# In-Database Machine Learning with CorgiPile: Stochastic Gradient Descent without Full Data Shuffle

Lijie Xu[†‡], Shuang Qiu[§], Binhang Yuan[†], Jiawei Jiang[†], Cedric Renggli[†], Shaoduo Gan[†], Kaan Kara[†], Guoliang Li[¶], Ji Liu[♯], Wentao Wu[♭], Jieping Ye[♮], Ce Zhang[†]

[†]ETH Zürich [‡]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[§]University of Chicago [¶]Tsinghua University [♯]Kwai Inc. [♭]Microsoft Research [♮]University of Michigan
[†]{firstname.lastname}@inf.ethz.ch
[¶]liguoliang@tsinghua.edu.cn [♯]ji.liu.uwisc@gmail.com [♭]wentao.wu@microsoft.com [§♮]{qiush, jpye}@umich.edu

## ABSTRACT

Stochastic gradient descent (SGD) is the cornerstone of modern ML systems. Despite its computational efficiency, SGD requires random data access that is inherently inefficient when implemented in systems that rely on *block-addressable secondary storage* such as HDD and SSD, e.g., in-DB ML systems and TensorFlow/PyTorch over large files. To address this impedance mismatch, various data shuffling strategies have been proposed to balance the convergence rate of SGD (which favors randomness) and its I/O performance (which favors sequential access).

In this paper, we first conduct a systematic empirical study on existing data shuffling strategies, which reveals that all existing strategies have room for improvement—they suffer in terms of I/O performance *or* convergence rate. With this in mind, we propose a simple but novel *hierarchical* data shuffling strategy, `CorgiPile`. Compared with existing strategies, `CorgiPile` *avoids* a full data shuffle while maintaining *comparable convergence rate* of SGD as if a full shuffle were performed. We provide a non-trivial theoretical analysis of `CorgiPile` on its convergence behavior. We further integrate `CorgiPile` into PostgreSQL by introducing three new *physical* operators with optimizations. Our experimental results show that `CorgiPile` can achieve comparable convergence rate to the full shuffle based SGD, and 1.6×-12.8× faster than two state-of-the-art in-DB ML systems, Apache MADlib and Bismarck, on both HDD and SSD.

## CCS CONCEPTS

• **Information systems** → **Database management system engines**; • **Computing methodologies** → **Machine learning**.

## KEYWORDS

In-database machine learning; Stochastic Gradient Descent; Shuffle

**ACM Reference Format:**
Lijie Xu, Shuang Qiu, Binhang Yuan, Jiawei Jiang, Cedric Renggli, Shaoduo Gan, Kaan Kara, Guoliang Li, Ji Liu, Wentao Wu, Jieping Ye, Ce Zhang. 2022. In-Database Machine Learning with CorgiPile: Stochastic Gradient Descent without Full Data Shuffle. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3526150


*SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA.*

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526150

## 1 INTRODUCTION

Stochastic gradient descent (SGD) is the cornerstone of modern ML systems. With ever-growing data volume, inevitably, SGD algorithms have to access data stored in the *secondary storage* instead of accessing the DRAM directly. This can happen in two prominent applications: (1) for many *in-database machine learning* (in-DB ML) scenarios, one has to assume that the data is stored on the secondary storage, managed by the buffer manager [5]; and (2) in *deep learning systems* such as TensorFlow [15], one needs to support out-of-memory access via a specialized scanner over large files.

*In-database Machine Learning.* In-DB ML has been a popular research area in the database community for years [22, 27, 34, 41, 43, 46, 53, 57, 59]. One major benefit of in-DB ML is that users do not need to move the data out of the database to another specialized ML platform, given that data movement is often time-consuming, error-prone, or even impossible (e.g., due to privacy and security compliance concerns). Instead, users can define their ML training jobs using SQL, e.g., training an SVM model with MADlib [4, 34] and Bismarck [27] can be done via a single SQL statement:

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

*A Fundamental Discrepancy.* As identified by previous work [27, 39, 69], one unique challenge of in-DB ML is that data can be *clustered* while shuffling is not always feasible. For example, the data is clustered by the label, where data with negative labels might always come before data with positive labels [27]. Another example is that the data is ordered by one of the features. These are common cases when there is a clustered B-tree index, or the data is naturally grouped/ordered by, e.g., timestamps. As SGD requires a random data order (shuffle over *all* data) to converge [24, 27, 31, 32, 50, 56, 58, 68], directly running sequential scans over such a clustered dataset can slowdown the convergence of SGD.

Meanwhile, when data are stored on block-addressable secondary storage such as HDD and SSD, it can be incredibly expensive to either shuffle the data *on-the-fly* when running SGD, or shuffle the data *once* with copy and run SGD over the shuffled copy, due to the amount of *random* I/O's. Sometimes, data shuffling might not be applicable in database systems—in-place shuffling might have an

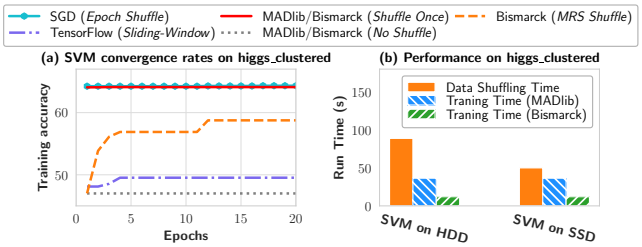



**Figure 1: The convergence rate and performance of SVM on clustered *higgs* dataset. (a) Today's SGD systems over secondary storage, including in-DB ML solutions (MADlib/Bismarck) and TensorFlow file scanner, are sensitive to clustered data order. (b) Forcing a full data shuffle before training accommodates this clustered data issue, but introduces large overhead that is often more expensive than training itself.**

impact on other indices, whereas shuffling over a data copy introduces a 2× storage overhead. *How to design efficient SGD algorithms without requiring even a single pass of full data shuffle?* Understanding this question can have a profound impact to the system design of both in-DB ML systems and deep learning systems.

*Existing Landscape and Challenges.* Various solutions have been proposed, in the context of both in-DB ML and deep learning systems. In Bismarck [27], the authors suggest a "multiplexed reservoir sampling" (MRS) shuffling strategy, in which two concurrent threads update the same model—one reads data sequentially with reservoir sampling and the other reads from a small, shuffled in-memory buffer. Alternatively, TensorFlow provides a shuffling strategy based on a *sliding window* over the data [12]. Both significantly improve the SGD convergence rate and have been widely adopted in practice. Despite these efforts, however, they suffer from some shortcomings. As illustrated in Figure 1(a), both strategies proposed by Bismarck and TensorFlow perform suboptimally given a *clustered* data order. Meanwhile, the idea of shuffling data once *before* training, i.e., the curve corresponding to "MADlib/Bismarck (*Shuffle Once*)," can accommodate for such convergence problem but also introduce a significant overhead as shown in Figure 1(b), which is consistent with the observations from previous work [27].

*Our Contributions.* Inspired by previous efforts, we study the following questions and make four contributions in this paper:

> *Can we design an SGD-style algorithm with DB-friendly shuffling strategy that can converge without requiring a full data shuffle? Can we provide a rigorous theoretical analysis on its convergence behavior? Can we integrate such an algorithm into DBMS to support in-DB ML?*

**C1. An Anatomy and Empirical Study of Existing Algorithms.** We start with a systematic evaluation 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 evaluate them in the context of using SGD to train generalized linear models and deep learning models, over a variety of datasets. Our evaluation reveals that existing strategies cannot *simultaneously* achieve good hardware efficiency (I/O performance) and statistical efficiency (convergence rate and converged accuracy). Specifically, *Epoch Shuffle* and *Shuffle Once* achieve the best statistical efficiency, since the data has been fully shuffled; however, their hardware efficiency is suboptimal given the 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 might converge slowly or even diverge. The other two strategies, *Sliding-Window Shuffle* and *MRS Shuffle*, can be viewed as a compromise between *Shuffle Once* and *No Shuffle*, which trade statistical efficiency for hardware efficiency. Nevertheless, both strategies suffer in terms of statistical efficiency (Section 3).

**C2. A Simple, but Novel, Algorithm with Rigorous Theoretical Analysis.** Motivated by the limitations of existing strategies, we propose `CorgiPile`, a novel SGD-style algorithm based on a *two-level hierarchical* data shuffle strategy.[1] The main idea is to first sample and shuffle the data at a *block level*, and then shuffle data at a *tuple level* within the sampled data blocks, i.e., first sampling data blocks (a batch of table pages per block), then merging the sampled blocks in a buffer, and finally shuffling the tuples in the buffer for SGD. While this two-level strategy seems quite simple, it can achieve both good hardware efficiency and statistical efficiency. Although the hardware efficiency is easy to understand—accessing random data blocks is much more efficient than accessing random tuples, especially when the block size is large, the statistical efficiency requires some non-trivial analysis. To this end, we further provide a rigorous theoretical study on the convergence behavior.

**C3. Implementation, Optimization, and Deep Integration with PostgreSQL.** While the benefit of `CorgiPile` for hardware efficiency is intuitive, its realization 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. Therefore, we operate at the "physical level" and enable in-DB ML inside PostgreSQL [10] via three new *physical operators*: `BlockShuffle`, `TupleShuffle`, as well as `SGD` operator for our customized SGD implementation[2]. We can then construct an *execution plan* for the SGD computation by chaining these operators together to form a pipeline, naturally following the built-in Volcano paradigm [29] of PostgreSQL. We also design a *double-buffering* mechanism to optimize the `TupleShuffle` operator.

**C4. Extensive Empirical Evaluations.** We conduct extensive evaluations to demonstrate the effectiveness of `CorgiPile`. Specifically, we compare the end-to-end performance of our PostgreSQL implementation with two state-of-the-art in-DB ML systems, MADlib and Bismarck. We show that `CorgiPile` achieves comparable training/testing accuracy to the best *Shuffle Once* baseline, but is significantly faster since it does not require full data shuffle. Other data shuffling strategies suffer from lower SGD convergence rate on *clustered* datasets. Overall, `CorgiPile` can achieve 1.6×-12.8× speedup compared to MADlib and Bismarck over clustered data. Furthermore, for the datasets ordered by features instead of the label, our `CorgiPile` still achieves comparable accuracy to the *Shuffle Once*, whereas *No Shuffle* suffers from lower accuracy.

[1] Although we give unquestionable love to dogs, the name 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 is available at https://github.com/DS3Lab/CorgiPile-PostgreSQL.

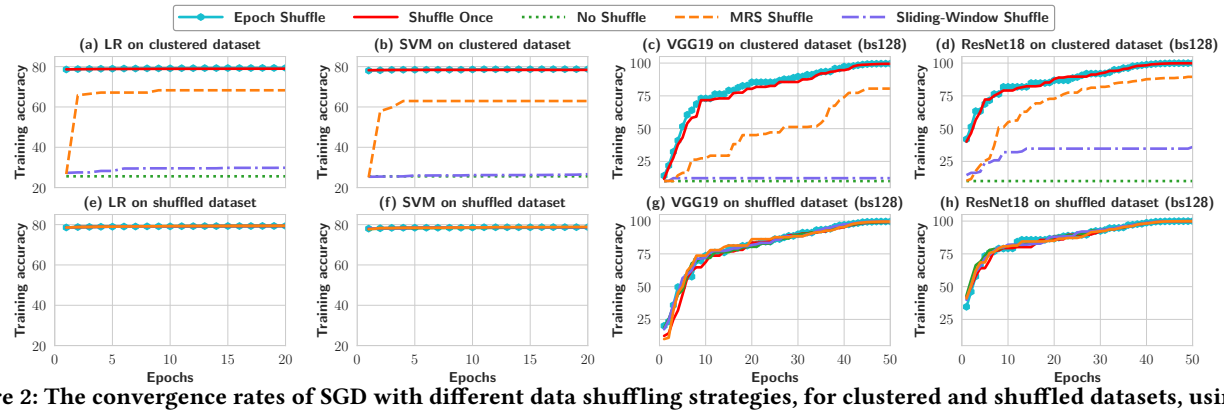



**Figure 2: The convergence rates of SGD with different data shuffling strategies, for clustered and shuffled datasets, using the same buffer size (10% of the dataset size) for MRS and Sliding-Window Shuffles.**

## 2 PRELIMINARIES

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

### 2.1 Stochastic Gradient Descent (SGD)

Given a dataset with $m$ training examples $\{\mathbf{t}_i\}_{i\in[m]}$, e.g., $m$ tuples if the training set is stored as a table in a database, the typical ML task essentially solves an optimization problem that can be cast into minimizing a finite sum over $m$ data examples w.r.t. model $\mathbf{x}$:

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

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

(1) **Initialization** – Initialize the model $\mathbf{x}$, often randomly.
(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 variant, where the random tuples are drawn *without replacement* [19, 24, 27, 68]. To achieve this, one *shuffles* all tuples *before* each epoch and sequentially scans these shuffled tuples. For each tuple, we compute the stochastic gradient and update the model parameters.
(3) **Termination** – The procedure ends when it converges (i.e., the parameters of model $\mathbf{x}$ no longer change) or has attained the maximum number of epochs.

### 2.2 In-database Machine Learning

There has been a plethora of work in the past decade focusing on in-DB ML [22, 27, 34, 36, 39, 41, 43, 46, 47, 53, 57, 59, 67, 69]. Most existing in-DB ML systems implement SGD as "User-Defined Aggregates" (UDA) [27, 34]. 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 [34] and DB4ML [37] assume that the training data has already been shuffled, so they do not perform any data shuffling. Other systems, such as Bismarck [27], 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.

## 3 DATA SHUFFLING STRATEGIES FOR SGD

In this section, we present a systematic analysis of data shuffling strategies used by existing in-DB ML systems. We consider five common data shuffling strategies: (1) *Epoch Shuffle*, (2) *Shuffle Once*, (3) *No Shuffle*, (4) *Sliding-Window Shuffle* [12], and (5) *MRS Shuffle* [27]. 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 [61] and ResNet [33].

**Experimental Setups.** We use the `criteo` dataset [3] for generalized linear models, and use the `cifar-10` image dataset [2] for deep learning. Each dataset has two versions: a *shuffled* version and a *clustered* version. In the *shuffled* version, all tuples are randomly shuffled, whereas in the *clustered* version all tuples are clustered by their *labels*. The use of *clustered* datasets is inspired by similar settings leveraged in [27], 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 in the database. SGD is then executed over this shuffled copy without any further shuffle during the execution of SGD. 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 set 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* shuffled and clustered datasets, confirming previous observations [27].

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

| Shuffling Strategy | Convergence Behavior | I/O Perf. | In-memory Buffer | Additional Disk Space |
|---|---|---|---|---|
| *No Shuffle* | Slow; Lower Accuracy | **Fast** | **No** | **No** |
| *Epoch Shuffle* | **Fast; High Accuracy** | Slow | Yes | 2× data size if in-place shuffle impossible |
| *Shuffle Once* | **Fast; High Accuracy** | Slow | Yes | 2× data size if in-place shuffle impossible |
| *MRS Shuffle* [27] | Worse than *Shuffle Once* | **Fast** | Yes | **No** |
| *Sliding-Window* [12] | Worse than *Shuffle Once* | **Fast** | Yes | **No** |
| `CorgiPile` | **Comparable to *Shuffle Once*** | **Fast** | Yes | **No** |

**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 [27]. 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. Simply running MADlib over a dataset picks the *No Shuffle* strategy.

**Convergence.** On *shuffled* data, *No Shuffle* can achieve comparable convergence rate to *Shuffle Once*. However, for *clustered* data, *No Shuffle* leads to a significantly lower model accuracy. This is not surprising, as SGD relies on a random data order to converge.

**Performance.** *No Shuffle* is the fastest data shuffling strategy, as it can *sequentially*, instead of randomly, access the data tuples [16].

## 3.3 "Sliding-Window Shuffle"

The *Sliding-Window Shuffle* strategy uses a sliding window to perform *partial* data shuffle as follows, which is used by TensorFlow [12].

(1) Allocate a sliding window and fill tuples 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.

**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 [27]. The first thread sequentially scans the dataset and performs *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 sampled tuples using another buffer $B_2$ for SGD, where tuples are simply copied from the buffer $B_1$.

**Convergence.** As illustrated in Figure 2, *MRS Shuffle* achieves higher accuracy than *Sliding-Window Shuffle* but lower accuracy than *Shuffle Once* when SGD converges. The reason is quite similar to that given to *Sliding-Window Shuffle*, as the shuffle based on reservoir sampling is again *partial* and therefore is insufficient when dealing with clustered data. Specifically, the order of the *dropped* tuples is also generally *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 thus decrease the model accuracy.

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

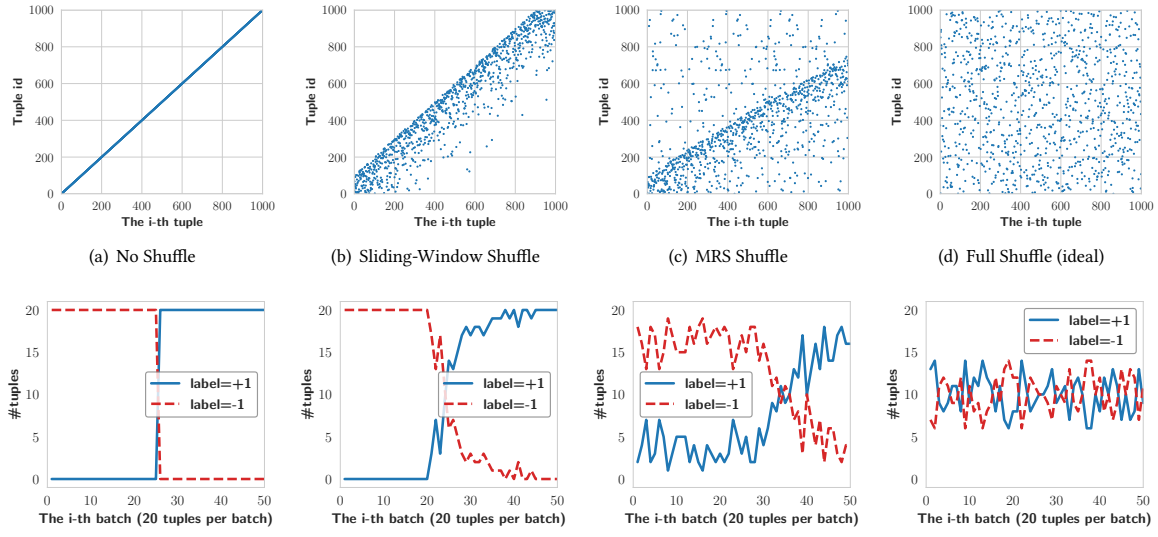
## 3.5 Analysis and Summary

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 it is too expensive to implement in in-DB ML systems. *Shuffle Once* also introduces significant data randomness, which is the best practice for in-DB ML.
- 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* and *MRS*) try to sacrifice data randomness for better I/O efficiency, leaving 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 distributions of* tuple_id *and corresponding* label*s after Sliding-Window and MRS Shuffle, and compare them with the ideal distributions from a full shuffle. Specifically, 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.*




(a) No Shuffle (b) Sliding-Window Shuffle (c) MRS Shuffle (d) Full Shuffle (ideal)

(e) No Shuffle (f) Sliding-Window Shuffle (g) MRS Shuffle (h) Full Shuffle (ideal)

**Figure 3: The tuple id distribution (a-d) and the corresponding label distribution (e-h) of data shuffling strategies. Tuple id denotes the tuple position after shuffling. #tuple denotes the number of negative/positive tuples in every 20 tuples shuffled.**

*We can observe that Sliding-Window 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(f) further confirms this, where almost all negative labels still appear before positive ones after shuffling. Similar patterns can be observed for MRS in Figures 3(c) and 3(g), though MRS has improved over Sliding-Window. In summary, the data randomness achieved by Sliding-Window or MRS is far from the ideal case, as shown in Figures 3(d) and 3(h).*

## 4 CORGIPILE

Inspired by 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 SGD.*

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 our I/O performance test on HDD and SSD [7]. In terms of statistical efficiency, as we will show, *given the same buffer size*, `CorgiPile` converges much better than *Sliding-Window* and *MRS*. Nevertheless, both the convergence analysis and its integration into PostgreSQL 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 $\ell_2$-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 $a_s \le C g_s$ for all $s$;
- $N$, the total number of blocks ($N \ge 2$);

**Algorithm 1** `CorgiPile` Algorithm

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

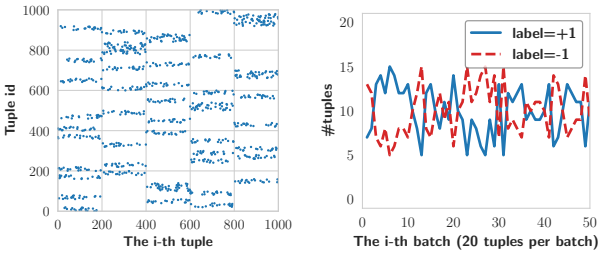
- $n$, the buffer size (i.e., the number of blocks kept in the buffer);
- $b$, the size (number of tuples) of each data block;
- $B_l$, the set of tuple indices in the $l$-th block ($l \in [N]$ and $|B_l| = b$);
- $m$, the number of tuples for the finite-sum objective ($m = Nb$);
- $f_i(\cdot)$, the 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;
- $\mu$-strongly convexity: function $F(\mathbf{x})$ is $\mu$-strongly convex if $\forall \mathbf{x}, \mathbf{y}$,

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

### 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




(a) Tuple id distribution of `CorgiPile` (b) Label distribution of `CorgiPile`

**Figure 4: The tuple id/label distribution of `CorgiPile`.**

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 [17, 19, 27, 30, 32].

(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 function averaging the gradients of all samples in the tuple indexed by $\boldsymbol{\psi}_s(k)$, and $\eta_s$ is the learning rate for gradient descent at the epoch $s$. 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 4(a). 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 4(b), is closer to a* uniform *distribution.*

**Performance.** While *No Shuffle* only requires *sequential* I/O, 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., 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, 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 `CorgiPile` is not trivial—even reasoning about the convergence of SGD with *sample without replacement* is an open question for decades [31, 32, 60, 66], not to say a hierarchical sampling scheme like ours. Luckily, a recent theoretical advancement [32] 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 analysis, one epoch represents going through all the tuples in the sampled $n$ blocks.

Assumption 1. *We make the following standard assumptions, as that in other previous work on SGD convergence analysis [20, 45]:*

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

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

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

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 the deep learning models) respectively, in order to show the correctness and efficiency of `CorgiPile`. The proof of the following theorems is available at [14].

*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 $\mu$-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*

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

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

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

**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 [32]. 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 [32].
- $\alpha = 0$: 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 vanilla SGD, we only randomly select one tuple from the dataset to update the model. It admits the convergence rate $O(\sigma^2/T)$. Comparing to the leading term $(1-\alpha)h_D(\sigma^2/T)$ in (2) for our algorithm, if $n \gg (h_D - 1)(N - 1)/h_D + 1$ (for $h_D > 0$), $(1-\alpha)h_D$ will be much smaller than 1, indicating that our algorithm outperforms vanilla SGD in terms of sample complexity. It is also worth noting that, even if $n$ is small, `CorgiPile` may still significantly outperform vanilla SGD. 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 $t_{\text{lat}}$ 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 $t_{\text{t}}$ is the time that one needs to transfer a single tuple. To reach an error of $\epsilon$, vanilla SGD requires, in physical time,

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

whereas `CorgiPile` requires

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

Because $(1-\alpha)\frac{h_D}{b} < 1$, `CorgiPile` always provides benefit over vanilla SGD in terms of the read/write latency $t_{\text{lat}}$. When $t_{\text{lat}}$ dominates the transfer time $t_{\text{t}}$, `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 $\alpha \le \frac{N-2}{N-1}$, choosing $\eta_s = \frac{1}{\sqrt{bn(1-\alpha)h_D\sigma^2 S}}$ and assuming $S \ge \frac{bn(\frac{104}{3}L+\frac{4}{3}L_H)^2}{\sigma^2(1-\alpha)h_D}$, 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*

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

*(2) When $\alpha = 1$, choosing $\eta_s = \frac{1}{(mS)^{\frac{1}{3}}}$ and assuming $S \ge (\frac{416}{3}L + \frac{16}{3}L_H)^3 b^2 n^3/N$, 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 IN-DATABASE IMPLEMENTATION

We integrate `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 such as MADlib [4, 34] and Bismarck [27]. Examples of the `params` include *learning_rate = 0.1*, *max_epoch_num = 20*, and *block_size = 10MB*. `CorgiPile` outputs various metrics after each epoch, such as *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 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, one for each of the three steps:

- `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 [29] that is also the query execution paradigm of PostgreSQL.

One challenge is the design and implementation of the *SGD* operator, which requires an *iterative* procedure that is not typically

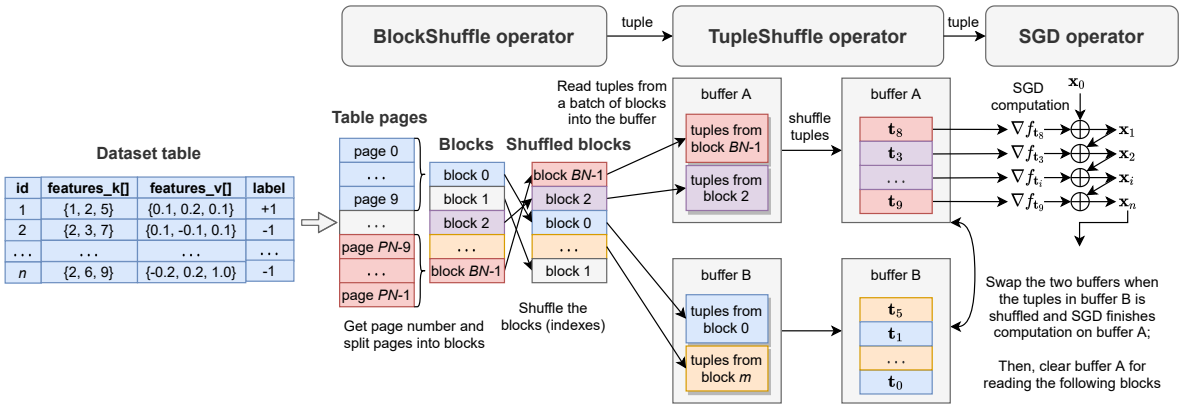



**Figure 5: The implementation of `CorgiPile` with new operators and the "double-buffering" optimization, in PostgreSQL.**

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.

Moreover, we store the training dataset as a table in PostgreSQL, using the schema of $\langle id, features\_k[], features\_v[], label \rangle$, which is similar to the one used by Bismarck [27]. 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 a `Name` in 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 Name`", which invokes the learned model for the prediction.

## 5.2 Physical Operators

The control flow of the three operators is shown in Figure 5, which leverages 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.

**BlockShuffle.** It first obtains the total number of pages using PostgreSQL's `RelationGetNumberOfBlocks()`. It then computes the number of blocks $BN$ by $BN = page\_num * page\_size / block\_size$. After that, it shuffles the block indices $[0, \ldots, BN-1]$ and obtains 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 somewhat similar to PostgreSQL's `Scan` operator, although the `Scan` operator reads pages sequentially instead of randomly.

**TupleShuffle.** It first allocates a buffer, and then pulls the tuples one by one from the `BlockShuffle` operator by invoking its `ExecTupleShuffle()`, i.e., `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 PostgreSQL. After that, the shuffled tuples are returned one by one to the `SGD` operator.

**SGD.** It first initializes an ML model in `ExecInitSGD()` and then executes SGD 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 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 EVALUATION

We evaluate `CorgiPile`, mainly focusing on in-DB ML systems. Our goal is to study the statistical and hardware efficiency of `CorgiPile` when applied to in-DB ML, i.e., whether it can achieve high accuracy and high performance in database systems. For this purpose, we compare our PostgreSQL-based implementation with two state-of-the-art systems, *Apache MADlib and Bismarck* with diverse models and datasets as follows. We first evaluate the linear models with standard SGD in PostgreSQL in Section 6.2. We further evaluate linear models with mini-batch SGD as well as other types of (continuous, multi-class, and feature-ordered) datasets in

**Table 2: Datasets. The first four are from LIBSVM [3]. For `criteo`, we extract 98M tuples from the `criteo` terabyte dataset. For `yfcc`, we extract 3.6M tuples from the `yfcc100m` dataset [63]; 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 in DB |
|---|---|---|---|---|
| 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 |

PostgreSQL in Section 6.3. In Section 6.4, we also briefly discuss `CorgiPile` for deep learning workloads. It shows an interesting but orthogonal line of future work to understand how to integrate `CorgiPile` as another file scanner in systems such as TensorFlow and PyTorch, which are not in-DB ML systems.

## 6.1 Experimental Setup

**Runtime.** We perform our experiments on a single *ecs.i2.xlarge* node in Alibaba Cloud. It 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. Moreover, `CorgiPile` only uses a single physical core, and we bind the two threads (see Section 5.3) to the same physical core using the "`taskset -c`" command. We run all experiments under CentOS 7.6, and we clear the OS cache before running each experiment.

**Datasets.** We use a variety of datasets in our evaluation, including dense/sparse and small/large ones as shown in Table 2. The datasets in Table 2 are stored in PostgreSQL for in-DB ML experiments. We focus on the evaluation over the *clustered* datasets, since SGD with various data shuffling strategies can achieve comparable convergence rates on the *shuffled* datasets, as shown in Figure 2.

**Models and Parameters.** For the evaluation on in-DB ML systems, we train two popular generalized linear models, logistic regression (LR) and support vector machine (SVM), that are also supported by Bismarck and 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.[3] Therefore, we leave it out of our end-to-end comparisons. Instead, we implemented *MRS Shuffle* in PyTorch and compare with it when we discuss the convergence rates of different data shuffling strategies (Figure 7). The hyperparameters include the learning rate, the decay factor, and the number of epochs (scanning the whole dataset per epoch). We use an exponential learning rate decay with 0.95. We set the number of epochs to 20 and use grid search to tune the best learning rate from {0.1, 0.01, 0.001}. We use the same initial parameters and hyperparameters among the compared MADlib, Bismarck, and `CorgiPile`.

**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`.

**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 high-dimensional datasets using the so-called TOAST [11] 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.

## 6.2 Evaluation on SGD with In-DB ML Systems

We first evaluate `CorgiPile` in terms of the end-to-end execution time. The compared systems include the *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 shuffling. We then analyse 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 report a sensitivity analysis on the impact of buffer sizes and block sizes in Section 6.2.4.

*6.2.1 End-to-end Execution Time.* Figure 6 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 running;[4] (2) the data caching time, i.e., the time spent on loading data from disk to the OS cache during the first epoch;[5] and (3) the execution time of all epochs.

From Figure 6, 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. Compared to *Shuffle Once* in MADlib and Bismarck, `CorgiPile` converges 2.9×-12.8× faster than MADlib and 2×-4.7× faster than Bismarck, 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 [37]. Moreover, for high-dimensional dense datasets, such as `epsilon` and `yfcc`, MADlib LR cannot finish even

[3] We have confirmed this with the author of Bismarck (private communication).

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

[5] 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.

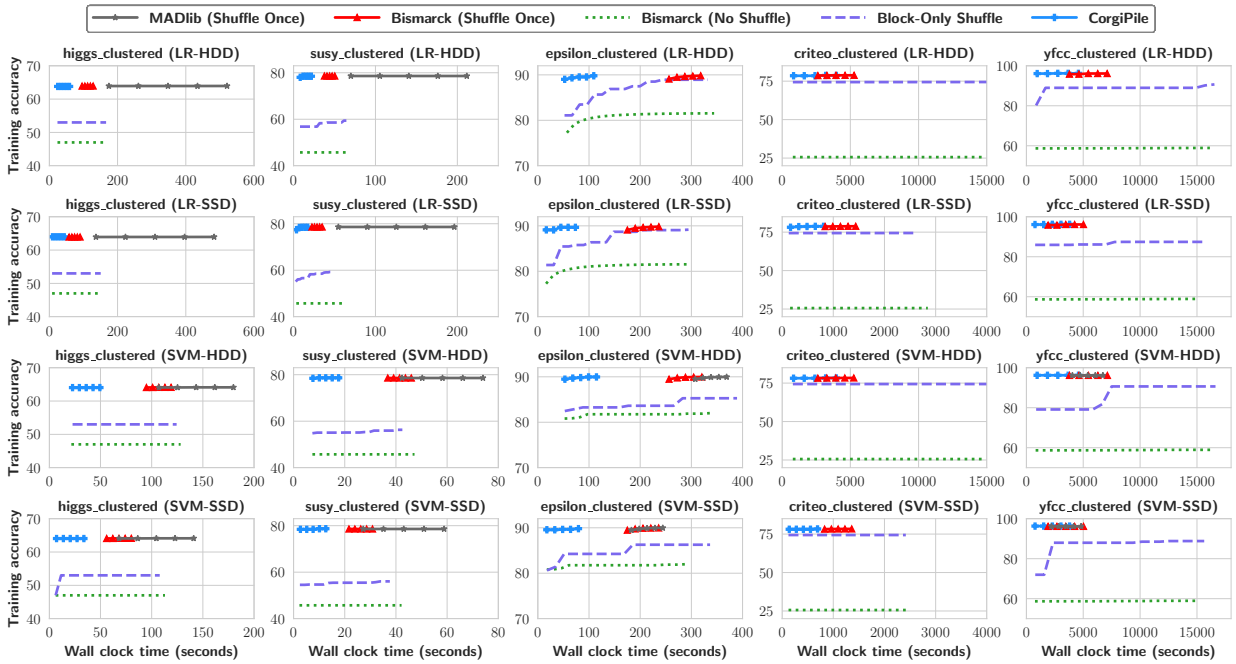



**Figure 6: The end-to-end execution time of SGD with different data shuffling strategies in PostgreSQL, for clustered datasets on HDD and SSD. We only show the first 5 epochs for *Shuffle Once* and `CorgiPile`, since they converge in 1-3 epochs.**

a single epoch within 4 hours, due to some expensive matrix computations on a metric named *stderr*.[6] 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`.

*6.2.2 Convergence rates.* For all datasets inspected, the gap between *Shuffle Once* and `CorgiPile` is below 1% for the final training/testing accuracy, as shown in Table 3. We attribute this to the fact that `CorgiPile` can yield good data randomness in each epoch of SGD (Section 4.2). *No Shuffle* results in the lowest accuracy when SGD converges, as illustrated in Figure 6. 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 *partial* random order, and the tuples in each block can all be negative or positive for 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 7 shows the convergence rates of all strategies, where *Sliding-Window*, *MRS*, and `CorgiPile` all use the same buffer size (10% of the dataset). As shown in Figure 7, *Sliding-Window Shuffle* suffers from lower accuracy, whereas *MRS Shuffle* only achieves comparable accuracy to *Shuffle Once* on `yfcc` but suffers on the other datasets.

*6.2.3 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 8. We make the following three observations.

[6]We have confirmed this behavior with the MADlib developers.

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

| | LR or VGG19 (SO \| Ours) | | SVM or ResNet18 (SO \| Ours) | |
|---|---|---|---|---|
| Dataset | Train acc. (%) | Test acc. (%) | Train acc. (%) | Test acc. (%) |
| higgs | 64.04 \| 64.07 | 64.04 \| 64.06 | 64.11 \| 64.22 | 63.93 \| 63.95 |
| susy | 78.61 \| 78.54 | 78.69 \| 78.66 | 78.61 \| 78.66 | 78.73 \| 78.66 |
| epsilon | 90.02 \| 90.01 | 89.77 \| 89.74 | 90.12 \| 90.11 | 89.81 \| 89.80 |
| criteo | 78.97 \| 78.91 | 78.77 \| 78.69 | 78.31 \| 78.41 | 78.45 \| 78.44 |
| yfcc | 96.43 \| 96.38 | 96.14 \| 96.11 | 96.35 \| 96.31 | 96.23 \| 96.20 |
| cifar-10 | 99.44 \| 99.10 | 92.29 \| 92.15 | 99.88 \| 99.86 | 93.82 \| 94.11 |

- 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 ~1.1× 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.

*6.2.4 Sensitivity Analysis.* We next study the effects of different buffer sizes, I/O bandwidths, and block sizes for `CorgiPile`.

**The effects of buffer size.** Figure 9 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

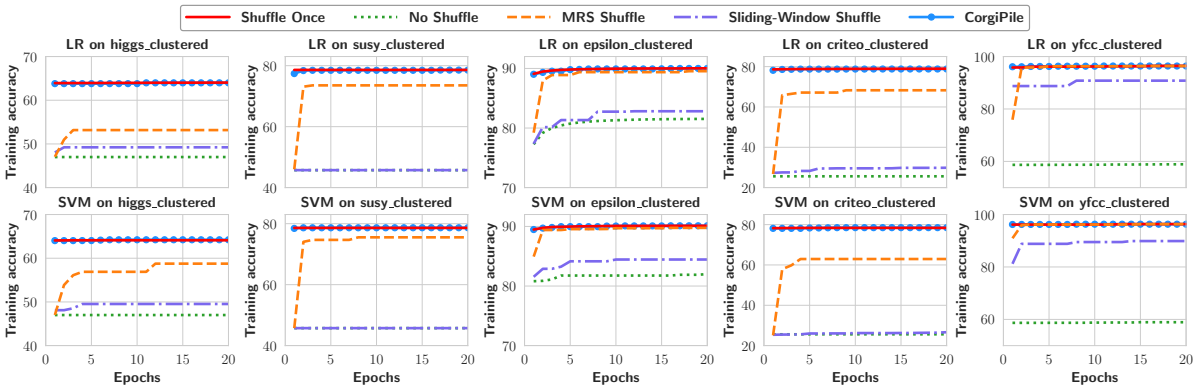



**Figure 7: The convergence rates of LR and SVM with different shuffling strategies for clustered datasets.**

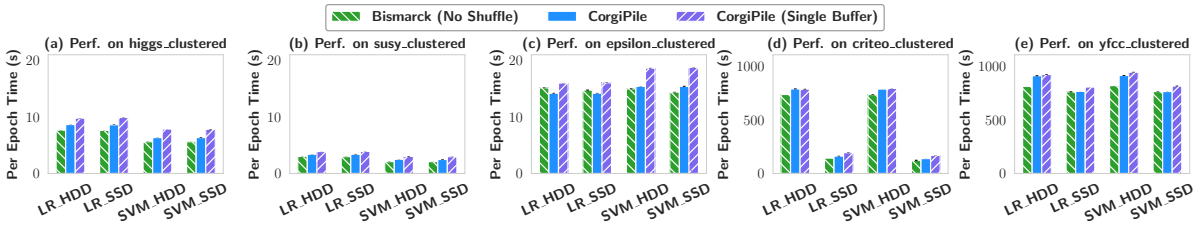



**Figure 8: The average per-epoch time of SGD with Bismarck (*No Shuffle*), `CorgiPile`, and `CorgiPile` with single buffer in PostgreSQL, for clustered datasets on HDD and SSD. It shows that `CorgiPile` is up to 11.7% slower than the fastest *No Shuffle*.**

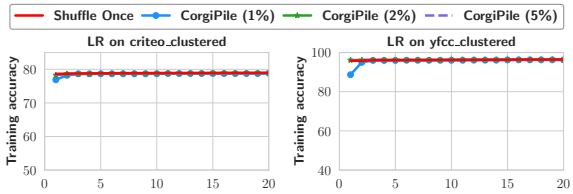



**Figure 9: `CorgiPile`'s convergence with varying buffer sizes.**

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%).

**The effects of I/O bandwidth.** As shown in Figure 8, for smaller datasets such as `higgs`, `susy`, and `epsilon`, `CorgiPile` on HDD and `CorgiPile` on SSD achieve the similar per-epoch times, since these datasets have been cached in memory after the first epoch. For larger datasets such as `criteo`, `CorgiPile` is faster on SSD than HDD, as expected. Interestingly, for `yfcc`, `CorgiPile` achieves similar performance on both HDD and SSD. The reason is that the TOAST compression on `yfcc` slows down data loading to only ~130 MB/s on both SSD and HDD. In contrast, for `criteo` without this compression, `CorgiPile` achieves ~700 MB/s on SSD and ~130 MB/s on HDD, as expected. These observations also hold for the *No Shuffle* on HDD/SSD for these datasets.

**The effects of block size.** We vary the block size in {2MB, 10MB, 50MB} on the large `criteo` and `yfcc` datasets. Figure 10 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). In practice, we recommend users

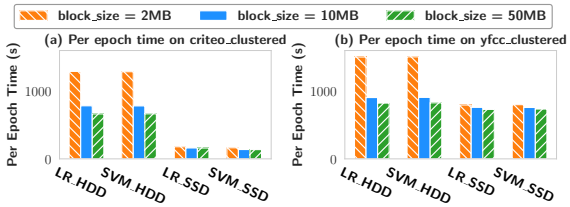



**Figure 10: Epoch time of `CorgiPile` with varying block sizes.**

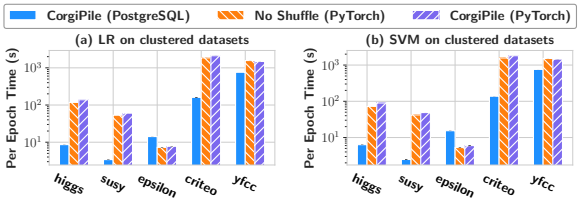



**Figure 11: The per-epoch time comparison between in-DB `CorgiPile` and PyTorch on SSD.**

to choose the smallest block size that can achieve high-enough I/O throughput, using I/O test commands such as "fio" in Linux.

*6.2.5 Performance comparison with PyTorch.* To further understand the performance gap between our in-DB `CorgiPile` and the start-of-the-art PyTorch outside DB, we compare them in two ways.

(1) `CorgiPile` in PostgreSQL vs. PyTorch: Figure 11 shows the per-epoch time comparison between `CorgiPile` in PostgreSQL and PyTorch with *No Shuffle*. For PyTorch, we load small datasets into memory before training to reduce the I/O overhead, and store the large `criteo` and `yfcc` datasets on disk. The comparison results in Figure 11 show that our in-DB `CorgiPile` is 2-16× faster than PyTorch on `higgs`, `susy`, `criteo`, and `yfcc` datasets. We speculate

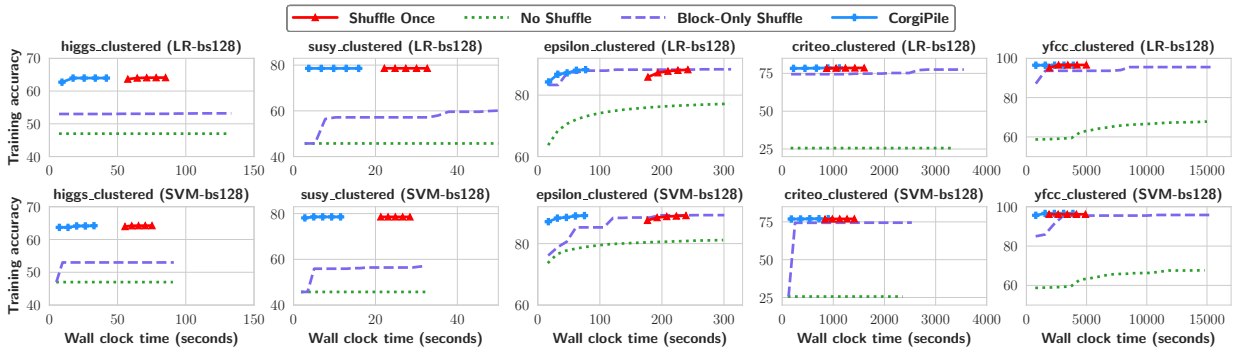



**Figure 12: The end-to-end execution time of LR and SVM using mini-batch SGD in PostgreSQL, for clustered datasets on SSD.**

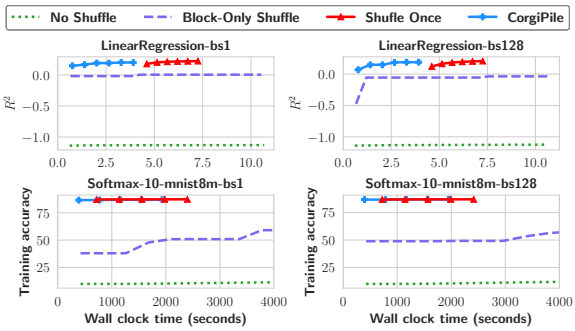



**Figure 13: The end-to-end execution time of linear and softmax regression in PostgreSQL, for clustered datasets on SSD.**

that this is because PyTorch has high overhead of Python-C++ invocations of forward/backward/update functions for each tuple, and these datasets have a large number (3-92 millions) of tuples. Only for the `epsilon` dataset, PyTorch is 2-3× faster than `CorgiPile`. The reason is that this dataset is compressed in DB by *TOAST* [11]. `CorgiPile` needs to decompress each tuple, while PyTorch directly computes on the in-memory uncompressed data.

(2) Outside DB: Figure 11 shows that PyTorch with `CorgiPile` introduces small (up to 16%) overhead compared to PyTorch with *No Shuffle*, which is consistent with what we observed inside DB.

## 6.3 Evaluation on Mini-Batch SGD and other types of datasets with In-DB ML Systems

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

*6.3.1 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.

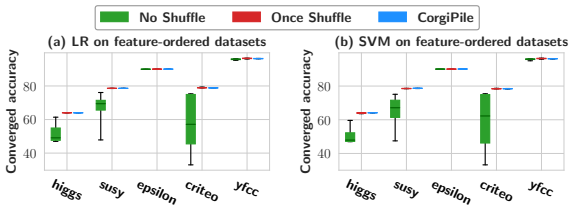



**Figure 14: The converged accuracy of LR and SVM on the datasets ordered by features instead of the label.**

*6.3.2 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 DB. For these cases, we further implement *linear regression* for training continuous dataset and *softmax regression* (i.e., multinomial logistic regression) for multi-class datasets, based on our in-DB operators in PostgreSQL. Figure 13 shows the end-to-end execution time of linear regression for continuous `YearPredictionMSD` dataset [3] and softmax regression for 10-class `mini8m` dataset [3], with different batch sizes on SSD. Our `CorgiPile` again achieves similar convergence rate and accuracy (i.e., coefficient of determination $R^2$ for linear regression) to the best *Shuffle Once*, but 1.6-2.1× faster.

*6.3.3 Beyond label-clustered datasets.* We conduct additional experiments using LR and SVM on all the binary-class datasets ordered by features instead of the labels. For low-dimensional `higgs` and `susy`, we sort each feature of them and report the statistics of the converged accuracy in Figure 14. For the other three high-dimensional datasets, we select 9 features such that 3/3/3 of them have the highest/lowest/median correlations with the labels.

Figure 14 shows that *No Shuffle* again leads to lower accuracy than *Shuffle Once*. Only for `yfcc` with image-extracted features and `epsilon` (with unknown features [1]), the accuracy gap is limited. In contrast, `CorgiPile` achieves similar converged accuracy to *Shuffle Once* on all the datasets. This implies simply scanning does not work on the datasets clustered by labels or by features.

## 6.4 Evaluation Beyond In-DB ML Systems

`CorgiPile` is not tied to in-DB ML; rather, it is a general data shuffling strategy for any SGD implementation. To understand its impact beyond in-DB ML systems and workloads, we implement the `CorgiPile` strategy as well as others in PyTorch and compare them on deep learning models, for both image classification and text classification. For image classification, we perform the classical

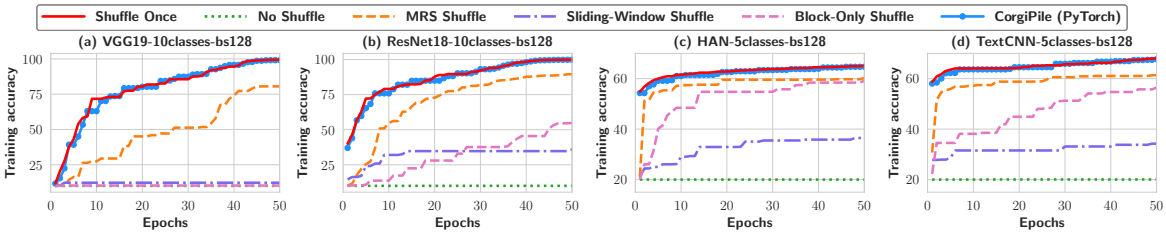



**Figure 15: The convergence rates of DLMs with different data shuffling strategies on the clustered image/text datasets.**

VGG19 and ResNet18 models on the *clustered* `cifar-10` dataset, which has 10 classes and is stored as binary data on disk. For text classification, we perform the classical HAN [65] and TextCNN [42] models with pre-trained word embeddings [54] on the clustered `yelp-review-full` dataset [70], which has 5 classes. Figure 15 illustrates the convergence rates with different data shuffling strategies and *batch_size* = 128. The buffer size is 10% of the whole dataset.

We observe that `CorgiPile` can achieve the similar accuracy and convergence rate to the best *Shuffle Once* baseline. This implies that `CorgiPile` can work well for non-convex optimization problems, too. In contrast, *Block-Only Shuffle* and *MRS Shuffle* suffer from lower accuracy, because they only generate partial random order of the data tuples. *Sliding-Window Shuffle*, which is used by TensorFlow, also suffers from lower accuracy.

## 7 RELATED WORK

*Stochastic gradient descent (SGD).* SGD is broadly used in machine learning to solve large-scale optimization problems [18]. It admits the convergence rate $O(1/T)$ for strongly convex objectives, and $O(1/\sqrt{T})$ for the general convex case [28, 51], where $T$ is the number of iterations. For non-convex optimization problems, an ergodic convergence rate $O(1/\sqrt{T})$ is proved by Ghadimi and Lan [28], and the convergence rate is $O(1/T)$ (e.g., [32]) under the Polyak-Łojasiewicz condition [55]. 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/standard SGD*.

*Data shuffling strategies for SGD.* In practice, random-shuffle SGD is a more practical and efficient way of implementing SGD [19]. 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 [17, 30, 32]. 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* [12] and *MRS Shuffle* [27]. Our empirical study shows that *Shuffle Once* achieves good convergence rate but suffers from low performance, whereas other strategies suffer from low accuracy when running on top of *clustered* data.

*In-DB ML.* Previous work [9, 22, 27, 34, 36, 39, 41, 43, 46–48, 53, 57, 59, 67, 69] has intensively discussed how to implement ML models on relational data, such as linear models [43, 53, 59], linear algebra [22, 46, 47], factorization models [57], neural networks [36, 47, 67] and other statistical learning models [41], 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 (UDAs). The representative in-DB ML tools are Apache MADlib [4, 34] and Bismarck [27], which use PostgreSQL's UDAs to implement SGD, and leverage SQL LOOP (Bismarck) or Python driver (MADlib) to implement iterations. Recently, DB4ML [37] 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 distributed database.* In recent years, there has been active research on integrating ML models into distributed database systems to enable scalable ML, such as MADlib on Greenplum [13], Vertica-ML [26], Google's BigQuery ML [6], Microsoft SQL Server ML Services [8], etc. Another trend is to leverage big data systems to build scalable ML models based on different architectures, e.g., MPI [23, 40], MapReduce [21, 49, 72], Parameter Server [25, 38, 64] and decentralization [44, 62]. Some of these systems can be integrated with distributed databases to support in-DB machine learning training, such as Spark MLlib [49] and SimSQL on Hadoop [35]. Recent work also started discussing how to integrate deep learning into databases [52, 71]. These efforts are orthogonal to `CorgiPile` and offer an exciting future direction to understand how to combine `CorgiPile` with these distributed learning efforts.

## 8 CONCLUSION

We have presented `CorgiPile`, a simple but novel data shuffling strategy for efficient SGD computation on top of block-addressable secondary storage systems such as HDD and SSD. `CorgiPile` adopts a two-level (i.e., block-level and tuple-level) hierarchical shuffle mechanism that avoids the computation and storage overhead of full data shuffling while retaining similar convergence rates of SGD as if a full data shuffle were performed. We provide a rigorous theoretical analysis on the convergence behavior of `CorgiPile` and further integrate it into PostgreSQL. Experimental evaluations demonstrate both statistical and hardware efficiency of `CorgiPile` when compared to state-of-the-art in-DB ML systems on top of PostgreSQL as well as deep learning systems.

## ACKNOWLEDGMENTS

CZ and the DS3Lab gratefully acknowledge the support from a SERI-funded ERC Starting Grant (TRIDENT 101042665), the Swiss National Science Foundation (Project Number 200021_184628, and 197485), Innosuisse/SNF BRIDGE Discovery (Project Number 40B2-0_187132), European Union Horizon 2020 Research and Innovation Programme (DAPHNE, 957407), Botnar Research Centre for Child Health, Swiss Data Science Center, Alibaba, Cisco, eBay, Google Focused Research Awards, Kuaishou Inc., Oracle Labs, Zurich Insurance, and the Department of Computer Science at ETH Zurich. LX is partially supported by the Youth Innovation Promotion Association, Chinese Academy of Sciences.

## REFERENCES


[1] 2008. Epsilon dataset. https://www.k4all.org/project/large-scale-learning-challenge/.
[2] 2009. CIFAR-10 dataset. http://www.cs.toronto.edu/~kriz/cifar.html.
[3] 2011. LIBSVM Data. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.
[4] 2022. Apache MADlib: Big Data Machine Learning in SQL. http://madlib.apache.org/.
[5] 2022. Buffer Manager of PostgreSQL. https://www.interdb.jp/pg/pgsql08.html.
[6] 2022. Google BigQuery ML. https://cloud.google.com/bigquery-ml/docs/introduction.
[7] 2022. I/O performance test on HDD and SSD with distinct block sizes. https://github.com/DS3Lab/CorgiPile-PostgreSQL/blob/corgipile/IO-perf.pdf.
[8] 2022. Microsoft SQL Server Machine Learning Services. https://docs.microsoft.com/en-us/sql/machine-learning/sql-server-machine-learning-services?view=sql-server-ver15.
[9] 2022. Oracle R Enterprise Versions of R Models. https://docs.oracle.com/cd/E11882_01/doc.112/e36761/orelm.htm.
[10] 2022. PostgreSQL. https://www.postgresql.org/.
[11] 2022. PostgreSQL TOAST. https://www.postgresql.org/docs/9.5/storage-toast.html.
[12] 2022. Sliding-Window Shuffle in TensorFlow. https://www.tensorflow.org/api_docs/python/tf/data/Dataset.
[13] 2022. The Greenplum MADlib extension. https://greenplum.docs.pivotal.io/6-19/analytics/madlib.html.
[14] 2022. The proof of our theorems in the theoretical analysis of CorgiPile. https://github.com/DS3Lab/CorgiPile-PostgreSQL/blob/corgipile/Proof.pdf.
[15] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*. USENIX Association, 265–283.
[16] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces* (1.00 ed.). Arpaci-Dusseau Books.
[17] Léon Bottou. 2009. Curiously fast convergence of some stochastic gradient descent algorithms. In *Proceedings of the symposium on learning and data science, Paris*, Vol. 8. 2624–2633.
[18] Léon Bottou. 2010. Large-Scale Machine Learning with Stochastic Gradient Descent. In *19th International Conference on Computational Statistics, COMPSTAT 2010*. 177–186.
[19] Léon Bottou. 2012. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*. Springer, 421–436.
[20] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. 2018. Optimization Methods for Large-Scale Machine Learning. *SIAM Rev.* 60, 2 (2018), 223–311.
[21] Zhuhua Cai, Zografoula Vagena, Luis Leopoldo Perez, Subramanian Arumugam, Peter J. Haas, and Christopher M. Jermaine. 2013. Simulation of database-valued markov chains using SimSQL. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013*. ACM, 637–648.
[22] Lingjiao Chen, Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. 2017. Towards Linear Algebra over Normalized Data. *Proc. VLDB Endow.* 10, 11 (2017), 1214–1225.
[23] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, et al. 2015. Xgboost: extreme gradient boosting. *R package version 0.4-2* 1, 4 (2015), 1–4.
[24] Christopher M De Sa. 2020. Random reshuffling is not always better. *Advances in Neural Information Processing Systems* 33 (2020).
[25] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. 2012. Large scale distributed deep networks. *Advances in neural information processing systems* 25 (2012), 1223–1231.
[26] Arash Fard, Anh Le, George Larionov, Waqas Dhillon, and Chuck Bear. 2020. Vertica-ML: Distributed Machine Learning in Vertica Database. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020*. ACM, 755–768.
[27] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a Unified Architecture for in-RDBMS Analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. 325–336.
[28] Saeed Ghadimi and Guanghui Lan. 2013. Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM Journal on Optimization* 23, 4 (2013), 2341–2368.
[29] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135.
[30] Mert Gürbüzbalaban, Asu Ozdaglar, and Pablo A Parrilo. 2019. Why random reshuffling beats stochastic gradient descent. *Mathematical Programming* (2019), 1–36.
[31] Mert Gürbüzbalaban, Asuman E. Ozdaglar, and Pablo A. Parrilo. 2021. Why random reshuffling beats stochastic gradient descent. *Math. Program.* 186, 1 (2021), 49–84.
[32] Jeff Z. HaoChen and Suvrit Sra. 2019. 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)*. PMLR, 2624–2633.
[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016*. 770–778.
[34] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (2012), 1700–1711.
[35] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. 2020. Declarative Recursive Computation on an RDBMS: or, Why You Should Use a Database For Distributed Machine Learning. *SIGMOD Rec.* 49, 1 (2020), 43–50.
[36] Dimitrije Jankov, Binhang Yuan, Shangyu Luo, and Chris Jermaine. 2021. Distributed Numerical and Machine Learning Computations via Two-Phase Execution of Aggregated Join Trees. *Proc. VLDB Endow.* 14, 7 (2021), 1228–1240.
[37] Matthias Jasny, Tobias Ziegler, Tim Kraska, Uwe Röhm, and Carsten Binnig. 2020. DB4ML - An In-Memory Database Kernel with Machine Learning Support. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020*. ACM, 159–173.
[38] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 463–478.
[39] Kaan Kara, Ken Eguro, Ce Zhang, and Gustavo Alonso. 2018. ColumnML: Column-store machine learning with on-the-fly data transformation. *Proceedings of the VLDB Endowment* 12, 4 (2018), 348–361.
[40] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017), 3146–3154.
[41] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. In-Database Learning with Sparse Tensors. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 325–340.
[42] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014*. 1746–1751.
[43] Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. 2015. Learning Generalized Linear Models Over Normalized Data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1969–1984.
[44] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. 2017. 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*. 5336–5346.
[45] Ji Liu and Ce Zhang. 2020. Distributed Learning Systems with First-Order Methods. *Found. Trends Databases* 9, 1 (2020), 1–100.
[46] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, Luis Leopoldo Perez, Dimitrije Jankov, and Christopher M. Jermaine. 2020. Scalable linear algebra on a relational database system. *Commun. ACM* 63, 8 (2020), 93–101.
[47] Shangyu Luo, Dimitrije Jankov, Binhang Yuan, and Chris Jermaine. 2021. Automatic Optimization of Matrix Implementations for Distributed Machine Learning and Linear Algebra. In *Proceedings of the 2021 International Conference on Management of Data*. 1222–1234.
[48] John MacGregor. 2013. *Predictive Analysis with SAP*. Bonn: Galileo Press.
[49] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* 17 (2016), 34:1–34:7.
[50] Konstantin Mishchenko, Ahmed Khaled Ragab Bayoumi, and Peter Richtárik. 2020. Random reshuffling: Simple analysis with vast improvements. *Advances in Neural Information Processing Systems* 33 (2020).
[51] Eric Moulines and Francis R Bach. 2011. Non-asymptotic analysis of stochastic approximation algorithms for machine learning. In *Advances in Neural Information Processing Systems*. 451–459.
[52] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A Data System for Optimized Deep Learning Model Selection. *Proc. VLDB Endow.* 13, 11 (2020), 2159–2173.
[53] Dan Olteanu and Maximilian Schleich. 2016. F: Regression Models over Factorized Views. *Proc. VLDB Endow.* 9, 13 (2016), 1573–1576.
[54] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014*. 1532–1543.
[55] Boris Teodorovich Polyak. 1963. Gradient methods for minimizing functionals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki* 3, 4 (1963), 643–653.

[56] Shashank Rajput, Anant Gupta, and Dimitris Papailiopoulos. 2020. Closing the convergence gap of SGD without replacement. In *International Conference on Machine Learning*. PMLR, 7964–7973.

[57] Steffen Rendle. 2013. Scaling Factorization Machines to Relational Data. *Proc. VLDB Endow.* 6, 5 (2013), 337–348.

[58] Itay Safran and Ohad Shamir. 2020. How good is SGD with random shuffling?. In *Conference on Learning Theory*. PMLR, 3250–3284.

[59] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016*. ACM, 3–18.

[60] Ohad Shamir. 2016. Without-replacement sampling for stochastic gradient methods. In *Advances in Neural Information Processing Systems*. 46–54.

[61] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015*.

[62] Hanlin Tang, Xiangru Lian, Ming Yan, Ce Zhang, and Ji Liu. 2018. $D^2$: Decentralized Training over Decentralized Data. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018 (Proceedings of Machine Learning Research, Vol. 80)*. PMLR, 4855–4863.

[63] Bart Thomee, David A. Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. 2016. YFCC100M: the new data in multimedia research. *Commun. ACM* 59, 2 (2016), 64–73.

[64] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE transactions on Big Data* 1, 2 (2015), 49–67.

[65] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alexander J. Smola, and Eduard H. Hovy. 2016. Hierarchical Attention Networks for Document Classification. In *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2016*. 1480–1489.

[66] Bicheng Ying, Kun Yuan, Stefan Vlaski, and Ali H Sayed. 2019. Stochastic Learning Under Random Reshuffling With Constant Step-Sizes. *IEEE Transactions on Signal Processing* 67, 2 (2019), 474–489.

[67] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2021. Tensor Relational Algebra for Distributed Machine Learning System Design. *Proc. VLDB Endow.* 14, 8 (2021), 1338–1350.

[68] Chulhee Yun, Suvrit Sra, and Ali Jadbabaie. 2021. 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)*. PMLR, 4653–4658.

[69] Ce Zhang and Christopher Ré. 2014. DimmWitted: A Study of Main-Memory Statistical Analytics. *Proceedings of the VLDB Endowment* 7, 12 (2014).

[70] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. 2015. Character-level Convolutional Networks for Text Classification. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015*. 649–657.

[71] Yuhao Zhang, Frank Mcquillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. 2021. Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches. *Proc. VLDB Endow.* 14, 10 (2021), 1769–1782.

[72] Zhipeng Zhang, Jiawei Jiang, Wentao Wu, Ce Zhang, Lele Yu, and Bin Cui. 2019. MLlib*: Fast Training of GLMs Using Spark MLlib. In *35th IEEE International Conference on Data Engineering, ICDE 2019*. IEEE, 1778–1789.