Pasteur: Scaling Private Data Synthesis to 1 Billion Rows

Antheas Kapenekakis
Aalborg University
antheas@cs.aau.dk

Daniele Dell'Aglio
Aalborg University
dade@cs.aau.dk

Minos Garofalakis
Athena Research Center
minos@athenarc.gr

Katja Hose
Aalborg University
khose@cs.aau.dk

ABSTRACT

As data synthesis becomes applicable to more research domains (medical, financial) and applications (dataset surrogates, privatized releases), new challenges arise, in terms of data complexity, evaluation, and reproducibility. Applications need to combine thousands of rows, spread across dozens of tables, with columns requiring machine learning models to parse domain knowledge. In such scenarios, preprocessing and encoding data in an ad-hoc style becomes intractable and suffers from low reproducibility. Furthermore, the amount of data easily becomes too large to fit into memory, and the large number of processing steps requires parallelization. Finally, proper data evaluation and other downstream tasks require a decoded version of the data, with column encodings that are fully reversible and additional steps to the synthesis process. In summary, this rise in complexity alludes to the need for a regimented process. In this paper, we propose an all-inclusive methodology for data synthesis, from raw data to evaluation, and implement it in a system, named Pasteur. Pasteur is modular, allowing the addition of new datasets, algorithms, and transformations, while parallelizing and scaling to larger-than-memory datasets. To showcase Pasteur's capabilities, we synthesize and evaluate a dataset of 1 billion rows (200 GB) with differential privacy in 1 hour, starting from raw data.

1 INTRODUCTION

Data synthesis is an emerging field in data science, with the aim of promoting responsible data sharing through anonymization of data [13], and correcting (sampling and societal) bias through the reshaping of datasets [10]. Up to this point, data synthesis research has focused on tabular datasets fitting in memory (up to 1 million rows, < 100 MB [2, 15, 16, 23, 25]). However, most of the interesting synthesis topics (multi-modal, hierarchical datasets and medical data) raise new scalability, reproducibility, and performance challenges, due to the larger amount and complexity of the data.

Current approaches for data synthesis roughly work as follows. For analyzing tabular datasets of less than 10 million rows and 5 GB in size, it is natural to load them in-memory and to preprocess and encode them by hand in the beginning of a project. Preprocessing involves converting the data to a set of categorical and numerical columns [8]. Then for encoding, depending on the approach, either numerical columns are discretized [2, 6, 14, 25], or all columns are converted to floating point, one-hot values [12, 22]. Furthermore, it is generally tractable to communicate the preprocessing and encoding through text or by providing the code that was used.

For evaluation, a set of metrics (appropriate for synthesis) is selected, and some code needed to selectively decode part of the data is written to the extent required. For example, to measure classifier performance on synthetic data created by a neural network algorithm, the target column can be converted from one-hot to categorical, leaving the rest as is. As long as the ideal encoding of columns is obvious (e.g., categorical columns have less than 100 unique values and numerical columns follow a pattern), this approach is simple and fast.

However, even in an ideal scenario, the sketched approach has limitations. Preprocessing and encoding data once is often the result of using a well-behaved curated dataset, such as the US Adult income dataset [3], which limits the algorithm's application to real-world data. To reproduce the results of a paper, the data needs to be encoded manually following the instructions in the paper or code samples, which might be inadequate or partial. Since the data is encoded in a specific way, it is easier to include metrics based on that encoding, and omit others, which would require additional effort in cross-encoding the result (e.g., numerical columns that are discretized are not converted to numerical values prior to evaluation). Finally, cross-comparison between algorithms that use different encodings requires significant additional effort, since the data has to be re-encoded and re-measured for each algorithm and dataset.

Scaling limitations – in terms of rows. The current approach becomes less viable in the presence of data that is not preprocessed, that is magnitudes larger, and that features multiple tables. Consider, for example, the MIMIC-IV dataset [11]; MIMIC contains 27 tables, which are interlinked, and requires special handling for loading certain tables into RAM (due to their sizes). MIMIC consists of a set of compressed CSVs (*.csv.gz*). Without type information, the largest table in MIMIC requires more than 256 GB of RAM. With type information, loading it requires half an hour. By working directly on raw input and output data (such as CSV), the synthesis process incurs performance penalties for preprocessing and loading that become significant for large datasets.

Scaling limitations – in terms of domain size. MIMIC features columns with disease and item codes that contain more than 50k unique values and timestamp values with minute accuracy and a range of years. When naively synthesizing those values, the resulting synthetic data becomes noise. The data requires expert domain knowledge to parse: disease codes fall within a human-made hierarchy, and timestamps need to be interpreted with respect to patient age, seasonality, day-of-week, time-of-day, etc. Furthermore, even with sufficient domain knowledge being available, we have two challenges arise. Given that the raw data is overtly complex to synthesize, to what point should it be simplified prior to synthesis? Given a level of simplification, how should the data be encoded to correctly transfer domain knowledge to the synthesis algorithm? Answering these questions requires multiple iterations of performing synthesis and evaluation on the resulting data.

Scaling limitations – in terms of encodings. Parsing and embedding expert knowledge into a column so that it can be encoded for input into a synthesis algorithm (such as encoding dates and disease codes) requires significant effort and testing. Without an abstraction layer, this effort has to be performed anew for each algorithm and dataset combination. For a proper evaluation of the synthetic data, the encoded output has to be decoded back to the original format,

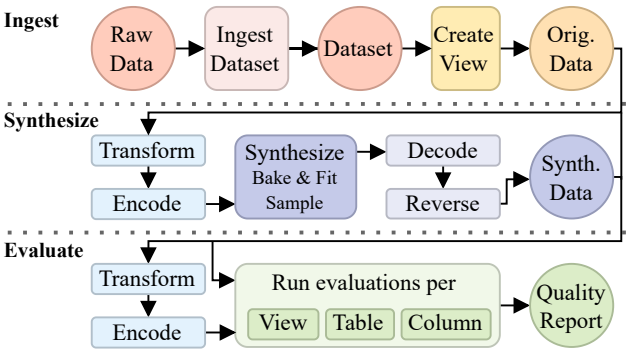

Figure 1: Pasteur methodology overview (see figure 2).

with additional overhead. While this issue can be avoided by preprocessing datasets into a set of categorical (fixed cardinality integers) and numerical (floating point) values, this lowers the utility of the resulting data and constraints present in the original data (such as 2 values originating from the same column becoming null at once) are hidden during evaluation. Synthetic data algorithms should be evaluated on the data in the form it is used by end-consumers, for synthetic data generation to become a mainstream process, and not after being preprocessed and featurized.

Scaling limitations – in terms of evaluation. While measuring the performance of a machine learning model using a set of key metrics is a relatively well-defined process (accuracy, precision and recall, AUROC, F1 score, latency, etc.), this is not the case if synthetic data is involved. The performance of synthetic data varies depending on the use case, each requiring different evaluation criteria [8]. For example, when providing a dataset surrogate in lieu of private data for a paper, the evaluation criteria would be the downstream analysis or model having identical results. For a dataset release that would be used for analyses by end users, performance on general queries, which can be simulated using 1–3 way distributions, would be measured. Without a foundation of the structure of metrics and how they should access data [8], solutions become specific to dataset and use cases, and results between papers are not comparable.

1.1 Contributions

An end-to-end methodology. To handle the complexity of the synthesis and evaluation process, we propose an end-to-end synthesis methodology (Figure 1), which begins with raw data and ends with evaluation. This methodology has three stages: ingestion, synthesis, and evaluation. The ingestion stage types and simplifies raw data so that it is fit for synthesis and splits it into two sets: one used for synthesis (work), and a hold-out used for evaluation (reference). Afterwards, the synthesis stage handles encoding and producing a synthetic version of the work set. Lastly, the evaluation stage compares the work, reference, and synthetic sets with each other in a suite of metrics, which we classify based on how they parse data into column, table, and view metrics.

The system Pasteur. We implement this methodology in a system we name Pasteur. Pasteur is designed for modularity and reproducibility: all parts of the methodology are implemented and are separated into extendable and hot-swappable modules. To ensure interoperability, we model the interactions between the modules. As a result, Pasteur enables the addition of new datasets, algorithms, and evaluation metrics, independently of each other. Since the ingestion process is codified, support for datasets can be added in Pasteur without requiring a license from data owners: users selectively obtain licenses and artifacts for datasets they are interested.

End-to-end scalability. All parts of Pasteur are designed for handling larger-than-memory datasets and linear parallelization. Pasteur uses a Directed Acyclic Graph (DAG) dependency-based pipeline to execute methodology steps in parallel. Furthermore, Pasteur partitions large datasets and schedules tasks per partition, through a common task queue. By controlling the number of parallel tasks and partition size, Pasteur maximizes CPU utilization while adapting to available system memory.

A range of synthesis algorithms [14, 15, 24, 25] depends on marginal calculations, which we found to become a bottleneck when working with large tables. For this reason, we developed a marginal calculation method that is supported by a C-based AVX2 algorithm and a variable memory allocation strategy for parallelizing. We achieve a speedup of 150x in single core compared to current implementations, while parallelizing linearly to 2000x with 16 cores (15x), with a fixed penalty for larger-than-memory datasets (35s for a table of 1 billion rows) per set of marginals.

We evaluate the performance of our system by performing synthesis on a larger-than-memory table (1 billion rows, 65 GB in-memory, 200 GB CSV). Pasteur synthesizes the table using 10 GB of RAM with a single core and scales row count dependent tasks linearly for an overall performance gain of 10x-14x and 75 GB of RAM for 16 cores/32 threads and 32 simultaneous tasks.

2 METHODOLOGY

The methodology (overview—Figure 1, expanded—Figure 2) consists of three main stages: ingestion, synthesis, and evaluation. We expand those stages into distinct processes. Ingestion expands to: dataset creation and view creation. Synthesis to: transformation, encoding, and synthesis. Evaluation expands differently depending on the metric, which can fit to the whole data at once, once per table, or once per column. All steps of the methodology should produce deterministic results. This allows for cross-machine comparisons and sharing intermediary results between executions. In Figure 2, we separate artifacts based their dependencies.

2.1 Dataset Creation

Raw data can consist of various formats: SQL queries, compressed archives (*.tar.gz*, *.zip*, *.rar*), CSV Files (*.csv*, *.csv.gz*), JSON files (*.json*), etc. All these formats share a common attribute: for large datasets, using them to load and save data becomes a bottleneck in the synthesis process. Loading data from those formats can require hours or, if the data is offsite (such as in a SQL server), upwards of days. Furthermore, raw data may not follow required naming conventions, have an appropriate structure, or can contain extraneous information which has to be removed prior to working with it.

The first step in our methodology is defining an intermediate structure that represents raw data in a form more amenable to processing, through typing the input data and packaging it in a

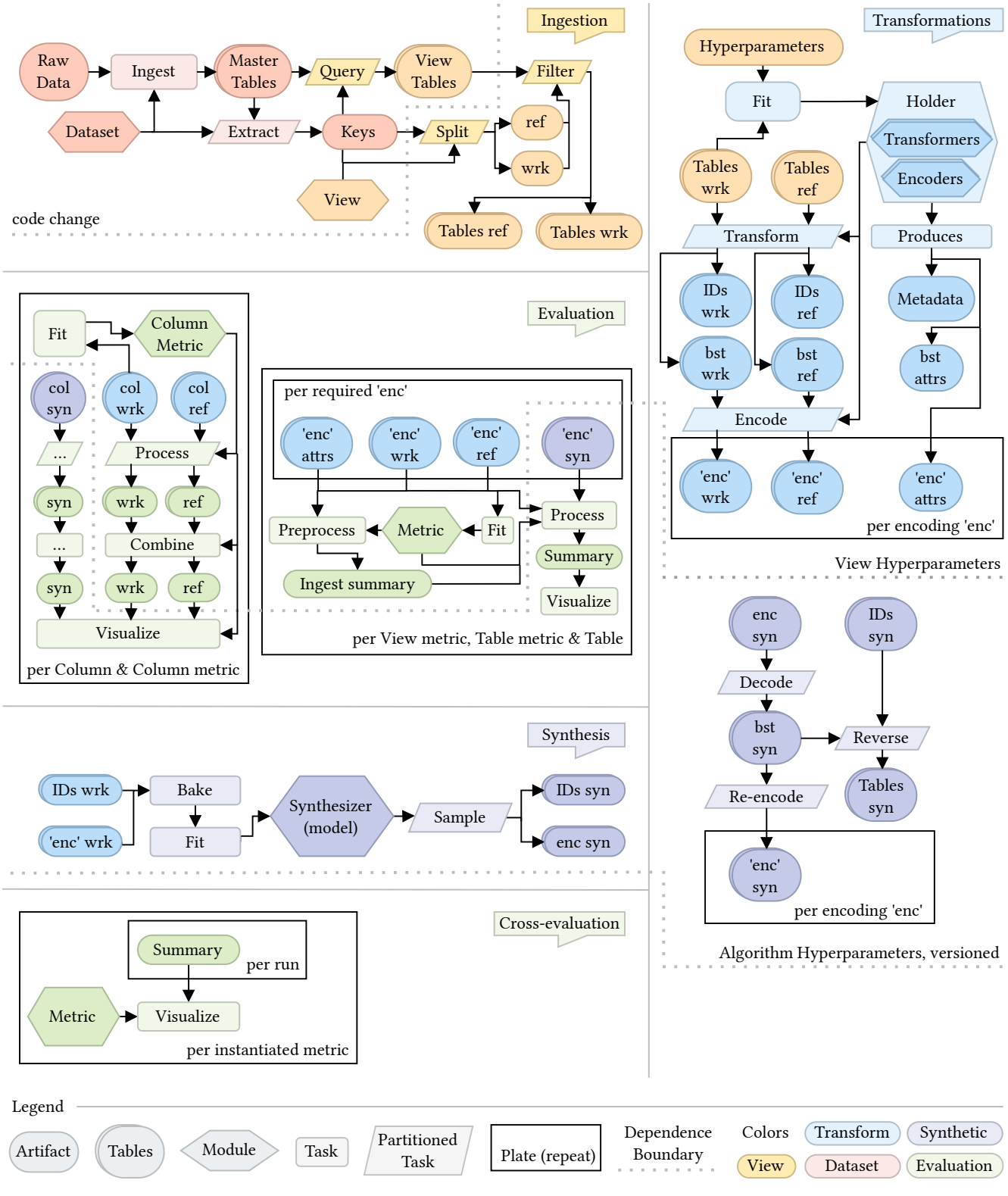

Figure 2: The individual segments of the Pasteur methodology.

performance optimized file format, with a verbose structure based on a set of guidelines that make documentation possible (harmonizing the structure of multi-year data, dropping extraneous columns, renaming columns and tables). We name this structure a Dataset.

A Dataset consists of set of typed Master Tables (2D matrices of arbitrary objects). In addition, it contains information about a kind of real world entity (patients, companies, customers, warehouses). To aid in evaluation, the Dataset contains a set of Keys that identify its entities, so that it can be partitioned.
The definition of a Dataset consists of:

- A list of raw data sources and how to load them.
- For each Master Table, an Ingest function and its raw data dependencies that can be used to compute the table.
- An Extraction function with its Master Table dependencies, which can be used to produce the Dataset's Keys.

The Ingest function's role is to type the input data by using more efficient data types, and to perform changes required to conform to the set of required guidelines. Since a Dataset is defined by code, its artifacts can be reused for a certain code version and data sources.

For example, the US Adult dataset is composed of two raw tables: *train* and *test*, and the *test* table appends a period to the income column categories. The Ingest function of Adult merges the raw tables *train* and *test*, harmonizes their income columns, and creates a new primary key, to form a single Master Table. We extract the Master Table primary key to act as the Dataset's Keys. Ingesting MIMIC is simpler, it consists typing its 27 tables and extracting the *subject_id* from the Patients table to use as the Dataset's Keys.

2.2 View Creation

If the raw sources of a Dataset have not preprocessed, it will feature an amount of detail that is intractable to synthesize (large number of tables, columns, relationships, and columns with large domain sizes). For synthesis, a partial and simplified version of the Dataset is required. Depending on the goals of synthesis, this version can be expressed be in terms of down-sampled columns, in portions of tables, or in a smaller number of rows. In addition, for testing synthetic algorithms, there might be a need to experiment with exaggerated versions of datasets to strain synthesis algorithms.

For these reasons, we define a structure we name a View (from the term database view). A View represents a simplified version of a Dataset, in the form that will be synthesized. Each View is based on a Dataset and contains a set of Tables. To aid in evaluation, a View can partition its Dataset's keys and based on those, its Tables.
The definition of a View consists of:

- For each Table, a Query function with its Master Table dependencies that is used to compute the Table.
- A Split function that partitions the Dataset's Keys.
- For each Table, a Filtering function, which filters the table based on the Key partitions.
- View Hyperparameters, to control splitting and transformation, and sets of Algorithm Hyperparameters, which control synthesis algorithms (ex. learning rate, privacy budget).

The Query functions should be deterministic, so View Tables can be reused until a code change. The Split function receives hyperparameters which define how it should portion data, so downstream artifacts are View hyperparameter specific. For evaluation, we define two sets of data: work and reference, and the Split and Filter functions are responsible for creating those sets. Work and Reference sets are a disambiguated version of the "train" and "test" sets used in traditional machine learning.

2.3 Transformation and Encoding

In current works, raw data is preprocessed in a process external to synthesis to a set of categorical and numerical columns [8]. Categorical columns correspond to a set of unique values with a fixed cardinality, and Numerical columns into floating point or integer values. Given this format, encoding is a rudimentary process that broadly depends on the class of algorithm. For example, when using an algorithm with discrete inputs, categorical columns are converted to integers. However, if we omit part of the initial simplification, this process breaks down. It makes it intractable to synthesize more complex inputs, such as dates or disease codes. There does not exist a standard way to encode these values, and they are influenced by domain knowledge: dates might be relative to others (e.g., discharge time is relative to admit time) and disease codes have high cardinality while respecting a human made hierarchy.

To allow handling more complex column types while integrating domain knowledge, we expand the encoding process into Transformation and Encoding. Transformation reversibly converts complex types (e.g., dates) into simple ones (numerical and categorical values) while integrating domain knowledge. Following, Encoding performs the processing required for a specific algorithm in hand in a reversible manner (e.g., discretizing for marginal algorithms and converting to one-hot for neural network algorithms). We standardize the layer in-between to a format we name the Base Transform Layer. This abstraction layer allows adding support for new column types to all algorithms at once and support for new algorithms to all column types at once.
Base Transform Layer. The abstraction of Base Transform Layer consists of two structures: the Content Table and Attributes (a metadata structure describing the table). Each initial Table column is mapped to an Attribute, which encapsulates a set of Values and a set of common conditions (e.g., all Values have to be null together).

Values represent columns in the Content Table. A Value can be Categorical or Numerical. Numerical Values are stored as floating point values and their metadata consists of their bins for when they are discretized. This is important: discretized columns in evaluation have to equal the ones in encoding. Example:

$$V_{\text{age}} = \{[0, 20), [20, 40), [40, 60), [60, 80), [80, 100]\} \quad (1)$$

Categorical Values are stored as unsigned integers. Their metadata structure consists of a Stratification (based on hierarchical values [24]). A Stratification is a tree where leafs correspond to unique values and nodes to value groupings. We specify two types of groupings: Categorical (python set syntax {}) and Ordinal (python array syntax []). Below, we have a nullable discretized age value and a disease value. The Stratification captures the ordinal nature of the age buckets, while preserving that the nullable value is separate. Likewise, it preserves that leukemia and melanoma are closer than fracture, but contain no ordinal nature.

$$S_{\text{age}} = \{\text{None}, [[0, 25), [25, 50), [50, 75), [75, 100]]\} \quad (2)$$

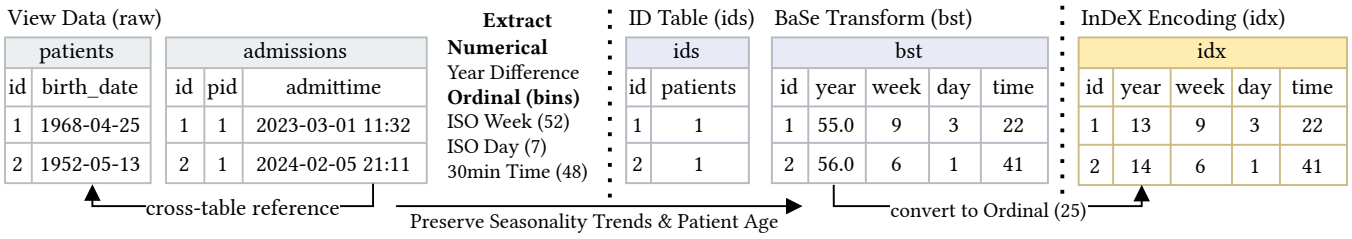

Figure 3: Transformation of the datetime column "admittime" of the table Admissions, preserving domain knowledge.

$$S_{\text{disease}} = \{\text{None}, \{\{\text{leukemia}, \text{melanoma}\}, \text{fracture}\}\} \quad (3)$$

For algorithms that use Categorical Values, a Stratification provides finer grained simplification rules: it is better to merge $[0, 25)$ and $[25, 50)$ rather than $[0, 25)$ and $[50, 75)$, None should be merged last. For neural network encodings, Stratifications reduce the dimensionality explosion of one-hot classification: each level in the tree can be encoded one-hot (for $\log n$), and ordinal groupings can be converted to floating point values.

Figure 3 showcases the reversible transformation and encoding of the column "admittime" by referencing the column "birth_date" into the Attribute "admittime" (certain mappings are not fully reversible; such as binning numbers and rounding times). This Attribute which holds the Values "year", "week", "day", "time". The Value "year" is set to be Numerical, to retain its full accuracy. The Values "week", "day", "time" are Stratified Values where all underlying values (ex. "10:30") are under an ordinal grouping. If the initial column of "admittime" was nullable, the attribute "admittime" could specify one common state for "null", where all Values would be expected to have the same value (0 or NaN). In this case, we would also modify the stratification for each value to hold the null value and the ordinal grouping under a categorical grouping, as we did for the S_{age} example in Equation 2.

The Attribute abstraction enables the decomposition of complex columns into independent random variables that share common

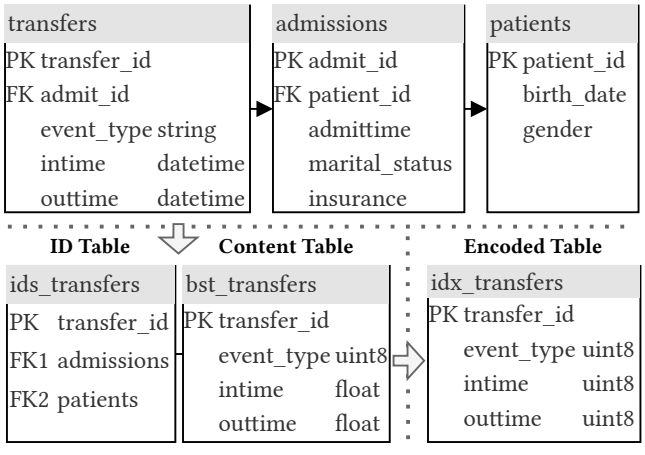

Figure 4: Transformation for the table Transfers of MIMIC.

conditions (such as being "null"). This allows informing the underlying synthesis algorithm at hand: for Neural Network algorithms, common conditions can be encoded as one-hot and shared between the Values, where if one of the common conditions is active, the Value indices are considered as Do Not Care.

Transformation & Encoding. Consider a View with N Tables, with a 1–M non-cyclic relationship to each other. Each Table consists of a set of columns, and a primary key we name its ID. We separate columns by referencing View hyperparameters into three types: content, ID, and unused. For each Table, the transformation process begins by recursively finding its parents based on ID column metadata and collecting the foreign key columns (IDs) that link them. We name the result the ID table: for each row in the original Table it contains the IDs that link it to all upstream parents.

The process continues by iterating over the content columns of the Table, transforming them based on View hyperparameters, and saving the results to a new table. A transformation may reference upstream columns in order to transform its column (such as for a relative date transform). If a reference column is on a parent table, we join the ID table with the parent table to fetch the appropriate values. The new table is named the Content Table and conforms to the Base Transform Layer format. We encode the Content Table into the Encoded Table using an encoding appropriate for the current algorithm at hand. For decoding, the process is performed in reverse, by decoding referenced columns first. We provide an example in Figure 4, where the Transfers table of a three Table View is transformed and encoded. Artifacts up to this point can be reused for a certain code version and View hyperparameters.

2.4 Synthesis

The end result of Transformation and Encoding is a set of tables with normalized information (ex. relative dates don't duplicate the year), integrated domain knowledge (through Stratification), and structure that is appropriate for input into a synthesis algorithm (through an appropriate Encoding). The design of a synthetic algorithm can then focus on ingesting and producing a set of high quality primitives (categorical, numerical values).

We separate the synthesis process into Bake, Fit, and Sample steps. Bake uses the provided data to create the graphical model skeleton without values. Fit samples the data to fill the graphical model. Either of the steps is optional (AIM [15] and MST [14] do not have a Fit step, Neural Network algorithms do not have a Bake step). Once the model is fit, it is used to sample an encoded version

of the data, which is then reverse-encoded and reverse-transformed. Artifacts post-synthesis depend on Algorithm Hyperparameters.

2.5 Evaluation

It is possible to condense the performance of a machine learning model to a small set of scalar numbers, which cover its accuracy and performance. Those numbers are generated from the training process itself, by fitting and scoring the model on the train and test sets. In data synthesis, this is not the case: each evaluation metric is calculated by executing arbitrary code on the resulting data and can produce dozens of numbers to be reviewed.

As an example, while the measure of data quality will vary depending on the intended use-case, we would, at minimum, require a measure of all 1-way and 2-way distributions to ensure all columns are synthesized correctly. For a 15 column table, this translates to 15 histograms and $N+N^2 = 240$ numbers. Conveying this information through the terminal or in a single results row is not possible, a multipage quality report is required. In addition, since this report is generated by executing arbitrary code, to ensure interoperability between Views, Algorithms, and different use-cases, we have to provide a formal definition of what a synthetic data metric is.

Required Data. We define metrics as having access to three sets of data: work, reference, and synthetic. The work and reference sets are independently sampled from original data at a preset ratio (ex. 80–20, 50–50). The work set is used to generate the synthetic set.

For metrics that compare synthetic to original data (ex. X^2, Kullback-Leibler divergence), the synthetic set and the reference set are both compared to the work set. The reference set acts as "faux" synthetic data to define the base similarity: synthetic data being more similar than reference data implies overfitting or a privacy leak, while less similar implies underfitting.

For metrics that compare the performance of synthetic data to real data, the work set acts as the real data, and the reference set acts as the real-world test set. For example, consider a classifier metric. We train two classifiers: one on the work set and one on the synthetic set. Performance on the reference set approximates behavior on real world data.

Metric Structure. We define metrics as being calculated in four steps: Fit, Preprocess (optional), Process and Visualize. In Fit, we use the work set and metadata to adjust the metric to the current View. The main step is Process, where a summary is computed from the work, reference and synthetic sets. The summary is a condensed version of the results which requires $O(1)$ memory compared to data size. Optionally, the Preprocess step is used to produce an ingest summary from work and reference, which can be re-used between synthesis executions in lieu of calculating it each time in Process. Finally, in the Visualize step, the summary is used to produce the quality report for the metric in a synthesis run.

To compare multiple synthesis executions, their summaries can be collected and combined in a Visualize step afterwards. Each metric's presentation is adjusted to account for multiple executions and the produced quality report displays a cross-comparison.

Different Types. We separate metrics based on their access patterns into three categories: Column, Table, and View. Table and View Metrics share working principles, with the difference being that Table metrics are performed once per table and View metrics once per synthesis. Table metrics only have access to a single table and its parents. Both can optionally use encoded data, so they can reuse the domain knowledge integrated into transformations. For example, classifier metrics receive both continuous and discrete encodings. Classifiers use a Value from the discrete encoding as their label and all continuous Attributes other than the one belonging to the label Value for input. Column metrics act similar to transforms, where we process each set independently into three summaries (work, reference, synthetic), which we merge into one combined summary. The full metric structure is shown in Figure 2.

3 SYSTEM DESIGN

In the previous section, we present an end-to-end methodology for data synthesis. In this section, we envelop this methodology in a system we name Pasteur and present its key architectural features.

3.1 Module System

Pasteur is designed to be a modular system. Dataset modules contain all the required information to create a Dataset. Likewise, View modules reference a Dataset and encapsulate all required information to create a View. The transformation and encoding process is managed by Transformer and Encoder modules. The Transformer modules are referenced in the system by the column type they manage and one is instantiated per Table column. The Encoder modules are referenced in the system based on the algorithm encoding they provide, and one is instantiated per Attribute in the Content Tables. Synthesizer modules encapsulate the Bake, Fit, and Sample steps of a synthetic data algorithm and store the model inside them. Finally, evaluation is provided by the modules Column Metric, Table Metric and View Metric. Figure 2 includes modules and how they encapsulate the methodology.

3.2 Partitioning

As we started to experiment with larger datasets we found them unwieldy to work with even if they fit in available memory. To work with a dataset in-memory, at least 2-5X of its size is required in available memory to store intermediary results. Spilling excess data to disk results in operations becoming magnitudes slower. For parallelizing tasks working on disjoint data, having each worker allocating memory proportional to the dataset is not feasible, and having workers share a single computational thread (or task) which

	Non-Partitioned Tables		Partitioned Tables		
PID	Information	Admissions	Prescription	ICU Vitals	
1	1 Row	5 Rows	23 Rows	1523 Rows	**Partition 1**
2	1 Row	9 Rows	102 Rows	5812 Rows	Patients 1-3
3	1 Row	2 Rows	57 Rows	503 Rows	←
4	1 Row	7 Rows	65 Rows	7432 Rows	**Partition 2**
5	1 Row	6 Rows	323 Rows	3244 Rows	Patients 4-6
6	1 Row	2 Rows	34 Rows	321 Rows	←

Figure 5: Partitioning example for the dataset MIMIC. All tables partition across the same patients.

works on a single copy of the dataset is computation specific and limits scaling to a few cores.

To solve the issues of memory use and parallelization, we design Pasteur to partition Datasets and Views, and to execute steps per-partition. With partitions, memory use is proportional to the partition size and workers, while disk access is limited to an initial load and final save for each partition. For parallelization, Pasteur loads, processes, and saves multiple partitions at a time, instead of parallelizing attempting to parallelize the calculations themselves, resulting in less overhead and higher parallelization than the use of a single computational thread.

Definitions. We define a View or Dataset as being partitioned if one or more of their Tables are partitioned. A partitioned Table consists of a set of similarly sized Tables we name Partitions. Each Partition is associated with a name. In the case a View or Dataset is partitioned, the subset of its Tables that are partitioned contain Partitions with the same names. Every entity in a View or Dataset is contained within Partitions of the same name. Example: in the MIMIC Dataset, we store patient's data across Partitions with the same name (Figure 5). By having each entity's data contained in one set of partitions, all required actions for that entity (transforming, encoding, measuring) without requiring loading the whole Tables.

3.3 Parallelization

Computation in Pasteur is composed by self-contained steps we name Tasks. For running Tasks, Pasteur provides a Task Queue which is backed by N workers (can execute up to N tasks simultaneously). We define Partitioned Tasks to be Tasks that are performed per View or Dataset Partition. Pasteur represents the methodology (Figure 2) as a Directed Acyclic Graph (DAG) dependency-based pipeline. Nodes in the DAG of the pipeline represent steps of the methodology. All nodes without conflicting dependencies are scheduled to run simultaneously. Each node is launched as a lightweight thread and is expected to create and schedule Tasks for heavy computation through the Task Queue. We assume each node's thread has negligible computational and memory requirements. This architecture is shown on Figure 6.

Sizing Partitions. For each pipeline execution, there is a Task that requires the most memory. This Task defines the base worker memory for that pipeline, M_{worker}. The total estimated required memory

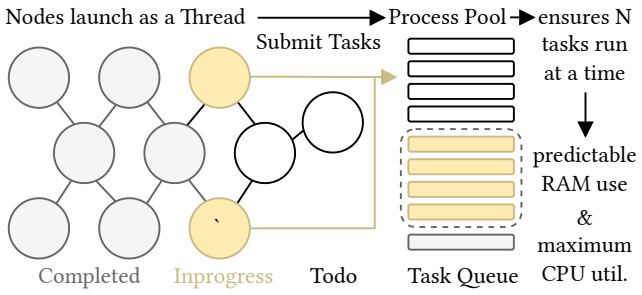

Figure 6: Parallelization architecture overview. Nodes feed a Task Queue which bounds RAM use.

for Pasteur to execute that pipeline becomes:

$$M_{\text{estimated}} = N \cdot M_{\text{worker}} \tag{4}$$

Correctly designed tasks require similar amounts of memory to each other, proportional to Partition size (ex. Partitioned Tasks).

$$M_{\text{worker}} = c \cdot M_{\text{partition}} \tag{5}$$

where c is a constant representing memory headroom (2-5).
From Equations 4 and 5 we have:

$$M_{\text{estimated}} \approx N \cdot c \cdot M_{\text{partition}} \tag{6}$$

To ensure maximum CPU utilization we set the number of workers N equal to the number of available logical cores. We also know the amount of available memory: $M_{\text{estimated}} = M_{\text{available}}$. We use the above as a guide to size our partitions:

$$M_{\text{partition,max}} \approx \frac{1}{c} \frac{M_{\text{available}}}{N} \tag{7}$$

To minimize runtime, partitions should be proportional to workers:

$$n_{partition} = kN \tag{8}$$

Where k is an integer. When a node schedules a set of Partitioned tasks, all workers will run with an equal amount of work, and finish in the minimum amount of time.
In sum, for maximum CPU utilization and stable memory usage:

- we set the number of workers N to the number of logical cores available in the system.
- We select the partition number $n_{partition}$ for the minimum k that respects $M_{\text{partition,max}}$.
- If the View is small and $n_{partition} \geq N$ degrades performance, we set $1 \leq n_{partition} < N$.

3.4 Marginal Calculations

Processor speed is measured in gigahertz, which are billions of hertz. Memory speed is measured in thousands of megatransfers per second, which are billions of transfers per second. In marginal calculations for tables where their size approaches billions of rows, processor instructions and memory accesses per row become important. An algorithm such as PrivBayes, when provided with hierarchical values, will require at minimum 10k marginal calculations. With a solution from the current state of the art, which requires 3m30s per 1 billion row marginal, synthesis requires 24 days. It becomes clear that we require a new marginal calculation method, which can parallelize linearly and is efficient in terms of clock cycles and memory accesses.

Preliminaries. Consider a distribution D with a set of N random values. The marginal distribution of a subset of D's variables is the probability distribution of those variables. In a computer system, we translate the definition to the following example: given a table T with columns X, Y, Z, K, L, the marginal distribution of X, Y, Z is a 3D matrix where the value at position (x, y, z) is the percentage of the combination (x, y, z) occurring in T.

Let us define each X in D to represent a hierarchical variable, which can be expressed as a tree with height H. X_i is the random variable that is formed when grouping the values of the level i nodes in X's hierarchical tree ($X_0 = X$). Stratifications (Section 2.3) are an extension of hierarchical variables that specify node child order. We have that $|X_i| < |X_j| \iff i > j$ and $|X_i| \leq |X| \; \forall \; 0 \leq i < H$,

where $|X|$ is the domain of X. For hierarchical variables, there exist deterministic functions such:

$$x_i = f_{i,j}(x_j) = f_i(x) \tag{9}$$

We create a grouping matrix $M_{H,|X|}$ per X where:

$$x_i = M_{i,x} \tag{10}$$

Marginal synthesis algorithms pose marginal requests where:

$$R = \{X_i,\ 0 \leq i < H_X \mid X \in D_R \subseteq D\} \tag{11}$$

Marginals are calculated in four steps:

(1) **Calculate variable groupings:**
$\hat{x_i} = M_{i,\hat{x}}$
(2) **Ravel variables to one index:**
$\hat{c} = \hat{x_i} + |X_i|\hat{y_j} + |Y_j||X_i|\hat{z_k} \ldots$
(3) **Create counting vector:**
$J_m = \text{count}(\hat{c} = m)$
(4) **Postprocess (reshape, normalize):**
marginal = reshape$(J, \left[|X_i|, |Y_j|, |Z_k|\right])$

Raveling the index and converting it into a counting vector (steps 2, 3) scale with row size and have to be calculated per marginal. Calculating variable groupings is a one time process. Postprocessing is negligible for tables with 50k+ rows.

Synthesis algorithms should batch and execute marginal requests without a causal dependence together: for larger-than-memory datasets, there is a fixed overhead associated with loading the dataset, and we achieve parallelization through performing multiple marginal calculations simultaneously.

Vectorized Algorithm. Our initial approach is an algorithm that performs steps 2 and 3 using vectorized instructions (ex. using NumPy arrays). All variable groupings are calculated once per table (step 1), rendering them negligible. Raveling to one index (step 2) is performed using standard vectorized operations (which use SIMD instruction sets) following the function shown in Step 2. Creating the counting vector (step 3) is performed by iterating over c and incrementing corresponding locations of a counting vector by 1 (with a compiled function, ex. NumPy bincount).

```
1:  procedure CALCULATE_MARGINALS_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
```

Figure 7: 256 bit marginal calculation for 16 bit marginals.

Consider a marginal for a table with N rows that covers M columns of that table (M is in the single digit range). When using vectorized operations for step 2, we perform one multiplication for each column apart from the first one (1 read, 1 write), which is stored in an intermediate buffer, and add it to the current index sum (2 reads, 1 write), which is a second temporary array initialized as the first column grouping. This results in a memory access complexity of $O(5N(M-1))$ and a space complexity of $O(2N)$. Finally, step 3 is performed by iterating over the index sum and increasing a marginal vector (2 reads, 1 write) for $O(3N)$ memory access complexity and $O(1)$ space complexity (assuming marginal size to be negligible). The total memory access complexity is $O(N(5(M-1)+3))$ and the space complexity is $O(2N)$. This algorithm achieves a great speedup compared to the implementations of the current state of the art, but is not optimal due to the intermediate array materialization. Its simplified complexity $O(MN)$ is optimal, so further improvements are linear.

Operation Fusing Algorithm. Our second approach performs operation fusing manually and only materializes the final counting vector, by storing all intermediate results in registers (space complexity of $O(1)$). We use SIMD extensions as in the previous algorithm. The AVX2 instruction set is widespread, so the formulation we show in this paper uses 256 bit registers for intermediate results. However, the algorithm is trivial to extend to 512 bits (for AVX-512), or to limit to 128 bits (for Armv8 Neon). For optimal results, we assume each variable grouping uses the smallest bit size unsigned integer based on domain size. In an initial step, we separate variable groupings based on bit size (arr_x with n_x groupings) and calculate a multiplier for each (m_x).

We then input the parameters in Algorithm 7, which showcases a version for marginals with domain sizes smaller than 16 bit. The algorithm uses a partially unrolled outer loop to handle 16 rows at a time. We unroll all inner loops by compiling a version for each combination of value sizes. 8 bit variable groupings are first expanded to 16 bit and all operations are performed using 16 bit vectorized instructions, except for the final increment, which can not be vectorized due to potentially incrementing the same memory location. We also create a variation for the rare case there is a larger marginal (more than $2^{16} = 65536$ domain size), where all groupings are first expanded to 32 bit. This algorithm features a memory access complexity of $O(N(M+2))$ and a space complexity of $O(1)$, which are optimal, and results in an estimated speedup of 5x.

Achieving Linear Scaling per Core. While the fused operation algorithm is faster than the vectorized algorithm, it reads and writes to a small subset of memory. As we will show in the experimental section, the writes to the counting vector can stall CPU pipelining if one marginal combination is over-represented. In addition, the reads of the algorithm create an effect of "read-read" contention when parallelizing, where if each core accesses the same memory for variable groupings, the overall execution becomes 2-3x slower. To solve for this issue, we propose a variable memory allocation strategy for workers when parallelizing, based on table size.

Using Shared Memory. The first strategy is to have workers access a shared pool of memory. For table sizes fitting in CPU cache, this solution offers no overhead and performs optimally, but its performance degrades for larger tables.
Using Unique Copy. The second strategy is to give each worker performing marginal calculations its own copy of the data. For L workers, provided available system memory exceeds $L+1$ times the dataset size this is possible, with the additional overhead of each worker copying the data prior to calculating marginals.
Using Unique Partition. The third strategy is splitting the table into partitions and having workers calculate marginals for each partition, which are merged in the end. This method has the most overhead, due to the high amount of interprocess communication required to merge the partition marginals. For large tables where the second allocation strategy is not possible and for out-of-core tables, this is the preferred solution. The overhead of this approach is static compared to row size and becomes negligible given a large enough table. Partitions are loaded from disk in the case of a larger-than-memory table for a fixed time penalty.

4 IMPLEMENTATION

We implement Pasteur as a Python library based on the packages Pandas [21], NumPy [7], PyArrow, and Kedro [1]. Kedro provides the system foundations, with its catalog, configuration, and pipeline. We extend Kedro to handle the increased complexity of Data Synthesis, by introducing a module system, parallelization primitives, automatic pipeline generation, and native out-of-core partitioned dataset support. For parallelization, Pasteur provides a set of primitive functions. The current implementations use Python's threading, multiprocessing and shared memory packages. By changing to a library such as Ray [18], Pasteur could offer horizontal scaling or adapt to cluster configurations.

4.1 Project Structure

Pasteur adopts Kedro's project based methodology. Users are expected to create a Pasteur project, which houses their configuration files, notebooks, and custom module implementations. Pasteur requires name spacing all hyperparameters and artifacts with the following rules, which enable the use of multiple datasets, views and algorithms per project:

- **Hyperparameter:** YAML syntax with top level tag <view>
  Example: tab_adult.tables.table.fields.income
- **Dataset:** <dataset>.<table> and <dataset>.raw@<source>
  Examples: adult.table, adult.raw@train, adult.raw@test
- **View:** <view>.<split>.<enc>_<table>
  Examples: tab_adult.ref.idx_table, tab_adult.wrk.table
- **Synthetic:** <view>.<alg>.<enc>_<table>
  Examples: tab_adult.aim.model, tab_adult.aim.idx_table

For each project, we define two working directories: a *raw data directory* and a *working directory*. The raw data directory hosts the raw sources of the datasets. It is not performance sensitive, may reside on a remote host (ex. S3, SQL server for queries), and may be read-only. The working directory stores the generated artifacts of the system and is performance sensitive (SATA SSD performance and above is enough). Since all system artifacts are generated in reasonable times (less than 1 hour), the working directory can utilize volatile media (scratch and RAM disks) and be ephemeral.

4.2 Memory Use Optimization and File Format

Proper Typing. We provide memory and save summaries for a table with 1 Billion rows, which we will synthesize, in Table 1. We estimate that loading this table from CSV without providing typing information would require 777 GB of RAM. For tables such as this, Pasteur allows users to provide typing information when loading raw data sources, which lowers memory use to 49 GB (parsing dates, using float32, dictionary data types, and variable integer sizes). Furthermore, a discrete encoding version of that table would require 136 GB of RAM with full width integers (64 bit). We expect a large proportion of categorical values to have less than 256 domain size (8 bit) and the vast majority less than 65536 (16 bit), so Pasteur automatically uses unsigned integers with the minimum bit width based on domain size, lowering memory use by 7-8x (to 24 GB).
Using Parquet. From Table 1, we see that compressed CSV requires hours to save and uncompressed CSV requires 197 GB in disk space. As seen in Figure 2, a single synthesis execution produces 6-9 artifact sets which equal the dataset size. We conclude that with uncompressed CSV total artifacts would reach 1-2 TB and with compressed CSV saving would require days. Therefore, we choose the Parquet file format for the intermediate artifacts of Pasteur. With parquet, a synthesis execution for 1 Billion rows requires around 100 GB. Furthermore, Pasteur uses partitioned data to work with large tables and saves files in parallel, which lowers saving time from 8m to less than 1m. Parquet files retain typing information, so information specified for raw data sources is retained throughout synthesis. Pasteur uses PyFileformat to open files, enabling the use of remote storage (such as S3) for both raw data and artifacts.

5 EVALUATION

5.1 Hardware

We execute experiments in a server with an AMD EPYC 7281 CPU (16c/32t) and 256 GB of quad-channel RAM. The raw directory is located on an 8 TB hard-drive and the working directory on a 256 GB SATA SSD drive. Parallelized experiments use 32 workers and single core experiments execute tasks on a single process. Since we use 16 cores, we define vertical scalability as being 16X faster than single core performance. The exception is Table 2, which is

Table 1: Memory use based on Format

		Original		Encoded	
	Format	Memory	Save t	Memory	Save t
Pandas[1]	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

[1]Data extrapolated to 1B rows from 15.5M rows.
object → category & date, int64 → uint8, → float32

Table 2: Marginal Calculation using Different Methods (In-memory)

		Arbitrary (SOTA)		Vectorized			Operation Fusing				
		Pandas	**NumPy**		JAX		C-based	**C AVX2**		**C AVX2** (stalled)	
Rows	Domain	value_counts	**histogramdd**	bincount	CPU	**GPU**		uint32	**uint16**	uint32	uint16
20k	2^{12}	2.39 ms	**2.99 ms**	99 us	211 us	312 us	118 us	27.9 us	**18.7 us**	47.0 us	46.5 us
500k	2^{12}	51.9 ms	**76.2 ms**	1.65 ms	4.13 ms	538 us	2.76 ms	613 us	**383 us**	1.09 ms	1.08 ms
1M	2^{12}	87.0 ms	**144 ms**	4.02 ms	8.33 ms	736 us	5.47 ms	1.20 ms	**742 us**	2.17 ms	2.16 ms
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**	4.34 ms	4.33 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**	10.9 ms	10.9 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**	21.7 ms	21.7 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**	10.9 ms	10.8 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**	10.9 ms	10.9 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**	8.69 ms	8.62 ms
5M	2^{20}	618 ms	**1.24 s**	71.9 ms	41.9 ms	**2.2 ms**	38.1 ms	**20.9 ms**	-	11.0 ms	-
5M	2^{24}	642 ms	**OVF**	79.1 ms	OVF	OVF	42.5 ms	**21.8 ms**	-	11.0 ms	-
5M	2^{28}	966 ms	**OVF**	127 ms	OVF	OVF	107 ms	**61.2 ms**	-	11.0 ms	-
5M	2^{32}	1.25 s	**OVF**	215 ms	OVF	OVF	254 ms	**97.6 ms**	-	11.1 ms	-

executed on a VM with AMD EPYC Rome cores and a T4 GPU to showcase JAX in GPU mode. For non-GPU methods, there is minimal deviation between the two machines.

5.2 Marginal Calculation

Algorithms To benchmark marginal calculation, we select a set of typical marginals as they would be used during synthesis and benchmark them for various row counts and domain sizes in Table 2. As a reference, we provide 2 implementations as used in the current state-of-the-art: Pandas' value counts and NumPy's histogramdd. Neither is optimized for marginal calculation, allowing the calculation of marginals with columns of arbitrary data types and floating point values respectively.

For the vectorized algorithms, we implement them using two libraries: NumPy and JAX. JAX advertises automatic operation fusing, but as it is mainly a GPU acceleration library, its use in CPU mode did not result in optimal results. For the fused operation algorithms, we implement them using C and intrinsics. We provide three implementations: one which uses standard C without the use of a SIMD instruction set, and two which use the AVX2 instruction set, for uint16 and uint32 domain marginals.

We conclude that all algorithms perform sufficiently well when used for a low number of marginals (< 500) and in-memory datasets such as those used in the current state-of-the-art (< 5M rows), without parallelization. For larger tables, the use of a faster algorithm becomes beneficial: we notice a 20x speedup for the vectorized NumPy bincount implementation. JAX in GPU offers the most impressive performance, for an additional 15x speed improvement. However, this speed improvement comes at the cost of occupying a GPU and in the limited use case where the table fits in VRAM. When the GPU is used for the synthesis algorithm or the table does not fit in VRAM, marginal calculation will occur the additional overhead of transfers to GPU memory. In addition, with GPU utilization close to 100 %, this method does not parallelize.

Finally, we present the AVX2 implementations as the benchmark for operation fusing marginal calculation. For the uint16 algorithm, we achieve a 5-8x speed-up over the vectorized NumPy implementation, which is inline with our complexity analysis. In total, this represents a gain of 150x over the current state-of-the-art without the use of parallelization. The final columns of Table 2 zero out the columns of the table, to showcase performance in the case of a non-spread out marginal due to CPU pipeline stalling (all increments are performed on index 0). In this case, both uint32 and uint16 feature equal performance as the majority of time is allocated to incrementing. An interesting effect of this is that even as the marginal domain size increases beyond the size of the CPU cache, calculation time remains the same, showcasing. the performance detriment of CPU cache page faults for very large marginals.

Table 3: Marginal Throughput on Parallelization

		Parallelized 16c/32t					
	One	Shared		Unique		Unique	
Rows	Core	Memory		Copy		Partition	
20k[1]	16.8k	**26.6k**	**1.6x**	26.6k	1.6x	-	
100k[1]	7.1k	**22.9k**	**3.2x**	23.3k	3.3x	-	
500k	1.6k	**14.4k**	**9.0x**	13.2k	8.3x	-	
1M	754	**9.0k**	**11.9x**	8.9k	11.8x	-	
5M	160	935	5.8x	**2.3k**	**14.4x**	-	
10M	81	413	5.0x	**1.2k**	**14.8x**	470	5.8x
50M	16.0	122	7.6x	**223**	**13.9x**	179	11.2x
100M	8.0	98.6	12.3x	-		**104**	**13.0x**
500M	1.6	18.6	11.6x	-		**25.5**	**15.9x**
1B	0.8	10.1	12.6x	-		**12.4**	**15.5x**

[1]Single thread IPC limits parallelization.

Parallelization. Following, we attempt parallelization with the three memory allocation attempts we outlined (Shared Memory, Unique Memory, Unique Partition) for multiple row counts and showcase the results in Table 3. For each row count, we load the table in-memory, calculate 10k to 4M marginals and average the result. We include single core performance as a reference.

As we discussed, Shared Memory features no overhead and performs well for table sizes fitting in the CPU cache. Once tables are large enough, Shared Memory underperforms and Unique Copy is faster. Due to the additional memory required, we only apply Unique Copy up to 50M rows. For larger tables, we provide Unique Partition, which we test for 10M rows and above.

With a combination of the three approaches, we achieve linear parallelization for tables beyond 5M rows. For smaller tables, the inter-process communication, scheduling, and postprocessing of marginals by a single thread becomes the bottleneck. Calculating a batch of marginals for larger-than-memory tables incurs a fixed time penalty of 7s to 1m depending on the available cores of the system, the table size, and the columns required to calculate them, after which point the performance mirrors Unique Partition. *In total, we note a 2000x speed improvement over the current state of the art, across all row numbers, when including parallelization.*

5.3 Synthesis Execution

In a final experiment shown in Table 4, we execute a synthesis experiment end-to-end to evaluate the performance of the system. We record execution time, the number of marginal calculations N required for synthesis, and a data quality measure to prove our resulting data is valid. For the quality measure, we use an average of normalized 2-way Kullback-Leibler divergence (KL), defined as:

$$KL_{ij} = \sum M_{ij} \log \frac{M_{ij}^{ref}}{M_{ij}^{wrk}}, \quad \text{KL} = \frac{1}{|C|^2} \sum_{i,j \in C} \frac{1}{1 + KL_{ij}} \tag{12}$$

where M_{ij} is a marginal for columns i, j, and C is a set of all columns.

We separate the Views by Dataset, and for each Dataset we measure its ingest time. For each View, we measure the time of its ingest tasks (which include all cache-able steps) and the KL measure for its reference set. Then, for each algorithm, we measure the number of marginals N it requires, the KL measure, and its execution time. The execution time includes sampling an equal number of rows, evaluation with per-column histograms, KL divergence for each column pair, and CS tests for each column.

Views and Datasets. We create three Views, based on 2 Datasets: Adult and MIMIC. To our knowledge, there is no viable Differentially Private multi-table algorithm, so we use tabular Views.

The Adult View mirrors the Adult dataset, with 15 columns (5 numerical, 10 categorical) which are encoded to 14 discrete Values. It portions 80 % for the work set and 20 % for the reference set from a total of 58k rows. For Numerical Values, it uses 20 buckets.

The Admissions View of MIMIC joins the Admissions table to the Patients table and retains 14 columns (4 datetime columns and 10 categorical) which are encoded to 19 discrete Values. It portions 80 % for the work set and 20 % for the reference set from a total of 418k rows. The datetime columns "dischtime", "deathtime" reference the column "admittime", which references the birth year of the patient. For synthetic data, we set the birthdate of patients to be 1/1/2000 (in MIMIC, birth year is randomized).

The ICU Charts View joins the tables ICU chartevents (330M rows), icu stays, and patient together, and duplicates the data 3 times for a total of 60 partitions and 989M rows. It retains 10 columns (4 datetimes, 1 numerical, and 5 categorical) which become 15 discrete encoded Values. The datetime columns "charttime", "outtime" reference the column "intime", which references the birth year of the patient. As with Admissions, we set birthdays to 1/1/2000. This View uses equal work and reference sets up to a reference set of 20M rows. Following, additional rows are allocated to the work set, which reaches 970M in total.

Synthetic Algorithms. For synthesis, we select three algorithms: PrivBayes [24], MST [14] and AIM [15]. PrivBayes is ideal for benchmarking our system, due to the fact that the majority of its compute is calculating marginals, which stress tests our system. We execute PrivBayes once with the original Values, which feature a limited height and result in a low complexity, and once with a manually rebalanced set of Stratified Values (PrivBayesRB) which raise PrivBayes' computational complexity and require 10k-1M marginals for synthesis. For the ICU Charts View, we limit PrivBayes's rebalanced Values to a complexity which results in around 15k marginal calculations. We use AIM and MST in their original implementations to showcase the system's ability to generalize and scale algorithms with minimal changes. Since AIM is a workload-aware algorithm, we use all 2-way marginals for its workload. MST and AIM sample from a small pool of marginals (< 500), so we compute all of them at the start (as discussed in Section 3.4, there is a fixed penalty per batch for out-of-core data).

In the ICU Charts View, we wish to retain the solution complexity for algorithms while increasing the row count. For marginal algorithms, this can be accomplished by keeping the ratio eN stable, where N is the number of rows and e is the privacy budget. In PrivBayes, the complexity is defined by a parameter theta, which is set by the ratio eN. In AIM and MST, the noise added to each marginal is proportional to e and the marginals are not normalized (proportional to N). We note a sublinear relationship between row count and execution time. We attribute this to fixed execution tasks requiring the same amount of time and row dependent ones increasing linearly. The exception is AIM for 1M, 10M rows, where due to the table being noisy it increases its step count.

Parallelization. To test the system parallelization, we execute the synthesis of 500M rows twice: once parallelized and once without parallelization (last row). At each execution, we measure RAM use. For ingestion, the peak RAM use is 70 GB for multicore (32 workers) and 25 GB for single core. For synthesis, the peak RAM use is 85 GB for multicore and 10 GB for single core (all algorithms). We split the MIMIC Dataset into 5 partitions and the resulting View into 60. We expect the ingestion process to require more RAM when ran in single core (25 GB) because of the larger partitions. The synthesis executions that follow are partitioned correctly and feature lower RAM use for single core (10 GB) and great parallelization for multicore (11-14x). For the algorithms AIM and MST, we use their original implementations, which are single-threaded, resulting in low parallelization (5.5x, 6.2x). With less than 3 GB per worker (or core), these experiments can be executed on most commercial VMs, which feature 2-4 GB per core.

Table 4: Synthesis Execution

			Ref	Ingest	PrivBayes			PrivBayesRB			MST			AIM		
Data	Rows	e	KL	**time**	N	time	KL	**N**	**time**	KL	N	time	KL	N	time	**KL**
Adult				**0:14**												
	50k	1	0.937	**0:21**	168	0:45	0.944	**10.5k**	**1:03**	**0.948**	120	3:16	0.937	120	13:35	0.912
MIMIC				**15:40**												
Admit.	550k	1	0.973	**0:26**	8k	1:17	0.954	**1.3M**	**8:46**	0.964	210	4:36	0.948	210	21:54	**0.983**
	1M	1	0.394	**3:27**	257	0:57	0.954	**11.3k**	**1:17**	0.930	136	3:51	0.956	136	187:59	**0.996**
ICU	10M	0.1	0.714	**4:45**	302	1:56	0.742	**17.7k**	**3:10**	0.717	136	4:47	0.953	136	218:01	**0.987**
Charts$^3_+$	100M	0.01	0.748	**6:20**	306	5:48	0.773	**16.3k**	**10:52**	0.755	136	7:43	0.984	136	8:35	**0.978**
	500M	0.002	0.779	**12:52**	307	16:37	0.770	**15.8k**	**36:37**	0.790	136	27:03	0.588	136	23:23	**0.992**
	1B	0.001	0.743	**21:25**	300	29:34	0.801	**16.0k**	**71:57**	0.797	136	41:35	0.995	136	30:01	**0.989**
1 core →	500M			2:19:08	(**11x**)	3:00:06	(**11x**)		8:16:25	(**13.6x**)		3:16:53	(**7.3x**)		2:24:35	(**6.2x**)

For AIM with 1 Billion rows we have: 15:40 for Dataset ingest, 21:25 for View ingest, and 30:01 for synthesis, which round to 1 hour for synthesizing and evaluating an 1 Billion row (200 GB CSV size) dataset, starting from raw data.

6 RELATED WORK

Systems. The system Synthetic Data Vault (SDV) [19] comprises the only previous work to provide a framework for data synthesis. It contains a common platform for data synthesis algorithms to build upon, with multiple evaluation datasets (SDGym) and metrics (SDMetrics), as well as a framework with transforms (RDT) that can be used to preprocess raw data. In addition, SDV features a metadata system that can instruct how to type and transform a raw CSV file, which we borrow for Pasteur with YAML files instead of JSON. The main limitations of SDV are that it is an exclusively in-memory synthesis framework, its transformer library is rudimentary (without integration of domain knowledge and only for continuous encodings), and it doesn't account for preprocessing.

ML-Ops. In the field of Machine Learning Operations (ML-Ops), multiple systems have been proposed to handle the machine learning workflow using a DAG pipeline: Prefect, Kubeflow, Apache Airflow, Luigi, Snakemake [17] and Kedro [1].

Synthesis Algorithms. Data Synthesis algorithms can be separated into two broad categories: Neural Network based and Marginal based. Notable Neural Network algorithms are PATE-GAN [12], CT-GAN [22], DP-GAN [9], DP-CTGAN [5], and ADS-GAN [23]. For tabular data, a new class of Marginal based algorithms has been shown to outperform Neural Network based algorithms when privacy is required [20]. Notable marginal algorithms are: MST [14], PrivBayes [24], AIM [15], PrivMRF [2], which are enhanced by using probabilistic graphical models on inference, through PrivPGM [16]. When privacy is required, most algorithms utilize the method Differential Privacy [4] to provide their privacy guarantees.

Evaluation For evaluation, prior work utilizes three types of metrics: visualization, synthetic, and modelling. Visualization metrics are histograms [20], target value class distribution [20], and Principal Component Analysis (PCA) maps. Synthetic metrics are synthetic (non-real) workloads performed on the data which produce a set of numbers indicating quality. The major ones are X^2, correlation matrices [20], and Kullback-Leibler (KL) Divergence. Modelling metrics [22] use a classifier to train for a target variable, where a dataset has one, and measure the behavior of classifiers trained on synthetic data versus ones trained on original data, with the specific metrics depending on the methodology.

7 CONCLUSION

Most interesting topics in Data Synthesis involve complicated and large datasets. In this paper, we introduce an end-to-end methodology designed to handle this complexity. Our methodology ensures reproducibility and transparency in the synthesis process, by including all processing steps from raw data to synthetic product and evaluation, and by being algorithm and dataset agnostic. We implement this methodology in a system we name Pasteur, which we open-source alongside this paper. Pasteur is designed to work with out-of-core datasets and linear vertical scaling, with a novel parallelization architecture and marginal calculation algorithm. Our evaluation shows that for a 16 core machine, Pasteur reaches up to 14X single core performance without memory over-allocation issues. In fact, Pasteur is able to synthesize and evaluate 1 Billion rows of synthetic data (200 GB) in around 1 hour.

8 ACKNOWLEDGEMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 955895.

REFERENCES

[1] Sajid Alam, Nok Lam Chan, Gabriel Comym, Yetunde Dada, Ivan Danov, Deepyaman Datta, Tynan DeBold, Lim Hoang, Jannic Holzer, Rashida Kanchwala, Ankita Katiyar, Amanda Koh, Andrew Mackay, Ahdra Merali, Antony Milne, Cvetanka Nechevska, Huong Nguyen, Nero Okwa, Joel Schwarzmann, Jo Stichbury, and Merel Theisen. 2022. *Kedro.* https://github.com/kedro-org/kedro

[2] Kuntai Cai, Jianxin Wei, Xiaoyu Lei, and Xiaokui Xiao. 2021. Data Synthesis via Differentially Private Markov Random Fields. *VLDB* (2021), 13.

[3] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[4] Cynthia Dwork. 2006. Differential Privacy. In *International Colloquium on Automata, Languages, and Programming*. Springer, 1–12.

[5] Mei Ling Fang, Devendra Singh Dhami, and Kristian Kersting. 2022. DP-CTGAN: Differentially Private Medical Data Generation Using CTGANs. In *Artificial Intelligence in Medicine*, Martin Michalowski, Syed Sibte Raza Abidi, and Samina Abidi (Eds.). Vol. 13263. Springer International Publishing, Cham, 178–188. https://doi.org/10.1007/978-3-031-09342-5_17

[6] Chang Ge, Shubhankar Mohapatra, Xi He, and Ihab F Ilyas. [n.d.]. Kamino: Constraint-Aware Differentially Private Data Synthesis. ([n. d.]), 14.

[7] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[8] Mikel Hernandez, Gorka Epelde, Ane Alberdi, Rodrigo Cilla, and Debbie Rankin. 2022. Synthetic Data Generation for Tabular Health Records: A Systematic Review. *Neurocomputing* 493 (July 2022), 28–45. https://doi.org/10.1016/j.neucom.2022.04.053

[9] Stella Ho, Youyang Qu, Bruce Gu, Longxiang Gao, Jianxin Li, and Yong Xiang. 2021. DP-GAN: Differentially Private Consecutive Data Publishing Using Generative Adversarial Nets. *Journal of Network and Computer Applications* 185 (2021), 103066. https://doi.org/10.1016/j.jnca.2021.103066

[10] Taeuk Jang, Feng Zheng, and Xiaoqian Wang. 2021. Constructing a Fair Classifier with Generated Fair Data. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 9 (May 2021), 7908–7916. https://doi.org/10.1609/aaai.v35i9.16965

[11] Alistair Johnson, Lucas Bulgarelli, Tom Pollard, Steven Horng, Leo Anthony Celi, and Roger Mark. [n.d.]. MIMIC-IV. https://doi.org/10.13026/6MM1-EK67

[12] James Jordon, Jinsung Yoon, and Mihaela van der Schaar. 2018. PATE-GAN: Generating Synthetic Data with Differential Privacy Guarantees. In *International Conference on Learning Representations*.

[13] Bo Liu, Ming Ding, Sina Shaham, Wenny Rahayu, Farhad Farokhi, and Zihuai Lin. 2021. When Machine Learning Meets Privacy: A Survey and Outlook. *Comput. Surveys* 54, 2 (March 2021), 31:1–31:36. https://doi.org/10.1145/3436755

[14] Ryan McKenna, Gerome Miklau, and Daniel Sheldon. 2021. Winning the NIST Contest: A Scalable and General Approach to Differentially Private Synthetic Data. arXiv:2108.04978 [cs]

[15] Ryan McKenna, Brett Mullins, Daniel Sheldon, and Gerome Miklau. 2022. AIM: An Adaptive and Iterative Mechanism for Differentially Private Synthetic Data. arXiv:2201.12677 [cs]

[16] Ryan Mckenna, Daniel Sheldon, and Gerome Miklau. 2019. Graphical-Model Based Estimation and Inference for Differential Privacy. In *Proceedings of the 36th International Conference on Machine Learning*. PMLR, 4435–4444.

[17] Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B. Hall, Christopher H. Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O. Twardziok, Alexander Kanitz, Andreas Wilm, Manuel Holtgrewe, Sven Rahmann, Sven Nahnsen, and Johannes Köster. 2021. Sustainable Data Analysis with Snakemake. *F1000Research* 10 (Jan. 2021), 33. https://doi.org/10.12688/f1000research.29032.1

[18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. arXiv:1712.05889 [cs, stat]

[19] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. 2016. The Synthetic Data Vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. 399–410. https://doi.org/10.1109/DSAA.2016.49

[20] Yuchao Tao, Ryan McKenna, Michael Hay, Ashwin Machanavajjhala, and Gerome Miklau. 2021. Benchmarking Differentially Private Synthetic Data Generation Algorithms. *arXiv:2112.09238 [cs]* (Dec. 2021). arXiv:2112.09238 [cs]

[21] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 56 – 61. https://doi.org/10.25080/Majora-92bf1922-00a

[22] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. 2019. Modeling Tabular Data Using Conditional GAN. *arXiv:1907.00503 [cs, stat]* (Oct. 2019). arXiv:1907.00503 [cs, stat]

[23] Jinsung Yoon, Lydia N. Drumright, and Mihaela van der Schaar. 2020. Anonymization Through Data Synthesis Using Generative Adversarial Networks (ADS-GAN). *IEEE Journal of Biomedical and Health Informatics* 24, 8 (Aug. 2020), 2378–2388. https://doi.org/10.1109/JBHI.2020.2980262

[24] Jun Zhang, Graham Cormode, Cecilia M. Procopiuc, Divesh Srivastava, and Xiaokui Xiao. 2017. PrivBayes: Private Data Release via Bayesian Networks. *ACM Transactions on Database Systems* 42, 4 (Oct. 2017), 25:1–25:41. https://doi.org/10.1145/3134428

[25] Zhikun Zhang, Tianhao Wang, Ninghui Li, Jean Honorio, Michael Backes, Shibo He, Jiming Chen, and Yang Zhang. [n.d.]. PrivSyn: Differentially Private Data Synthesis. ([n. d.]), 18.