low row numbers because there are not enough for multiple partitions and Unique copy requires a copy of the dataset for each core, and there is not enough memory for it beyond 50M rows.

These results confirm our theoretical analysis. Shared Memory works well for smaller numbers of rows, whereas for higher numbers ($> 1M$), after the

	ICU Charts Dataset (1B)			
	Original		Categorical	
Format	Space	t^1	Space	t^1
Mem Naive	777 GB	-	136 GB	-
Optimized	49 GB	-	24 GB	-
File .csv	197 GB	17m	41 GB	11m
.csv.gz	31 GB	3h	14 GB	2h
.pq	19 GB	10m	12 GB	8m
.pq partition	19 GB	1m	12 GB	1m

[1]Save (write) time

Table 3: Memory Format performance

		Parallelized 16c/32t					
Rows	Single Core	Shared Memory		Unique Copy		Unique Partition	
20k[1]	16.8k	**27k**	**2x**	26.6k	1.6x	-	
100k[1]	7.1k	**23k**	**3x**	23.3k	3.3x	-	
500k	1.6k	**14k**	**9x**	13.2k	8.3x	-	
1M	754	**9k**	**12x**	8.9k	12x	-	
5M	160	935	5.8x	**2.3k**	**14x**	-	
10M	81	413	5.0x	**1.2k**	**15x**	470	6x
50M	16.0	122	7.6x	**223**	**14x**	179	11x
100M	8.0	98.6	12x	-		**104**	**13x**
500M	1.6	18.6	12x	-		**25**	**16x**
1B	0.8	10.1	13x	-		**12**	**16x**

[1]Single thread orchestration limit

Table 4: Parallelization Strategies (rows/s)

dataset stops fitting in the CPU cache, total memory bandwidth is limited. By giving each core its own copy of the dataset, Unique Copy maximizes memory bandwidth and is faster. Due to the additional memory required, it can only be applied up to 50M rows. Afterwards, Unique Partition performs well, as the dataset size increases to amortize the cost of merging the partition marginals. By varying the approach based on dataset size (bolded results) it is possible to achieve linear parallelization (13x-16x for 16 hyper-threaded cores) for tables beyond 5M rows with great parallelization down to 500k rows. If the dataset is trivial ($\leq$ 100k rows), the single thread that orchestrates the marginals and distributes them to cores becomes the overhead.

5.4 Synthesis Pipelines

With all system components in place, we perform an experiment with end-to-end synthesis pipelines while using three state-of-the-art PGM algorithms: AIM [13], MST [12], PrivBayes [22] and three datasets (Adult, Admissions, ICU Charts). For the ICU Charts 3 dataset, we evaluate it by subsampling it with different row counts and use 60 partitions starting at 100M rows. In other datasets, we use a single partition. For all algorithms, we use hyper-parameters recommended by their authors (e.g., AIM uses all two-way column combinations as a workload) and only vary the privacy budget ϵ.

A pipeline in this experiment consists of three parts: pre-ingestion (transform raw data into a set of tables in Parquet format that are readily available), ingestion (create the dataset table and convert it to categorical), and synthesis (fit, sample the PGM algorithm, and evaluate the resulting data). For evaluation metrics, as a dummy workload, we calculate per-column histograms, KL divergence for each column pair, and χ^2 tests for each column. These are paral-

Table 5: Performance of the Synthesis Pipeline

			Ingest		PrivBayes		MST		AIM		**PrivBayesHV**	
Dataset	Rows	ϵ	**pre**	**data**	time	N	time	N	time	N	**time**	**N**
Adult	50k	1	**1m**	**1m**	1m	168	3m	120	13m	120	**1m**	**10.5k**
MIMIC												
Admit.	550k	1	**16m**	**30s**	1m	8k	5m	210	22m	210	**9m**	**1.3M**
ICU Charts	1M	1		**3m**	1m	257	4m	136	187m	136	**1m**	**11.3k**
	10M	0.1		**5m**	2m	302	5m	136	218m	136	**3m**	**17.7k**
	100M	0.01	**16m**	**6m**	6m	306	8m	136	9m	136	**11m**	**16.3k**
	500M	0.002		**13m**	17m	307	27m	136	23m	136	**37m**	**15.8k**
	1B	0.001		**21m**	30m	300	42m	136	30m	136	**72m**	**16.0k**
1 core → 500M				**139m**	180m	(11x)	197m	(7.3x)	145m	(6.2x)	**196m**	**(13.6x)**

lelizable and contribute to the pipeline runtime. Since our optimizations do not change the output of the PGM algorithms, we omit qualitative results.

To ensure comparability of the evaluation results between different row counts on the ICU Charts dataset, we keep the ratio $\epsilon \cdot N$ constant, where N is the row number of the table and ϵ is the privacy budget hyperparameter of differential privacy, the mechanism used to ensure privacy. This is because the complexity of all three surveyed algorithms scales with that ratio. PrivBayes uses noise reversly proportional to ϵ and keeps the signal-to-noise ratio of marginals constant, where the signal is proportional to N. AIM and MST instead work by adding noise to denormalized marginals (sum to N) in reverse proportion to ϵ; where a larger ϵ equates to less noise and a larger signal-to-noise ratio. Therefore, when we vary the number of rows in the ICU Charts dataset, we update ϵ such that $\epsilon \cdot N$ remains constant.

Our evaluation results are illustrated in Table 5, which shows the execution time in minutes and the number of calculated marginals. The runtime of the approaches increases sub-linearly with the growing number of rows, e.g., AIM requires 30 minutes for 1B rows and 23 minutes for 500M rows. The results meet our expectations; as other than marginals, we do not parallelize the algorithms themselves, which have a fixed runtime. There is a small exception with AIM using the ICU Charts dataset at 1M and 10M rows. The ICU Charts dataset is noisy by its nature and especially with a low row count, it causes AIM, an iterative algorithm, to try to overfit it and repeatedly perform measurements (we re-ran the experiment multiple times with different dataset samples and obtained similar results).

Finally, we evaluate the influence of parallelization, by executing the ICU Charts dataset with 500M rows using a single core and compare it to multicore. In dataset ingestion, the peak RAM use is 70 GB for multicore (32 logical cores) and 25 GB for single core. For the rest of the steps, the peak RAM use is 85

GB for multicore and 10 GB for single core (all algorithms). PrivBayes has great parallelization for multicore (11-14x) as it contains a small fixed overhead. AIM and MST have a larger fixed overhead per execution, where it is large enough to result in non-linear parallelization (7.3x, 6.2x respectively).

6 Related Work

Synthesis Algorithms. Data Synthesis algorithms fit into two broad categories: Neural Network (NN) and Marginal based. Notable Neural Network algorithms are PATE-GAN [10], CT-GAN [19], DP-GAN [8], and ADS-GAN [20]. For tabular data, a class of Probabilistic Graphical Model (PGM) based algorithms has been shown perform well when privacy is required [18]. Notable algorithms are: MST [12], PrivBayes [22], AIM [13], PrivMRF [3], which are enhanced through inference, shown in PrivPGM [14]. For privacy, the mechanism Differential Privacy [7] (DP) provides strong guarantees.
Data Synthesis Systems. Synthcity [17] and Synthetic Data Vault (SDV) [16] are collections of algorithms and evaluation metrics for data synthesis. Neither attempts to perform scaling nor larger-than-memory synthesis.
Machine Learning Orchestration. In Machine Learning Operations (MLOps), multiple systems have been proposed to parallelize machine learning workflows using DAG pipelines in an inter-task context: Prefect, Kubeflow, Apache Airflow, Luigi, and Kedro [1]. For intra-task parallelization, Dask [5] and Spark [21] have been proposed to parallelize computations themselves. The previous solutions focus on cluster environment scaling, causing a performance overhead (shown in Section 5) when the data fits in a single server. Our approach is tailored to a single server environment, which is where a large segment of medical data processing occurs due to GDPR limitations. In those environments, the low overhead of our approach (launching processes versus containers or cross-server communication) makes it optimal.

7 Conclusion

Privacy-aware data synthesis has become an important research area in fields such as Medicine. Yet available systems and frameworks suffer from limited scalability and do not scale to the current needs of real-world medical applications and datasets. Hence, in this paper, we proposed, Pasteur, a system that achieves linear scaling and stable memory usage, by employing parallelization, optimized data representations, and an optimized method of computing marginals (the core bottleneck in PGM algorithms). Our experiments show that Pasteur supports data sets of up to 1 billion on a single CPU in the period of one hour.

Acknowledgements. This project received funding from the European Union's Horizon 2020 research and innovation programme under Marie Skłodowska-Curie (grant No 955895), the Poul Due Jensens Fond (Grundfos Foundation), and the Novo Nordisk Foundation (grant number NNF23OC0083510).

References

1. Alam, S., Chan, N.L., Comym, G., Dada, Y., Danov, I., Datta, D., DeBold, T., Hoang, L., Holzer, J., Kanchwala, R., Katiyar, A., Koh, A., Mackay, A., Merali, A., Milne, A., Nechevska, C., Nguyen, H., Okwa, N., Schwarzmann, J., Stichbury, J., Theisen, M.: Kedro (12 2022)
2. Bradbury, J., Frostig, R., Hawkins, P., Johnson, M.J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., Zhang, Q.: JAX: composable transformations of Python+NumPy programs (2018)
3. Cai, K., Wei, J., Lei, X., Xiao, X.: Data Synthesis via Differentially Private Markov Random Fields. VLDB (2021)
4. Cai, K., Xiao, X., Cormode, G.: Privlava: Synthesizing relational data with foreign keys under differential privacy. Proc. ACM Manag. Data **1**(2), 142:1–142:25 (2023)
5. Dask Development Team: Dask: Library for dynamic task scheduling (2016)
6. Dua, D., Graff, C.: UCI machine learning repository (2017)
7. Dwork, C.: Differential privacy (2006)
8. Ho, S., Qu, Y., Gu, B., Gao, L., Li, J., Xiang, Y.: DP-GAN: Differentially private consecutive data publishing using generative adversarial nets (2021)
9. Johnson, A., Bulgarelli, L., Pollard, T., Horng, S., Celi, L.A., Mark, R.: MIMIC-IV
10. Jordon, J., Yoon, J., van der Schaar, M.: PATE-GAN: Generating Synthetic Data with Differential Privacy Guarantees (Sep 2018)
11. Liu, B., Ding, M., Shaham, S., Rahayu, W., Farokhi, F., Lin, Z.: When machine learning meets privacy: A survey and outlook. ACM Comput. Surv. (2022)
12. McKenna, R., Miklau, G., Sheldon, D.: Winning the NIST Contest: A scalable and general approach to differentially private synthetic data (Aug 2021)
13. McKenna, R., Mullins, B., Sheldon, D., Miklau, G.: AIM: An Adaptive and Iterative Mechanism for Differentially Private Synthetic Data (Jan 2022)
14. Mckenna, R., Sheldon, D., Miklau, G.: Graphical-model based estimation and inference for differential privacy. In: Proceedings of the 36th International Conference on Machine Learning. pp. 4435–4444. PMLR (May 2019)
15. Wes McKinney: Data Structures for Statistical Computing in Python. In: Stéfan van der Walt, Jarrod Millman (eds.) Proceedings of the 9th Python in Science Conference. pp. 56 – 61 (2010)
16. Patki, N., Wedge, R., Veeramachaneni, K.: The Synthetic Data Vault (Oct 2016)
17. Qian, Z., Cebere, B.C., van der Schaar, M.: Synthcity: facilitating innovative use cases of synthetic data in different data modalities (2023)
18. Tao, Y., McKenna, R., Hay, M., Machanavajjhala, A., Miklau, G.: Benchmarking Differentially Private Synthetic Data Generation Algorithms (Dec 2021)
19. Xu, L., Skoularidou, M., Cuesta-Infante, A., Veeramachaneni, K.: Modeling Tabular data using Conditional GAN (2019)
20. Yoon, J., Drumright, L.N., van der Schaar, M.: Anonymization Through Data Synthesis Using Generative Adversarial Networks (ADS-GAN). IEEE Journal of Biomedical and Health Informatics **24**(8), 2378–2388 (Aug 2020)
21. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: a unified engine for big data processing (2016)
22. Zhang, J., Cormode, G., Procopiuc, C.M., Srivastava, D., Xiao, X.: PrivBayes: Private Data Release via Bayesian Networks. ACM Transactions on Database Systems **42**(4), 25:1–25:41 (Oct 2017)
23. Zhang, Z., Wang, T., Li, N., Honorio, J., Backes, M., He, S., Chen, J., Zhang, Y.: PrivSyn: Differentially Private Data Synthesis