The Case for Adding Privacy-Related Offloading to Smart Storage

Claudiu Mihali
Aalto University, Finland

Anca Hangan
UTCN, Romania

Gheorghe Sebestyen
UTCN, Romania

Zsolt István
IT University, Denmark

ABSTRACT

It is important to ensure that personally identifiable information (PII) is protected within large distributed systems and is used only for intended purposes. Achieving this is challenging and several techniques have been proposed for privacy-preserving analytics, but they typically focus on the end hosts only. We argue that future storage solutions should include, in addition to emerging compute offload, also privacy-related operators. Since many privacy operators, such as perturbation and anonymization, take place as the very first step before other computations, query offload to a Smart Storage device might be only feasible in the future if privacy-related operators can also be offloaded.

In this work we demonstrate that privacy-preserving operators can be implemented in hardware without reducing read bandwidths. We focus on perturbations and extend an FPGA-based network-attached Smart Storage solution to show that it is possible to provide these operations at 10Gbps line-rate while using only a small amount of additional FPGA real-estate. We also discuss how future faster smart storage nodes should look like in the light of these additional requirements.

CCS CONCEPTS

• **Information systems → Storage architectures**; • **Hardware → Reconfigurable logic and FPGAs**; • **Security and privacy → Data anonymization and sanitization**.

ACM Reference Format:
Claudiu Mihali, Anca Hangan, Gheorghe Sebestyen, and Zsolt István. 2021. The Case for Adding Privacy-Related Offloading to Smart Storage. In *The 14th ACM International Systems and Storage Conference (SYSTOR '21), June 14–16, 2021, Haifa, Israel.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3456727.3463769

SYSTOR '21, June 14–16, 2021, Haifa, Israel

ACM ISBN 978-1-4503-8398-1/21/06…$15.00
https://doi.org/10.1145/3456727.3463769

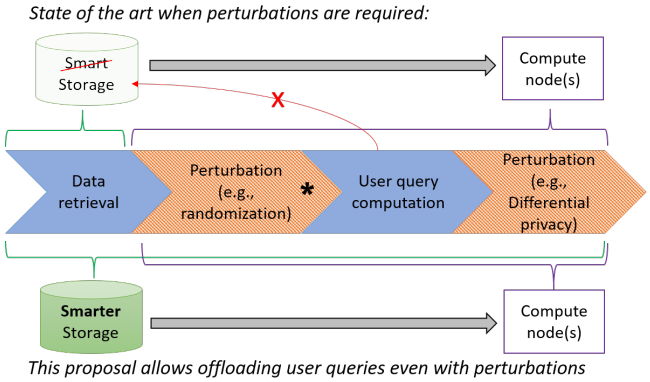


Figure 1: Some perturbations take place before data is being processed by queries, so unless they can be performed by the storage node, user query offloading cannot be shifted to the Smart Storage node either.

1 INTRODUCTION

Big Data has revolutionized our lives but has also created challenges in the areas of privacy and data protection. As a result, there is a push for stronger data protection rules and companies are investing in tools for tighter control, monitoring, and transformation of personally identifiable information (PII). Data perturbations [9, 16, 23] emerged as a powerful tool for ensuring that data retains utility (e.g., to be used to prototype analytical models, compute aggregate statistics, release anonymized information) without risking data leaks. Data perturbation operations are being incorporated in a growing number of systems [11, 17, 27, 29] and we expect for this trend to continue. In processing pipelines, in general, perturbations can take place both *before* any further query-specific computation (e.g., rotation-based perturbation [8]) and *after* the results of a query have been computed (e.g, applying differential privacy [15] to aggregates), as shown in the right-hand side of Figure 1.

Smart Storage, that is, storage with offloading capabilities, is an idea that has been around for a long time [3] and is seeing increasing adoption in datacenters [2, 21] and even in cloud services, such as Amazon Redshift Aqua [1]. Many of the state-of-the-art systems are based on specialized hardware, such as FPGAs [20, 22, 26], due to their energy efficiency and predictable behavior. Regardless of the inner design choices and deployment model of smart storage nodes,

one challenge always lies in mapping high level processing to the available resources and splitting between storage nodes and software. When factoring in the need for privacy-related operators in future pipelines, the usefulness of smart storage could be dramatically reduced unless they can offload such operators at the beginning of the pipelines as well. Motivated by this, it is important to start exploring privacy-related processing options today, to ensure that storage nodes with offload capabilities will remain relevant going forward.

In this work we study *what is the cost of offering data perturbation on-the-fly* inside specialized storage nodes. We extend Caribou [20], an FPGA-based network-attached key-value store, with rotation-based privacy-preserving perturbation [9, 34] – resulting in a *Smarter Storage* node. The chosen operator is representative of a wide range of perturbations (the step marked in Figure 1 with $*$) and is one of the more compute intensive ones requiring matrix operations on real numbers. Our results show that such perturbations can be carried out at the 10Gbps network line-rate, while requiring only a small amount of additional FPGA resources. We integrate with a Python library and evaluate training classifiers directly on perturbed data.

In summary, this work brings the following contributions:

- We make the case for adding privacy-related operators to the "toolbox" of Smart Storage nodes and show that such *Smarter Storage* can be built with FPGAs.
- We present a general pipeline structure for FPGAs for rotation-based perturbations, with a performance that matches the 10Gbps network rate by design.
- We provide an open-source implementation[1] that enables transparent integration with software applications written in high level languages, such as Python.

2 BACKGROUND AND RELATED WORK

2.1 Smart Storage

Recent years have seen a wide-spread deployment of data analytics systems that span several machines and access datasets stored on distributed storage, including notable examples such as Amazon Redshift [1] and Snowflake [10]. This distributed setup introduces data movement bottlenecks at all levels of the architecture and motivates the renewed interest in active storage [3], with numerous related works in the last decade [7, 12, 18, 21, 33] to build solutions that are power-efficient but, at the same time, offer low latency and high throughput access to data.

It is possible to build distributed storage using nodes that execute both data management and processing logic on specialized hardware, such as Field Programmable Gate Arrays (FPGAs). This can be seen as a specialization of the server hardware without compromising on the general-purpose nature of the key-value store service being offered. Using specialized hardware, such as FPGAs, shows promising results [20, 22, 26, 33]. One of the advantages of using FPGA-based nodes is that they can be designed such that the in-storage processing steps are performed without reducing read or write bandwidths to the nodes. This is done by structuring computation as pipelines that process input at a fixed rate. Recent successes reported in Redshift Aqua [1] and the Samsung SmartSSDs [2] brought these solutions into the mainstream in clouds and datacenters alike.

2.2 Privacy-preserving Perturbations

Data perturbation is a privacy preservation technique that alters the values of tuples in a dataset to disguise the sensitive information while preserving the particular properties that are critical for building meaningful data analytics models. Many data perturbation techniques enable applying common data mining operations directly onto modified data, with a minimal impact on the overall accuracy of the obtained results. The intent of these techniques is to allow legitimate users the ability to access aggregate statistics and relations between the data elements while protecting the individual identity of the records.

Among privacy-preserving techniques, data perturbation is conceptually simple, since in many cases it can be reduced to applying arithmetic operations to all the rows, columns, or individual records in a database. Examples of perturbation include: Additive perturbation [30], Random rotation [8], Geometric perturbation [9], Microaggregation [13], Condensation [4] and Differential Privacy [15].

There are several data mining techniques that are invariant to geometric, rotation-based perturbations, as proven in [9], strengthening the case for using this method as a first step for exploring more complex data perturbation algorithms. Such data mining techniques include kNN Classifiers, Support Vector Machines, Linear Regression Models, Clustering Models based on Euclidean distance, and many others.

2.3 Data Protection Frameworks

Most data perturbation solutions require two components: one is the actual data transformation (relying, e.g., on arithmetic, matrix operations and sampling from random distributions), and the other is a high-level controlling algorithm that reasons about privacy budgets and determines the parameters of the transformations. Differential privacy (DP) is one example that provides a strong privacy guarantee [15] but requires maintaining a privacy budget constraint. This limits the number and types of queries to be performed before some information leakage occurs. The data perturbation method used in this work targets different data processing operators

[1] https://github.com/zistvan/caribou/tree/privacy-prototype

than DP and does not require keeping track of privacy budgets – nonetheless, our prototype platform could incorporate operators that require an external controller. Such controllers already exist as part of frameworks that track and control data flows across distributed systems. This area has been explored for more than two decades [6, 24, 27, 28, 32, 36] and we assume that in the future a logically centralized policy management framework will be available to control, among other aspects, the way data perturbations are performed and who can access what datasets. For this reason, we consider the controller-related aspects external to this work and we focus on the mechanisms required for performing the data perturbation and making sure that the operator is controllable from software without overhead.

3 ROTATION PERTURBATION PRIMER

We chose to use 3D Rotation Perturbation [9, 31, 34] due to its conceptual simplicity but, at the same time, good performance for classification workloads. The algorithm works on numerical data in tabular form with at least three attributes (columns). The columns of the dataset are randomly grouped into triplets so that each column is covered by the grouping, with repeating entries when necessary. For example, if we have a dataset with five columns, a suitable random grouping would be: (C4, C1, C3), (C2, C4, C5). Each row will then produce triplets of numerical values according to the column grouping, which are interpreted as points in the three-dimensional space. The perturbation rotates them by a random angle, modifying their positions while maintaining specific geometrical properties which are used for training classifiers (e.g., the Euclidean distance relative to the origin and also the Euclidean distance relative to each other).

Rotations in 3D Euclidean space are realized using rotation matrices of size 3-by-3 and, for determining the rotation angle that maximizes privacy, we use the algorithm described in [34]. In a nutshell, it works as follows: First, as a preprocessing step, the data is normalized with MIN_MAX to bring the values in the same range such that differences in the magnitudes of the values do not influence the choice of the rotation matrix. After computing all the privacy metrics corresponding to all *(angle, axes combination)* pairs in 1-degree steps, we choose the pair which gives the maximum privacy metric and output the general rotation matrix that results from it. The time complexity of the algorithm is $O(m \times n)$, where m represents the number of attributes (columns) of the dataset and n represents the number of entries (rows) of the dataset. The runtime of this algorithm that determines the rotation matrix could be reduced by subsampling.

For each independent table (dataset), a different column grouping, and rotation matrix are pre-computed and stored as key-value pairs into the KVS. The client library can choose at run-time between any number of perturbation matrices. For generality, we target double-precision floating point numbers and carry out the computation as such on the FPGA.

3.1 Example Use-case in a Company

Our example use-case targets an enterprise that stores large datasets about their clients, including private information. The company has data officers who define company-wide policies, and only applications that have been explicitly authorized and audited can gain access to user's data in plaintext. Overall, the goal of the company is to protect data against *honest but curious* employees while, at the same time, retaining as much of its usefulness as possible.

In this context, if there is a team that would like to propose a new product idea requiring the development of a classifier based on a dataset already available at the company, they would have to get full access permission to the data from the officers to verify the feasibility of their idea. Alternatively, they could get access to the data in a perturbed form for exploration, to train and retrain their classifiers identifying promising directions and building a proof-of-concept around it. In cases where it is necessary for integration with other applications, once the prototype shows credible benefits, the team can be allowed by management and the data officer to gain access to real user data. While it is possible to build meaningful applications on top of perturbed data (e.g., when using differential privacy), in the case of the rotation-based perturbation, retraining on real data is probably the better decision. Note that, in the above example, the mechanisms to obtain user consent or to make sure that the use is within the allowed ones, is orthogonal to this work.

4 BUILDING SMARTER STORAGE

In this work we extend the multi-tenant version of Caribou [19, 20]. Caribou is an FPGA-based network-attached smart storage solution that exposes a simple key-value store interface and can handle at 10Gbps line-rate for virtually any mix of read/write operations.

Caribou is "smart" in that it mitigates the data movement bottleneck between storage nodes and software applications by offering different filtering operators inside the storage node while matching the streaming read bandwidth of the underlying storage. These filtering operations are exposed to software as so called "getConditional" operations. These are similar to gets but values are only transmitted from the storage if they match a specific condition (e.g., a regex).

In addition to the hardware modules, Caribou has a layered client library [25], written in Go, that executes on regular software nodes (Figure 2). In addition to the expected get/set/delete and getConditional operations for key-value pairs, Caribou's client library supports operations on Parquet

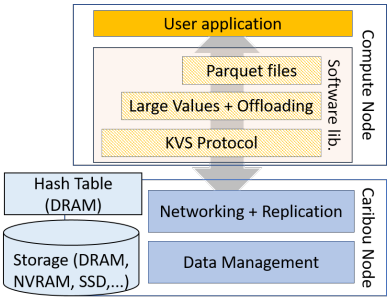


Figure 2: Caribou is an open-source FPGA-based Smart Storage solution with a companion library.

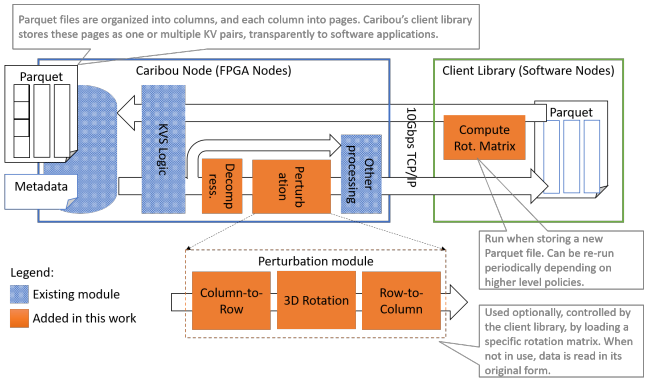


Figure 3: Our prototype consists of a software component that pre-computes rotation matrices and provides the right interfaces to clients, and a hardware component that decompresses Parquet pages and performs the rotation on tuples.

files, a columnar file format that is widely used in distributed data processing frameworks. Upon storage, a Parquet file is transparently decomposed by the library into data pages which are set as individual key-value pairs. A metadata page makes it possible to reconstruct the file when retrieving the data as key-value pairs.

In this work we extend Caribou with several new operations, shown in Figure 3. These perform the perturbation on the read path of the FPGA and include new API entries in the software library to handle additional meta-data and rotation matrices. Furthermore, we add a command that retrieves a subset of three pages (three column fragments) from a Parquet file with rotations applied to them (n.b., the three page-oriented operation is due to the nature of the perturbation we chose).

We chose Parquet as the underlying data format due to its wide-spread use in analytics. Even though it is columnar in its layout, internally, the perturbation module works on a stream of tuples (more like rows). Hence, we provide modules translating between columns and tuples. These are not Parquet-specific and, therefore, other columnar file formats could be used. The design could also be run on row-oriented data with little modification and no loss of performance. In general, our proposed modules are independent of the underlying key-value store (KVS) design and could be integrated with other systems, such as BlueCache [37].

4.1 Software Components

4.1.1 Accessing Parquet files (un)perturbed. We extended the Go-based client library of Caribou with operations that allow accessing data stored as Parquet files both with and without applying perturbations. In the latter case, the perturbation matrix can be updated at run-time, with the additional cost of a single SET operation. It is also worth pointing out that the perturbation operations themselves are not specialized to Parquet files and could be used with other columnar, or even row-based, file formats.

The existing client library exposes both a "raw" interface for setting/getting individual key-value pairs, and an interface that operates at the level of Parquet files and their columns and pages. We rely on the raw interface to store and load rotation metadata. When reading with perturbation, a group of three values has to be read, which, for Parquet files is equivalent to three pages. For simplicity, we provide a function to retrieve a Parquet file with selected columns (automatically grouped into sets of three) perturbed. The reason for performing the perturbations selectively is that, for instance for classifications, the category data of tuples is needed in its original form to perform learning. The Go client is integrated into a Python front-end in order to perform the actual training and classification using Python's rich set of machine learning libraries.

The example use-case (Section 3.1) applies privacy perturbations onto datasets destined for machine learning use. Such datasets often have a very large number of columns and not all of them are useful for learning. This is because some of them might be correlated thus giving too much weight onto a certain characteristic of the data or they might add noise to the final predictions. For this reason, we have provided functionality to selectively read columns. However, the software interface exposed to Python works only with whole columns, without support for partial and range-based access to the selected columns. This is because, in our current implementation we use an off-the-shelf Parquet serialization library and lack fine-grained control over the page sizes in a Parquet file which brings the challenge of having to deal with data columns being decomposed into pages of various lengths. Thus, in order to avoid accumulating leftover data into buffers because of non-matching page sizes, we restrict the client to load only entire columns at once, as opposed to

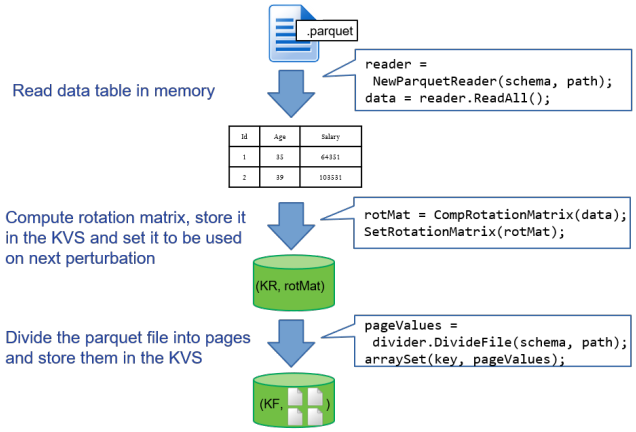


Figure 4: Steps done for storing a file.

individual pages. Since all dataset columns have the same number of rows, at the end of the read operations no leftovers will remain.

4.1.2 Perturbation metadata. The perturbation key is unique for each dataset and consists of a random partitioning of the column set into triplets and a rotation matrix. This key is stored as a key-value pair and must be loaded into the perturbation module before the perturbed access. Protecting the key against unauthorized access can be enforced with access control, which is orthogonal to our work.

The algorithm for determining rotation matrices includes a step for random column partitioning to make the rotation matrix non-deterministic in case the algorithm is re-run on the same dataset. We used Richard Durstenfeld's version of the Fisher-Yates shuffle [14] to obtain a random permutation of the column order. To build the random partition, we pick ordered triplets from this permutation and if the number of columns is not divisible by three, we pick some other columns at random to be paired with the ones which do not yet have a complete triplet.

The cost of loading the rotation matrix in the FPGA circuit is negligible both from a time- and a BRAM-capacity-perspective which means that data accesses by different clients, requiring different rotation matrices, can be interleaved. In the current implementation this is only limited by the idiosyncrasies of the Column-to-Row module explained in Section 4.2.2.

4.1.3 File storage workflow. Figure 4 illustrates the steps taken when storing a Parquet file in the KVS. When a client calls the specific API function to store a parquet file, first, the file will be read in memory and the column permutation will be computed. After that, the algorithm described in Section 3, that computes the rotation matrix for a specific dataset, is performed on client-side. The resulting matrix will then be stored in the KVS and loaded into the Rotation module whenever accessing its associated Parquet file.

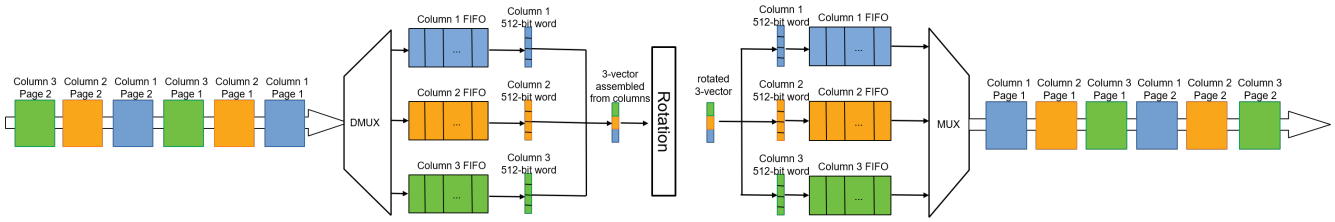
Independently of these operations, the Parquet file will be stored using the same operations as in Caribou [25]. Hence, there is no additional overhead associated to storing the file or updating it over time.

4.1.4 File access from software. There are two main operations that are used to interface with the FPGA in order to access data: Get and GetPerturbed. The Get operation simply fetches a page referenced by a key. The GetPerturbed operation is similar to the Get operation, with the difference being (a) that in the requests sent to the FPGA, it instructs the KVS to stream the page through the rotation module. And (b) that it has to be invoked in groups of three pages, belonging to different columns. If the columns are not divided into pages of equal sizes, this GetPerturbed operation should be used to retrieve all three columns in an interleaved manner, to avoid leftover data or mismatched outputs in the buffers. This does not mean, however, that all pages have to be equal, and the modules are able to handle pages of different sizes, as long as for each column the same number of rows is retrieved.

Since both the Decompression and Perturbation modules can be activated/deactivated in order to allow for complete flexibility upon data storage (compressed/decompressed) and retrieval (perturbed/non-perturbed), the GetPerturbed operation encodes the flags that decide which functioning mode shall be used by each module to serve the current client requests. This operation is, in fact, derived from the GetConditional operation of Caribou, that encodes the configuration of each query offload module on the device.

To retrieve sets of three columns, we rely on bulk synchronous requests, meaning that we issue multiple Gets or GetPerturbeds in the same TCP packet that will be sent to the FPGA. As already mentioned, an important aspect when accessing perturbed data is the order of the requests. These have to be performed in groups of three, interleaved, corresponding to the three columns that are grouped in order to be perturbed. The triplets are chosen according to the random partitioning, which was performed in the file storage stage, since the rotation matrix was computed in order to maximize the privacy metric for this specific partitioning of the column set.

Another feature that we incorporated in the client is the ability of selectively turning off perturbation for some columns. This is needed, for instance, for the "output" column used for training the machine learning models. Since the result of the model has to be compared to the values of such columns, they shall not be perturbed. To this end, we provide API functions to specify the special "output" column and thus, when retrieving the perturbed dataset, the Perturbation module will be deactivated when fetching that column.



(a) Column To Row module internal architecture.

(b) Row To Column module internal architecture.

Figure 5: Transformations between columnar and row-wise layout are performed in a streaming fashion. Throughput is constant and additional latency is only incurred for the first page of a request.

4.2 Hardware Components

In the following, we present in detail the design of each processing step in our prototype: starting with the (optional) Decompression module that decompresses Parquet pages to allow for in-storage perturbation of compressed data, then the Column-to-Row and Row-to-Column transformation modules that work on groups of three columns and transpose the columnar data at page-granularity for the 3D rotation to take place on tuples, and finally the core of the perturbation module. We also explain how several perturbation "cores" are run in parallel to ensure 10Gbps line-rate processing.

4.2.1 Snappy Decompression. Parquet files, as well as many other columnar formats, employ lightweight compression by default to save storage capacity. Therefore, we considered it important to demonstrate that the perturbation can be carried out even if the underlying data has been stored in a compressed format. One of the compression algorithms used in Parquet files is Google's Snappy. This algorithm works similarly to LZ77 [38], encoding data as either "literal bytes" or as references to the window of recently emitted data. To implement this algorithm in an FPGA, two main components are required: a state machine to control which data is output (from input or from window) and a BRAM memory module to hold the contents of the data window (as a circular buffer).

Data is received as commands that are decoded one by one and the state machine decides whether the input is a literal byte, and if yes, it will write it to a sliding window memory and to the output of the module at the same time. If it is a copy command, it will look up the word in the memory implementing the sliding window. Herein lies the main challenge of the decompression module, because the BRAM memory introduces additional latency on reads. This results in stalls when previous characters have to be looked up. As a result, even though the throughput of the decompression unit can reach up to 1 B/cycle, with high compression ratios it drops to 0.5 B/cycle. For this reason, several decompression units are deployed in parallel, and pages read from the key-value store are routed to decompression units in a data-parallel fashion. Such an approach increases latency but, with enough parallel units, can guarantee line-rate behavior.

4.2.2 Column/Row Transformation. Similarly to other recent work on in-FPGA processing for ML workloads [5], our implementation has to also transform data from a column-oriented layout, well suited for storage and compression, to tuples, which are used in the actual ML operations and are manipulated by the rotation. To maintain the same schema as the original data, the perturbed tuples have to be re-transformed to columns. In our case, the perturbation takes place on groups of three columns. For this reason, our implementation is optimized for the case of receiving pages belonging to one of three columns and producing a tuple with three attributes.

The Column To Row module's internal architecture is presented in Figure 5a. Data is read from the value storage area (which maps to DDR memory in Caribou on our prototyping FPGA platform) in 512 bit words. Pages are stored as individual key-value pairs and, for perturbation purposes, groups of pages of a three column set are retrieved. The pages are accessed in an interleaved fashion, as shown in the figure, and stored in three buffers. The buffers should be deep enough to hold an entire page of each Parquet column (in our prototype this size is 2KB). Since the pages do not have equal lengths, we implemented a mechanism for handling "leftovers" in the buffers from larger pages and dealing with potentially variable length metadata at the beginning of pages.

The design of the Column To Row module does not require the clients to retrieve entire columns but access is more efficient (and requires no additional client-side or FPGA-side state) if retrieval is done such that the aggregate number of items retrieved for each column is equal, that is, that the end of the pages is aligned.

Once the perturbation is performed, the Row To Column Module takes the rotated 3-vectors and splits them into columns and into pages. Figure 5bshows the architecture

of this transformation module, that is the reverse of the Column to Row module. Once enough data has been enqueued in the first column buffer to form a page, it is emitted.

Even though the row-wise data can be processed without requiring a notion of pages, we maintain the page length information for one of the columns to be able to reconstruct metadata fields at a later step. It has to be noted that the rotation operation itself works on a stream of tuples and is oblivious of page-wise organization of data. Therefore, using smaller or larger pages will not affect in any way the cost or perturbation. Using very small pages, however, could introduce inefficiencies at other layers, such as the TCP/IP stack or the receiving clients.

4.2.3 Perturbation: Random Rotation Operator. Computing on floating point numbers using Verilog/VHDL is fairly complex. When combined with the requirement to implement a matrix to vector multiplication, the task introduces further control flow complications. For this reason, we have designed the matrix-vector multiplication module using high-level synthesis from C++ code (Xilinx HLS). Our goal was to obtain high throughput and low latency for the specific operation of multiplying a 3x3 matrix with a 3-vector. Since other modules in Caribou do not use DSPs (arithmetic units on the FPGA), we can afford using them as a way to achieve this goal, instead of implementing the multiplication using logic resources.

In order to meet the 10Gbps throughput performance requirement, at the default 156.25MHz network clock rate, the perturbation needs to output 8B/cycle. However, the multiplication core only outputs a 3-vector of double-precision floating-point numbers every 18 clock cycles $8 * 3/18 = 1.33\ B/cycle$. To increase the throughput to the desired level, we employ data level parallelism by instantiating six matrix-vector multiplication modules that work in parallel, thus achieving a throughput of 8 B/cycle, that is, 10Gbps line-rate.

4.3 Open Challenges

This work is a first step in the exploration of the idea of *Smarter Storage*, and there are several open challenges that we will tackle in future work:

How to manage the perturbation parameters from a global controller? There is rich related work in building frameworks that track and control data accesses in databases and distributed systems. Our proposal is complementary in its nature but, as described in the example use-case (Section 3.1), in practical deployments it will require the presence of a policy management framework to generate and manage rotation matrices (privacy-related meta-data in general) in the key-value store. The same high-level management would also be required to provide access control to the different name-spaces on the KVS to ensure that unauthorized users cannot access perturbation meta-data. Our prototype design exposes API "hooks" for integrating such a controller.

Could one modify the perturbation operation? In this work we chose a specific type of data perturbation operator in its "vanilla" configuration (Section 3). Naturally, there could be several different modifications to the algorithm that make reconstructing the original data more difficult, or that maintains different properties of the dataset. These can likely be achieved by extending the implemented rotation module with other arithmetic steps (e.g, addition, normalization) or by using larger matrices, to intermix more rows at each step. Modifications of this type can be added to our prototype without requiring fundamental changes.

Could one provide Differential Privacy inside the storage? In the presence of a policy management framework, the *Smarter Storage* nodes could also be extended with perturbation operators that provide differential privacy, either directly on the data or after near-data processing has taken place on the FPGA (e.g., after computing group-by aggregates or histograms on the FPGA). This is an achievable goal because the difficulty in providing such functionality is external to the storage and consists of enforcing high level privacy policies and keeping track of privacy budgets.

5 EVALUATION

The evaluation considers the scenario of individual clients storing and retrieving data from the key-value store, simulating the behavior of cloud tenants running machine learning jobs. Even though a client can have many parallel tasks, it will still be heavily impacted by the data retrieval rate from the storage nodes.

For the client machine we use a dual-socket server with Intel Xeon Silver 4114 CPUs and 10Gbps networking. Caribou is running on a Xilinx VCU1525 board which has a Virtex UltraScale+ FPGA and 10Gbps networking. The board provides 64GB of DDR4 memory that we use for the purposes of storage. Since in this work we only modify the value processing pipeline of Caribou, as long as our modules can handle 10Gbps-line-rate traffic, our design inherits the 10Gbps-line-rate behavior of Caribou for both Sets and Gets [20].

5.1 Inside the FPGA

The privacy pipeline inside the FPGA consists of four operations. First, an optional decompression step, followed by column-to-row conversion, for pages in groups of three, then rotation, and finally row-to-column conversion. In the following we discuss the bandwidth of these modules as a function of page sizes. Since these modules provide streaming operations with a fixed overhead per input page, their performance is independent of the total file size and is only impacted by the page sizes. In the following, when we report

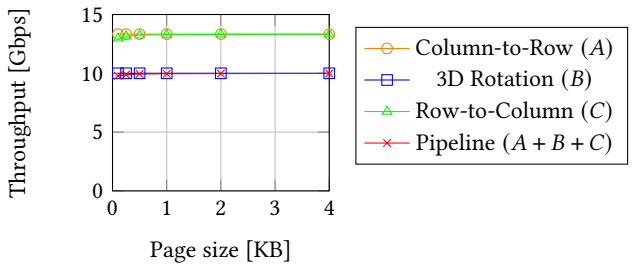


Figure 6: All privacy-related processing modules can sustain at least 10Gbps as a soon as the page size is larger than 512 B. Parquet files typically contain multi-KB pages.

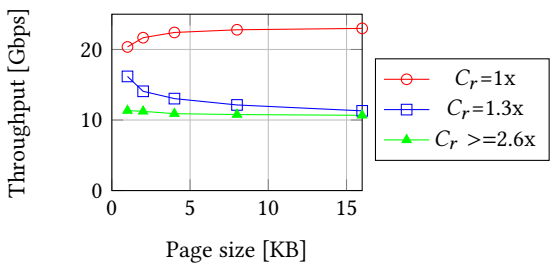


Figure 7: The throughput of the decompression engine (16 parallel modules) depends on the compression ratio (C_r) but it can guarantee a lower bound of 10Gbps

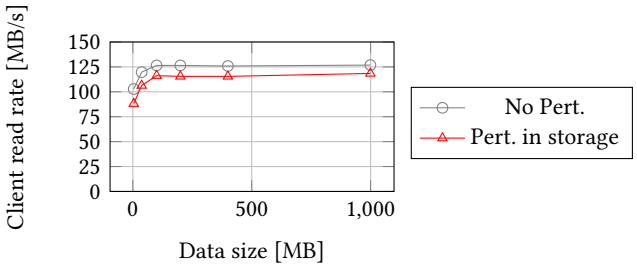


Figure 8: From the client's perspective, using perturbation in storage reduces the observed throughput only marginally.

throughput as a function of page size, this signifies page requests in triplets being issued asynchronously until reaching 4MBs in accumulated data size.

The latency of the modules is negligible when compared to the round-trip times over the network and is linear in the size of the pages (the two conversion modules have to perform a store and forward buffering on three pages at line-rate). To measure the throughput of these modules, we have instrumented the FPGA circuit and measured the throughput of the individual modules using synthetic datasets.

In Figure 6 we show the throughput of the three processing steps inside the Perturbation module. The Column/Row manipulation steps are able to process data at more than 10Gbps for any input size. The rotation step reaches the 10Gbps line-rate requirement as soon as the size of the pages is above 512 B. As a result, the perturbation module as a whole can also sustain 10Gbps data movement. The somewhat lower performance with very small inputs is acceptable because Parquet files typically have page sizes in the KB range. Small pages yield lower throughputs since the pipelines are not filled and we are not able to take full advantage of instruction-level parallelism. Another reason for the lower throughput when the pages are small is the data alignment: if the input pages are small and not aligned to the size of the module inputs, the last block of the pages will have mostly padding bits and few useful ones. The smaller the page size, the more frequent this situation is – however, once the page size reaches 1KB, these overheads become negligible.

In contrast to the other steps, that are content agnostic in their throughput behavior, for the decompression module the ratio of "literal" bytes and those that result in lookups in the sliding window of the Snappy algorithm changes the rate at which output can be produced. Each decompression module will range between 1 and 0.5 B/cycle output rate. For this reason, we deployed 16 parallel decompression modules to guarantee 10Gbps output rate even for data that is very heavily compressed. The results for different page sizes and compression ratios can be seen in Figure 7.

The above results demonstrate that the internal processing pipeline fulfills the 10Gbps line-rate requirement for all realistic Parquet page-sizes and ensures a predictable behavior in the storage node. Given it's streaming interfaces, this perturbation module can be combined with other offloading to take place on the FPGA after its execution.

5.2 From the client's perspective

For end-to-end evaluation, we measure the time it takes for a client to retrieve a perturbed dataset that was previously stored in the key-value store. We use a single-threaded client design to simplify the integration with Python but using multiple threads would be possible (as already mentioned, the throughput of the underlying key-value store reaches 10Gbps line-rate in most operation mixes).

We have compared the case when the key-value storage system performs the data perturbation with the case when the dataset is fetched from the key-value store unperturbed. The cost of storing a Parquet file is not impacted by the perturbation module (Section 4.1.3) and the rotation matrix construction can be done outside the critical path of performance. To run the experiments, we have generated synthetic

Module	CLBs (%)	BRAMs (%)	DSPs (%)
KVS (no offload)	27k (18.4%)	729 (33.7%)	6 (<0.1%)
Decompression	4k (2.9%)	288 (13.3%)	0 (0%)
Perturbation	9k (6.2%)	59.5 (2.7%)	486 (7.1%)
Total used	41k (28.1%)	1076.5 (49.8%)	492 (7.1%)
Total available	147k (100%)	2160 (100%)	6840 (100%)

Table 1: Privacy pipeline (optional decompression and 3D rotation-based perturbation) resource consumption on an XCVU9P FPGA.

input Parquet files by duplicating the entries from the dataset in order to reach files of different sizes. We have disabled compression by default to ensure that all access methods utilize the network bandwidth identically.

Figure 8 shows the read bandwidth that individual clients see when accessing the key-value store, with or without perturbation. As can be seen in the figure, the difference is not significant: the regular read bandwidth is 125MB/s for larger files (around 100MB/s for files smaller than 10MB) and with perturbation enabled, the read bandwidth is 115MB/s (resp. 90MB/s for small files). We do not compare the performance versus performing these operations on the client side because the goal of this work is to demonstrate that such perturbations can be pushed down to the storage node, allowing this way other types of processing to happen after the perturbation inside the same nodes.

Overall, the data rate remained consistent when we varied the size of the data fetched from the key-value store, so the number of requests performed at the key-value store does not influence negatively the processing throughput. Note that since in earlier work we demonstrate that Caribou saturates 10Gpbs bandwidth on the same type of FPGA as used for this experiment, and the previous subsections showed that the privacy pipeline is able to sustain 10Gbps line-rate, we omit experiments with multiple clients for brevity.

5.3 Integration with Python

In order to show that it is possible to integrate our prototype into a Python application that uses Sci-Kit Learn, we have trained several popular machine learning models with a publicly available dataset, the Pima Indians Diabetes Dataset[2] from Kaggle. It combines personal information (e.g., age of a person) with medical details, based on which, predictions can be made whether the person has diabetes. We considered this as an example of existing private data at a company (e.g., insurance company), that can be used to explore ideas of for new applications/insights on the dataset by a product group.

We trained several machine learning classifiers on the original and perturbed data to compare the quality metrics of resulting models. The types of classifiers are: Decision Tree

[2]https://www.kaggle.com/uciml/pima-indians-diabetes-database

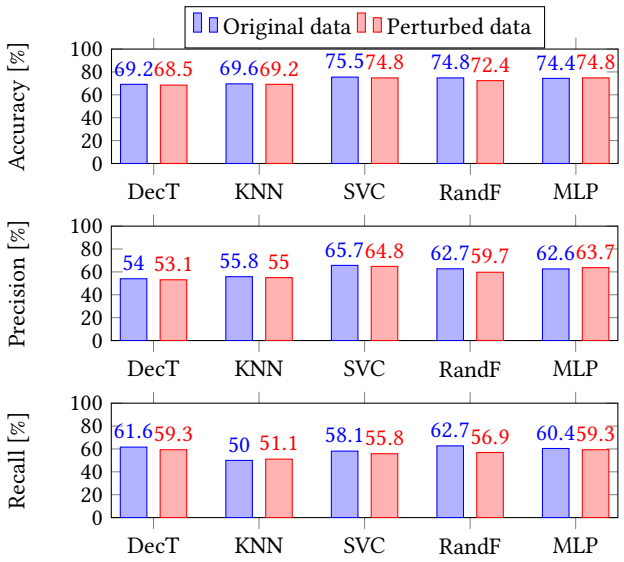


Figure 9: Model quality comparison when using original vs perturbed data

(DecT), K-Nearest Neighbors (KNN), Support Vector (SVC), Random Forest (RandF), Multi-layer Perceptron (MLP).

Figure 9 shows the results of training on (un)perturbed data by a Python application using the Go client and the Sci-Kit Learn library for training. As it can be seen from the graphs, the data perturbation algorithm only minimally impacts all of the models tested for all of the three metrics. In these experiments, training time dominates the runtime: it is several orders of magnitude larger than the time for retrieving data from the storage into the main memory of the nodes. For this reason we omit the training times and, for simplicity, focus on the evaluation of the utility of perturbations.

6 HOW TO DESIGN FUTURE SMARTER STORAGE NODES?

Systems such as BlueDBM [22] and Samsung's SmartSSD [2] demonstrate the benefits of using FPGAs to implement smart storage thanks to the increased energy efficiency and the predictable line-rate behavior. When designing smart storage devices using such specialized hardware, two important questions arise: *Q1)* How many different types of processing can an FPGA-based a platform provide in the future? and *Q2)* How scalable is the underlying design for targeting 40/100Gbps networks that are becoming commodity?

To answer the first question, we need to consider the resource consumption of the 10Gbps prototype. Table 1 shows that, when sized for 10Gbps operation, the privacy pipeline, consisting of decompression and rotation, occupies only a fraction of the Xilinx XCVU9P chip in terms of configurable logic blocks (CLBs). On-chip memory (BRAM) usage is somewhat more accentuated because the decompression module

incorporates large buffers. Together with the KVS logic, the prototype occupies one-third of CLBs and half of BRAMs, allowing for extensions with further, more complex, data perturbation logic or for other types of query offloading.

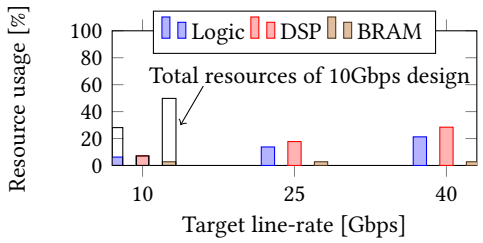
To answer the second question of scaling the modules for faster processing, e.g., 25Gbps or 40Gbps, the main increase in cost would be driven by the number DSPs and BRAMs. In our design, the DSPs are used for the floating-point matrix-vector multiplication, the BRAM is used for buffers and the CLBs are used for the glue logic. For this reason, the number of DSPs used is directly proportional with the number of floating-point operations that we need to simultaneously perform, so if we add more matrix-vector multiplication modules, or if we increase the matrix dimensions, the number of DSPs will increase as well.

We model the expected resource consumption for 25 and 40Gbps line-rates by extrapolating from the cost of individual processing "cores" in our 10Gpbs design. We show the numbers for the rotation perturbation module in Figure 10a and for the decompression module in Figure 10b. For the former, to increase its throughput, the only modification required is to increase the number of rotation cores because the rest of the pipeline, including the row-to-column transformations, operate at a wide-enough bus width to handle 40Gbps load with minimal adjustments compared to the current prototype. For this reason, Figure 10a shows that only the DSP and logic resources would increase visibly.

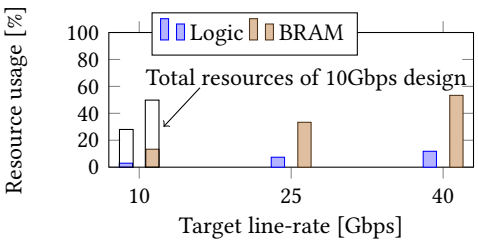
For Decompression, shown in Figure 10b, reaching 40Gbps operation is more expensive. Throughput can be scaled by deploying more parallel cores but this results in a steep increase in BRAM usage. This is because each decompression core has both large input buffers and internal memory for storing the sliding window of the algorithm.

If targeting faster networks, it will already become challenging to fit the KVS and the privacy pipeline modules on the FPGA, with little space left for additional processing. This would be a limiting bottleneck even if the frequency of the modules would be increased by a factor of 2 (reducing their resource footprint by 2x). In our opinion, one way forward could involve FPGAs that also incorporate CPU cores, such as the Xilinx MPSoC family, or devices with more powerful DSPs and ALUs, such as the recently announced Xilinx Versal [35]. In such a device, one could use small CPU cores or more capable DSPs for performing the matrix operations required for the perturbation, and perhaps decompression, and dedicate the FPGA resources to more packet-oriented processing at which they excel.

To evaluate the feasibility of running the perturbation storage-side in software, we have modified the prototype's client. A single thread of our server could perform the rotation operation at a rate of ±190 MB/s, which hints that



(a) Scaling rotation perturbation throughput.



(b) Scaling decompression throughput.

Figure 10: Higher target data rates within the privacy pipeline require a linear increase in resources.

several cores could potentially reach 25Gbps and faster line-rates. Even though ARM cores found, for instance, in the Xilinx MPSoC devices are less powerful than the Xeon cores in our server, it is realistic to assume that by using multi-threading and further optimizations, software running close to the FPGA fabric could be used for perturbation operations.

7 CONCLUSION

In this work, we highlight the importance of including privacy-related offloading in future programmable Smart Storage solutions. We show the feasibility of building such Smarter Storage with specialized hardware, namely FPGAs. We prototyped our idea by extending an open-source FPGA-based key-value store and showed that, in addition to the usual key-value management and query offload functionality, it is possible to implement a rotation-based data perturbation operator that runs at the 10Gbps network line-rate.

When targeting 10Gbps networks, the perturbation module and leaves enough real estate free on the device to be combined with other types of more traditional query processing inside the FPGA. For faster network speeds we have identified the expected FPGA-resource bottlenecks and discuss alternative design options.

Even though this work focuses on a specific perturbation type, the underlying matrix-vector multiplication operation is representative for a wide range of perturbations. The internal operations of the privacy pipeline on the FPGA can be considered as a blueprint for implementing other, similar, privacy operator offload in the future.

ACKNOWLEDGMENTS

We would like to thank Xilinx for their generous donation of software and hardware. The project has been partially funded by the Novo Nordisk Foundation (NNF20OC0064411).

REFERENCES

[1] Advanced query accelerator (aqua) for amazon redshift (accessed 03/06/2020). https://pages.awscloud.com/AQUA-Preview.html.

[2] Samsung smartssd computational storage drive (accessed 03/06/2020). https://samsungsemiconductor-us.com/smartssd/index.html.

[3] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. *ACM SIGPLAN Notices*, 33(11), 1998.

[4] C. C. Aggarwal and P. S. Yu. A condensation approach to privacy preserving data mining. In *Advances in Database Technology - EDBT 2004*, pages 183–199, Berlin, Heidelberg, 2004. Springer.

[5] G. Alonso, Z. Istvan, K. Kara, M. Owaida, and D. Sidler. doppiodb 1.0: Machine learning inside a relational engine. *IEEE DE Bull*, 42(2), 2019.

[6] E. Arfelt, D. Basin, and S. Debois. Monitoring the gdpr. In *European Symposium on Research in Computer Security*, pages 681–699. Springer, 2019.

[7] K. Chapman, M. Nik, B. Robatmili, S. Mirkhani, and M. Lavasani. Computational storage for big data analytics. In *Tenth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS'19)*, 2019.

[8] K. Chen and L. Liu. A random rotation perturbation approach to privacy preserving data classification. 2005.

[9] K. Chen and L. Liu. Geometric data perturbation for privacy preserving outsourced data mining. *Knowledge and information systems*, 29(3):657–695, 2011.

[10] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226, 2016.

[11] B. Ding, J. Kulkarni, and S. Yekhanin. Collecting telemetry data privately. In *Advances in Neural Information Processing Systems 30*, December 2017.

[12] J. Do, Y.-S. Kee, J. M. Patel, et al. Query processing on smart SSDs: opportunities and challenges. In *SIGMOD'13*.

[13] J. Domingo-Ferrer and J. M. Mateo-Sanz. Practical data-oriented microaggregation for statistical disclosure control. *IEEE Transactions on Knowledge and Data Engineering*, 14(1):189–201, 2002.

[14] R. Durstenfeld. Algorithm 235: Random permutation. *Communications of the ACM*, 7(7):420, 1964.

[15] C. Dwork. Differential privacy. *Automata, Languages and Programming*, pages 265–284, 2006.

[16] C. Dwork, A. Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[17] U. Erlingsson, V. Pihur, and A. Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1054–1067, 2014.

[18] IBM. IBM Netezza data warehouse appliances, 2012. http://www.ibm.com/software/data/netezza/.

[19] Z. István, G. Alonso, and A. Singla. Providing multi-tenant services with FPGAs: Case study on a key-value store. In *FPL'18*, pages 119–124, 2018.

[20] Z. István, D. Sidler, and G. Alonso. Caribou: intelligent distributed storage. *PVLDB*, 10(11):1202–1213, 2017.

[21] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *PVLDB*, 9(12), 2016.

[22] S. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, and S. X. and. Bluedbm: An appliance for big data analytics. In *ISCA'15*, pages 1–13, 2015.

[23] H. Kargupta, S. Datta, Q. Wang, and K. Sivakumar. On the privacy preserving properties of random data perturbation techniques. In *Third IEEE international conference on data mining*, pages 99–106. IEEE, 2003.

[24] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–17, 2018.

[25] L. Kuhring, E. Garcia, and Z. István. Specialize in moderation—building application-aware storage services using fpgas in the datacenter. In *11th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[26] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: high-performance in-memory key-value store with programmable nic. In *SOSP'17*, pages 137–152, 2017.

[27] F. D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 19–30, 2009.

[28] A. Mehta, E. Elnikety, K. Harvey, D. Garg, and P. Druschel. Qapla: Policy compliance for database-backed systems. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1463–1479, 2017.

[29] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler. Gupt: privacy preserving data analysis made easy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 349–360, 2012.

[30] K. Muralidhar, R. Parsa, and R. Sarathy. A general additive data perturbation method for database security. *Management Science*, 45(10):1399–1415, 1999.

[31] S. R. Oliveira and O. R. Zaiane. Privacy preserving clustering by data transformation. *Journal of Information and Data Management*, 1(1):37–37, 2010.

[32] A. Pretschner, M. Hilty, and D. Basin. Distributed usage control. *Communications of the ACM*, 49(9):39–44, 2006.

[33] B. Salami, G. A. Malazgirt, O. Arcas-Abella, A. Yurdakul, and N. Sonmez. AxleDB: A novel programmable query processing platform on FPGA. *Microprocessors and Microsystems*, 51:142–164, 2017.

[34] S. Upadhyay, C. Sharma, P. Sharma, P. Bharadwaj, and K. Seeja. Privacy preserving data mining with 3-d rotation transformation. *Journal of King Saud University-Computer and Information Sciences*, 30(4):524–530, 2018.

[35] K. Vissers. Versal: The xilinx adaptive compute acceleration platform (acap). In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 83–83, 2019.

[36] F. Wang, R. Ko, and J. Mickens. Riverbed: enforcing user-defined privacy constraints in distributed web services. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 615–630, 2019.

[37] S. Xu, S. Lee, S.-W. Jun, M. Liu, J. Hicks, et al. BlueCache: A scalable distributed flash-based key-value store. *PVLDB*, 10(4):301–312, 2016.

[38] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3), 1977.