The VLDB Journal manuscript No.
(will be inserted by the editor)

Stochastic Gradient Descent without Full Data Shuffle

with Applications to In-Database Machine Learning and Deep Learning Systems

Lijie Xu · Shuang Qiu · Binhang Yuan · Jiawei Jiang · Cedric Renggli · Shaoduo Gan · Kaan Kara · Guoliang Li · Ji Liu · Wentao Wu · Jieping Ye · Ce Zhang

Received: date / Accepted: date

Abstract Modern machine learning (ML) systems commonly use Stochastic Gradient Descent (SGD) to train ML models. However, SGD relies on random data order to converge, which usually requires a full data shuffle. For in-DB ML systems and deep learning systems with large datasets stored on *block-addressable secondary storage* such as HDD and SSD, this full data shuffle leads to low I/O performance — the data shuffling time can be even longer than the training itself, due to massive random data accesses. To balance the convergence rate of SGD (which favors data randomness) and its I/O performance (which favors sequential access), previous work has proposed several data shuffling strategies.

In this paper, we first perform an empirical study on existing data shuffling strategies, showing that these strategies suffer from either low performance or low convergence rate. To solve this problem, we propose a simple but novel *two-level* data shuffling strategy named `CorgiPile`, which can *avoid* a full data shuffle while maintaining *comparable* convergence rate of SGD as if a full shuffle were performed. We further theoretically analyze the convergence behavior of `CorgiPile` and empirically evaluate its efficacy in both in-DB ML and deep learning systems. For in-DB ML systems, we integrate `CorgiPile` into PostgreSQL by introducing three new *physical* operators with optimizations. For deep learning systems, we extend single-process `CorgiPile` to multi-process `CorgiPile` for the parallel/distributed environment and integrate it into PyTorch. Our evaluation shows that `CorgiPile` can achieve comparable convergence rate with the full-shuffle based SGD for both linear models and deep learning models. For in-DB ML with linear models, `CorgiPile` is 1.6×-12.8× faster than two state-of-the-art systems, Apache MADlib and Bismarck, on both HDD and SSD. For deep learning models on ImageNet, `CorgiPile` is 1.5× faster than PyTorch with full data shuffle.

Keywords In-Database Machine Learning, Stochastic Gradient Descent, Data Shuffle, Deep Learning

· Lijie Xu, Cedric Renggli, Shaoduo Gan, Kaan Kara,
ETH Zürich, {firstname.lastname}@inf.ethz.ch
· Shuang Qiu, Binhang Yuan,
Hong Kong University of Science and Technology
· Jiawei Jiang, Wuhan University
· Guoliang Li, Tsinghua University
· Ji Liu, Meta
· Wentao Wu, Microsoft Research
· Jieping Ye, University of Michigan
· Ce Zhang, University of Chicago

1 Introduction

Stochastic gradient descent (SGD) is a popular iterative optimization algorithm that has been widely used in machine learning systems. With the growing data volume, SGD algorithms have to access the data stored on the *secondary storage* instead of main memory. There are two prominent scenarios: (1) *In-database machine learning* (in-DB ML) systems have to access the data tables stored on the secondary storage via the buffer manager [26]; (2) *Deep learning (DL) systems* such as TensorFlow [30] and PyTorch [68] need specialized data loader/scanner to access the datasets that are stored in parallel/distributed file systems.

In-database ML and deep learning systems In-DB ML systems and deep learning systems have been extensively studied for many years [51,43,73,58,67,37,71,57,61]. For in-DB ML, its major benefit is that users do not need to move the data out of DB to another specialized ML platform, since it is often time-consuming or even infeasible (due to privacy or security concerns). With the help of in-DB ML systems such as MADlib [2,51] and Bismarck [43], users can train an ML model (e.g., SVM) using a simple SQL query as follows:

```
SELECT svm_train(table_name, parameters).
```

For deep learning systems such as PyTorch and TensorFlow, they usually provide users with simple `Dataset/DataLoader` APIs to load data from secondary storage into memory and further into GPUs, as shown in the following lines of code. Deep learning systems can automatically perform model training in `train()` with SGD, using a number of GPUs.

```
train_data = Dataset(dataset_path, args)
train_loader = DataLoader(train_data, args)
train(train_loader, model, args)
```

A Fundamental Discrepancy A fundamental problem is that SGD requires random data order to converge, but the data is usually not guaranteed to be stored in a random order, for both in-DB ML and deep learning systems. As identified by previous work [43,84,55], the worst case is that the data is stored in a *clustered* order. For example, if the data is clustered by labels, data with negative labels might always come before data with positive labels [43]. Another example is that the data is ordered/clustered by one of the features. There are common cases when the data is naturally ordered by some features such as timestamps, usernames/types, item prices, or there is a clustered B-tree index on a subset of the feature columns or the label column (if data is stored in a relational database). In these cases, directly running sequential scans over the clustered data can slow down the convergence of SGD.

A common solution is to perform a full data shuffle on the original data. However, when data is stored on *block-addressable* secondary storage such as HDD and SSD, it can be extremely time-consuming to either randomly access the data during SGD, or shuffle the data *once* with data copy and run SGD over the shuffled copy, due to massive *random* I/O's. For example, shuffling a 50GB dataset in PostgreSQL using '`ORDER BY RANDOM()`' took about 50 mins in our experiments, and shuffling a scalability dataset in DB did not finish even in one day, as reported by previous work [43]. Moreover, sometimes, it is infeasible to shuffle the data in DB — in-place shuffling might have an impact on other indices, whereas shuffling over a data copy leads to $2\times$ storage overhead. Likewise, the parallel/distributed file systems such as HDFS and Lustre [69] do not support/recommend randomly accessing small data tuples, which will significantly degrade the I/O performance. *How efficient SGD algorithms can be designed without requiring even a single pass of full data shuffle?* Understanding this question can have a profound impact on the system design of both in-DB ML and deep learning systems.

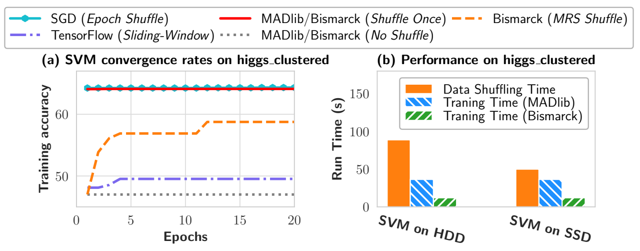


Fig. 1: The convergence rate and performance of SVM on the *higgs* dataset clustered by labels, with different data shuffling strategies. (a) Today's ML systems, including in-DB ML systems (e.g., MADlib and Bismarck) and TensorFlow, are sensitive to the data order. (b) Forcing a full data shuffle before training accommodates this clustered data issue, but introduces large overhead that is often more expensive than training itself.

Existing Landscape and Challenges To solve the data shuffling problem of SGD, previous work has proposed several data shuffling strategies in the context of in-DB ML or deep learning systems. TensorFlow adopts a *sliding-window* based shuffling strategy, which constantly loads data into a buffer and randomly fetches data from the buffer for SGD [19]. Bismarck [43] proposes a "multiplexed reservoir sampling" (MRS) shuffling strategy, which leverages two threads to update the model concurrently. One thread reads the data sequentially with reservoir sampling, while the other thread reads data from a small in-memory buffer filled with the sampled data. Although these strategies improve the I/O performance, they suffer from convergence shortcomings. As demonstrated in Figure 1(a), both strategies proposed by Bismarck and TensorFlow suffer from lower accuracy given a *clustered* data. In contrast, shuffling data once *before* training, i.e., the curve corresponding to "MADlib/Bismarck (*Shuffle Once*)", can accommodate such convergence problem but introduce a significant overhead as shown in Figure 1(b).

Our Contributions Inspired by these previous efforts, we ask the following questions in this paper:

Can we design an SGD-style algorithm with efficient data shuffling strategy that can converge without requiring a full data shuffle? Can we provide a rigorous theoretical analysis on the convergence behavior of such an algorithm? Further, can we integrate such an algorithm into database systems as well as deep learning systems?

In this paper, we systematically study these questions and make the following contributions.

C1. An Anatomy and Empirical Study of Existing Algorithms. We first conduct a systematic empirical study of existing data shuffling strategies for SGD, including (1) *Epoch Shuffle*, which performs a full shuffle before each epoch, (2) *Shuffle Once*, (3) *No Shuffle*, (4) *Sliding-Window Shuffle*, and (5) *MRS Shuffle*. We compare them by using SGD to train generalized linear models and deep learning models, over both label-clustered and feature-ordered datasets. Our study reveals that existing strategies cannot *simultaneously* achieve good hardware efficiency (i.e., I/O performance) and statistical efficiency (i.e., convergence rate and converged accuracy). Specifically, *Epoch Shuffle* and *Shuffle Once* achieve the best statistical efficiency in terms of convergence rate of SGD, since the data has been fully shuffled; however, their hardware efficiency is suboptimal due to additional shuffle overhead and storage overhead. In contrast, *No Shuffle* achieves the best hardware efficiency as no data shuffle is required; however, its statistical efficiency suffers as it has the lowest accuracy. The other two strategies, *Sliding-Window Shuffle* and *MRS Shuffle*, perform like a compromise between *Shuffle Once* and *No Shuffle*, but still suffer in terms of statistical efficiency (Section 3).

C2. A Simple but Novel Algorithm with Rigorous Theoretical Analysis. To address the limitations of existing strategies, we propose `CorgiPile`, a novel SGD-style algorithm with a *two-level hierarchical* data shuffling strategy.[1] The main idea is to first sample and shuffle the data at *block level*, and then shuffle data at *tuple level* within the sampled data blocks. That is, we first randomly sample data blocks (e.g., one block refers to a batch of table pages in DB) into a buffer, and then shuffle the tuples from all the blocks in the buffer for SGD. While this two-level strategy seems simple, it can achieve both good hardware efficiency and statistical efficiency. The hardware efficiency is intuitive—randomly accessing data blocks is much faster than randomly accessing small tuples, especially for large block size. However, the statistical efficiency requires some non-trivial analysis. To this end, we further present a rigorous theoretical study on the convergence behavior of `CorgiPile`.

C3. Implementation, Optimization, and Deep Integration with PostgreSQL. For in-DB ML, we aim to integrate `CorgiPile` with PostgreSQL, which requires careful design, implementation, and optimization. Unlike previous in-DB ML systems such as MADlib and Bismarck that integrate ML algorithms using User-Defined Aggregates (UDAs), our technique requires a deeper system integration since it needs to directly interact with the buffer manager and pages. Therefore, we operate at the "physical level" and enable in-DB ML inside PostgreSQL [17] via three new *physical operators*: a `BlockShuffle` operator, a `TupleShuffle` operator, and an `SGD` operator for our customized SGD implementation. We can then construct an *execution plan* for the SGD computation by chaining these operators together to form a pipeline, which naturally follows the built-in Volcano query execution paradigm [45] of PostgreSQL. We also design a *double-buffering* mechanism to optimize the `TupleShuffle` operator, to reduce the data copy and shuffle overhead[2].

C4. Multi-Process `CorgiPile` and Integration with PyTorch. Today's deep learning systems usually work in the parallel/distributed environment with multiple processes and GPUs. To adapt to this environment, we further extend single-process `CorgiPile` to multi-process `CorgiPile`, by enhancing the tuple-level shuffle. The multi-process `CorgiPile` also contains three operators: `BlockShuffle`, `TupleShuffle`, and `SGD`. For block-level shuffling, we randomly distribute data blocks to different processes. For tuple-level shuffling, we use multi-buffer based shuffling instead of single-buffer based shuffling—in each process we allocate a local buffer to read blocks and shuffle their tuples. The `SGD` operator performs mini-batch SGD and *synchronizes* the computation of gradients/parameters among different processes for each batch. We demonstrate that multi-process `CorgiPile` generates random data order similar to that of single-process `CorgiPile`. We further integrate multi-process `CorgiPile` into PyTorch and wrap it as a new `CorgiPileDataset` API for ease of use[3]

C5. Comprehensive Empirical Evaluations. We perform comprehensive evaluations to demonstrate hardware efficiency and statistical efficiency of `CorgiPile`. For in-DB ML, we compare our PostgreSQL implementation with two state-of-the-art in-DB ML systems, Apache MADlib and Bismarck, in terms of both convergence rate and end-to-end performance. The results show that `CorgiPile` achieves comparable model accuracy to the best *Shuffle Once* baseline on both *label-clustered* and *feature-ordered* datasets. Meanwhile, `CorgiPile` gains 1.6×-12.8× speedup compared to MADlib and Bismarck, since it does not require the full data shuffle. In contrast, other strategies suffer from lower convergence rate or lower accuracy. For deep learning systems,

[1] Although we have an unquestionable love for dogs, the name of `CorgiPile` comes from the shuffling strategy that is a combination of *pile shuffle* and *corgi shuffle*, two commonly used strategies to shuffle a deck of cards.

[2] The code of `CorgiPile` in PostgreSQL is available at `https://github.com/DS3Lab/CorgiPile-PostgreSQL`.

[3] The code of `CorgiPile` in PyTorch is available at `https://github.com/DS3Lab/CorgiPile-PyTorch`.

we compare CorgiPile with other shuffling strategies in PyTorch using deep learning models for image classification. The results are similar to those in PostgreSQL—CorgiPile in PyTorch again achieves similar model accuracy compared to the (best) *Shuffle Once* baseline, while other data shuffling strategies result in lower accuracy. Specifically, on ImageNet CorgiPile is 1.5× faster than *Shuffle Once* to converge, using 8 GPUs.

This paper is an extension of our previous work [79]. Our new contributions beyond [79] are the following:

– To enable CorgiPile to work in a parallel/distributed environment, we extend the previous single-process CorgiPile to multi-process CorgiPile (Section 6). We further implement multi-process CorgiPile in PyTorch, by enhancing two operators BlockShuffle and TupleShuffle and wrapping them as a new CorgiPileDataset API.
– We demonstrate that multi-process CorgiPile can obtain random data order similar to that of single-process CorgiPile for mini-batch SGD (Section 6.3).
– We evaluate the multi-process CorgiPile with deep learning models for image classification (Section 7.3). On ImageNet [10], CorgiPile achieves comparable model accuracy to the (best) *Shuffle Once* baseline but is 1.5× faster to converge.
– We expand the evaluation in [79] with datasets ordered by subsets of features. The new results reinforce the result that CorgiPile is comparable with *Shuffle Once*, whereas the other approaches suffer from lower accuracy and/or lower convergence rate.
– We extend the evaluation in [79] by comparing convergence rate among different shuffling strategies for mini-batch linear models such as LR and SVM.

Paper Organization We first review the SGD algorithm and its implementation in Section 2. We next perform an empirical study on the existing data shuffling strategies for SGD in Section 3. We then present our CorgiPile strategy and provide a theoretical analysis on its convergence in Section 4. We detail our implementation of CorgiPile inside PostgreSQL in Section 5. We present the multi-process CorgiPile and its implementation within PyTorch in Section 6. We compare the end-to-end performance and convergence rate of CorgiPile with other baseline approaches in Section 7. We summarize related work in Section 8 and conclude in Section 9.

2 Preliminaries

In this section, we briefly review the standard SGD algorithm and its implementation in state-of-the-art in-DB ML and deep learning systems.

2.1 Stochastic Gradient Descent (SGD)

Given a dataset with m training examples $\{\mathbf{t}_i\}_{i\in[m]}$, i.e., m tuples if these examples are stored as a table in a database, typical ML tasks essentially solve an optimization problem of minimizing a finite sum over m training examples with respect to model $\mathbf{x}$.

$$F(\mathbf{x}) = \frac{1}{m}\sum\nolimits_{i=1}^{m} f_i(\mathbf{x})$$

Here, each f_i corresponds to the loss over each training tuple $\mathbf{t}_i$. SGD is an *iterative* algorithm that takes as input hyperparameters such as the learning rate η and the maximum number of epochs S. It works as follows:

1. **Initialization** – Initialize the parameters of model $\mathbf{x}$, often randomly or as zero.
2. **Iterative computation** – In each iteration, it draws a (batch of) tuple $\mathbf{t}_i$, *randomly with replacement*, computes the stochastic gradient $\nabla f_i(\mathbf{x})$ and updates the parameters of model $\mathbf{x}$. In practice, most systems implement a more efficient variant, where the random tuples are drawn *without replacement* [34,43,40,83]. To achieve this, SGD *shuffles* all tuples *before* each epoch and sequentially scans these shuffled tuples. For each tuple, SGD computes the stochastic gradient and updates the model parameters.
3. **Termination** – The iterative computation ends when it converges (i.e., the parameters of model $\mathbf{x}$ no longer change) or has attained the pre-defined maximum number of epochs.

2.2 In-database Machine Learning Systems

There has been a plethora of work in the past decade focusing on in-DB ML [84,51,43,73,58,67,37,71,57,61,52,62,82,55]. Most existing in-DB ML systems implement SGD algorithm using "User-Defined Aggregates" (UDA) [43,51]. In detail, each epoch of SGD is done via an invocation of the corresponding UDA function, where the parameters of model $\mathbf{x}$ are treated as the *state* and updated for each tuple.

To implement the data shuffling step required by SGD, different in-DB ML systems adopt distinct strategies. For example, some systems such as MADlib [51] and DB4ML [53] assume that the training data has already been shuffled, so they do not perform any data shuffling. Other systems, such as Bismarck [43], do not make this assumption. Instead, they either perform a *pre-shuffle* of the data in an offline manner and then store the shuffled data as a replica in the database, or perform *partial* data shuffling based on sampling technologies such as *reservoir sampling* and *sliding-window* *sampling*. As we will see in the next section, such partial data shuffling strategies, despite alleviating the computation and storage overhead of the preshuffle strategy, raise new issues regarding the convergence of SGD, since the data is insufficiently shuffled and does not follow the purely random order required.

2.3 Deep Learning Systems

Deep learning systems such as PyTorch and TensorFlow are now widely used in industry and academia for AI tasks, including image classification, natural language processing, speech recognition, etc. These systems usually leverage the SGD optimizer or its variants [72, 23, 24] for training deep learning models. To facilitate data loading, these systems classify datasets into two types, namely, *map-style* datasets and *iterable-style* datasets. Map-style datasets refer to datasets whose tuples can be randomly accessed by indices. For example, if an image dataset is stored in an in-memory array as $\langle$`image`, `label`$\rangle$ tuples, it is a map-style dataset that can be randomly accessed by the array index. Iterable-style datasets refer to datasets that can only be accessed in sequence, which is usually used for datasets that cannot fit in memory. It is easy to shuffle map-style datasets since we only need to shuffle the indices and access the tuples based on the shuffled indices. However, if map-style datasets are stored on secondary storage such as HDD/SSD, this random access usually leads to low I/O performance (see Figure 4). To alleviate this problem, TensorFlow provides *sliding-window shuffle* using sliding-window based sampling. As we will see in Section 3, sliding-window shuffle often results in low accuracy of the trained model.

3 Study of Data Shuffling Strategies for SGD

In this section, we present a systematic analysis of data shuffling strategies used by existing ML systems. We consider five common data shuffling strategies: (1) *Epoch Shuffle*, (2) *Shuffle Once*, (3) *No Shuffle*, (4) *Sliding-Window Shuffle* [19], and (5) *MRS Shuffle* [43]. We use diverse SGD workloads, including generalized linear models such as logistic regression (LR) and support vector machine (SVM), as well as deep learning models such as VGG [75] and ResNet [50].

Experimental Setups. We use the `criteo` dataset [13] for generalized linear models, and use the `cifar-10` image dataset [4] for deep learning. Each dataset has two versions: a *label-clustered* (or *clustered* for short) version and a *feature-ordered* version. In the *clustered* version, all tuples are clustered by their *labels*, whereas in the *feature-ordered* version all tuples are ordered by the first feature without loss of generality—we observed similar results by ordering other features, as shown in [79](ref. Section 6.3.3). The usage of *clustered* datasets is inspired by similar settings leveraged in [43], with the goal of testing the worst-case scenarios of data shuffling strategies for SGD. For example, the clustered version of `criteo` dataset has the *negative* tuples (with "-1" labels) ordered before the *positive* tuples (with "+1" labels).

3.1 "Shuffle Once" and "Epoch Shuffle"

The *Shuffle Once* strategy performs an offline shuffle of all data tuples, either in-place or by storing the shuffled tuples as a copy. SGD is then executed over this shuffled copy. Albeit a simple (but costly) idea, it is arguably a strong baseline that many state-of-the-art in-DB ML systems assume when they take as input an already shuffled dataset. For *Epoch Shuffle*, it shuffles the training dataset before each training epoch. Therefore, the data shuffling cost of *Epoch Shuffle* grows *linearly* with respect to the number of epochs.

Convergence. As illustrated in Figure 2, *Shuffle Once* can achieve a convergence rate comparable to *Epoch Shuffle* on *both* clustered and feature-ordered datasets, confirming previous observations [43].

Performance. Although *Shuffle Once* reduces the number of data shuffles to only once, the shuffle itself can be very expensive on large datasets due to the random access of tuples, as we will show in our experiments. Previous work has also reported that shuffling a huge dataset could not be finished in one day [43]. Another problem of *Shuffle Once* is that, when in-place shuffle is not feasible, it needs to duplicate the data, which can double the space overhead.

3.2 "No Shuffle"

The *No Shuffle* strategy does not perform any data shuffle at all, i.e., the SGD algorithm runs over the given data order in each epoch. Running MADlib over a dataset or running PyTorch over an iterable-style dataset (`IterableDataset`) picks the *No Shuffle* strategy.

Convergence. On both *clustered* and *feature-ordered* data, *No Shuffle* suffers from the lowest model accuracy. This is not surprising, as SGD relies on random data order to converge.

Performance. *No Shuffle* is the fastest among the five data shuffling strategies, as it can always *sequentially*, instead of randomly, access the data tuples [31].

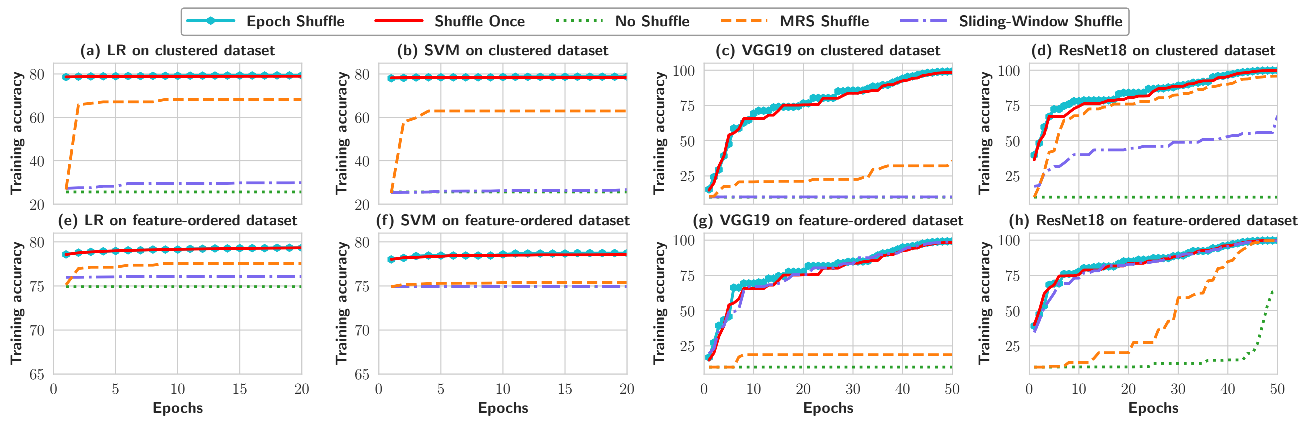


Fig. 2: The convergence rates of SGD with different data shuffling strategies for both label-clustered and feature-ordered datasets, using the same buffer size (10% of the dataset size) for MRS and Sliding-Window Shuffles. LR and SVM use the standard SGD, while VGG19 and ResNet18 use mini-batch SGD with batch size of 64.

3.3 "Sliding-Window Shuffle"

The *Sliding-Window Shuffle* strategy leverages a sliding window to perform *partial* data shuffling, which is used by TensorFlow [19]. It includes the following steps:

1. Allocate a sliding window and fill tuples into the window as they are scanned.
2. Randomly select a tuple from the window and use it for the SGD computation. The slot of the selected tuple in the window is then filled in by the next incoming tuple.
3. Repeat (2) until all tuples are scanned.

Convergence. As illustrated in Figure 2, for clustered datasets, *Sliding-Window Shuffle* can achieve higher model accuracy than *No Shuffle* but lower accuracy than *Shuffle Once* when SGD converges. The reason is that this strategy shuffles the data only *partially*. For two data examples $\mathbf{t}_i$ and $\mathbf{t}_j$ where $\mathbf{t}_i$ is stored much earlier than $\mathbf{t}_j$ ($i \ll j$), it is likely that $\mathbf{t}_i$ is still selected before $\mathbf{t}_j$. As a result, on the clustered datasets used in our study, negative tuples are more likely to be selected (for SGD) before positive ones, which distorts the training data seen by SGD and leads to low model accuracy. This accuracy degradation also happens in feature-ordered datasets when training LR and SVM. For VGG and ResNet on the feature-orded `cifar-10` dataset, *Sliding-Window Shuffle* achieves only 0.3%-0.4% lower accuracy than *Epoch Shuffle* and *Shuffle Once*. The reason is that ordering the feature pixels of `cifar-10` images results in good data randomness.

Performance. *Sliding-Window Shuffle* can achieve I/O performance comparable to *No Shuffle*, as it also only needs to *sequentially* access the data tuples with limited additional CPU overhead to maintain and sample from the sliding window.

3.4 "Multiplexed Reservoir Sampling Shuffle"

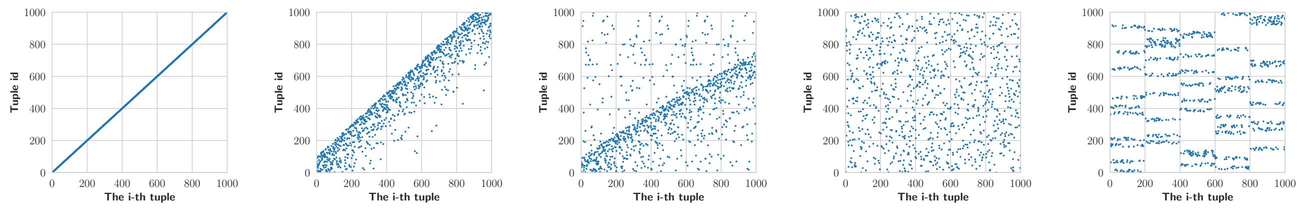
Multiplexed Reservoir Sampling (MRS) Shuffle uses two concurrent threads to read tuples and update a shared model [43]. The first thread sequentially scans the dataset and performs the *reservoir sampling*. The *sampled* (i.e., selected) tuples are stored in a buffer B_1, and the *dropped* (i.e., not selected) ones are used for SGD. The second thread loops over the tuples from another buffer B_2 for SGD, where tuples in B_2 are simply copied from B_1 (by swapping B_1 and B_2 once reservoir sampling ends). PyTorch's shuffling strategy uses pure reservoir sampling [29], which is a weaker version of (i.e., has lower convergence rate than) *MRS Shuffle*, as detailed in Section 3.4 in [43]. Therefore, we use *MSR Shuffle* instead of reservoir sampling as one baseline.

Convergence. As illustrated in Figure 2, *MRS Shuffle* achieves higher accuracy than *Sliding-Window Shuffle* but lower accuracy than *Shuffle Once* for clustered datasets. The reason is quite similar to that given to *Sliding-Window Shuffle*, as the shuffle based on reservoir sampling is again *partial*. Specifically, the order of the *dropped* tuples is also *increasing*, i.e., if $i \ll j$, $\mathbf{t}_i$ is likely to be processed by SGD before $\mathbf{t}_j$. Moreover, looping over the sampled tuples may lead to suboptimal data distribution—the sampled tuples in the looping buffer B_2 may be used multiple times, which can cause data skew and lower model accuracy (e.g., the accuracy of VGG/ResNet on the feature-ordered `cifar-10` dataset).

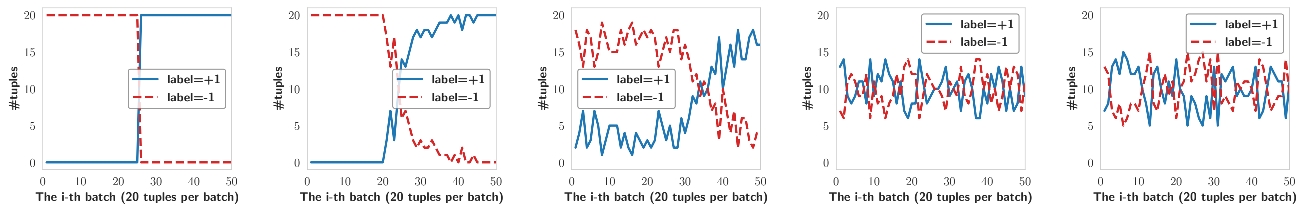
Performance. *MRS Shuffle* is fast, as the first thread only needs to *sequentially* scan the tuples for reservoir sampling. However, it is slightly slower than *Sliding-Window Shuffle* and *No Shuffle*, as there is a second thread that loops over the buffered tuples.

Table 1: A Summary of Different Data Shuffling Strategies, where **bold** fonts represent the "ideal" scenario. We assume all strategies that require an in-memory buffer have reasonably large buffer size, e.g., 10% of dataset size.

Shuffling Strategy	**Convergence Behavior**	**I/O Perf.**	**Buffer**	**Additional Disk Space**
No Shuffle	Slow; Lower Accuracy	**Fast**	**No**	**No**
Epoch Shuffle	**Fast; High Accuracy**	Slow	Yes	2× data size
Shuffle Once	**Fast; High Accuracy**	Slow	Yes	2× data size
MRS Shuffle [43]	Worse than *Shuffle Once*	**Fast**	Yes	**No**
Sliding-Window [19]	Worse than *Shuffle Once*	**Fast**	Yes	**No**
`CorgiPile`	**Comparable to *Shuffle Once***	**Fast**	Yes	**No**



(a) No Shuffle (b) Sliding-Window (c) MRS Shuffle (d) Full Shuffle (ideal) (e) Our `CorgiPile`



(f) No Shuffle (g) Sliding-Window (h) MRS Shuffle (i) Full Shuffle (ideal) (j) Our `CorgiPile`

Fig. 3: The tuple id distribution (a-e) and corresponding label distribution (f-j). Tuple id denotes the tuple position after shuffling. #tuple refers to the number of negative/positive tuples in every 20 tuples shuffled.

3.5 Analysis and Summary

Table 1 summarizes the characteristics of different data shuffling strategies. As discussed, the effectiveness of data shuffling strategies for SGD largely depends on two somewhat conflicting factors, namely, (1) the degree of *data randomness* of the shuffled tuples and (2) the *I/O efficiency* when scanning data from disk. There is an apparent *trade-off* between these two factors:

– **The more random the tuples are, the better the convergence rate of SGD is.** *Epoch Shuffle* introduces data randomness at the highest level, but is too expensive to implement in in-DB ML and deep learning systems. *Shuffle Once* also introduces good data randomness, which is usually the best practice in terms of SGD convergence for in-DB ML systems.
– **A higher degree of randomness implies more random disk accesses and thus lower I/O efficiency.** As a result, the *No Shuffle* strategy is the best in terms of I/O efficiency.

The other strategies (*Sliding-Window Shuffle* and *MRS Shuffle*) try to sacrifice data randomness for better I/O efficiency, leaving a room for improvement.

Example 1 To better understand these issues, consider a clustered dataset with 1,000 tuples, each of which has a tuple-id *and a* label, *where* tuple-id *of the i-th tuple is i. The first 500 tuples are negative and the next 500 tuples are positive. Figure 3 plots the* tuple-id *distributions and corresponding* label *distributions after Sliding-Window Shuffle and MRS Shuffle, with a comparison to the ideal distributions from a full shuffle. The* tuple-id *distribution illustrates the positions of the tuples after shuffling, whereas the* label *distribution illustrates the number of negative/positive tuples in every 20 tuples shuffled.*

We can observe that Sliding-Window Shuffle results in a "linear"-shape distribution of the tuple-id *after shuffling, as shown by Figure 3(b), which suggests that the tuples are almost* not *shuffled. The corresponding label distribution in Figure 3(g) further confirms this, where almost all negative labels still appear before positive ones after shuffling. Similar patterns can be observed for MRS Shuffle in Figures 3(c) and 3(h), though MRS Shuffle has improved over Sliding-Window Shuffle. In summary, the data randomness achieved by Sliding-Window Shuffle or MRS Shuffle is far from the ideal case, as shown in Figures 3(d) and 3(i).* In contrast, we will see in the next section that the data randomness of our CorgiPile is closer to the ideal full shuffle (Figures 3(e) and 3(j)).

4 CorgiPile

As illustrated in the previous section, data shuffling strategies used by existing ML systems can be suboptimal when dealing with data that are not fully shuffled. Although recent efforts have significantly improved over baseline methods, there is still a large room for improvement. Inspired by these previous efforts, we present a simple but novel data shuffling strategy named CorgiPile. The key idea of CorgiPile lies in the following two-level hierarchical shuffling mechanism:

> *We first randomly select a set of blocks (each block refers to a set of contiguous tuples) and put them into an in-memory buffer; we then randomly shuffle all tuples in the buffer and use them for the SGD computation.*

Despite its simplicity, CorgiPile is highly effective. In terms of hardware efficiency, when the block size is large enough (e.g., 10MB+), a random access on the *block* level can be as efficient as a *sequential* scan, as shown in the I/O performance test on HDD and SSD in Figure 4. In terms of statistical efficiency, as we will show, *given the same buffer size*, CorgiPile converges much better than *Sliding-Window Shuffle* and *MRS Shuffle*. Nevertheless, both the convergence analysis and its integration into PostgreSQL and PyTorch are non-trivial. In the following, we first describe the CorgiPile algorithm precisely and then present a theoretical analysis on its convergence behavior.

Notations and Definitions. The following is a list of notations and definitions that we will use:
- $\|\cdot\|$: the ℓ_2-norm for vectors and the spectral norm for matrices;
- $\lesssim$: For two arbitrary vectors a, g, we use $a_s \lesssim g_s$ to denote that there exists a certain constant C that satisfies $a_s \leq C g_s$ for all s;
- N, the total number of blocks ($N \geq 2$);
- n, the buffer size (i.e., the number of blocks kept in the buffer);
- b, the size (number of tuples) of each data block;
- B_l, the set of tuple indices in the l-th block ($l \in [N]$ and $|B_l| = b$);
- m, the number of tuples for the finite-sum objective ($m = Nb$);
- $f_i(\cdot)$, the function associated with the i-th tuple;
- $\nabla F(\cdot)$ and $\nabla f_i(\cdot)$, the gradients of the functions $F(\cdot)$ and $f_i(\cdot)$;
- $H_i(\cdot) := \nabla^2 f_i(\cdot)$, the Hessian matrix of the function $f_i(\cdot)$;
- $\mathbf{x}^*$, the global minimizer of the function $F(\cdot)$;
- $\mathbf{x}_k^s$, the model $\mathbf{x}$ in the k-th iteration at the s-th epoch;
- μ-strongly convexity: function $F(\mathbf{x})$ is μ-strongly convex if $\forall \mathbf{x}, \mathbf{y}$,

$$F(\mathbf{x}) \geq F(\mathbf{y}) + \langle \mathbf{x} - \mathbf{y}, \nabla F(\mathbf{y}) \rangle + \frac{\mu}{2}\|\mathbf{x} - \mathbf{y}\|^2. \quad (1)$$

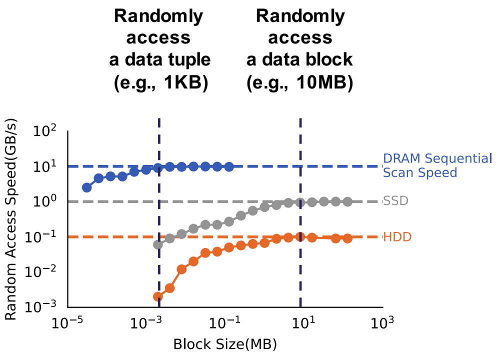


Fig. 4: Random Access Performance vs. Block Size. Randomly accessing a data block is faster than randomly accessing a data tuple. Larger block size results in more sequential accesses on data tuples and fewer cache misses. When data block is large (e.g., 10MB), random block access can be as efficient as sequential scan.

Algorithm 1 CorgiPile Algorithm

```
1: Input: $N$ blocks with $m$ total tuples, total epochs $S$ ($S \geq 1$), $a \geq 1$, $F(\cdot) = \frac{1}{m}\sum_{i=1}^{m} f_i(\cdot)$.
2: Initialize $\mathbf{x}_0^0$;
3: for $s = 0, \cdots, S$ do
4:     Randomly pick $n$ blocks without replacement, each containing $b$ tuples. Load these blocks into the buffer;
5:     Shuffle tuple indices among all $n$ blocks in the buffer and obtain the permutation $\boldsymbol{\psi}_s$;
6:     for $k = 1, ..., bn$ do
7:         Update $\mathbf{x}_k^s = \mathbf{x}_{k-1}^s - \eta_s \nabla f_{\psi_s(k)}\left(\mathbf{x}_{k-1}^s\right)$;
8:     end for
9:     $\mathbf{x}_0^{s+1} = \mathbf{x}_{bn}^s$;
10: end for
11: Return $x_{bn}^S$;
```

4.1 The CorgiPile Algorithm

Algorithm 1 illustrates the details of CorgiPile. At each epoch (say, the s-th epoch), CorgiPile runs the following steps:

1. (**Sample**) Randomly sample n blocks out of N data blocks *without replacement* and load the n blocks into the buffer. Note that we use *sample without replacement* to avoid visiting the same tuple multiple times for each epoch, which can converge faster and is a standard practice in most ML systems [32,34,46,49,43].

2. (**Shuffle**) Shuffle all tuples in the buffer. We use $\boldsymbol{\psi}_s$ to denote an ordered set, whose elements are the indices of the shuffled tuples at the s-th epoch. The size of $\boldsymbol{\psi}_s$ is bn, where b is the number of tuples per block. $\boldsymbol{\psi}_s(k)$ is the k-th element in $\boldsymbol{\psi}_s$.
3. (**Update**) Perform gradient descent by scanning each tuple with the shuffle indices in $\boldsymbol{\psi}_s$, yielding the updating rule

$$\mathbf{x}_k^s = \mathbf{x}_{k-1}^s - \eta_s \nabla f_{\boldsymbol{\psi}_s(k)}\left(\mathbf{x}_{k-1}^s\right),$$

where $\nabla f_{\boldsymbol{\psi}_s(k)}(\cdot)$ is the gradient of the function associated with the data sample with index $\boldsymbol{\psi}_s(k)$, and η_s is the learning rate for gradient descent at the epoch s. We initialize $\mathbf{x}_0^0$ and the parameter update is performed for all $k = 1, ..., bn$ in one epoch.

Intuition behind `CorgiPile`. Before we present the formal theoretical analysis, we first illustrate the intuition behind `CorgiPile`, following the same example used in Section 3.5.

Example 2 Consider the same settings as those in Example 1. Recall that `CorgiPile` contains both block-level and tuple-level shuffles. Suppose that the block-level shuffle generates a random order of blocks as {b20, b8, b45, b0, ...} and the buffer can hold 10 blocks. The tuple-level shuffle will put the first 10 blocks into the buffer, whose tuple_id*s are {b20[400, 419], b8[160, 179], b45[900, 919], b0[0, 19], ...}. After shuffling, the buffered tuples will have random* tuple_id*s in a large* non-contiguous *interval that is the union of {[0, 19], [160, 179], ..., [900, 919]}, as shown in the first 200 tuples in Figure 3(e). The buffered tuples therefore follow a random order closer to what is given by a full shuffle. As a result, the corresponding label distribution, as shown in Figure 3(j), is closer to a* uniform *distribution.*

Performance. While *No Shuffle* only requires *sequential* I/O's, our `CorgiPile` needs to (1) randomly access blocks, (2) copy all tuples in these blocks into a buffer, and (3) shuffle the tuples inside the buffer. Here, random accessing a block means randomly picking a block and reading the tuples of this block from secondary storage (e.g., the disk) into memory. If the block size is large enough, the I/O performances of random and sequential accesses are close. `CorgiPile` incurs additional overheads for *buffer copy* and in-memory *shuffle*. However, these I/O overheads can be hidden via standard techniques such as double buffering. As we will show in our experiments on PostgreSQL, the optimized version of `CorgiPile` only incurs 11.7% additional overhead compared to the most efficient *No Shuffle* baseline.

4.2 Convergence Analysis

Despite its simplicity, the convergence analysis of our `CorgiPile` is not trivial—even reasoning about the convergence of SGD with *sample without replacement* is an open question for decades [74, 47, 81, 49], not to say a hierarchical sampling scheme like ours. Fortunately, a recent theoretical advancement [49] provides us with the technical language to reason about `CorgiPile`'s convergence. In the following, we present a novel theoretical analysis for `CorgiPile`.

Note that in our following analysis, one epoch denotes going through all tuples in the sampled n blocks.

Assumption 1 *We make the following standard assumptions, as that in other previous work on SGD convergence analysis [35, 60]:*

1. *$F(\cdot)$ and $f_i(\cdot)$ are twice continuously differentiable.*
2. *L-Lipschitz gradient: $\exists L \in \mathbb{R}_+$, $\|\nabla f_i(\mathbf{x}) - \nabla f_i(\mathbf{y})\| \le L\|\mathbf{x}-\mathbf{y}\|$ for all $i \in [m]$.*
3. *L_H-Lipschitz Hessian matrix: $\|H_i(\mathbf{x}) - H_i(\mathbf{y})\| \le L_H\|\mathbf{x}-\mathbf{y}\|$ for all $i \in [m]$.*
4. *Bounded gradient: $\exists G \in \mathbb{R}_+$, $\|\nabla f_i(\mathbf{x}_k^s)\| \le G$ for all $i \in [m]$, $k \in [K-1]$, and $s \in \{0, 1 \ldots, S\}$.*
5. *Bounded Variance: $\mathbb{E}_\xi[\|\nabla f_\xi(\mathbf{x}) - \nabla F(\mathbf{x})\|^2] = \frac{1}{m}\sum_{i=1}^m \|\nabla f_i(\mathbf{x}) - \nabla F(\mathbf{x})\|^2 \le \sigma^2$ where ξ is the random variable that takes the values in $[m]$ with equal probability $1/m$. Here σ^2 denotes the upper bound of the variance for sampling the gradient $\nabla f_\xi(\mathbf{x})$.*

Factor h_D. In our analysis, we use the factor h_D to characterize the upper bound of a block-wise data variance:

$$\frac{1}{N}\sum_{l=1}^{N}\|\nabla f_{B_l}(\mathbf{x}) - \nabla F(\mathbf{x})\|^2 \le h_D\frac{\sigma^2}{b},$$

where $b = |B_l|$ is the size of each data block (recall the definition of b). Here, h_D is an essential parameter to measure the "cluster" effect within the original data blocks. Let's consider two extreme cases: 1) ($h_D = 1$) all samples in the data set are fully shuffled, such that the data in each block follows the same distribution; 2) ($h_D = b$) samples are well clustered in each block, for example, all samples in the same block are identical. Therefore, the larger h_D, the more "clustered" the data.

We now present the results for both strongly convex objectives (corresponding to generalized linear models) and non-convex objectives (corresponding to deep learning models) respectively, in order to show the correctness and efficiency of `CorgiPile`. Due to the space limitation, we detail the proof of the following theorems in the Appendix of our technical report [80].

Strongly convex objective We first show the result for strongly convex objective that satisfies the strong convexity condition (1).

Theorem 1 *Suppose that $F(\mathbf{x})$ is a smooth and μ-strongly convex function. Let $T = Snb$, that is, the total number of samples used in training and $S \geq 1$ is the number of tuples iterated, and choosing $\eta_s = \frac{6}{bn\mu(s+a)}$ where $a \geq \max\left\{\frac{8LG+24L^2+28L_HG}{\mu^2}, \frac{24L}{\mu}\right\}$, under Assumption 1, `CorgiPile` has the following convergence rate*

$$\mathbb{E}[F(\bar{\mathbf{x}}_S) - F(\mathbf{x}^*)] \lesssim (1-\alpha)h_D\sigma^2\frac{1}{T} + \beta\frac{1}{T^2} + \gamma\frac{m^3}{T^3}, \tag{2}$$

where $\bar{\mathbf{x}}_S = \frac{\sum_s (s+a)^3\mathbf{x}_s}{\sum_s (s+a)^3}$, and

$$\alpha := \frac{n-1}{N-1}, \quad \beta := \alpha^2 + (1-\alpha)^2(b-1)^2, \quad \gamma := \frac{n^3}{N^3}.$$

Tightness. The convergence rate of `CorgiPile` is tight in the following sense:

- $\alpha = 1$: It means that $n = N$, i.e., all tuples are fetched to the buffer. Then `CorgiPile` reduces to full shuffle SGD algorithm [49]. In this case, the upper bound in Theorem 1 is $O(1/T^2 + m^3/T^3)$, which matches the result of the full shuffle SGD algorithm [49].
- $\alpha = 0$: It means that $n = 1$, i.e., only sampling one block each time. Then `CorgiPile` is very close to *mini-batch* SGD (by viewing a block as a mini-batch), except that the model is updated once per data tuple. Ignoring the higher-order terms in (2), our upper bound $O(h_D\sigma^2/T)$ is consistent with that of mini-batch SGD.

Comparison to vanilla SGD. In the vanilla SGD, we only randomly select one tuple from the dataset to update the model. It admits the convergence rate $O(\sigma^2/T)$. For our algorithm, when T is sufficiently large, the term $(1-\alpha)h_D(\sigma^2/T)$ in (2) will be dominating. If $n \gg (h_D - 1)(N-1)/h_D + 1$ for $h_D > 0$ (i.e. sampling sufficiently many blocks), the factor $(1-\alpha)h_D$ in the dominating term will be much smaller than 1. Therefore, ignoring the higher order terms in (2) for a large T, our algorithm admits a faster convergence rate compared to $O(\sigma^2/T)$ for the vanilla SGD. It is also worth noting that, even if n is small, `CorgiPile` may still significantly outperform vanilla SGD in practice. Assuming that reading a random single tuple incurs an overhead of $t_{\text{lat}}+t_{\text{t}}$ and reading a block of b tuples incurs an overhead of $t_{\text{lat}}+bt_{\text{t}}$, where t_{lat} is the "latency" for one read/write operation that does not grow linearly with respect to the amount of data that one reads/writes (e.g., SSD read/write latency or HDD "seek and rotate" time), and t_{t} is the time that one needs to transfer a single tuple. To reach an error of ϵ, vanilla SGD requires time

$$O\left(\frac{\sigma^2}{\epsilon}t_{\text{lat}} + \frac{\sigma^2}{\epsilon}t_{\text{t}}\right),$$

whereas `CorgiPile` requires time

$$O\left((1-\alpha)\frac{h_D}{b}\cdot\frac{\sigma^2}{\epsilon}t_{\text{lat}} + (1-\alpha)h_D\cdot\frac{\sigma^2}{\epsilon}t_{\text{t}}\right).$$

Because $(1-\alpha)\frac{h_D}{b} < 1$, `CorgiPile` always provides benefit over vanilla SGD in terms of the read/write latency t_{lat}. When t_{lat} dominates t_{t}, `CorgiPile` can outperform vanilla SGD even for small buffers.

Non-convex objective We further conduct an analysis on objectives that are non-convex or satisfy the Polyak-Łojasiewicz condition, which leads to similar insights on the behavior of `CorgiPile`.

Theorem 2 *Suppose that $F(\mathbf{x})$ is a smooth function. Letting $T = Snb$ be the number of tuples iterated, under Assumption 1, `CorgiPile` has the following convergence rate:*

1. *When $\alpha \leq \frac{N-2}{N-1}$, choosing $\eta_s = \frac{1}{\sqrt{bn(1-\alpha)h_D\sigma^2 S}}$ and assuming $S \geq \frac{bn(\frac{104}{3}L+\frac{4}{3}L_H)^2}{\sigma^2(1-\alpha)h_D}$, we have*

$$\frac{1}{S}\sum_{s=1}^{S}\mathbb{E}\|\nabla F(\mathbf{x}_0^s)\|^2 \lesssim (1-\alpha)^{1/2}\frac{\sqrt{h_D}\sigma}{\sqrt{T}} + \beta\frac{1}{T} + \gamma\frac{m^3}{T^{\frac{3}{2}}},$$

where the factors are defined as

$$\alpha := \frac{n-1}{N-1}, \quad \beta := \frac{\alpha^2}{1-\alpha}\frac{1}{h_D\sigma^2} + (1-\alpha)\frac{(b-1)^2}{h_D\sigma^2},$$

$$\gamma := \frac{n^3}{(1-\alpha)N^3};$$

2. *When $\alpha = 1$, choosing $\eta_s = \frac{1}{(mS)^{\frac{1}{3}}}$ and assuming $S \geq (\frac{416}{3}L + \frac{16}{3}L_H)^3 b^2 n^3/N$, we have*

$$\frac{1}{S}\sum_{s=1}^{S}\mathbb{E}\|\nabla F(\mathbf{x}_0^s)\|^2 \lesssim \frac{1}{T^{\frac{2}{3}}} + \gamma'\frac{m^3}{T},$$

where we define $\gamma' := \frac{n^3}{N^3}$.

We can apply a similar analysis as that of Theorem 1 to compare `CorgiPile` with vanilla SGD, in terms of convergence rate, and reach similar insights.

5 Implementation in the Database

We have integrated `CorgiPile` into PostgreSQL. Our implementation provides a simple SQL-based interface for users to invoke `CorgiPile`, with the following query template:

```
SELECT * FROM table TRAIN BY model WITH params
```

This interface is similar to that offered by existing in-DB ML systems like MADlib [51,2] and Bismarck [43]. Examples of the `params` include *learning_rate = 0.1*, *max_epoch_num = 20*, and *block_size = 10MB*. `CorgiPile` outputs various metrics after each epoch, including the *training loss*, *accuracy*, and *execution time*.

The Need of a Deeper Integration Unlike existing in-DB ML systems, we choose not to implement our `CorgiPile` strategy using User-Defined Aggregates (UDAs). Instead, we choose to integrate `CorgiPile` into PostgreSQL by introducing physical operators. *Is it necessary for such a deeper integration with database system internals, compared to a potential UDA-based implementation without modifying the internals?*

While a UDA-based implementation is conceptually possible, it is not natural for `CorgiPile`, which requires accessing low-level data layout information such as table pages, tuples, and buffers. A deeper integration with database internals makes it much easier to reuse such functionalities that have been built into the core APIs offered by database system internals but not yet have been externally exposed as UDAs. Moreover, such a physical-level integration opens up the door for more advanced optimizations, such as double-buffering that will be illustrated in Section 5.3.

5.1 Design Considerations

As discussed in Section 4.1, `CorgiPile` consists of three steps: (1) block-level shuffling, (2) tuple-level shuffling, and (3) SGD computation. Accordingly, we design three physical operators:

– `BlockShuffle`, an operator for randomly accessing blocks;
– `TupleShuffle`, an operator for buffering a batch of blocks and shuffling their tuples;
– `SGD`, an operator for the SGD computation.

We then chain these three operators together to form a pipeline, and implement the `getNext()` method for each operator, following the classic Volcano-style execution model [45] that is also the query execution paradigm of PostgreSQL.

One challenge is the implementation of the *SGD* operator, which requires an *iterative* procedure that is not typically supported by database systems. We choose to implement it by leveraging the built-in *re-scan* mechanism of PostgreSQL to reshuffle and reread the data after each epoch.

We store datasets as tables in PostgreSQL using the schema of $\langle id, features\_k[], features\_v[], label\rangle$, which is similar to the one used by Bismarck [43]. For sparse datasets, *features_k*[] indicates which dimensions have non-zero values, and *features_v*[] refers to the corresponding non-zero feature values. For dense dataset, only *features_v*[] is used.

Currently, we store the (learned) machine learning model as an in-memory object (a C-style Struct) with an ID in the PostgreSQL's kernel instead of using UDA. Users can initialize the model hyperparameters via the query. For the inference, users can execute a query as "`SELECT table PREDICT BY model ID`", which invokes the learned model for prediction.

5.2 Physical Operators

The control flow of the three operators is illustrated in Figure 5, which leverages a PostgreSQL's pull-style dataflow to read tuples and perform the SGD computation. In the following, we assume that the readers are familiar with the structure of PostgreSQL's operators, e.g., functions such as `ExecInit()` and `getNext()`.

After parsing the input query, `CorgiPile` invokes `ExecInit()` of each operator to initialize their states such as ML models and I/O buffers. At each epoch, the `SGD` operator pulls tuples from the `TupleShuffle` operator for SGD computation, which further pulls tuples from the `BlockShuffle` operator. The `BlockShuffle` operator is responsible for shuffling blocks and reading their tuples. We now present the implementation of these operators.

(1) BlockShuffle: This operator first obtains the total number of pages by PostgreSQL's internal function as `RelationGetNumberOfBlocks()`. It then computes the number of blocks BN by $BN = page\_num * page\_size / block\_size$. After that, it shuffles the block indices $[0, \dots, BN-1]$ and gets shuffled block ids, where each block corresponds to a batch of contiguous table pages. For each shuffled block id, it reads the corresponding pages using `heapgetpage()` and returns each fetched tuple to the `TupleShuffle` operator. The `BlockShuffle` operator is similar to PostgreSQL's `Scan` operator, although the `Scan` operator reads pages sequentially instead of randomly.

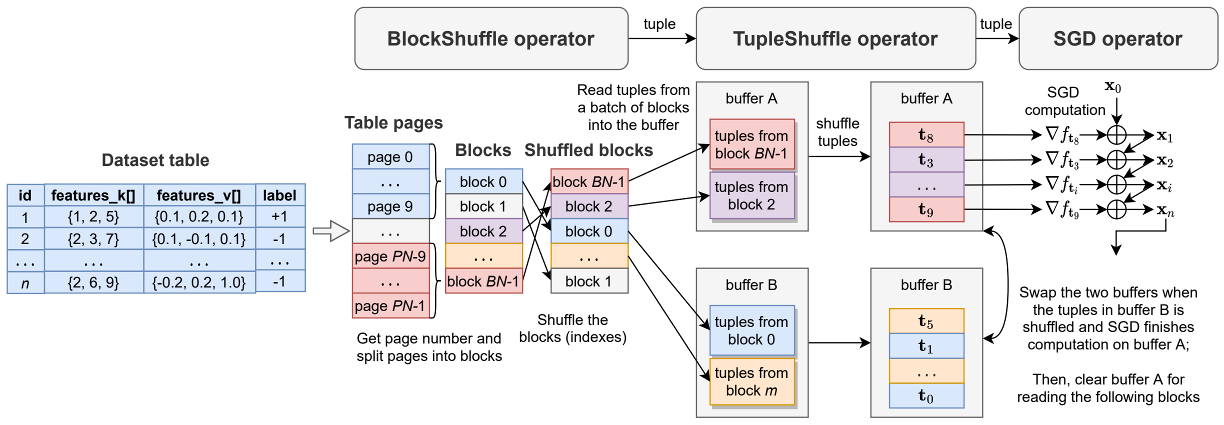


Fig. 5: `CorgiPile` in PostgreSQL, with three new operators and the "double-buffering" optimization.

(2) TupleShuffle: It first allocates a buffer, and then pulls the tuples one by one from the `BlockShuffle` operator by invoking its `ExecTupleShuffle()`, namely `getNext()`. Each pulled tuple is transformed to an `SGDTuple` object, which is then copied to the buffer. Once the buffer is filled, it shuffles the buffered tuples, which is similar to how the `Sort` operator works in the PostgreSQL. After that, the shuffled tuples are returned one by one to the `SGD` operator.

(3) SGD: It first initializes an ML model in the `ExecInitSGD()` and then executes SGD computation in `ExecSGD()`. At each epoch, `ExecSGD()` pulls tuples from `TupleShuffle` one by one, and runs SGD computation. Once all tuples are processed, an epoch ends. It then has to reshuffle and reread the tuples for the next epoch, using the *re-scan* mechanism of PostgreSQL. Specifically, after each epoch, `SGD` invokes `ExecReScan()` of `TupleShuffle` to reset the I/O states of the buffer. It further invokes `ExecReScan()` of `BlockShuffle` to reshuffle the block ids. After that, `SGD` operator can reread shuffled tuples via `ExecSGD()` for the next epoch. This is similar to the behavior of multiple table/index scans in PostgreSQL's `NestedLoopJoin`.

5.3 Optimizations

As discussed in Section 4.1, `CorgiPile` introduces additional overheads for *buffer copy* and *shuffle*. To reduce them, we use a double-buffering strategy as shown in Figure 5. Specifically, we launch two concurrent threads for `TupleShuffle` with two buffers. One *write* thread is responsible for pulling tuples from `BlockShuffle` into one buffer and shuffling the buffered tuples; the other *read* thread is responsible for reading tuples from another buffer and returning them to `SGD`. The two buffers are swapped once one is full and the other has been consumed by `SGD`. As a result, the data loading (i.e., block-level and tuple-level shuffling) and SGD computation can be executed concurrently, reducing the overhead.

6 Multi-process `CorgiPile` in PyTorch

We also integrated `CorgiPile` into PyTorch, one state-of-the-art deep learning system. The main challenge is how to extend our single-process `CorgiPile` to work in the parallel/distributed environment of deep learning systems, which usually use multiple processes with multiple GPUs to train models. For example, PyTorch offers a `DistributedDataParallel` (DDP) mode [7] for multi-process training, where PyTorch runs multiple processes in a single machine with multiple GPUs or across a number of machines to train models.

6.1 A Multi-process Mode of `CorgiPile`

`CorgiPile` can be naturally extended to work in a multi-process mode, by enhancing the tuple-level shuffle under the data-parallel computation paradigm. As mentioned in Section 4.1, `CorgiPile` contains both block-level shuffle and tuple-level shuffle. As shown in Figure 6(a), we can naturally implement block-level shuffle by randomly distributing data blocks to different processes. For tuple-level shuffle, we can use *multi-buffer* based shuffling instead of single-buffer based shuffling — in each process we allocate a local buffer to read blocks and shuffle their tuples. The deep learning system can then read the shuffled tuples when running SGD to perform the forward/backward/update computation as well as gradient/parameter communication/synchronization among different processes.

We implement this enhanced multi-process `CorgiPile` as a new `CorgiPileDataset` API in PyTorch:

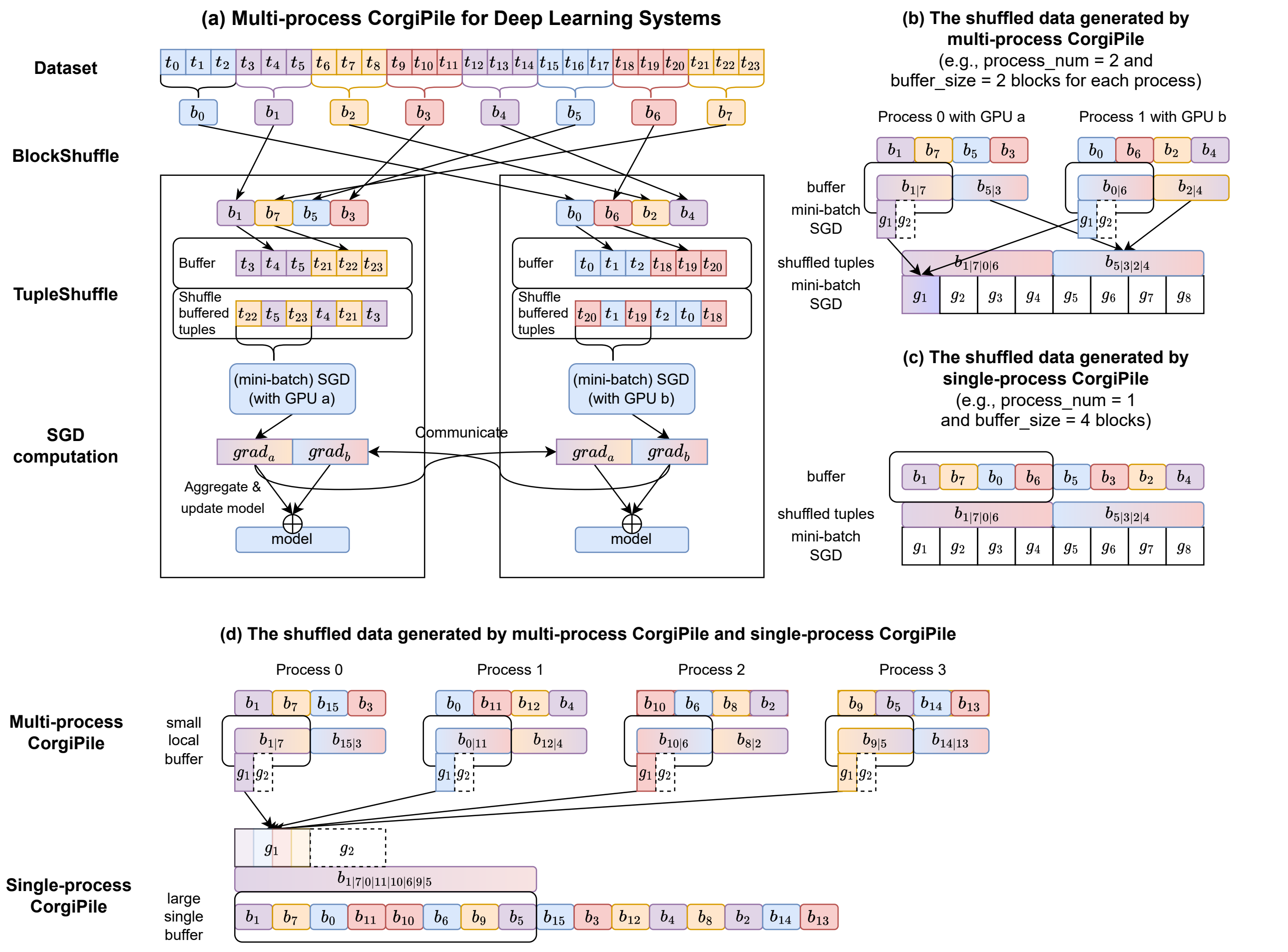

(a) Multi-process CorgiPile for Deep Learning Systems
Dataset
BlockShuffle
TupleShuffle
SGD computation
Buffer
Shuffle buffered tuples
(mini-batch) SGD (with GPU a)
(mini-batch) SGD (with GPU b)
Communicate
Aggregate & update model
model
(b) The shuffled data generated by multi-process CorgiPile
(e.g., process_num = 2 and buffer_size = 2 blocks for each process)
Process 0 with GPU a
Process 1 with GPU b
buffer
mini-batch SGD
shuffled tuples
(c) The shuffled data generated by single-process CorgiPile
(e.g., process_num = 1 and buffer_size = 4 blocks)
(d) The shuffled data generated by multi-process CorgiPile and single-process CorgiPile
Process 0
Process 1
Process 2
Process 3
Multi-process CorgiPile
small local buffer
Single-process CorgiPile
large single buffer

Fig. 6: (a) The implementation of `CorgiPile` in a parallel/distributed environment (e.g., PyTorch) with multiple processes and GPUs. (b) The shuffled data generated by multi-process `CorgiPile` is similar to (c) the shuffled data generated by single-process `CorgiPile`. (d) further confirms this statement using more (four) processes.

```
train_dataset = CorgiPileDataset(dataset_path,
                    block_index, other_args)
train_loader = torch.utils.data.DataLoader(
                    train_dataset, other_args)
train(train_loader, model, other_args).
```

Similar to usage of the original `Dataset` API, users only need to initialize the `CorgiPileDataset` with necessary parameters and then use it as usual in the `DataLoader` API offered by PyTorch. The `train()` method constantly extracts a batch of tuples from `DataLoader` and then performs mini-batch SGD. Multi-process `CorgiPile` can achieve random data order similar to that of the single-process `CorgiPile` (Section 6.3).

6.2 Implementation Details

We next detail the implementation of multi-process `CorgiPile` in PyTorch:

(1) Block partitioning: We first partition the dataset into blocks. In a parallel/distributed environment, we typically store the dataset on the block-based parallel/distributed file systems such as HDFS [9], Amazon EBS [1], and Lustre [22]. For example, the ETH Euler cluster [6] uses Lustre, which reads/writes data in blocks (by default 4 MB) [14] and does not allow users to store/read massive small files like raw images in a directory. Therefore, for training ∼150GB `ImageNet` with 1.3 million raw images [10] that cannot be fit into memory, we need to convert these images into binary data files such as the widely-used iterable dataset `TFRecords` [25, 20] and store them in Lustre before training. In addition, we build a block index to identify the start/end of each block, by using the block information provided by the file system or indexing tools such as `PyTorch-TFRecord` [20]. If the dataset itself contains tuple index (e.g., the *map-style* dataset in PyTorch), we can also partition the dataset into blocks based on the tuple index.

(2) Block shuffle: Each process randomly picks BN/PN blocks, where BN is the number of blocks and PN is the number of processes. We implement block shuffle in our `CorgiPileDataset` API. At the beginning of each epoch, it first shuffles the block indices and then splits the indices into PN parts. The i-th process only reads the blocks with indices in the i-th part.

(3) Tuple shuffle: Each process first allocates a small buffer in memory and then constantly reads the blocks into the buffer. Once the buffer is full, the process will shuffle the buffered tuples. This is implemented in `CorgiPileDataset` as its `iter()` method, which reads blocks into a buffer, shuffles their tuples, and returns the shuffled tuples one by one. The buffer size here is much smaller than that used in single-process `CorgiPile`—if we set $buffer\_size = BS$ in single-process `CorgiPile`, we can choose $buffer\_size = BS/PN$ for each local buffer in multi-process `CorgiPile`.

(4) SGD computation: After block shuffle and tuple shuffle, each process performs mini-batch SGD on the shuffled tuples. Unlike single-process `CorgiPile` that performs mini-batch SGD on the whole dataset with $batch\_size = bs$, each process in multi-process `CorgiPile` performs mini-batch SGD on partial dataset with a smaller batch size (bs/PN) and updates the model with *gradient synchronization* every batch. As shown in Figure 6(a), after each batch the processes will synchronize/aggregate the gradients using a communication protocol (e.g., `AllReduce`); each process then updates its local copy of the ML model. This procedure is encapsulated inside the `train()` method, which automatically performs gradient computation/communication/synchronization and model update every time after reading a batch of tuples from `CorgiPileDataset`.

6.3 Single-process vs. Multi-process `CorgiPile`

The shuffled data order of multi-process `CorgiPile` is comparable to that of single-process `CorgiPile`. Indeed, any data order generated by multi-process `CorgiPile` can also be generated by single-process `CorgiPile` (see Theorem 3). Here, we use a simple example (shown in Figure 6) to demonstrate this. As shown in Figure 6(a), there are two processes and each randomly picks four blocks from the dataset. Each process can read two blocks into the buffer at once and shuffle their tuples in the buffer. As shown in Figure 6(b), the shuffled tuples of process 0 are in sequence from block 1/7 (denoted as $b_{1|7}$) and then from block 5/3. Likewise, the shuffled tuples of process 1 are in sequence from block 0/6 and then from block 2/4. Since PyTorch sequentially performs mini-batch SGD on the first $batch\_size/PN$ tuples of each process (denoted as g_1 on block 1/7 and block 0/6) and aggregates their gradients (sums and averages g_1) every batch, this parallel mini-batch SGD is equivalent to the mini-batch SGD on the first $batch\_size$ tuples from block 1/7/0/6 (i.e., g_1 on $b_{1|7|0|6}$) in a single process. Therefore, from the view of the whole dataset, PyTorch with multi-process `CorgiPile` performs mini-batch SGD on the tuples first from block 1/7/0/6 and then from block 5/3/2/4. This is similar to the data order generated by single-process `CorgiPile` in Figure 6(c), where the buffer size is PN times larger. Here, $PN = 2$ and the buffer can keep 4 blocks at once.

To demonstrate more general cases with $PN > 2$, we increase the number of processes from two to four in Figure 6(d). In this case, each process first loads two blocks into the local buffer, shuffles their tuples, and then performs mini-batch SGD on a number of ($batch\_size/PN$) shuffled tuples in each batch. For the first batch, PyTorch computes the gradient g_1 in each process and then aggregates them. The aggregated g_1 can be viewed as the result of performing mini-batch SGD on the first $batch\_size$ shuffled tuples from block 1/7/0/11/10/6/9/5 (denoted as $b_{1|7|0|11|10|6|9|5}$). This $b_{1|7|0|11|10|6|9|5}$ can also be generated in a single process, by shuffling blocks as shown in Figure 6(d) and then shuffling their tuples in a large single buffer. Thus, given a data order generated by multi-process `CorgiPile`, we can also find an equivalent data order generated by single-process `CorgiPile`. The same observation holds for the next batches (e.g., g_2).

Theorem 3 *Any order of data tuples generated by the multi-process `CorgiPile` can also be generated by the single-process `CorgiPile`.*

Proof Suppose that the dataset contains n blocks. After the block-level shuffle, the shuffled blocks are denoted as $b_1, b_2, \ldots, b_n$. Suppose that the buffer can keep m blocks. The next step is tuple-level shuffle for both single-process `CorgiPile` and multi-process `CorgiPile`.

For single-process `CorgiPile`, it puts m blocks into the buffer each time. Without loss of generality, suppose that the buffer holds $[b_1, b_2, \ldots, b_m]$. Then, after tuple-level shuffle, each tuple in the buffer comes from a mixture of $[b_1, b_2, \ldots, b_m]$, denoted as $b_{1|2|\ldots|m}$.

For multi-process `CorgiPile`, suppose that there are p processes. Each process i has a smaller buffer that can hold $\frac{m}{p}$ blocks. We use round-robin to assign blocks to processes, i.e., the block b_i goes to the process $i\%p$. Suppose that $m\%p = 0$, i.e., m is a multiple of p, and $m = k \cdot p$ The process j will buffer $[b_j, b_{p+j}, \ldots, b_{(k-1)p+j}]$ for $1 \le j \le p$. After tuple-level shuffle, tuples in process j come from the mixture of $b_{j|p+j|\ldots|(k-1)p+j}$. Note that mini-batch SGD will sequentially compute the gradients of shuffled tuples in each process and then sum them together. As a result, mini-batch SGD will compute the gradients of tuples from the union $\{b_{j|p+j|\ldots|(k-1)p+j}\}_{j=1}^{k}$, which is equivalent to $b_{1|2|\ldots|m}$, i.e., the shuffled tuples generated by the single-process CorgiPile.

7 Evaluation

We evaluate CorgiPile in both in-DB ML and deep learning systems, to study the statistical and hardware efficiency of CorgiPile, i.e., whether it can achieve both high accuracy and high performance. For in-DB ML systems, we compare our PostgreSQL-based implementation with two state-of-the-art systems, *Apache MADlib and Bismarck* with diverse linear models and datasets[4]. We first evaluate linear models with standard SGD in PostgreSQL (Section 7.2). We further evaluate linear models with mini-batch SGD as well as other types of (continuous, multi-class) datasets in PostgreSQL. For deep learning systems, we compare CorgiPile with other shuffling strategies in PyTorch, using image classification workloads (Section 7.3).

7.1 Experimental Setup

7.1.1 Runtime

For in-DB ML workloads, we perform the experiments on a single *ecs.i2.xlarge* node in Alibaba Cloud, which has 2 physical cores (4 vCPU), 32 GB RAM, 1000 GB HDD, and 894 GB SSD. The HDD has a maximum 140 MB/s bandwidth, and the SSD has a maximum 1 GB/s bandwidth. We run all experiments in PostgreSQL under CentOS 7.6, and we clear the OS cache before running each experiment.

For deep learning workloads, we perform them on ETH Euler cluster [6] as batch jobs. Each job can use maximum 16 CPU cores, 160 GB RAM, and 8 NVIDIA GeForce RTX 2080 Ti GPUs. The datasets are stored in the cluster's block-based Lustre parallel file system.

7.1.2 Datasets

For in-DB ML, we use a variety of datasets in our evaluation, including dense/sparse and small/large ones with two classes as shown in Table 2. The datasets in Table 2 are stored in PostgreSQL for in-DB ML experiments and we use both label-clustered and feature-ordered datasets. For deep learning, we use both the cifar-10 dataset with 10 classes [4] and the ImageNet dataset with 1,000 classes [10] for image classification.

[4] Although MADlib contains DL models, these models are essentially backended/trained by TensorFlow [28]. TensorFlow uses a sliding-window shuffling strategy, which we implemented and compared against in PyTorch.

Table 2: Datasets. The first four are from LIBSVM [13]. For criteo, we extract 98M tuples from the criteo terabyte dataset. For yfcc, we extract 3.6M tuples from the yfcc100m dataset [77]; the *outdoor* and *indoor* tuples are marked as negative (-1) and positive (+1). #Tuples like 4.5/0.5M refer to 4.5M tuples for training and 0.5M tuples for testing.

Name	Type	#Tuples	#Features	Size
higgs	dense	10.0/1.0M	28	2.8 GB
susy	dense	4.5/0.5M	18	0.9 GB
epsilon	dense	0.4/0.1M	2,000	6.3 GB
criteo	sparse	92/6.0M	1,000,000	50 GB
yfcc	dense	3.3/0.3M	4,096	55 GB
ImageNet	image	1.3/0.05M	224*224*3	150 GB
cifar-10	image	0.05/0.01M	3,072	178 MB

7.1.3 Models and Parameters

Models for in-DB ML systems. For the evaluation on in-DB ML systems, we mainly train two popular generalized linear models, logistic regression (LR) and support vector machine (SVM), that are also supported by Bismarck and MADlib. We briefly report the evaluation results for other liner models such as linear regression and softmax regression, which are currently only supported by MADlib. Currently, Bismarck and MADlib only support two of the baseline data shuffling strategies, namely, *No Shuffle* and *Shuffle Once*, which we compare our PostgreSQL-based implementation against. Note that the code of *MRS Shuffle* has not been released by Bismarck yet. Therefore, we leave it out of our end-to-end comparisons. Instead, we implemented *MRS Shuffle* by ourselves in PyTorch and compare with it when we discuss the convergence behavior of different data shuffling strategies (like Figure 8).

Models for DL system. For the evaluation on deep learning system, we perform the classical VGG19 and ResNet18 models on the cifar-10 dataset, and perform more complex ResNet50 model on the ImageNet dataset.

Model hyperparameters. The model hyperparameters include the learning rate, the decay factor, and the maximum number of epochs. By default, we use an exponential learning rate decay with 0.95. We set the number of epochs to 20 for in-DB ML and 50 for deep learning models. Only for ResNet50 on ImageNet, we set the number of epochs to 100 and decay the learning rate every 30 epochs, following the official PyTorch-ImageNet

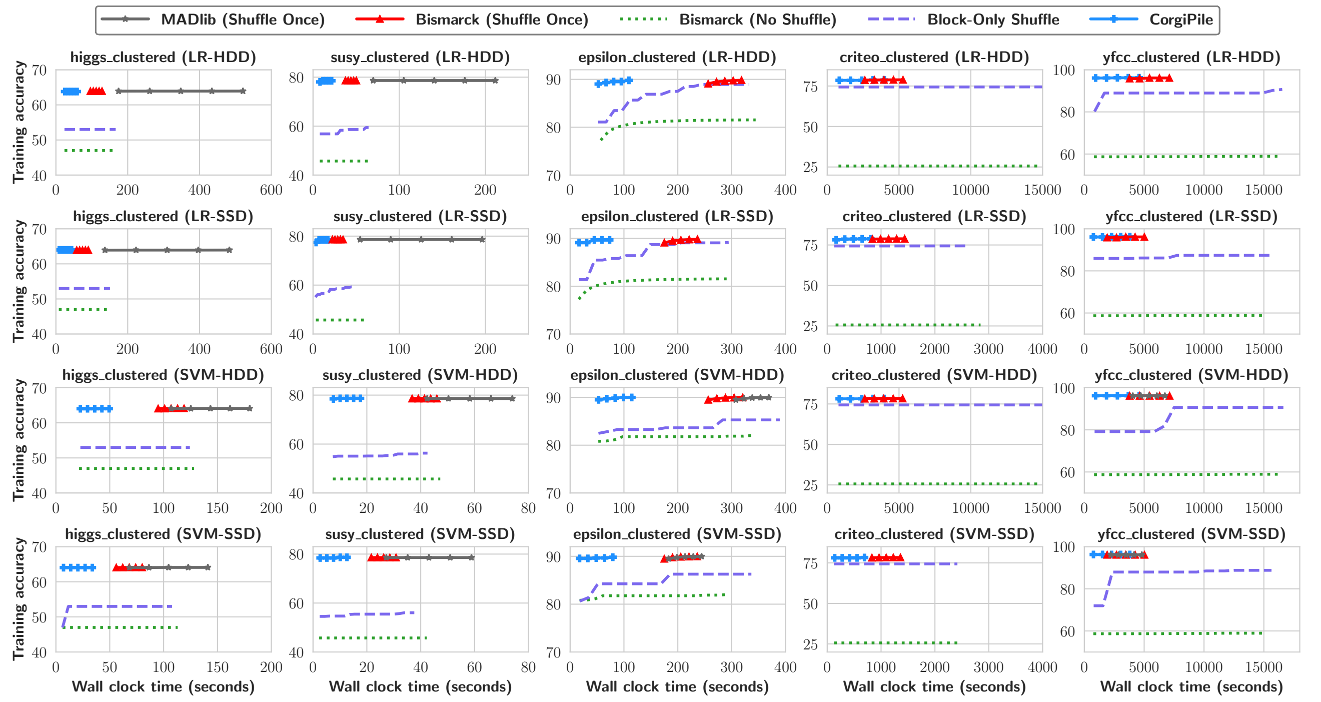


Fig. 7: The end-to-end execution time of SGD with different data shuffling strategies in PostgreSQL, for clustered datasets on HDD and SSD. *Block-Only Shuffle* refers to `CorgiPile` without tuple-level shuffle. We only show the first 5 epochs for *Shuffle Once* and `CorgiPile`, as they converge in 1-3 epochs due to better shuffled data order. We show all the 20 epochs for other shuffling strategies to observe if they can converge to high accuracy.

code [11]. We use grid search to tune the best learning rate from {0.1, 0.01, 0.001}. For in-DB ML, we use the same initial parameters and hyperparameters among the compared systems, including MADlib, Bismarck, and `CorgiPile`.

7.1.4 Settings of `CorgiPile`

`CorgiPile` has two more parameters, i.e., the buffer size and the block size. We experiment with a diverse range of buffer sizes in {1%, 2%, 5%, 10%} and the block size is chosen in {2MB, 10MB, 50MB}. We always use the same buffer size (by default 10% of the whole dataset size) for *Sliding-Window Shuffle*, *MRS Shuffle*, and our `CorgiPile`.

7.1.5 Settings of PostgreSQL

For PostgreSQL, we set the `work_mem` to be the maximum RAM size and tune `shared_buffers`. Note that PostgreSQL can further compress the high-dimensional datasets using the so-called TOAST [18] technology, which tries to compress large field value or break it into multiple physical rows. For our dense `epsilon` and `yfcc` datasets with 2,000+ dimensions, PostgreSQL uses TOAST to compress their *features_v* columns.

7.2 Evaluation on SGD with In-DB ML Systems

For in-DB ML, we first evaluate `CorgiPile` in terms of the end-to-end execution time. The compared systems include *No Shuffle* and *Shuffle Once* strategies in MADlib and Bismarck, as well as a simpler version of our `CorgiPile` named *Block-Only Shuffle*, to see how `CorgiPile` behaves without tuple-level shuffle. We then analyze the convergence rates, in comparison with other strategies, including *MRS Shuffle* and *Sliding-Window Shuffle*. We finally study the overhead of `CorgiPile` by comparing the per-epoch execution time of `CorgiPile` with the fastest *No Shuffle* baseline.

In the following, we set the buffer size to 10% of the whole dataset and block size to 10 MB for all methods. We choose these settings according to our sensitivity analysis in Section 7.2.4.

7.2.1 End-to-end Execution Time

Figure 7 presents the end-to-end execution time of SGD for in-DB ML systems, for *clustered* datasets on both HDD and SSD. The end-to-end execution time includes: (1) the time for shuffling the data, i.e., *Shuffle Once* needs to perform a full data shuffle before SGD starts

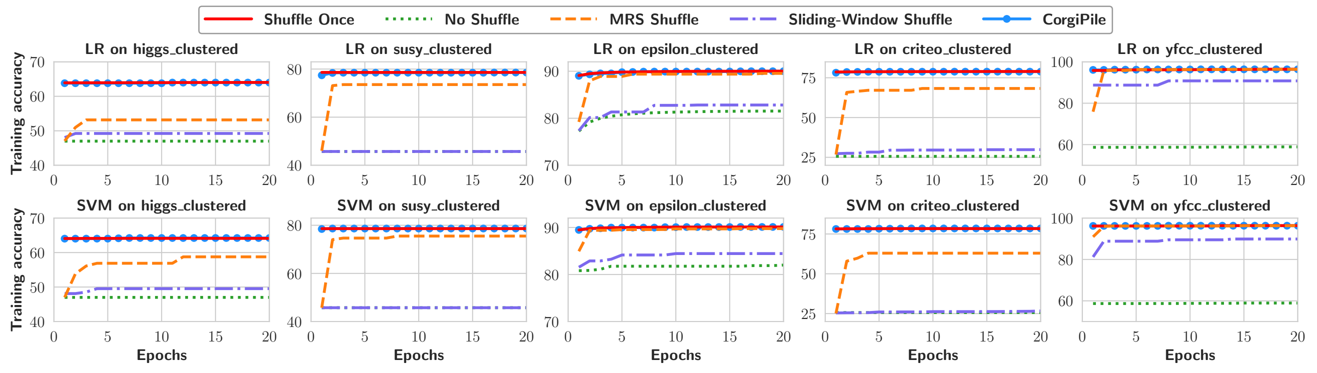


Fig. 8: The convergence rates of LR and SVM with different shuffling strategies, for clustered datasets.

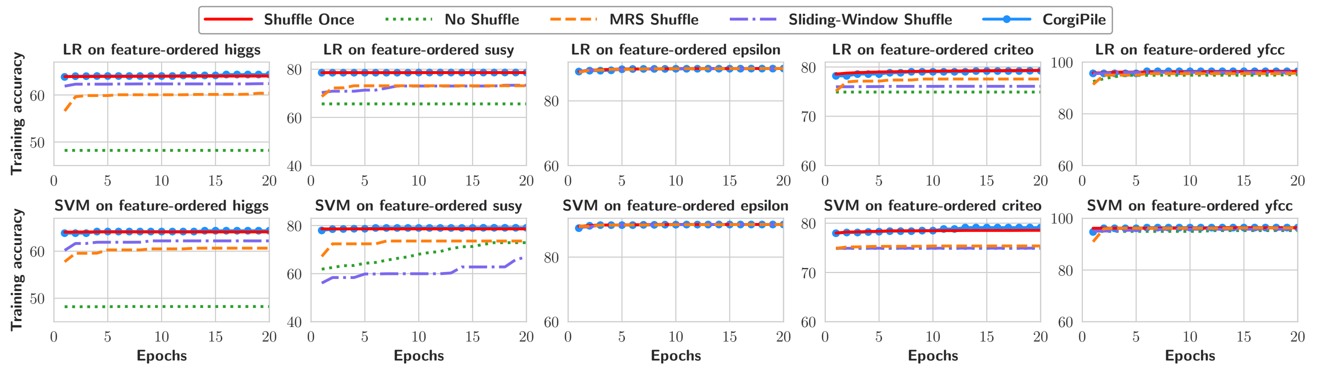


Fig. 9: The convergence rates of LR and SVM with different shuffling strategies, for feature-ordered datasets.

running;[5] (2) the data caching time, i.e., the time spent on loading data from disk to the OS cache during the first epoch;[6] and (3) the execution time of all epochs.

From Figure 7, we can observe that `CorgiPile` converges the fastest among all systems, and simultaneously achieves comparable converged accuracy to the best *Shuffle Once* baseline, usually within 1-3 epochs because of the large number of data tuples. In particular, `CorgiPile` is 2.9×-12.8× faster than MADlib and 2.0×-4.7× faster than Bismarck, to converge to the same accuracy when data is stored on HDD and SSD. This is due to the eliminated data shuffling time. For example, for the clustered `yfcc` dataset on HDD, `CorgiPile` can converge in 16 minutes, whereas *Shuffle Once* in Bismarck needs 50 minutes to shuffle the dataset and another 15 minutes to execute the first epoch (to converge). That is, when `CorgiPile` converges, *Shuffle Once* is still performing data shuffling. For other datasets like `criteo` and `epsilon`, similar observations hold. Moreover, data shuffling using `ORDER BY RANDOM()` in PostgreSQL, as implemented by *Shuffle Once* in MADlib/Bismarck, requires 2× disk space to generate and store the shuffled data. Therefore, `CorgiPile` is both more efficient and requires less space.

MADlib is slower than Bismarck given that it performs more computation on some auxiliary statistical metrics and has less efficient implementation [53]. Moreover, for high-dimensional dense datasets, such as `epsilon` and `yfcc`, MADlib's LR cannot finish even a single epoch within 4 hours, due to some expensive matrix computations on a metric named *stderr*.[7] MADlib's SVM implementation does not have this problem and can finish its execution on high-dimensional dense datasets. In addition, MADlib currently does not support training LR/SVM on sparse datasets such as `criteo` dataset.

7.2.2 Convergence rate comparison

For all datasets inspected, the gap between *Shuffle Once* and `CorgiPile` is below 1% for the final testing accuracy, as shown in Table 3. We attribute this to the fact that `CorgiPile` can yield good data randomness in each

[5] Therefore, *Shuffle Once* in MADlib and Bismarck starts later than the others.

[6] This is determined by the I/O bandwidth. Since SSD has higher I/O performance than HDD, the GLMs' first epoch on SSD starts earlier than that on HDD.

[7] We have confirmed this behavior with MADlib developers.

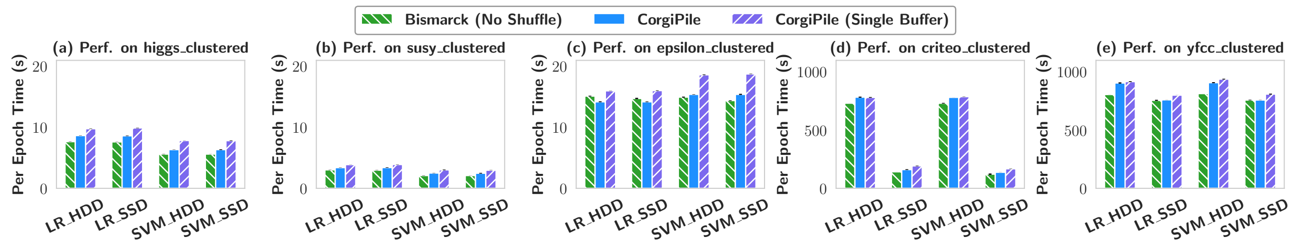


Fig. 10: The average per-epoch time of SGD with Bismarck (*No Shuffle*), `CorgiPile`, and `CorgiPile` with single buffer in PostgreSQL, for clustered datasets on HDD and SSD. It shows that `CorgiPile` is up to 11.7% slower than the fastest *No Shuffle*.

Table 3: The final testing accuracy of *Shuffle Once* (SO) and `CorgiPile`.

Dataset	LR (SO | `CorgiPile`)	SVM (SO | `CorgiPile`)
higgs	64.04 | 64.06	63.93 | 63.95
susy	78.69 | 78.66	78.73 | 78.66
epsilon	89.77 | 89.74	89.81 | 89.80
criteo	78.77 | 78.69	78.45 | 78.44
yfcc	96.14 | 96.11	96.23 | 96.20

epoch of SGD (Section 4.2). *No Shuffle* results in the lowest accuracy when SGD converges, as illustrated in Figure 7. The *Block-Only Shuffle* baseline, where we simply omit tuple-level shuffle in `CorgiPile`, can achieve higher accuracy than *No Shuffle* but lower accuracy than *Shuffle Once*. The reason is that *Block-Only Shuffle* can only yield a *partially* random order, and the tuples in each block can all be negative or positive for the clustered data.

Since *MRS Shuffle* and *Sliding-Window Shuffle* are not available in the current MADlib/Bismarck, we use our own implementations (in PyTorch) and compare their convergence rates. Figure 8 shows the convergence rates on clustered datasets for all strategies, where *Sliding-Window*, *MRS*, and `CorgiPile` all use the same buffer size (10% of the whole dataset). As shown in Figure 8, *Sliding-Window Shuffle* suffers from lower accuracy, whereas *MRS Shuffle* only achieves comparable accuracy to *Shuffle Once* on `epsilon` and `yfcc` but suffers on the other datasets. We further perform these strategies on the feature-ordered datasets with results in Figure 9. Although *No Shuffle*, *MRS*, and *Sliding-Window* achieve higher accuracy on the feature-ordered datasets, they still have gaps with the *Shuffle Once* and `CorgiPile` for the `higgs`, `susy`, and `criteo` datasets. Only for the `epsilon` (with synthetic features [5]) and `yfcc` with image-extracted features, they can achieve similar convergence rate to *Shuffle Once* and our `CorgiPile`. We observe the similar results for mini-batch SGD, which will be detailed in Section 7.2.5.

7.2.3 Per-epoch Overhead

To study the overhead of `CorgiPile`, we compare its per-epoch execution time with the fastest *No Shuffle* baseline, as well as the single-buffer version of `CorgiPile`, as shown in Figure 10. We make the following three observations.

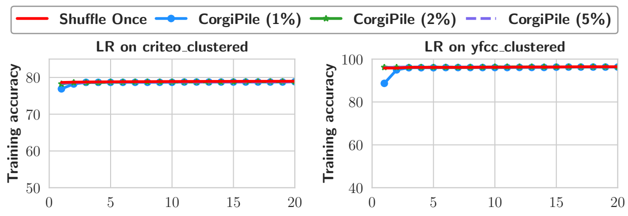
– For small datasets with in-memory I/O bandwidth, the average per-epoch time of `CorgiPile` is comparable to that of *No Shuffle*.
– For large datasets with disk I/O bandwidth, the average per-epoch time of `CorgiPile` is up to $\sim$1.1$\times$ slower than that of *No Shuffle*, i.e., it incurs at most an additional 11.7% overhead, due to buffer copy and tuple shuffle.
– By using double-buffering optimization, `CorgiPile` can achieve up to 23.6% shorter per-epoch execution time, compared to its single-buffering version.

The above results reveal that `CorgiPile` with double-buffering optimization can introduce limited overhead (11.7% longer per-epoch execution time), compared to the best *No Shuffle* baseline.

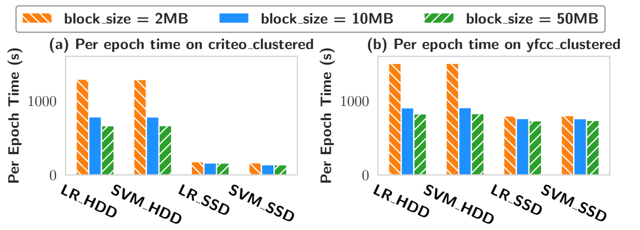
7.2.4 Sensitivity Analysis

We next study the effects of different buffer sizes and block sizes for `CorgiPile`.

The effects of buffer size. Figure 11(a) reports the convergence behavior of `CorgiPile` on the two largest datasets with different buffer sizes: 1%, 2%, and 5% of the dataset size. We see that `CorgiPile` only requires a buffer size of 2% to maintain the same convergence behavior as *Shuffle Once*. With a 1% buffer, it only converges slightly slower than *Shuffle Once*, but achieves the same final accuracy. On the other hand, as discussed in previous sections, *Sliding-Window Shuffle* and *MRS Shuffle* achieve a much lower accuracy even when given a much larger buffer (10%).



(a) `CorgiPile`'s convergence rate with varying buffer sizes. The block size is 10MB and the batch size is 1.



(b) Per-epoch time of `CorgiPile` with varying block sizes. The buffer size is 10% of the dataset and the batch size is 1.

Fig. 11: The effects of buffer size and block size on `CorgiPile`.

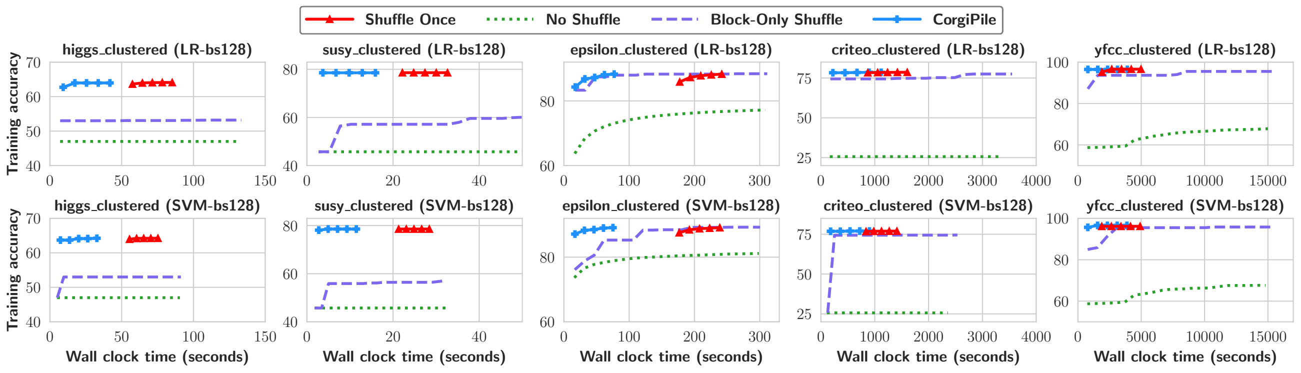


Fig. 12: The end-to-end execution time of LR and SVM using mini-batch SGD ($batch\_size = 128$) in PostgreSQL, for clustered datasets on SSD.

The effects of block size. We vary the block size in {2MB, 10MB, 50MB} on the large `criteo` and `yfcc` datasets. Figure 11(b) shows that the per-epoch time decreases as the block size increases from 2MB to 50MB, due to the higher I/O bandwidth (throughput). However, the time difference between 10MB and 50MB is limited (under 10%), because using 10MB has achieved the highest possible I/O bandwidth (130 MB/s on HDD). As the I/O performance of `CorgiPile` depends on the random accessing speed of blocks. A key question is how to choose an appropriate block size. In practice, as illustrated in Figure [12], we recommend users to choose the smallest block size that can achieve similar I/O bandwidth to the sequential read on their devices. To assist `CorgiPile` users on this task, we further developed a tool that explores the relationship between block size and disk I/O bandwidth via profiling, which is available at [27].

In the previous experiments, we focused on the standard SGD algorithm, which updates the model per tuple. Since it is also common to use mini-batch SGD, we implemented mini-batch SGD for *CorgiPile*, *Once Shuffle*, *No Shuffle*, and *Block-Only Shuffle*, using our in-DB operators in PostgreSQL. We compare these shuffling strategies only based on our PostgreSQL implementations, since MADlib and Bismarck currently do not support mini-batch SGD for linear models.

7.2.5 Mini-batch LR and SVM models

We first perform LR and SVM using mini-batch SGD on the clustered datasets. Figure 12 illustrates the end-to-end execution time of these two models in PostgreSQL on SSD. The result is similar to that of the standard SGD. Our `CorgiPile` achieves comparable convergence rate and accuracy to *Shuffle Once* but 1.7-3.3× faster than it to converge. Other strategies like *No Shuffle* and *Block-Only Shuffle* suffer from either lower converged accuracy or lower convergence rate.

In comparison with other shuffling strategies, Figure 13 and Figure 14 demonstrate the convergence rates of different shuffling strategies with $batch\_size = 128$, for both clustered datasets and feature-ordered datasets, respectively. We observe that `CorgiPile` (as well as the best *Shuffle Once* baseline) often significantly outperforms *Sliding-Window Shuffle* and *MRS Shuffle*, in terms of convergence rate and/or model accuracy, on both clustered and feature-ordered datasets. This result reveals that our `CorgiPile` also works for mini-batch SGD while other shuffling strategies are suboptimal.

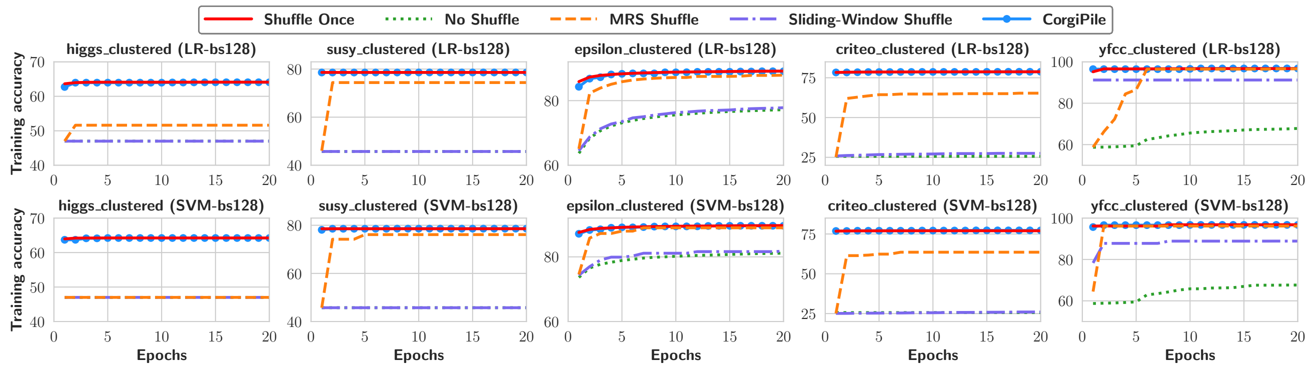


Fig. 13: The convergence rates of LR and SVM using mini-batch SGD ($batch\_size = 128$), for clustered datasets.

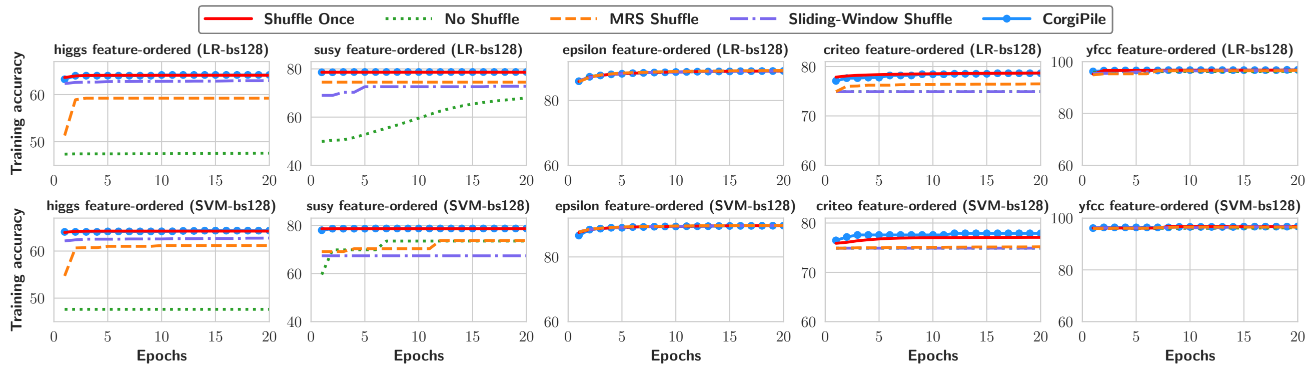


Fig. 14: The convergence rates of LR and SVM using mini-batch SGD (bs = 128), for feature-ordered datasets.

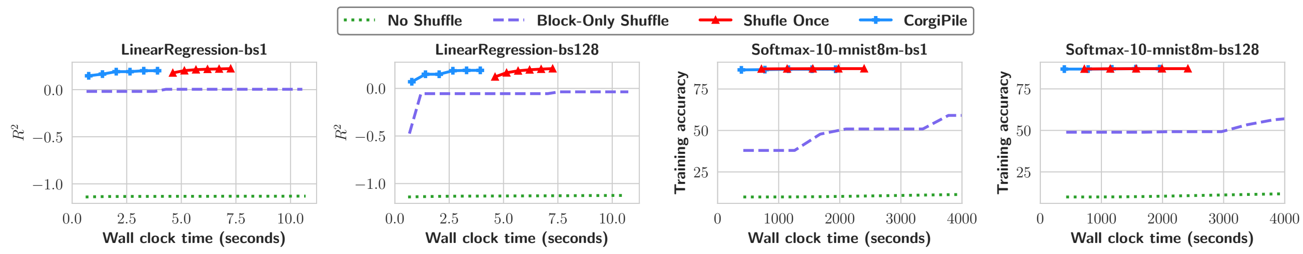


Fig. 15: The end-to-end time of linear and softmax regression in PostgreSQL using different batch sizes (bs = 1 and bs = 128), for clustered datasets on SSD.

7.2.6 Linear Regression and Softmax Regression Models

Apart from LR/SVM on binary-class datasets, users may also want to train ML models on continuous and multi-class datasets in the database. Thus, we further implemented *linear regression* for training continuous dataset and *softmax regression* (i.e., multinomial logistic regression) for multi-class datasets, based on our in-DB operators inside PostgreSQL. Figure 15 shows the end-to-end execution time of linear regression for the continuous `YearPredictionMSD` clustered dataset [13] and softmax regression for the 10-class `mini8m` clustered dataset [13], with different batch sizes on SSD. `CorgiPile` again achieves convergence rate and model accuracy (i.e., coefficient of determination R^2 for linear regression) similar to *Shuffle Once*, but is 1.6-2.1× faster to converge.

7.3 Evaluation with Deep Learning System

`CorgiPile` is a general data shuffling strategy for any SGD implementation. To understand its impact on deep learning systems and workloads, we implement the `CorgiPile` strategy as well as others in PyTorch

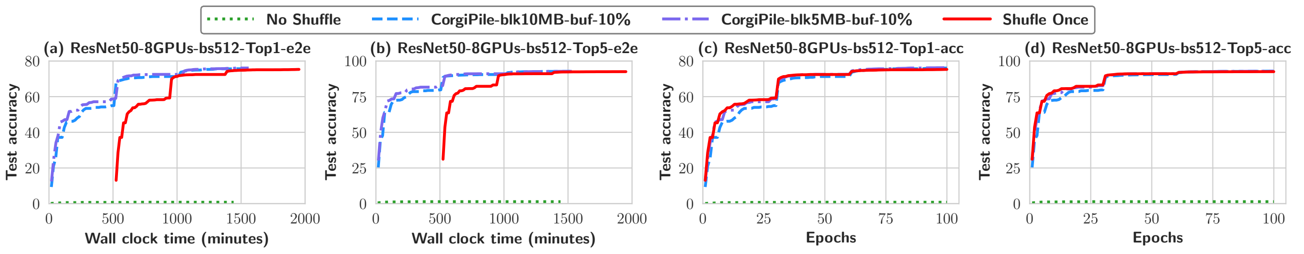


Fig. 16: The convergence rates of ResNet50 with different data shuffling strategies, for the clustered ImageNet dataset. Note that the original ImageNet dataset is clustered by the labels. TopN refers to the Top-N accuracy.

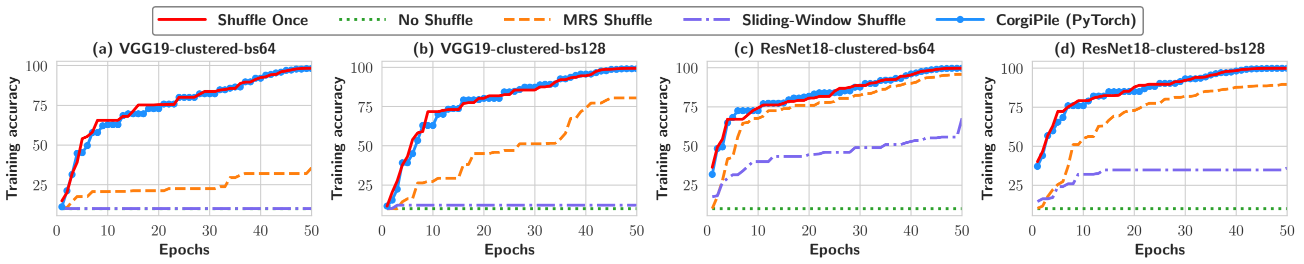


Fig. 17: The convergence rates of deep learning models with different data shuffling strategies and batch sizes, for the clustered 10-class `cifar-10` image dataset.

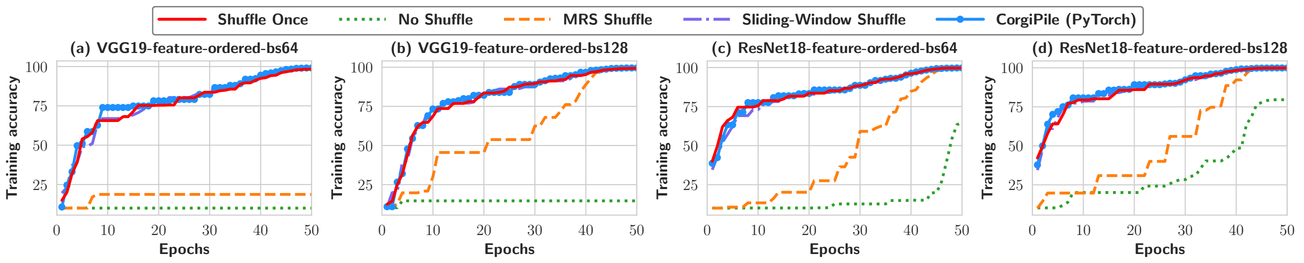


Fig. 18: The convergence rates of deep learning models with different data shuffling strategies and batch sizes, for the feature-ordered 10-class `cifar-10` image dataset.

and compare them using deep learning models, for image classification. In the following parts, we first evaluate the end-to-end performance and convergence rate of `CorgiPile` on the `ImageNet` dataset. We then study the convergence rate of `CorgiPile` in detail.

7.3.1 Performance Comparison

To evaluate the performance of `CorgiPile` in PyTorch, we train ResNet50 on `ImageNet`, which has 1.3 million images in 1,000 classes. We run this experiment using multi-process `CorgiPile` with 8 GPUs and 16 CPU cores in our cluster. We evaluate two different block sizes (5MB and 10MB, with about 50 and 100 images per block), as our cluster reads data in terms of 4MB+ blocks. The batch size is set to 512 images, so each process performs SGD computation on $512/8 = 64$ images per batch. The buffer size of each process is 1.25% of the whole dataset, thus the total buffer size of all processes is 10% of the whole dataset. The number of data loading threads for each process is set to two, as we have twice as many CPU cores as GPUs. The learning rate is initialized as 0.1 and is decayed every 30 epochs with multiplicative factor of 0.1.

Figure 16 illustrates the end-to-end execution time of ResNet50 model on the large `ImageNet` dataset, using different shuffling strategies. We report both the Top 1 and Top 5 accuracy. From Figure 16(a) and 16(b), we can observe that `CorgiPile` is 1.5× faster than *Shuffle Once* to converge and the converged accuracy of `CorgiPile` is similar to that of *Shuffle Once*. The main reason of the slowness of *Shuffle Once* is that it needs about 8.5 hours to shuffle the large (∼150 GB) `ImageNet` dataset and store the shuffled dataset in our cluster (using about 8.5 hours to randomly access raw images and merge them into large binary files [20], as Lustre file system does not allow users to store/access small raw image files [3]). Lustre is widely used in HPC clusters and has been used by many of the top supercomputers and large multi-cluster sites.[8] In contrast, `CorgiPile` eliminates this long data shuffling time. The second reason is that our `CorgiPile` has limited per-epoch overhead. Although `CorgiPile` has block shuffle and tuple shuffle overhead, the per-epoch time of `CorgiPile` with 5MB or 10MB block is only ∼15% longer than that of the fastest *No Shuffle* baseline, which is similar to that observed in the previous experiments on PostgreSQL. Recall that `CorgiPile` reads data in terms of blocks, which is comparable to sequential read on block-based parallel file system.

We further compare the convergence rate in Figure 16(c) and 16(d). We can see that the convergence rates of `CorgiPile` with 5MB/10MB block sizes are comparable to that of *Shuffle Once*. Although `CorgiPile` with 10MB block size has lower convergence rate than *Shuffle Once* in the first 30 epochs, it can catch up in the following epochs and converge to similar accuracy.

7.3.2 Convergence rate comparison

To compare `CorgiPile` with other data shuffling strategies, we perform deep learning (VGG19 and ResNet18) models on `cifar-10` image dataset using a single GPU. The `cifar-10` dataset contains 50,000 training images in 10 classes, and we use both the clustered and feature-ordered `cifar-10` dataset.

Figure 17 illustrates the convergence rates of VGG19 and ResNet18 models with different batch sizes (64 and 128) for the clustered `cifar-10` dataset. The buffer size is 10% of the whole dataset and the block size is set to 100 images per block. This figure shows that `CorgiPile` achieves comparable convergence rate and accuracy to the *Shuffle Once* baseline, whereas other strategies suffer from lower accuracy due to the partially random order of the shuffled tuples. Specifically, the *Sliding-Window Shuffle* used by TensorFlow only performs better than *No Shuffle*, and suffers from large (50%+) accuracy gap with *Shuffle Once* and `CorgiPile`.

We further repeat the experiments on the feature-ordered `cifar-10` dataset, and Figure 18 presents the results. It again shows that the convergence rate of `CorgiPile` is comparable to *Shuffle Once*, whereas *No Shuffle* and *MRS Shuffle* suffer from lower accuracy or lower convergence rate. Only *Sliding-Window Shuffle* can achieve similar convergence rate compared to *Shuffle Once* and `CorgiPile`.

The above results indicate that `CorgiPile` can achieve both good statistical efficiency and hardware efficiency for deep learning models on non-convex optimization problems. When integrated to PyTorch, `CorgiPile` is 1.5× faster than the *Shuffle Once* baseline on the large `ImageNet` dataset in our experiments.

[8] `https://en.wikipedia.org/wiki/Lustre_(file_system)`

8 Related Work

Stochastic gradient descent (SGD). SGD is broadly used in machine learning to solve large-scale optimization problems [33]. It admits the convergence rate $O(1/T)$ for strongly convex objectives, and $O(1/\sqrt{T})$ for the general convex case [65,44], where T refers to the number of iterations. For non-convex optimization problems, an ergodic convergence rate $O(1/\sqrt{T})$ is proved in [44], and the convergence rate is $O(1/T)$ (e.g., [49]) under the Polyak-Łojasiewicz condition [70]. In the analysis of the above cases, the common assumption is that data is sampled uniformly and independently *with replacement* in each epoch. We call SGD methods based on this assumption as *vanilla SGD.*

Data shuffling strategies for SGD. In practice, full shuffle SGD is a more practical and efficient way of implementing SGD [34]. In each epoch, the data is reshuffled and iterated one by one *without replacement.* Empirically, it can also be observed that random-shuffle SGD converges much faster than vanilla SGD [32,46, 49]. In Section 3, we empirically studied the state-of-the-art data shuffling strategies for SGD, including *Epoch Shuffle*, *No Shuffle*, *Shuffle Once*, *Sliding-Window Shuffle* [19] and *MRS Shuffle* [43]. Our empirical study shows that *Shuffle Once* achieves good convergence rate but suffers from low performance, whereas other strategies suffer from low accuracy. In addition, there has been previous work on *bi-sampling* [48], which has been used in the context of online aggregation [39]. Bi-sampling first selects/samples pages using Bernoulli sampling and then shuffles/samples the tuples inside each page. Unlike `CorgiPile`, it does not shuffle tuples across pages, which is similar to the *Block-Only Shuffle* used in our experimental evaluation.

In-DB ML. Previous work [84,51,43,73,58,67,37,71,57, 61,52,62,82,55,63,16] has intensively discussed how to implement ML models on relational data, such as linear models [73,58,67], linear algebra [37,61,62], factorization models [71], neural networks [52,62,82] and other statistical learning models [57], using Batch Gradient Descent (BGD) or SGD, over join or self-defined matrix/tensors, etc. The most common way of integrating ML algorithm into RDBMS is to use User-Defined Aggregate Functions (UDA). The representative in-DB ML tools are Apache MADlib [51,2] and

Bismarck [43], which use PostgreSQL's UDAs to implement SGD, and leverage SQL LOOP (Bismarck) or Python driver (MADlib) to implement iterations. Recently, DB4ML [53] proposes another approach called *iterative transactions* to implement iterative SGD/graph algorithm in DB. However, it still uses/assumes the *Shuffle Once* strategy as that of Bismarck/MADlib. Since the source code of DB4ML has not been released yet, we only compare with MADlib and Bismarck.

Scalable ML for the distributed data systems. In recent years, there are active research on integrating ML models into distributed database systems to enable scalable ML, such as MADlib on Greenplum [21], Vertica-ML [42], Google's BigQuery ML [8], Microsoft SQL Server ML Services [15], etc. Another trend is to leverage big data systems to build scalable ML models based on different architectures, e.g., MPI [38,56], MapReduce [64,86,36], Parameter Server [41,78,54] and decentralization [59, 76]. Recent work also started discussing how to integrate deep learning into databases [85,66]. Our CorgiPile is a general data shuffling strategy for SGD and has been integrated into PostgreSQL and PyTorch. We believe that CorgiPile can be potentially integrated into more above distributed data systems.

9 Conclusion

We have presented CorgiPile, a novel data shuffling strategy for efficient SGD computation on top of block-addressable secondary storage systems. It adopts a two-level hierarchical shuffle mechanism that avoids the computation and storage overhead of full data shuffling while retaining similar convergence rate of SGD as if a full data shuffle were performed. We provide theoretical analysis on the convergence behavior of CorgiPile and further integrate it into both PostgreSQL and PyTorch. Experimental evaluation demonstrates both statistical and hardware efficiency of CorgiPile when compared to state-of-the-art in-DB ML and deep learning systems.

References

1. Amazon Elastic Block Store (EBS). https://aws.amazon.com/ebs
2. Apache MADlib: Big Data Machine Learning in SQL. http://madlib.apache.org/
3. Best practices on Lustre parallel file systems. https://scicomp.ethz.ch/wiki/Best_practices_on_Lustre_parallel_file_systems
4. CIFAR-10 dataset. http://www.cs.toronto.edu/~kriz/cifar.html
5. Epsilon dataset. https://www.k4all.org/project/large-scale-learning-challenge/
6. ETH Euler Cluster. https://scicomp.ethz.ch/wiki/Euler
7. Getting Started With Distributed Data Parallel. https://pytorch.org/tutorials/intermediate/ddp_tutorial.html
8. Google BigQuery ML. https://cloud.google.com/bigquery-ml/docs/introduction
9. Hadoop HDFS Architecture. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html
10. ImageNet dataset. https://www.image-net.org/
11. ImageNet training in PyTorch. https://github.com/pytorch/examples/tree/main/imagenet
12. I/O performance test on HDD and SSD with distinct block sizes. https://anonymous.4open.science/r/corgipile/IO-perf.pdf
13. LIBSVM Data. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
14. Lustre reads/writes data in blocks. https://scicomp.ethz.ch/wiki/Conda
15. Microsoft SQL Server Machine Learning Services. https://docs.microsoft.com/en-us/sql/machine-learning/sql-server-machine-learning-services?view=sql-server-ver15
16. Oracle R Enterprise Versions of R Models. https://docs.oracle.com/cd/E11882_01/doc.112/e36761/orelm.htm
17. PostgreSQL. https://www.postgresql.org/
18. PostgreSQL TOAST. https://www.postgresql.org/docs/9.5/storage-toast.html
19. Sliding-Window Shuffle in TensorFlow. https://www.tensorflow.org/api_docs/python/tf/data/Dataset
20. TFRecord format for PyTorch. https://github.com/vahidk/tfrecord
21. The Greenplum MADlib extension. https://greenplum.docs.pivotal.io/6-19/analytics/madlib.html
22. The Lustre file system. https://www.lustre.org/
23. The optimization algorithms in PyTorch. https://pytorch.org/docs/stable/optim.html
24. The optimization algorithms in TensorFlow. https://www.tensorflow.org/api_docs/python/tf/keras/optimizers
25. The TFRecord format for storing a sequence of binary records. https://www.tensorflow.org/tutorials/load_data/tfrecord
26. Buffer Manager of PostgreSQL. https://www.interdb.jp/pg/pgsql08.html (2022)
27. Bandwidth Bench. https://github.com/JerryLead/bandwidth-bench (2024)
28. MADlib-Deep Learning. https://madlib.apache.org/docs/latest/group__grp__keras.html (2024)
29. PyTorch's shuffling strategy. https://pytorch.org/data/main/generated/torchdata.datapipes.iter.Shuffler.html (2024)
30. Abadi, M., Barham, P., Chen, J., et al: Tensorflow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, pp. 265–283. USENIX Association (2016)
31. Arpaci-Dusseau, R.H., Arpaci-Dusseau, A.C.: Operating Systems: Three Easy Pieces, 1.00 edn. Arpaci-Dusseau Books (2018)
32. Bottou, L.: Curiously fast convergence of some stochastic gradient descent algorithms. In: Proceedings of the symposium on learning and data science, Paris, vol. 8, pp. 2624–2633 (2009)
33. Bottou, L.: Large-scale machine learning with stochastic gradient descent. In: 19th International Conference on Computational Statistics, COMPSTAT 2010, pp. 177–186 (2010)

34. Bottou, L.: Stochastic gradient descent tricks. In: Neural networks: Tricks of the trade, pp. 421–436. Springer (2012)
35. Bottou, L., Curtis, F.E., Nocedal, J.: Optimization methods for large-scale machine learning. SIAM Rev. **60**(2), 223–311 (2018)
36. Cai, Z., Vagena, Z., Perez, L.L., Arumugam, S., Haas, P.J., Jermaine, C.M.: Simulation of database-valued markov chains using simsql. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, pp. 637–648. ACM (2013)
37. Chen, L., Kumar, A., Naughton, J.F., Patel, J.M.: Towards linear algebra over normalized data. Proc. VLDB Endow. **10**(11), 1214–1225 (2017)
38. Chen, T., He, T., Benesty, M., Khotilovich, V., Tang, Y., Cho, H., et al.: Xgboost: extreme gradient boosting. R package version 0.4-2 **1**(4), 1–4 (2015)
39. Cheng, Y., Zhao, W., Rusu, F.: Bi-level online aggregation on raw data. In: Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017, pp. 10:1–10:12. ACM (2017)
40. De Sa, C.M.: Random reshuffling is not always better. Advances in Neural Information Processing Systems **33** (2020)
41. Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al.: Large scale distributed deep networks. Advances in neural information processing systems **25**, 1223–1231 (2012)
42. Fard, A., Le, A., Larionov, G., Dhillon, W., Bear, C.: Vertica-ml: Distributed machine learning in vertica database. In: Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, pp. 755–768. ACM (2020)
43. Feng, X., Kumar, A., Recht, B., Ré, C.: Towards a unified architecture for in-rdbms analytics. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, pp. 325–336 (2012)
44. Ghadimi, S., Lan, G.: Stochastic first-and zeroth-order methods for nonconvex stochastic programming. SIAM Journal on Optimization **23**(4), 2341–2368 (2013)
45. Graefe, G.: Volcano - an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng. **6**(1), 120–135 (1994)
46. Gürbüzbalaban, M., Ozdaglar, A., Parrilo, P.A.: Why random reshuffling beats stochastic gradient descent. Mathematical Programming pp. 1–36 (2019)
47. Gürbüzbalaban, M., Ozdaglar, A.E., Parrilo, P.A.: Why random reshuffling beats stochastic gradient descent. Math. Program. **186**(1), 49–84 (2021)
48. Haas, P.J., Koenig, C.: A bi-level bernoulli scheme for database sampling. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2004, pp. 275–286. ACM (2004)
49. HaoChen, J.Z., Sra, S.: Random shuffling beats SGD after finite epochs. In: Proceedings of the 36th International Conference on Machine Learning, ICML 2019,, *Proceedings of Machine Learning Research*, vol. 97, pp. 2624–2633. PMLR (2019)
50. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, pp. 770–778 (2016)
51. Hellerstein, J.M., Ré, C., Schoppmann, F., Wang, D.Z., Fratkin, E., Gorajek, A., Ng, K.S., Welton, C., Feng, X., Li, K., Kumar, A.: The madlib analytics library or MAD skills, the SQL. Proc. VLDB Endow. **5**(12), 1700–1711 (2012)
52. Jankov, D., Yuan, B., Luo, S., Jermaine, C.: Distributed numerical and machine learning computations via two-phase execution of aggregated join trees. Proc. VLDB Endow. **14**(7), 1228–1240 (2021)
53. Jasny, M., Ziegler, T., Kraska, T., Röhm, U., Binnig, C.: DB4ML - an in-memory database kernel with machine learning support. In: Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, pp. 159–173. ACM (2020)
54. Jiang, J., Cui, B., Zhang, C., Yu, L.: Heterogeneity-aware distributed parameter servers. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 463–478 (2017)
55. Kara, K., Eguro, K., Zhang, C., Alonso, G.: Columnml: Column-store machine learning with on-the-fly data transformation. Proceedings of the VLDB Endowment **12**(4), 348–361 (2018)
56. Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., Liu, T.Y.: Lightgbm: A highly efficient gradient boosting decision tree. Advances in neural information processing systems **30**, 3146–3154 (2017)
57. Khamis, M.A., Ngo, H.Q., Nguyen, X., Olteanu, D., Schleich, M.: In-database learning with sparse tensors. In: Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, pp. 325–340. ACM (2018)
58. Kumar, A., Naughton, J.F., Patel, J.M.: Learning generalized linear models over normalized data. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1969–1984. ACM (2015)
59. Lian, X., Zhang, C., Zhang, H., Hsieh, C.J., Zhang, W., Liu, J.: Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In: Proceedings of the 31st International Conference on Neural Information Processing Systems, pp. 5336–5346 (2017)
60. Liu, J., Zhang, C.: Distributed learning systems with first-order methods. Found. Trends Databases **9**(1), 1–100 (2020)
61. Luo, S., Gao, Z.J., Gubanov, M.N., Perez, L.L., Jankov, D., Jermaine, C.M.: Scalable linear algebra on a relational database system. Commun. ACM **63**(8), 93–101 (2020)
62. Luo, S., Jankov, D., Yuan, B., Jermaine, C.: Automatic optimization of matrix implementations for distributed machine learning and linear algebra. In: Proceedings of the 2021 International Conference on Management of Data, pp. 1222–1234 (2021)
63. MacGregor, J.: Predictive Analysis with SAP. Bonn: Galileo Press (2013)
64. Meng, X., Bradley, J.K., Yavuz, B., et al: Mllib: Machine learning in apache spark. J. Mach. Learn. Res. **17**, 34:1–34:7 (2016)
65. Moulines, E., Bach, F.R.: Non-asymptotic analysis of stochastic approximation algorithms for machine learning. In: Advances in Neural Information Processing Systems, pp. 451–459 (2011)
66. Nakandala, S., Zhang, Y., Kumar, A.: Cerebro: A data system for optimized deep learning model selection. Proc. VLDB Endow. **13**(11), 2159–2173 (2020)
67. Olteanu, D., Schleich, M.: F: regression models over factorized views. Proc. VLDB Endow. **9**(13), 1573–1576 (2016)
68. Paszke, A., Gross, S., Massa, F., et al: Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, pp. 8024–8035 (2019)

69. Petersen, T.K.: Inside the lustre file system. SEAGATE Technology paper (2015)
70. Polyak, B.T.: Gradient methods for minimizing functionals. Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki **3**(4), 643–653 (1963)
71. Rendle, S.: Scaling factorization machines to relational data. Proc. VLDB Endow. **6**(5), 337–348 (2013)
72. Ruder, S.: An overview of gradient descent optimization algorithms. CoRR **abs/1609.04747** (2016)
73. Schleich, M., Olteanu, D., Ciucanu, R.: Learning linear regression models over factorized joins. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, pp. 3–18. ACM (2016)
74. Shamir, O.: Without-replacement sampling for stochastic gradient methods. In: Advances in Neural Information Processing Systems, pp. 46–54 (2016)
75. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: 3rd International Conference on Learning Representations, ICLR 2015 (2015)
76. Tang, H., Lian, X., Yan, M., Zhang, C., Liu, J.: D^2: Decentralized training over decentralized data. In: Proceedings of the 35th International Conference on Machine Learning, ICML 2018, *Proceedings of Machine Learning Research*, vol. 80, pp. 4855–4863. PMLR (2018)
77. Thomee, B., Shamma, D.A., Friedland, G., Elizalde, B., Ni, K., Poland, D., Borth, D., Li, L.: YFCC100M: the new data in multimedia research. Commun. ACM **59**(2), 64–73 (2016)
78. Xing, E.P., Ho, Q., Dai, W., Kim, J.K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., Yu, Y.: Petuum: A new platform for distributed machine learning on big data. IEEE transactions on Big Data **1**(2), 49–67 (2015)
79. Xu, L., Qiu, S., Yuan, B., Jiang, J., Renggli, C., Gan, S., Kara, K., Li, G., Liu, J., Wu, W., Ye, J., Zhang, C.: In-database machine learning with corgipile: Stochastic gradient descent without full data shuffle. In: SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022, pp. 1286–1300. ACM (2022)
80. Xu, L., Qiu, S., Yuan, B., Jiang, J., Renggli, C., Gan, S., Kara, K., Li, G., Liu, J., Wu, W., Ye, J., Zhang, C.: Stochastic gradient descent without full data shuffle. CoRR **abs/2206.05830** (2022). DOI 10.48550/arXiv.2206.05830. URL `https://doi.org/10.48550/arXiv.2206.05830`
81. Ying, B., Yuan, K., Vlaski, S., Sayed, A.H.: Stochastic learning under random reshuffling with constant step-sizes. IEEE Transactions on Signal Processing **67**(2), 474–489 (2019)
82. Yuan, B., Jankov, D., Zou, J., Tang, Y., Bourgeois, D., Jermaine, C.: Tensor relational algebra for distributed machine learning system design. Proc. VLDB Endow. **14**(8), 1338–1350 (2021)
83. Yun, C., Sra, S., Jadbabaie, A.: Open problem: Can single-shuffle SGD be better than reshuffling SGD and gd? In: Conference on Learning Theory, COLT 2021, *Proceedings of Machine Learning Research*, vol. 134, pp. 4653–4658. PMLR (2021)
84. Zhang, C., Ré, C.: Dimmwitted: A study of main-memory statistical analytics. Proceedings of the VLDB Endowment **7**(12) (2014)
85. Zhang, Y., Mcquillan, F., Jayaram, N., Kak, N., Khanna, E., Kislal, O., Valdano, D., Kumar, A.: Distributed deep learning on data systems: A comparative analysis of approaches. Proc. VLDB Endow. **14**(10), 1769–1782 (2021)
86. Zhang, Z., Jiang, J., Wu, W., Zhang, C., Yu, L., Cui, B.: Mllib*: Fast training of glms using spark mllib. In: 35th IEEE International Conference on Data Engineering, ICDE 2019, pp. 1778–1789. IEEE (2019)