Optimizing Large-Scale Machine Learning over Groups

Side Li
University of California, San Diego
s7li@eng.ucsd.edu

Arun Kumar
University of California, San Diego
arunkk@eng.ucsd.edu

ABSTRACT

Many applications that use large-scale machine learning (ML) increasingly prefer different models for subgroups (e.g., countries) to improve accuracy, fairness, or other desiderata. We call this emerging popular practice *learning over groups*, analogizing to GROUP BY in SQL, albeit for ML training instead of SQL aggregates. From the systems standpoint, this practice compounds the already data-intensive workload of ML model selection (e.g., hyperparameter tuning). Often, thousands of models may need to be trained, necessitating high-throughput parallel execution. Alas, most ML systems today focus on training one model at a time or at best, parallelizing hyperparameter tuning. This status quo leads to resource wastage, low throughput, and high runtimes. In this work, we take the first step towards enabling and *optimizing* learning over groups from the data systems standpoint for three popular classes of ML: linear models, neural networks, and gradient-boosted decision trees. Analytically and empirically, we compare standard approaches to execute this workload today: task-parallelism and data-parallelism. We find neither is universally dominant. We put forth a novel hybrid approach we call *grouped learning* that avoids redundancy in communications and I/O using a novel form of parallel gradient descent we call Gradient Accumulation Parallelism (GAP). We prototype our ideas into a system we call *Kingpin* built on top of existing ML tools and the flexible massively-parallel runtime Ray. An extensive empirical evaluation on large ML benchmark datasets shows that Kingpin matches or is up to 14x faster than state-of-the-art ML systems, including Ray's native execution and PyTorch DDP.

1 INTRODUCTION

Machine Learning (ML) over large-scale data is now common. Increasingly, many ML applications seek to train separate models for separate subgroups based on various attributes, e.g., country or zipcode. This is a new form of GROUP BY aggregation, albeit for ML, not SQL aggregates. We call this process *learning over groups*. It helps applications for various reasons such as accuracy, fairness, and/or ease of ML deployment. For instance, some groups' data distributions may be simpler than the whole population, helping raise accuracy. Emerging non-technical business needs, such as privacy and regulatory compliance, may also necessitate learning over groups. For instance, online advertising platforms build disaggregated partner-specific models, with each groups' training data organized as a separate pipeline [20].

During ML model building, *model selection* is typically inevitable to control underfitting vs overfitting, e.g., via hyperparameter tuning [36, 61]. Practitioners often compare tens to hundreds of models [21, 50]. Learning over groups only amplifies this load many times, since model selection is needed for each group's model. For instance, if one compares 30 models on a group and has 50 groups, this whole process results in the training of 1500 models. At this scale, it is impractical to be building models one by one. *High-throughput* parallel ML systems are needed to train models *en masse*.

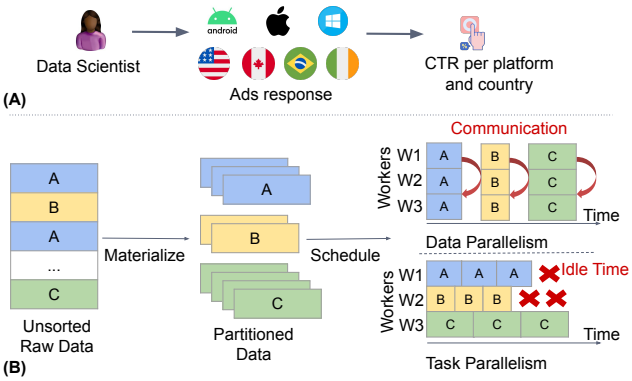

Figure 1: (A) Example for learning over groups. (B) Illustration of existing parallel approaches.

Example. Consider a data scientist at a Web advertising team modeling click-through rate. She tries a logistic regression model on the whole population. She then has a hunch that separate models per country and mobile platform can raise accuracy, as Figure 1(A) shows. She materializes each group's data subset on a distributed platform such as HDFS and then runs model selection for each group. After all model configurations of all groups finish, she picks the best model per group for further analyses.

As the example shows, learning over groups proceeds in two steps: (1) ETL to create and load groups' data subsets and (2) schedule ML model selection for all groups. Given the high volume of models to be trained, *parallelism on a cluster* is critical. There are two dominant existing approaches to execute this workload in parallel: *task parallelism (TP)* or *data parallelism (DP)*. We now briefly explain both of them, contrast their tradeoffs to explain why we need new approaches, and then present our novel approach. Figure 2(A) summarizes the key contrasts.

Existing Approaches and Drawbacks. ETL for TP needs each group's data subset in entirety. Then, randomly assign each worker the full dataset of a group. Repeat for each group. TP must balance data sizes assigned to workers to avoid imbalances. But in full generality, this becomes an NP-Hard multi-way partitioning problem [58]. Note that TP can raise the storage footprint substantially, since the dataset is fully copied across workers. In contrast, ETL foor DP is more straightforward: split each group's data subset evenly across all workers. DP does not raise storage footprint.

Scheduling model selection works as follows. TP spawns training for each group independently on a worker; workers do not talk to each other. But due to imbalanced task assignments and training times on workers, TP often results in idle times. This is especially

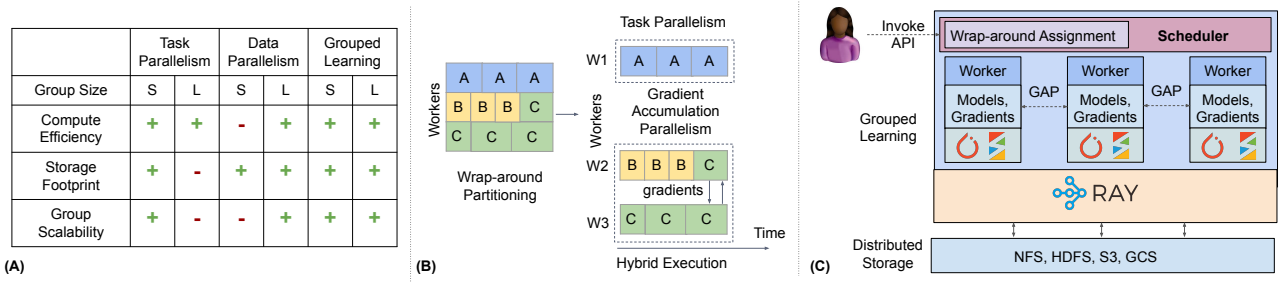

Figure 2: (A) Contrasting task-parallelism, data-parallelism, and our approach, grouped learning. S/L stand for small and large. Group scalability refers to handling large numbers of groups. (B) Illustration of grouped learning's hybrid execution with GAP. (C) Architecture of our system, Kingpin.

problematic when group sizes are disproportionately distributed. In contrast, DP trains each model of each group using the power of all workers simultaneously. But this can incur enormous communication cost for distributed training on a cluster, especially for ML workloads based on stochastic gradient descent (SGD) [41, 60]. DP is also an overkill for small groups. In general, real-world datasets typically have a mix of both small and large groups.

Recent work in the DB literature proposed a hybrid of TP and DP named Model Hopper Parallelism (MOP) for SGD workloads [37, 50]. Naturally, one may wonder if MOP can resolve the above TP-DP dichotomy. Alas, we find that MOP does not directly suit the setting of learning over groups for two reasons. First, MOP is akin to DP in sharding datasets across all workers. This is still an overkill for small groups. Second, MOP is tied to SGD's access patterns. But we seek to study learning over groups from a first-principles standpoint for other key ML access patterns too.

Desiderata. We seek to *optimize* learning over groups with the following desiderata. (1) *Generality:* Benefit multiple kinds of key ML access patterns. (2) *Scalability:* Scale along multiple axes, including number of groups, group sizes, cluster sizes, and model selection search space sizes. (3) *Efficiency:* Avoid the issues of TP, DP, and MOP, while retaining all their benefits. (4) *Non-disruptive integration:* Ideally, achieve all the above without needing to change the internal code of existing popular ML systems.

Our Approach. We perform an in-depth analysis of the access patters of learning over groups for 3 popular classes of ML: generalized linear models (GLMs), deep learning (DL), and gradient boosted decision trees (GBDT). We devise analytics cost models to account for multiple aspects–computation, network, and memory–to compare the efficiency of alternative approaches. Using our analytical models, we explored the tradeoff space thoroughly and gleaned insights for designing our new approach, which we explain next.

We call our approach *grouped learning*. It has three parts: (1) a simple and highly general epoch-level scheduling template, (2) a non-uniform data partitioning strategy for ETL, and (3) a novel form of parallel ML execution we call Gradient Accumulation Parallelism (GAP). The first part is inspired by Cerebro [50] and helps us meet the desiderata of generality and non-disruptive integration. It lets us unify all 3 of those classes of ML and support multiple forms of model selection. We support grid/random searches for now but it is possible to support AutoML heuristics too.

For the second and third parts, we formalize our optimization problem from first principles as an MILP. It is NP-Hard. So, we decompose ETL and scheduling. For ETL, we adopt and adapt an algorithm called *wrap-around* from the operations research literature [34]. It lets us cut communication costs substantially, while avoiding replication. Scheduling is tied to data placement. Most small groups are trained in a task-parallel manner. Large groups get sharded–typically non-uniformly–across workers. For such groups, we use GAP to reduce communication costs further. GAP is a new form of "bulk asynchronous" parallelism, a sister of MOP. Figure 2(B) illustrates our approach. Put together, wrap-around and GAP help us meet the desiderata of scalability and efficiency.

We prototype all of our ideas into a system we call *Kingpin* on top of the state-of-the-art distributed computation engine, Ray [48]. Figure 2(C) illustrates our system architecture. We evaluate Kingpin empirically on two large real-world ML benchmark datasets: Criteo [19] and Cityscapes [18]. Kingpin matches or outperforms existing approaches to learn over groups, with speedups up to 14x. Deeper analysis of resource utilization logs also validates that Kingpin's gains come from avoiding unnecessary network communications and disk I/O, as well as by reducing idle times on workers, thus affirming the benefits of our new techniques.
In summary, this paper makes the following contributions:

- To the best of our knowledge, this is the first work to enable and optimize the analogue of GROUP BY for ML at scale, a process we call learning over groups.
- We perform an in-depth analytical comparison of existing approaches to learn over groups and their tradeoffs for 3 main classes of ML: GLMs, DL, and GBDT.
- Based on our analyses, we devise a novel approach, grouped learning, that mitigates the issues of existing approaches, while still being easy to implement.
- As part of our approach, we adopt and adapt the wrap-around algorithm for shard placement and devise a novel form of bulk-asynchronous parallelism, GAP.
- We implement our ideas in a scalable ML system Kingpin, built on top of Ray and existing ML tools. A thorough empirical evaluation shows that Kingpin matches or surpasses strong existing baselines, with speedups up to 14x.

2 BACKGROUND AND PRELIMINARIES

2.1 Gradient-Based Optimization in ML

Gradient-based optimization is a highly popular mathematical substrate in ML. Many popular ML models are defined as *minimization* problems over model parameters and training data [29]. Given training data $D = \{(x_i, y_i)\}_{i=1}^{n}$, x_i, y_i denote features and targets; n is the number of examples. Many ML models compute $\text{argmin}_w L(w)$, where L is their loss function and w are model parameters, using optimization procedures such as gradient descent. We now explain them briefly; for more mathematical details, please see [24, 52].

The simplest optimization procedure is batch gradient descent (BGD). It does a full pass over sD to compute the gradient of L at an initial $w_{(0)}$, given by: $\nabla L(w) = \sum_{i=1}^{n} \nabla l(w^T x_i, y_i)$. The *update step* follows: $w_{(1)} \leftarrow w_{(1)} - \alpha \nabla L(w = w_{(0)})$; this is the "descent." BGD repeats this process multiple times, each called an *epoch*, until *convergence* to an optimal. Second-order batch methods are more popular than BGD for convex losses, like in GLMs [42]. They compute the second derivative of L, the Hessian, too. A popular second-order method is limited-memory BFGS (LBFGS). It approximates the Hessian with a fixed-size history of the last k gradients.

Stochastic Gradient Descent (SGD) is more popular than BGD for non-convex losses, like in DL, a popular form of ML, especially over unstructured data. One takes a random sample of D at a time, called a *mini-batch*, and estimates ∇L using that. Sampling is typically done *without replacement*, which can be done at scale using a random shuffle of D followed by a sequential pass. SGD typically converges much faster than BGD on large-scale data because it performs many updates to w within a single epoch. Optionally, the dataset is reshuffled between epochs [24].

Finally, GBDTs are a popular form of ML over structured data that also use gradients. They build an ensemble of weak models (typically short decision trees) to minimize L. A difference to BGD/SGD is that while GD descends along the gradient to update model parameters, gradient-boosting in GBDT performs descent against the gradient by adding new models. Thus, GBDT needs a full pass over D to run inference using all existing models.

2.2 Distributed Data Access Patterns in ML

Looking across the 3 common forms of optimization in ML, we can organize them into 3 main kinds of *data access patterns* based on amenability to distributed execution:

1) Algebraic: BGD and LBFGS compute algebraic aggregates, akin to SQL SUM [28]. Thus, an epoch is easily parallelized on a cluster with independent shards and workers; partial gradients are added by an aggregator in the end. This is logically equivalent to single-worker execution.

2) Sequential: An SGD epoch needs sequential access to the shuffled dataset. During a pass, each mini-batch gradient update depends on the previous, making SGD inherently hard to parallelize at the full-dataset scale. We call this pattern sequential-parallelizable. Techniques such as Parameter Server [41] and Horovod [60] are sometimes used for data-parallel SGD.

3) Sampling: In this pattern, gradients used for updates are downsampled from the full dataset. This is typical in distributed GBDT tools such as LightGBM [33]. The histogram-based GBDT performs parallel gradient computations as with LBFGS and constructs multiple histograms from local gradients [25]. Then it finds the best splits on the combined global histogram.

2.3 Assumptions and Notation

Unless otherwise mentioned, we assume that D fits in the cluster's total memory. We do allow data spills to disk; this is treated as an extension, explained later. We assume the ETL step materializes group data subsets and stores them in cheap (possibly ephemeral) networked storage. We fetch data during training by reading over the network; we cache data on workers if possible and necessary. For model selection, we focus on grid/random searches with a fixed number of hyperparameter configurations (configs). Grid/random searchers are the most popular forms of models selection in practice [16] but as we will explain later, Kingpin's design is amenable to easily plugging in AutoML procedures in future work.

Table 1 lists our notation. For simplicity sake, we assume all groups reuse the same s; this is also standard in ML practice. For simplicity of exposition in the next section, h denotes number of training epochs with one variable but Kingpin supports convergence-based criteria as well and does not need a fixed number of epochs up front. Likewise, the single m is for simplicity of exposition; different configs/groups can have different model sizes.

Symbol	Meaning
G	Set of groups; G_i is the i^{th} group
g	Number of groups ($\|G\|$)
n	Number of examples in D; n_i is number of examples in G_i.
p	Number of workers
s	Number of hyperparameter configurations.
h	Number of training epochs/iterations
b	Mini-batch size used in SGD
m	Model size

Table 1: Notation used in this paper.

2.4 Problem Statement

The problem of learning over groups is the following: orchestrate the given model selection workloads for all given groups on a given provisioned cluster. The goal is to minimize completion time, also called *makespan*. We want to satisfy all 4 desiderata listed in Section 1. This is challenging from a formal standpoint because we need to kill two birds with one stone: ETL for non-uniform partitioning and scheduling of model selection. It is challenging from the systems standpoint because we seek to build a unified system for 3 key access patterns in ML across GLMs, DL, and GBDT.

3 LEARNING OVER GROUPS

We first explain the existing approaches for the 3 classes of ML in terms of access patterns. We distill them into analytical cost models to offer a more in-depth understanding. Our goal is *not* to build some sort of cost-based optimizer but to study and explore the tradeoff space analytically to derive insights that help us devise our new approach. Section 4 will present our approach, which dominates these existing approaches due to formally grounded reasons.

3.1 Task Parallelism (TP)

In the ML world, the most common way to train models of groups in task parallelism (TP). A group's dataset is copied to each worker. A scheduler gets each worker to train some groups and/or configs. Each worker trains its assigned config(s) until convergence.

We identify two flavors: *group task parallelism (GTP)* and *model task parallelism (MTP)*, based on the granularity of tasks. The former defines tasks around groups; the latter, around configs. GTP places entire model selection of a given group on one worker to finish end to end. MTP breaks apart the model selection of a group to place one config on a worker at a time. MTP is more prevalent when compute resources are abundant.

Both flavors load a group's data over the network once. But MTP caches data on local disk so that it is not read again for a different config of the same group. Note that if there are more configs than workers ($s > p$), a worker may train more than one config per group even in MTP.

Compute Cost. Gradient computations would account for most of the compute time, while some cycles would also be used to update models. Therefore, we model the total compute cost as:

$$\begin{aligned} Cost_{Comp} &= \sum_{i=1}^{g} s \cdot h \cdot [f_{grad}(n_i) + f_{update} \cdot \eta] \\ &= s \cdot h \cdot [f_{grad}(n) + f_{update} \cdot g \cdot \eta] \end{aligned}$$

where $f_{grad}(1)$ is the time to compute the gradient for one example, f_{update} is the time to perform update to models, and η is the number of updates to models per iteration. $\eta = \frac{n}{b}$ for *sequential* workloads, and $\eta = 1$ for *algebraic* and *sampling* workloads.

I/O and Memory Cost. Iterative ML computations result in rounds of scans over the whole dataset. Training on one group requires a full scan of the dataset's corresponding subset. For GTP, we cache each group's data in memory for training all configs. For MTP, if $s > p$, we cache data on disk and load them $s - p$ times to train additional configurations. Thus, I/O and memory cost is:

$$\begin{aligned} Cost_{IO} &= f_{mem}(s \cdot h \cdot \sum_{i=1}^{g} n_i) + f_{disk}[\max(0, s-p) \cdot s \cdot \sum_{i=1}^{g} n_i] \\ &= f_{mem}(s \cdot h \cdot n) + f_{disk}[\max(0, s-p) \cdot s \cdot n] \end{aligned}$$

where f_{mem} is the cost of reading one example in memory, f_{disk} is the cost of loading one example from disk and f_{disk} gives non-zero only in MTP.

Network Cost. Training models in TP requires little to no network communication except fetching data from storage before training. So the network cost is a lump-sum cost of loading data to workers. For MTP, we copy the full dataset D to each worker.

$$\begin{aligned} Cost^{GTP}_{Network} &= f_{network}(n) \\ Cost^{MTP}_{Network} &= f_{network}(n \cdot p) \end{aligned}$$

where $f_{network}$ is the cost of fetching one example over network.

Idle Cost. Workers may go idle in GTP when too few groups and configs (or too many workers) are present, or groups are highly skewed. For MTP, skew is less of a concern if we have enough tasks to parallelize ($s \cdot g > p$). Random placement of tasks on workers will amortize such idle times. The max idle time MTP can have is when training the largest group for one config on one worker.

$$\begin{aligned} Cost^{GTP}_{Idle} =& f_{network}(n_{max}) + f_{mem}(s \cdot h \cdot n_{max}) \\ &+ s \cdot h \cdot [f_{grad}(n_{max}) + f_{update} \cdot \eta] \\ Cost^{MTP}_{Idle} =& f_{disk}(n_{max}) + f_{mem}(h \cdot n_{max}) \\ &+ h \cdot [f_{grad}(n_{max}) + f_{update} \cdot \eta] \end{aligned}$$

where n_{max} is the number of examples in the largest group G_{max}.

Total Runtime. We cannot merely add the three costs directly because the actual runtime depends on the underlying hardware. For example, in a memory-optimized cluster, the I/O cost will contribute less to the total runtime. Thus, we model total runtime with some cost parameters:

$$TotalTime = (\alpha Cost_{Comp} + \beta Cost_{IO} + \gamma Cost_{Network})/p + \delta Cost_{Idle}$$

We tune the cost parameters α, β, γ and δ using offline calibration runs, akin to RDBMSs. In particular, β may dominate when D does not fit in memory. For the rest of this section, we reuse this template of analytical cost models and cost parameters.

3.2 Data Parallellism (DP)

This approach shards the dataset of each group on each worker. Thus, it improves upon TP by exploiting all workers for each config, which can reduce worker idling for contiguous periods. But this also means DP needs communication among workers to aggregate partial gradients across workers. So, DP's cost model differs from TP primarily in the network and idle costs. Compute and I/O costs of DP are similar to GTP; we skip those for brevity.

3.2.1 ***Algebraic / LBFGS.*** Recall that algebraic workloads such as LBFGS can be easily partitioned across shards/workers by computing partial sums independently and then aggregating them.

Network Cost. Each worker computed local partial gradient (and Hessian approximation, baked into m) independently. It then sends its partial gradient to a peer per epoch. Also, there is the cost of loading training data. Together, the network cost is:

$$Cost_{Network} = f_{network}(n + s \cdot h \cdot g \cdot m)$$

Idle Cost. A worker can be idle while waiting for its peers to calculate their partial gradient. This discrepancy of runtime among workers also happens frequently in a cluster of homogenous nodes. Many factors, such as the state of hardware and numerical instability, contribute to it. With LBFGS, this process happens every epoch. f_{sync} is the blocking time spent on synchronization once.

$$Cost_{Idle} = f_{sync}(s \cdot h \cdot g)$$

3.2.2 ***Sequential / SGD.*** Unlike LBFGS, SGD with DP has the downside of massive amounts of communication (and often, synchronization) among workers for mini-batch gradient computations. This leads to overhead that compound the runtime distributed SGD in the DP regime.

State-of-the-art systems for distributed data-parallel SGD leverage techniques from the high-performance computing, especially the *all-reduce* scheme from Message Passing Interface (MPI) to synchronize mini-batch gradient computations. Horovod [60] is

Figure 3: Analytical cost model-based plots for key scalability axes. Cost parameters are calibrated from real empirical runs presented later (Section 6) on Algebraic/LBFGS on Criteo dataset with grouping attribute Country. The dataset size is 488GB; Country has 18 groups; ratio of largest group, n_{max}/n, is 0.26. We set $p = 4$ workers. For each plot here, we vary one variable, while fixing all other workload properties.

an exemplar, as is PyTorch DDP [43]. These systems saturate the network and make the best usage of all workers' compute capacity.

Network Cost. Network cost is where SGD and LBFGS differ fundamentally. Synchronizations among workers happen after *each* mini-batch instead of once per epoch. The network cost is as follows:

$$Cost_{Network} = f_{network}(n + s \cdot h \cdot m \cdot \frac{n}{b})$$

Idle Cost. SGD triggers many more rounds of network synchronizations and thus sees higher idle cost.

$$Cost_{Idle} = f_{sync}(s \cdot h \cdot \frac{n}{b})$$

3.2.3 ***Sampling / GBDT.*** For this workload, each worker first builds local statistics by sampling the local data partition. Then these local statistics are merged. This may not return the equivalent model obtained using a non-parallel approach because it essentially performs independent sampling across different workers. Yet, this approach still works in practice and is implemented in the popular GBDT tool LightGBM.

Network Cost. At each iteration, a worker computes sampled statistics and sends it to its peers. Together, the network cost is:

$$Cost_{Network} = f_{network}(n + s \cdot h \cdot g \cdot l)$$

where l is the size of sampled statistics.

Idle Cost. Idles time arises when aggregating local statistics for boosting.

$$Cost_{Idle} = f_{sync}(s \cdot h \cdot g)$$

3.3 Contrasting Task- and Data-Parallelism

Why are TP and DP not "good enough" for learning over groups? In short, each bakes in some tradeoffs on key scalability axes that make it substantially suboptimal in many realistic scenarios. Using our analytical cost models, we contrast them to expose such tradeoffs in Figure 3. Note that compute cost does *not* differ much across them; I/O costs do differ but can be optimized by faster storage and network. The key differentiating factors that make this tradeoff space non-trivial are inherent differences in communication cost/complexity and varying chances of worker idle times. The plots use cost parameters calibrated based on real empirical runs on Criteo dataset (Section 6); due to space constraints we provide their details in the appendix.

The overall takeaway is neither TP nor DP dominates the other on all axes. MTP is worse than both GTP and DP in this case mainly due to its repeated calls to reload data. GTP and DP are often comparable but have many crossovers. When the dataset size goes down or number of groups goes up, GTP dominates; but with more workers or more skew in group sizes, DP dominates. As a teaser for comparison, our approach in Kingpin (Section 4-5) is shown too–it matches or significantly dominates all these alternatives on all these scalability axes. We summarize key systems design issues with both TP and DP as it relates to learning over groups:

(1) TP Suffers from Imbalances. GTP faces idle times due to *imbalances* in groups across workers. Its optimal scheduling problem is known to be NP-Hard via a reduction from multi-way number partitioning [58]. But even with a near-optimal scheduling heuristic, group size skews will still cause idle times due to the fundamental *indivisibility* of a task in TP. As an extreme (but not uncommon) example, if a dataset has one very large group, the time to train that group on a worker will utterly bottleneck the cluster. Thus, TP as practiced today has inherent efficiency limits.

(2) TP Wastes Storage/Network. MTP mitigates GTP's issues with imbalances by placing different configs of a group on different workers. But this requires a *full copy* of that group's data to each worker. While storage is relatively cheaper, it is still a concern at scale, e.g., 1TB blows up to 10TB on a 10-node cluster! Of course, reading from remote storage (e.g., S3) can avoid such blowups. But due to the iterative nature of gradient-based ML this ends up being highly wasteful of the network instead! Recent work showed that this approach can have even 100x higher network costs [50]. Thus, TP as practiced today has inherent scalability limits as well.

(3) DP has Inherent Communication Costs. DP avoids the above issues of TP by splitting each config across all workers. Alas, this leads to new bottleneck: inevitable *communication rounds* and *synchronization* to aggregate gradients. Our analytical cost models show that this cost grows linearly with many key factors: number of configurations s, number of epochs h, number of batches (for SGD) and number of groups g. In fact, regardless of the class of ML, DP gets significantly slower as g increases. Thus, DP as practiced today also has inherent efficiency limits.

4 OUR APPROACH: GROUPED LEARNING

We now dive into our approach, grouped learning (GL), a novel hybrid of TP and DP. We start with some intuition and an overview,

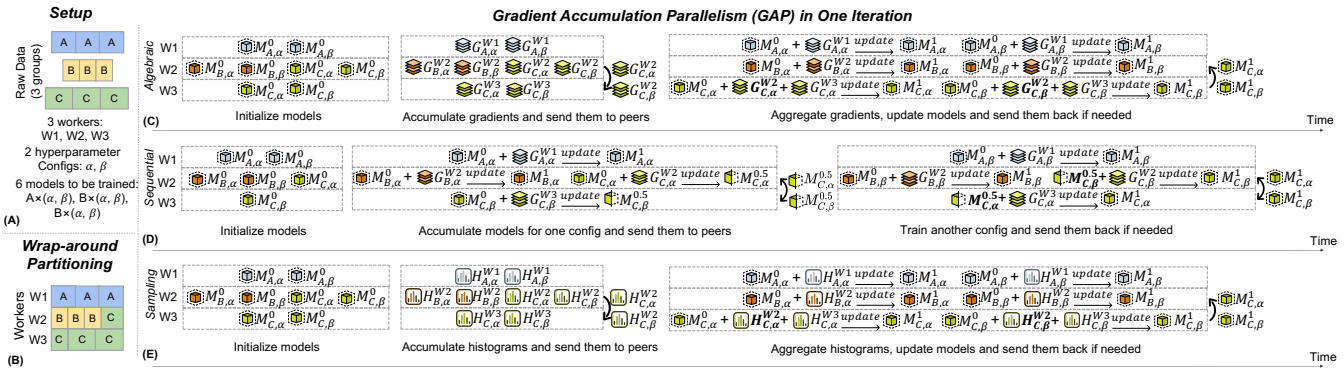

Figure 4: Illustration of our entire approach for an epoch/iteration. $M^k_{i,j}$ is the model of group i and config j after k-th iteration. $G^w_{i,j}$ (resp. $H^w_{i,j}$) is the gradient (resp. histograms) of group i and config j obtained at w worker. Items in bold are sent from other workers asynchronously. (A) Setup of workload inputs. (B) Wrap-around partitioning on the dataset. (C) GAP execution for Algebraic (LBFGS). (D) GAP execution for Sequential (SGD). (E) GAP execution for Sampling (GBDT).

then formalize our problem, present our algorithms, and finally present its analytical cost model and a key extension.

4.1 Intuition, Overview, and Technical Novelty

Our intuition is as follows: DP is too fine-grained, while TP is too coarse-grained. This leads to their fundamental issues laid out in Section 3.3. Philosophically, with GL we "take things apart" to go down from TP's level to avoid its issues (imbalances, resource bloat) and "put things back better" to go up from DP's level to avoid its issues (high communication costs). Concretely, GL reduces idle times from imbalances (vs TP), avoids data copying bloat (vs TP), and avoid needless communication/synchronization (vs DP). We achieve all this by assembling three things:

(1) A simple and general **two-level scheduling template**, inspired in part by Cerebro [50]. By decoupling per-epoch and across-epoch scheduling, we can support many kinds of iterative gradient-based ML (GLMs, DL, and GDBT) and many model selection heuristics in a unified way, including with *varying numbers of epochs per config*. The template itself is not novel but our application of it to the problem of learning over groups is novel.

(2) A **non-uniform data partitioning** strategy for ETL based on a suitable algorithm from the operations research world: *wrap-around*. It enables GL to *holistically optimize data and computation placement* across groups and configs at every epoch boundary. The wrap-around algorithm itself is not novel but our application of it to this large-scale ML systems setting is novel.

(3) A **novel form of parallel ML execution** per epoch: Gradient Accumulation Parallelism (GAP). To the best of our knowledge, GAP is only the second-known form of "bulk asynchronous" parallelism, inspired by its (complementary) sister MOP [50]. GAP works for all 3 major ML access patterns laid out in Section 2.2.

The precise workflow of GL is as follows, given the cluster, full dataset, groups, and model selection workloads. First, partition the dataset to favor TP using wrap-around. A group's data is not split across workers unless really needed. If a group's data is "too" large, it is sharded across workers. Second, place the epochs of the current set of configs on the workers and shards based on the scheduler's decision. Third, execute the training of configs in a hybrid-parallel manner, using TP for those that can and GAP for those that cannot. Next, we explain the mechanics of GAP first and then formalize our scheduling and partitioning problems.

4.2 Gradient Accumulation Parallelism (GAP)

GAP does the following: given a large set of configs and potentially sharded data per config, execute gradient-based ML using a hybrid of TP and DP. The interesting part is the operation of GAP on sharded data of a group. Basically, it *accumulates* gradients-related artifacts and ship them across workers asynchronously and in parallel. The degree of parallelism is determined by the number of configs and shards. We dive into how GAP works for each class of ML. Figure 4(C–E) illustrate these.

(1) Algebraic (LBFGS): Split gradients algebraically as usual; many workers can simultaneously train the same config to obtain local partial gradients. DP would aggregate partial gradients *eagerly* in one go, but GAP has this twist: aggregate partial gradients *lazily* and *asynchronously* over time as dictated by our scheduler.

(2) Sequential (SGD): Parallelize at the granularity of a group; two workers can *not* simultaneously train the same config. This ensures *logical equivalence to sequential SGD*, helping accuracy. When a worker is done with a group's config, it can move on to another config of that group, while the previous config is shipped to another worker. In this sense, GAP is partially inspired by MOP.

(3) Sampling (GBDT): We parallelize the sample statistics as usual. We repeat swapping of parallelizable artifacts among workers until GAP finishes one iteration of all configs of all groups. Our scheduler coordinates the scheduling and partitioning across workers.

Overall, GAP has the following invariants based on the ML access patterns, with the last 2 being potentially relaxed for Algebraic and Sampling workloads: (1) **Data sharding:** D is sharded across workers, not fully copied. (2) **Worker exclusivity:** A worker handles only one config at a time. (3) **Config isolation:** A config is handled by at most one worker at a time. (4) **Group isolation:** Configs of a group do not run concurrently across workers.

4.3 Formal Scheduling Problem

Our scheduler tackles the two-fold problem of ETL/partitioning and scheduling of configs. We now state it formally as a mixed-integer

Symbol	Description
C	Makespan (per epoch)
$N \in \lvert R\rvert^{g \times p}$	$N_{i,k}$ indicates the proportion of data of group G_i on k^{th} worker
$T \in \lvert R\rvert^{g \times s \times p}$	$T_{i,j,k}$ is the runtime of training group G_i and j^{th} configuration on k^{th} worker
$X \in \lvert R\rvert^{g \times s \times p}$	$X_{i,j,k}$ is the start time of the execution of group G_i and j^{th} configuration on k^{th} worker
$O \in \{0,1\}^{g \times g \times s \times p}$	$O_{i,i',j,k} = 1 \Longleftrightarrow X_{i,j,k} < X_{i',j,k}$, $N_{i,k} > 0$, $N_{i',k} > 0$
$P \in \{0,1\}^{g \times s \times p \times p}$	$P_{i,j,k,k'} = 1 \Longleftrightarrow X_{i,j,k} < X_{i,j,k'}$, $N_{i,k} > 0$, $N_{i,k'} > 0$
$Q \in \{0,1\}^{g \times s \times s \times p}$	$Q_{i,j,j',k} = 1 \Longleftrightarrow X_{i,j,k} < X_{i,j',k}$, $N_{i,k} > 0$
V	Very large value. Time to train one model on the whole dataset

Table 2: Additional notation used in MILP.

linear program (MILP). Table 2 summarizes the extra notation. The objective is to minimize makespan C to train all configs of all groups for one epoch. We do a pilot run to get the runtime of a config on a group's dataset to set V.

$$\text{Obj.} \quad \min_{C,N,X,O,P,Q} \quad C \tag{1}$$

$$\forall i, i' \in [1,..,g], \forall j, j' \in [1,...,s], \forall k, k' \in [1,...,p]$$

$$\text{s.t.} \quad C \geq X_{i,j} + T_{i,j} \tag{1a}$$

$$X_{i,j} \geq 0 \tag{1b}$$

$$\sum_{j=1}^{p} N_{i,j} = \frac{n_i}{n} \tag{1c}$$

$$X_{i,j,k} \geq X_{i',j,k} + T_{i',j,k} - V \cdot N_{i',k} \cdot O_{i,i',j,k} \tag{1d}$$

$$X_{i',j,k} \geq X_{i,j,k} + T_{i,j,k} - V \cdot N_{i,k} \cdot (1 - O_{i',i,j,k}) \tag{1e}$$

$$X_{i,j,k} \geq X_{i,j,k'} + T_{i,j,k'} - V \cdot N_{i,k'} \cdot P_{i,j,k,k'} \tag{1f}$$

$$X_{i,j,k'} \geq X_{i,j,k} + T_{i,j,k} - V \cdot N_{i,k} \cdot (1 - P_{i,j,k',k}) \tag{1g}$$

$$X_{i,j,k} \geq X_{i,j',k} + T_{i,j',k} - V \cdot N_{i,k} \cdot Q_{i,j,j',k} \tag{1h}$$

$$X_{i,j',k} \geq X_{i,j,k} + T_{i,j,k} - V \cdot N_{i,k} \cdot (1 - Q_{i,j',j,k}) \tag{1i}$$

The variables to optimize over are the start times X, group isolation assignments O, worker exclusivity assignments P, config isolation assignments Q, and data partitioning N. The constraints enforce the four invariants of GAP as listed above: (1c) ensures data sharding; (1d) and (1e) ensure group isolation; (1f) and (1g) ensure worker exclusivity; (1h) and (1i) ensure config isolation; (1a) and (1b) define makespan and sanity of start times.

One might now wonder if we can use an MILP solver such as Gurobi. But our problem turns out to be a fusion of two classic NP-Hard problems: multi-way number partitioning [58] and open-shop scheduling [27]. Since the total number of configs across groups can even be in the thousands, MILP solvers may be too slow.

Due to the above, we adopt the following two-step heuristic approach that is both efficient and offers near-ideal makespans in practice, certainly significantly faster than the prior art approaches (TP and DP). First, partition a set of numbers (in our case, data for groups) into a collection of subsets so that each collection's sums are as equal as possible. Second, find an optimal schedule to execute these collections/groups to minimize the makespan.

Algorithm 1 The Wrap-around Algorithm

```
Input: Group Information: G, g, n. Number of workers: p
Output: Partitioning schema
C* = max{n/p, max_{i∈[1,...,g]}(n_i)}
Initialize A ∈ |R|^{g×p} = [[0, ..., 0]]
cur_filled = 0, cur_worker = 1
for group i in G do
    if n_i + cur_filled <= C* then
        A_{i,cur_worker} = n_i,
        cur_filled += n_i
    else
        while n_i > 0 do
            if C* − cur_filled <= n_i then
                A_{i,cur_worker} = C* - cur_filled
                n_i -= C* - cur_filled
                cur_worker += 1, cur_filled = 0
            else
                A_{i,cur_worker} = n_i
                cur_filled += n_i, n_i = 0
Return A
```

4.4 The Wrap-around Algorithm for ETL

If we relax N in the MILP to real space, the ETL part becomes more manageable. In the operations research literature such a relaxation is called "pliable shop" scheduling [34]. That prior work proposed a linear-time algorithm that is a good fit for our (relaxed) scenario: *wrap-around* algorithm. The idea is to compute the expected capacity/makespan and keep adding data for groups onto a worker. If adding a group would exceed the current worker's optimal capacity, we shard the data and continue adding the overflow part to next worker. The order groups are put onto workers itself entails a schedule to manage executions. Based on all of our notation so far, the optimal makespan will now be:

$$C^* = max\{\frac{n}{p}, \max_{i \in [1,...,g]}(n_i)\} \cdot V \cdot s/n \tag{2}$$

Prior work [34] also showed that the wrap-around algorithm is *provably optimal*; we refer the interested reader to [46] for their proof. Note that $\max_{i \in [1,...,g]}(n_i)$ comes from the rationale that we cannot execute a job in parallel but in a strictly sequential manner. Overall, this algorithm is helpful to ensure sequential execution order on sharded data. If we end up needing to shard in between an example (non-integral splits), we round it by rolling over the extra example to the next worker. Just one example off from the optimal has virtually no impact on the makespan. Thus, we modify it slightly and present the pseudocode above. Figure 4(B) illustrates the algorithm.

4.5 Putting It All Together

Combining wrap-around and GAP, we first evaluate workers' expected capacities for at each epoch based on simple statistics on the

data and hyperparameter search space. Workers read their assigned data over the network. GAP is used for training all configs for that epoch. We accumulate training artifacts and ship them around workers asynchronously. When a worker finishes training a config on its (shard of a) group, it propagates gradient-related results to another worker with the same group data. The last worker will finally add up all gradients, run the update step for the optimization procedure, and then broadcast updated models back to relevant workers. Thus, all workers are kept busy almost always during training instead of waiting for instructions from a centralized manager.

See Figure 4 again for the end-to-end illustration of how wrap-around and GAP for all 3 classes of ML workloads. Note that we synchronization over the network is only needed when a group's data is not entirely on one worker. All in all, grouped learning scales seamlessly on all axes explained in Section 3: dataset size, cluster size, number of groups, and group sizes.

4.6 Analytical Cost Model

We now present our approach's analytical cost models that we used for Figure 3 in Section 3. The compute and I/O costs are the same as that of DP; we skip them for brevity.

Network Cost. This is a key advantage of GAP: it does not incur much network cost, especially for SGD, even if a group's dataset is sharded across workers. The actual cost per epoch is as follows. Note it is is linear in the number of shards of a group, which in the worst case is only the number of workers p (say, for a super large group). This is in contrast to DP for SGD (e.g., Horovod or PyTorch DDP), which are orders of magnitude higher.

$$Cost_{Network} = f_{network}(n + s \cdot h \cdot m \cdot p)$$

Idle Cost. A worker may still be idle when waiting for its peers to finish their parts. But this synchronization happens only once per epoch, no matter how many groups or configurations we have.

$$Cost_{Idle} = f_{sync}(h)$$

4.7 General Extension

So far, our approach can be easily layered onto existing ML popular frameworks for LBFGS and SGD-based ML (e.g., PyTorch) *without modifying their internal code*. But for GBDT, an implementation nuance is that this is not possible with just the pure wrap-around algorithm. We will explain the implementation part of this nuance later in Section 5.1. But for now we present an algorithmic extension that will let us support GBDT systems (e.g., LightGBM) seamlessly as well, preserving the high generality of our approach.

Basically, we decouple gradient computation and its subsequent use for the ML/optimization procedures. We then modify the MILP to account for this. There are only so many ways to partition a group's dataset: 1, 2, or upto p shards. Formally, this is akin to setting a constraint on $N_{i,j}$ as follows:

$$\forall c \in [1, ..., p], N_{i,j} = 0 \text{ or } N_{i,j} = \frac{n_i}{n \cdot c}$$

It is also equivalent to having the same upper and lower bound for each type of split. Even if we relax it to linear space, this bound

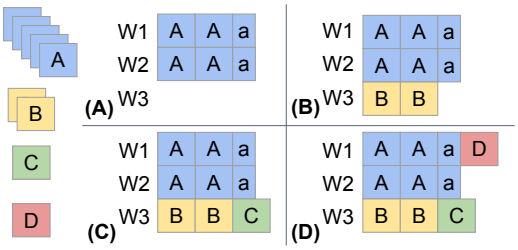

Figure 5: Illustration of the constrained wrap-around algorithm. Optimal makespan is 3. (A) Split A into two partitions, because 5/3 is rounded to 2. (B) No split on B, and put it on the worker with least data assigned. (C) and (D) No split.

setting makes the problem a variant of the pliable job scheduling problem, which is NP-Hard [34]. Thus, we propose a new heuristic.

Algorithm 2 The Constrained Wrap-around Algorithm

Input: Group Information: G, g, n. Number of workers: p
Output: Partitioning schema
$C^* = \frac{n}{p}$
Sort G by its size in decreasing order
Initialize $A \in |R|^{g \times p} = [[0, ..., 0]]$
// A priority queue for workers, sorted by assigned data
Q = [(0,1),...,(0,p)] // (filled, worker index)
for group i in G **do**
    best_num_splits = round $\frac{n_i}{C^*}$ to nearest integer
    for each split **do**
        cur_filled, $worker$ = Q.pop()
        A_{i,cur_worker} += n_i / best_num_splits
        Q.push((cur_filled + n_i / best_num_splits, $worker$))
Return A

Constrained Wrap-around Algorithm. Intuitively, we want to split a group into as few shards as possible because the more shards there are, the higher the network cost. Ideally we do not shard a group at all, resembling TP, albeit with no copying. As a heuristic, we always split a group to its nearest ratio with the optimal makespan: $round(n_i \cdot p/n)$. We call this heuristic the constrained wrap-around algorithm. Algorithm 2 presents the pseudocode. Figure 5 presents an illustration. The key difference now is that we first sort the groups in G in decreasing order of size to decide whether to split large groups first. Smaller groups usually have that ratio < 1; so, we need not split them at all. Empirically, we find that this heuristic achieves near-optimal makespans for realistic workloads; due to space constraints, we present more details of this comparison in the appendix.

5 SYSTEM IMPLEMENTATION SPECIFICS

Kingpin is a new ML system that implements our ideas of grouped learning, including GAP and the wrap-around algorithm. As a strategic decision, we chose to prototype Kingpin on top of Ray, a recent highly scalable runtime engine for compute-intensive ML/AI workloads [48]. We weighed a prototype on Spark too but found Ray better for 2 reasons. First, Ray has new systems capabilities that

avoid extra I/Os for intermediates relative to Spark, both for baselines (TP and DP) and for Kingpin. Second, Spark's JVM-based data caching leads to extra overheads for copying and shipping data to ML tools (we use Pytorch and LightGBM).

5.1 Overview

We adopt an extensible architecture that can talk to multiple ML tools, as Figure 2(B) showed. Ray helps parallelize workloads on physical machines. We still use Spark, albeit only for materializing group data subsets if the full dataset comes in raw.

A user specifies the following in Kingpin APIs: group metadata (with syntactic sugar), model_creator, optimizer_creator, hyperparameter search space, and cluster configuration (number of CPUs, GPUs, etc.). Our scheduler then runs the wrap-around algorithm and spawns workers to load corresponding data. All workers are started as native Ray *actors* connected by PyTorch's distributed primitives [43]. Intermediate results (e.g., gradient artifacts) are saved in Ray's *in-memory object store*, with the option to spill to disk for fault tolerance. In the end, users will get back serialized copies of all groups' models, including selected best models.

For ML training, Kingpin essentially acts as an abstraction enveloping popular ML tools without modifying their internal code, e.g., for tensor compilation or hardware optimizations. Currently, we support the APIs for PyTorch (for LGBFS and SGD) and LightGBM (for GBDT). It is relatively straightforward to add support for their rival tools if needed (e.g., TensorFlow or XGBoost).

As a detail about the LightGBM integration, it ships histograms during distributed training. But these histogram data structures are baked deep into its C++ code and are hard to decouple. LightGBM provides developers an extensibility option to compute gradients using a custom loss function. But to use this, we need to rewrite existing loss functions written in C++ (e.g., cross-entropy) in Python, which may become a backward compatibility issue for our software. We decided to avoid this deeper dependency with a different approach: *extend* the wrap-around algorithm itself in our scheduler–as explained in Section 4.7–so that we can restrict ourselves to using only the user-facing APIs of LightGBM.

5.2 Data API

5.2.1 ***ETL and Group Metadata.*** In general, the *how* of the ETL step is complementary. If an existing pipeline outputs partitioned files with corresponding metadata to distributed storage, Kingpin can take it from there. Our ETL functions are a lightweight abstraction on top of Spark. The user inputs the unsorted raw data file; the output is fine-grained partition files of each group and their metadata. Our API let users specify raw data path, type of data (CSV, images, or binary), and max file size (default: 25 MB). The max file size balances overheads vs parallelism. The wrap-around algorithm needs the ability to split a group's data at a fine granularity. But for ease of implementation, we use a coarser granularity of small files; we still get near-optimal makespans. We also provide a separate API to generate metadata (file path, number of examples in each file partition, and group) at scale using all I/O and network resources available.

5.2.2 ***Data Loading.*** Once the wrap-around algorithm assigns data partitions, it is up to the workers to load data for ML. But during ETL, we create 100s or even 1000s of small (shard) files. To the best of our knowledge, most ML systems still lack good support for reading multiple text files in parallel. The most common practice of loading multiple files is to load image data, where each image file is just one example. But we must also deal with 1000s of small text files, each with 1000s of training examples. To meet this need, we created an *efficient parallel data loader* to read 1000s of small text files. Our loader supports two modes: *eager* and *lazy*. Eager means we read all files listed in the metadata beforehand; lazy reads data on the fly as requested. Users can also specify the number of CPU cores to use and whether to cache shards in memory or on disk.

5.3 Moving Gradients and Models Around

Running GAP means shipping gradient artifacts and models among workers. These artifacts can add up to 100s of GBs of memory on a worker based on the ML algorithms and groups given. Thus, we need an efficient way to manage them in memory. We use the distributed in-memory object store in Ray, which allows simple put-get operations and pass-by-references in a cluster. The workflow is as follows: put gradients/models in the local object store and get reference IDs back. When we ship gradients/models, send reference IDs. When a worker needs to update models, it gets objects by asking the local object-store. If the local object store does not have it, Ray will consult other workers to find it and ship it for us. Overall, we ship gradients/and models lazily and asynchronously. Based on the partitioning created by the wrap-around algorithm, we have multiple workers initialize different groups; this also amortizes memory footprints.

6 EXPERIMENTS

We now present an extensive empirical evaluation seeking to answer three questions. (1) How does Kingpin compare with the existing approaches of MTP, GTP, and DP on runtimes, accuracy, and resource utilization? (2) Did our analytical cost-model based discussion correctly predict the trends? (3) Does Kingpin offer good scalability on key axes?

Datasets. We use two large ML benchmark datasets with different groups: Criteo Sponsored Search Data [19] and Cityscapes [18]. Criteo is a product ad click binary classification task with numeric and categorical features. We pick Country and Partner as meaningful groups for 2 datasets. As standard ML practice, we use random hashing to convert categorical features into feature vectors. Cityscapes is a popular image dataset for semantic understanding of urban street scenes in German and Swiss cities. Table 3 lists the dataset and group statistics. Figure 6 provides the group size distributions. We hold out a random 10% of data in each group as its validation set.

Dataset	On-disk size	Format	Count	Group attribute	# Groups
Criteo	488GB	CSV	14M	Country	18
				Partner	280
Cityscapes	54GB	PNG	21146	City	44

Table 3: Dataset details after preprocessing.

Workloads. We run end-to-end test for all three ML access patterns. (1) Algebraic: Logistic regression with LBFGS (history size 10). (2) Sequential: A UNet [55] neural network with 8 filters for semantic

Workload/Model	Dataset	Group-By attribute	Optimizer	Hyper-parameter search space
Algebraic - LR	Criteo	Country Partner	LBFGS	Learning rate: [1,0.5,0.1], L1 regularization: [1,0.5,0.1,0.05]
Sequential - UNet	Cityscapes	City	Adam	Learning rate: [1e-2,1e-3,1e-4,1e-5], L2 regularization: [0,1e-3,1e-2]
Sampling - GBDT	Criteo	Country Partner	N/A	Learning rate: [1,0.5,0.1,0.05], Num of leaves: [10,20,30]

Table 4: Model selection workloads of all 3 ML access patterns. All configs are run for 10 epochs/iterations.

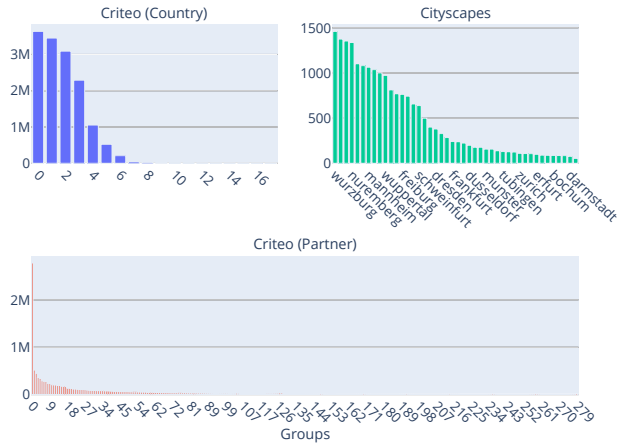

Figure 6: Data distributions of groups in Criteo and Cityscapes datasets. Note that group names in Criteo are just numbers for anonymous reasons.

segmentation, Adam as the SGD method, and $b = 32$. (3) Sampling: GBDT. For all 3, we run standard grid search for hyperparameter tuning. Table 4 lists the grids.

Experimental Setup. We use two clusters: CPU-only for Algebraic and Sampling; GPU-enabled for Sequential. Both clusters have 4 nodes on CloudLab [22]. Each CPU-only node has 2 Intel E5-2660 v2 10-core CPUs, 256GB RAM, 2 1TB HDDs, and 10 Gbps network. Each GPU node has 2 Intel Xeon Silver 4114 10-core CPUs, 192GB RAM, 2 1TB HDDs, 1 480GB SSD and an Nvidia P100 GPU. All nodes run Ubuntu 16.04. We use Ray v1.1.0, PyTorch v1.7, and LightGBM v3.1.1. All training and validation datasets are stored on HDFS hosted in the same cluster.

Baselines. GTP, MTP, and DP implemented on top of Ray's native ML infrastructure are our baselines. Ray offers native and robust TP scheduling, while RaySGD runs DP using PyTorch's DDP [43]. For GBDT, DP runs LightGBM in data-parallel mode on sockets [6]. We make all baselines as strong as can be with our best-effort engineering. All baselines use the same data loader we implement for Kingpin. Also, we scale DP to run more processes when a workload alone cannot saturate compute resources on one worker; the same applies to GTP. We spawn multiple configs to train in parallel on one worker using shared memory store in Ray [1]. For MTP, we do not perform this optimization because it will lead to training groups simultaneously on one worker. Loading all their corresponding data to memory at once is infeasible.

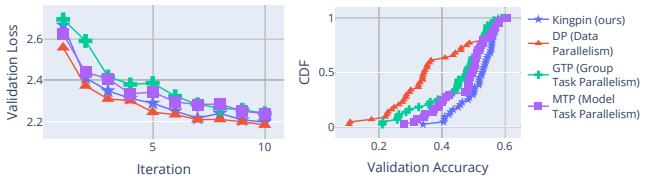

Figure 7: Learning curves of Wurzburg and CDFs of validation accuracies of all groups in the Cityscapes dataset.

6.1 End-to-End Results

We load all data shards to workers' local memory first before training for all experiments. Different approaches will trigger this process multiple times–thus, we must include data loading time from remote storage in the end-to-end runtimes. For GTP and MTP, we use a FIFO queue to schedule out groups and configs. For DP, we train one group at a time on all worker nodes. For GBDT, Kingpin uses the general extension in Section 4.7. We split the discussion into runtimes and learning curves.

Total Runtimes. Table 5 shows the results. We see Kingpin is either significantly faster than the baselines in most cases cases, while matching the best baselines in some cases. On Algebraic over Criteo-Country, Kingpin is 2.7x faster than GTP and 12.6x faster than MTP. This is expected because logistic regression training is dominated by I/O and data movement. MTP wastefully reloads data repeatedly. DP is about 1.8x slower than Kingpin and 1.4x faster than GTP. Overall, Kingpin dominates all the existing approaches.

On Algebraic over Criteo-Partner (a much larger group), Kingpin's runtime is comparable to what it saw for Country, thus showing its ability to scale well with large numbers of groups. In contrast, DP becomes dramatically slower due to its inherently high communication costs. GTP is also comparable to what it saw for Country; MTP, slightly slower than before.

On Sampling (GBDT) over Criteo for both grouping attributes, Kingpin is comparable to GTP. DP is marginally slower on Country but sees 4x slowdown on Partner due its larger number of groups. MTP's is still very slow, due to its repeated data loading overheads.

Finally, on Sequential (UNet) over Cityscapes, Kingpin is comparable to GTP; their GPU utilizations are also comparable. MTP is only 1.4x slower than Kingpin; this gap is lower now because deep learning is compute-bound and MTP's data loading disadvantage is less significant here. DP is 4.3x slower than Kingpin due to its inherent communication costs even though PyTorch DDP is a state-of-the-art system for data-parallel SGD.

Learning curves. For Algebraic and Sampling, all approaches (GTP, MTP, DP, and Kingpin) have virtually indistinguishable convergence behaviors. So, we elide them for brevity. For Sequential

Workload	Algebraic - Logistic Regression				Sampling - GBDT				Sequential - UNet		
Dataset	Criteo (country)		Criteo (partner)		Criteo (country)		Criteo (partner)		Cityscapes		
Systems	E2E runtime (min)	Avg CPU (%)	E2E runtime (min)	Avg CPU (%)	E2E runtime (min)	Avg CPU (%)	E2E runtime (min)	Avg CPU (%)	E2E runtime (min)	Avg CPU (%)	Avg GPU (%)
Kingpin (ours)	86	87.6	95	78.6	42*	85.0*	55*	66.5*	563	28.1	83.5
Group Task Parallelism	226	81.3	232	79.1	42	87.7	52	66.1	570	13.2	86.0
Model Task Parallelism	1082	35.6	1312	44.7	578	63.2	703	56.0	780	13.9	69.0
Data Parallelism	158	70.0	711	23.5	50	73.1	208	20.8	2414	13.8	95.3

Table 5: End-to-end results on Criteo and Cityscapes for all workloads. All of the experiments are conducted on the Ray infrastructure. GTP and MTP run the native task-parallel executions in Ray. DP runs PyTorch DDP under the hood for Logistic Regression and UNet. For GBDT, DP runs native LightGBM in data-parallel mode and *Kingpin* runs the general extension.

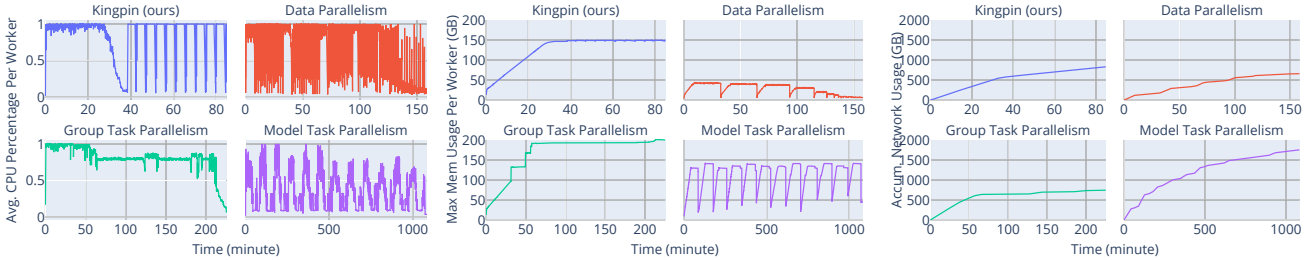

Figure 8: CPU, Memory and Network Usage of Algebraic on Criteo (Country)

(SGD), different approaches go through a group's dataset in different orders, likely leading to different learning curves. For all of them, we shuffled each group's dataset once up front, which often suffices for SGD [24]. Figure 7 shows the learning curves on the largest group of in Cityscapes (Wurzburg), as well as a CDF of the validation accuracies achieved by the best models for all groups after 10 epochs. The learning curves on the largest group largely overlap, showing that all approaches are suitable from the accuracy standpoint–their runtimes and resource efficiency are what differentiate them. Recall that Kingpin, just like GTP and MTP, ensures sequential-equivalence for SGD. In the CDF, we see that DP leads to somewhat worse models than the others; this is just an artifact of the batch size being fixed, since PyTorch DDP raises the "effective" batch size [50]. Tuning b too will mitigate that.

Summary. Kingpin is the most resource-efficient approach for learning over groups across all 3 major ML access patterns we studied. It also scales well with many groups and is often significantly faster than prior art without sacrificing on accuracy.

6.2 Resource Usage Drill-down

We now dive into the resource utilization of all approaches for processor, memory, and network. Due to space constraints, we focus only on Algebraic (LBFGS) on Criteo-Country here. Sequential (SGD) on Cityspaces, including GPU, and all disk utilization plots are available in the appendix. Figure 8 shows the average CPU usage, max memory usage per worker, and network usage for all compared approaches.

Compute (CPU). Kingpin saturates the CPU most of the time while having about 10 falloffs overall. These falloffs are from the synchronization among all workers and occur at our epoch-level scheduling boundaries. Falloffs are more frequent in DP due to its exchanging of gradients after every group, config, and epoch. GTP makes the fullest use of CPU but gradually it sees some workers becoming idle. MTP has the strangest CPU usage with 12 spikes. After poring over the detailed logs, we find that these spikes correspond to 12 sets of hyperparameter configs. MTP evenly distributes configs instead of groups to workers. Training one group and config at a time is not able to saturate all CPU cores in this experiment.

Memory. We see that Kingpin ramps up memory at the beginning and keeps needed data in memory till the end. DP does the same, except that it loads one group at a time, leading to less memory used per worker. GTP loads all data to Ray's shared memory store and trains multiple configs by reading data from the store. Thus, we see its memory usage is also increasing sharply in the beginning and stays there till the end. MTP loads groups' data to memory repeatedly; so, we see 12 spikes on this plot too. Note that GTP tends to consume more memory due to extra serialization in Ray's shared memory store.

Network. These plots concur with our analytical cost models. Kingpin first reads data and then keeps gradients flowing around. DP also loads data and exchanges gradients after each epoch. GTP and MTP only load data over the network once; no more network usage is triggered during training, except MTP copies almost all groups' data to each worker. One unexpected observation here is that Kingpin triggers more network communications than DP but GAP's asynchronous nature amortizes them to yield significantly lower end-to-end runtimes anyway.

6.3 Scalability Discussion

Worker Scalability. We now vary the number of workers for Algebraic on Criteo-Country to show their speedup behaviors. Figure 9 shows the results. The curves validate our analytical cost model-based discussion for all approaches, with one difference being DP does not scale as well as we expected (its curve is flat here). As

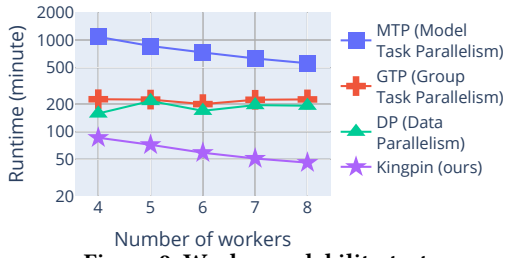

Figure 9: Worker scalability test.

the number of workers increases, Kingpin and MTP see speedups, while MTP benefits the most from more workers. This is because MTP can easily parallelize any group and config on any worker. DP's runtimes do not show any speedup at all mainly due to its high network communication costs. GTP's runtimes also barely change no matter how many workers there are; its total runtime is bounded by the time spent on the largest group.

Group Scalability. Our analytical cost models showed that the runtimes of all approaches except DP are agnostic to the number of groups. DP faces more overhead when training more groups. Table 5 confirmed this on Criteo, with Partner having 15x as many groups as Country. The differences in runtimes on these two attributes are marginal for Kingpin, GTP, and MTP, although we do see minor slowdown on Partner. This minor slowdown comes from the overhead of managing (create, read, and write) more models for all groups. DP suffers from a lot due to its communication costs and idleness as the number of groups went up.

7 RELATED WORK

In-RDBMS ML. There is much prior work on integrating ML with data systems. A key example is MADlib [24, 30], which integrates ML with RDBMSs via UDFs. Many DB and cloud vendors also support ML on their DBMSs, e.g., Google BigQuery ML [4, 63], Amazon Redshift ML [2], and more [7, 9, 23]. Several works also bring ML to dataflow systems, e.g., Mahout [11] for Hadoop and MLlib [47], MLlib* [66], PS2 [65], Horovod-on-Spark [5], and TensorFrames [10] for Spark. All these systems are *complementary* to our work; none of them optimize learning over groups. Our goal is to study the fundamental systems tradeoffs of this workload at scale, not to integrate ML with data systems. The "keyed models" API in Spark Scikit-learn [8] is the closest to our work in that it trains models in UDFs for each value of a grouping attribute. But it does not fully support model selection for groups and its execution approach is GTP, which suffers from the issues we explained in Section 3.3.

Higher-Level ML Systems. Our work is inspired in part from the recent line of in the DB world on "factorized ML" [17, 35, 38, 53, 54, 57, 64]. They optimize ML over joins of tables. For instance, [17, 42] push linear algebra operators down through joins to avoid materializing the join output. In contrast, our work enables and optimizes a GROUP BY abstraction for scalable ML model selection. Ease.ml [32, 44] and SystemML/SystemDS [12–15] are higher-level end-to-end ML platforms; Samsara [56] offers declarative ML on dataflow systems. SystemML also uses some hybrid-parallel execution schemes [15] but it is aimed at bulk linear algebra, not the 3 key gradient-based ML access patterns and model selection workloads we study. MLearn is a declarative language that compiles ML pipelines into SQL [59]. All of these works are *orthogonal* to ours. Our goal is *not* to build a general high-level ML system from scratch. Instead, we enable and optimize a GROUP BY abstraction for scalable ML model selection. Toward that goal, we devise a novel parallel ML execution scheme and unify the right mix of data partitioning, scheduling, and other system design decisions in a novel and effective manner.

Model Selection Heuristics and Systems. ASHA [40], Hyperband [39], Hyperopt [39] and PBT [31] are recent examples of methods that scale hyperparameter tuning. They are all *orthogonal* to our work; Kingpin can support them on top in the future. Ray Tune [45], Google Vizier [26], and Dask Hyperband [62] are model selection systems that implement some of the above heuristics; they are all task-parallel systems and thus suffer from the issues we explained in Section 3.3. Horovod [60] and SparkDL [3] offer some model selection support but they are data-parallel systems and thus suffer from the issues we explained in Section 3.3. Overall, none of these systems aim to holistically optimize learning over groups for model selection at scale. That said, they can adopt our techniques in the future to improve their efficiency.

Model Hopper Parallelism (MOP). We view GAP as a sister of MOP [37, 49, 50] in that both hybridize TP and DP. But they are fundamentally different and complementary in rationale, generality, and technique. MOP is aimed at Sequential (SGD) on a single large dataset that may not fit on one worker. GAP is aimed at learning over groups, not a single dataset, and supports Algebraic and Sampling access patterns too, not just SGD. MOP-based systems such as Cerebro [51] can be extended to support learning over groups via a lazy group-based subselection on the fly during model selection. However, such an approach will become too slow due to the overhead caused by the relatively low selectivity of most groups in practice, e.g., see the distributions in Figure 6.

8 CONCLUSION AND FUTURE WORK

Learning over groups is becoming a common practice among practitioners of large-scale ML. Existing approaches to scale this workload using task- or data-parallelism fail to view this process holistically and treat each group as an individual task, which results in poorer resource efficiency, lower model building throughput, and higher total runtimes. We compare existing approaches in depth analytically and empirically and then design a novel approach we call *grouped learning* to orchestrate learning over groups for large-scale ML model selection holistically and efficiently. We devise a novel parallel ML execution approach we call GAP and support 3 popular classes of ML models: generalized linear models, neural networks, and gradient-boosted decision trees. We adopt a new non-uniform data partitioning scheme suitable for this workload. All of our ideas are easy to integrate with existing ML tools, as we show by building our system, Kingpin, on top Ray and popular ML tools. Both analytically and empirically we find that Kingpin is often substantially faster than the alternative approaches. We hope our work helps ML practitioners interested in training ever more numerous models on group-specific data at scale to achieve more customization for ML applications. As for future work, we aim to extend Kingpin to support AutoML search heuristics, integrate it with Cerebro, and also enable cloud-native scalable execution.

REFERENCES

[1] [n.d.]. 10x Faster Parallel Python Without Python Multiprocessing. https://towardsdatascience.com/10x-faster-parallel-python-without-python-multiprocessing-e5017c93cce1. Accessed: 2021-1-29.

[2] [n.d.]. Amazon Redshift ML. https://aws.amazon.com/redshift/features/redshift-ml/. Accessed: 2021-1-29.

[3] [n.d.]. Deep Learning Pipelines for Apache Spark. https://github.com/databricks/spark-deep-learning. Accessed: 2021-1-29.

[4] [n.d.]. Google BigQuery ML. https://cloud.google.com/bigquery-ml/docs. Accessed: 2021-1-29.

[5] [n.d.]. Horovod on Spark. https://github.com/horovod/horovod/blob/master/docs/spark.rst. Accessed: 2021-1-29.

[6] [n.d.]. LightGBM Parallel Learning Guide. https://lightgbm.readthedocs.io/en/latest/Parallel-Learning-Guide.html. Accessed: 2021-1-29.

[7] [n.d.]. Oracle Machine Learning. https://www.oracle.com/data-science/machine-learning/. Accessed: 2021-1-29.

[8] [n.d.]. Scikit-learn integration package for Apache Spark. https://github.com/databricks/spark-sklearn. Accessed: 2021-1-29.

[9] [n.d.]. SQL Server Machine Learning Services. https://docs.microsoft.com/en-us/sql/machine-learning/sql-server-machine-learning-services?view=sql-server-2017. Accessed: 2021-1-29.

[10] [n.d.]. TensorFrames. https://github.com/databricks/tensorframes. Accessed: 2021-1-29.

[11] Robin Anil, Gokhan Capan, Isabel Drost-Fromm, Ted Dunning, Ellen Friedman, Trevor Grant, Shannon Quinn, Paritosh Ranjan, Sebastian Schelter, and ÃĈâĂŞzgÃĈÂijr YÃĎÂślmazel. 2020. Apache Mahout: Machine Learning on Distributed Dataflow Systems. *Journal of Machine Learning Research* 21, 127 (2020), 1–6. http://jmlr.org/papers/v21/18-800.html

[12] Matthias Boehm, Iulian Antonov, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, and Benjamin Rath. 2019. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. *CoRR* abs/1909.02976 (2019). arXiv:1909.02976 http://arxiv.org/abs/1909.02976

[13] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1425âĂŞ1436. https://doi.org/10.14778/3007263.3007279

[14] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1755âĂŞ1768. https://doi.org/10.14778/3229863.3229865

[15] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas R. Burdick, and Shivakumar Vaithyanathan. 2014. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow.* 7, 7 (March 2014), 553âĂŞ564. https://doi.org/10.14778/2732286.2732292

[16] Xavier Bouthillier and Gaël Varoquaux. 2020. *Survey of machine-learning experimental methods at NeurIPS2019 and ICLR2020.* Research Report. Inria Saclay Ile de France. https://hal.archives-ouvertes.fr/hal-02447823

[17] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2017. Towards Linear Algebra over Normalized Data. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1214âĂŞ1225. https://doi.org/10.14778/3137628.3137633

[18] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. 2016. The Cityscapes Dataset for Semantic Urban Scene Understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

[19] CriteoLabs. 2018. Criteo Sponsored Search Conversion Log Dataset. Retrieved November 20, 2020 from https://ailab.criteo.com/criteo-sponsored-search-conversion-log-dataset/

[20] CriteoLabs. 2018. Spark Custom Partitioner. Retrieved November 20, 2020 from https://labs.criteo.com/2018/06/spark-custom-partitioner/

[21] Jeffrey Dunn. 2016. Introducing FBLearner Flow: FacebookâĂŹs AI backbone. https://engineering.fb.com/2016/05/09/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/.

[22] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19).* USENIX Association, Renton, WA, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[23] 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 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20).* Association for Computing Machinery, New York, NY, USA, 755âĂŞ768. https://doi.org/10.1145/3318464.3386137

[24] 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).* ACM, 325–336.

[25] Fangeheng Fu, Jiawei Jiang, Yingxia Shao, and Bin Cui. 2019. An Experimental Evaluation of Large Scale GBDT Systems. *Proc. VLDB Endow.* 12, 11 (July 2019), 1357âĂŞ1370. https://doi.org/10.14778/3342263.3342273

[26] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Halifax, NS, Canada) *(KDD '17).* Association for Computing Machinery, New York, NY, USA, 1487âĂŞ1495. https://doi.org/10.1145/3097983.3098043

[27] Teofilo Gonzalez and Sartaj Sahni. 1976. Open Shop Scheduling to Minimize Finish Time. *J. ACM* 23, 4 (Oct. 1976), 665âĂŞ679. https://doi.org/10.1145/321978.321985

[28] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.* 1, 1 (Jan. 1997), 29âĂŞ53. https://doi.org/10.1023/A:1009726021843

[29] Trevor Hastie et al. 2001. *The Elements of Statistical Learning: Data mining, Inference, and Prediction.* Springer-Verlag.

[30] Joseph M. Hellerstein, Christoper 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 (Aug. 2012), 1700âĂŞ1711. https://doi.org/10.14778/2367502.2367510

[31] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Population Based Training of Neural Networks. arXiv:1711.09846 [cs.LG]

[32] Bojan Karlaš, Ji Liu, Wentao Wu, and Ce Zhang. 2018. Ease.Ml in Action: Towards Multi-Tenant Declarative Learning Services. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 2054âĂŞ2057. https://doi.org/10.14778/3229863.3236258

[33] 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. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[34] S. Knust, N. Shakhlevich, Stefan Waldherr, and C. WeiÃ§. 2019. Shop Scheduling Problems with Pliable Jobs. *Journal of Scheduling* (04 2019). https://doi.org/10.1007/s10951-019-00607-9

[35] Arun Kumar, Mona Jalal, Boqun Yan, Jeffrey Naughton, and Jignesh M. Patel. 2015. Demonstration of Santoku: Optimizing Machine Learning over Normalized Data. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1864âĂŞ1867. https://doi.org/10.14778/2824032.2824087

[36] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel. 2016. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Rec.* 44, 4 (May 2016), 17âĂŞ22. https://doi.org/10.1145/2935694.2935698

[37] Arun Kumar, Supun Nakandala, Yuhao Zhang, Side Li, Advitya Gemawat, and Kabir Nagrecha. 2021. Cerebro: A Layered Data Platform for Scalable Deep Learning *(CIDR'21).* http://cidrdb.org/cidr2021/papers/cidr2021_paper25.pdf

[38] Arun Kumar, Jeffrey 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* (Melbourne, Victoria, Australia) *(SIGMOD '15).* Association for Computing Machinery, New York, NY, USA, 1969âĂŞ1984. https://doi.org/10.1145/2723372.2723713

[39] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *Journal of Machine Learning Research* 18, 185 (2018), 1–52. http://jmlr.org/papers/v18/16-558.html

[40] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2020. A System for Massively Parallel Hyperparameter Tuning. arXiv:1810.05934 [cs.LG]

[41] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14).* USENIX Association, Broomfield, CO, 583–598. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

[42] Side Li, Lingjiao Chen, and Arun Kumar. 2019. Enabling and Optimizing Non-Linear Feature Interactions in Factorized Linear Algebra. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19).* Association for Computing Machinery, New York, NY, USA, 1571âĂŞ1588. https://doi.org/10.1145/3299869.3319878

[43] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. arXiv:2006.15704 [cs.DC]

[44] Tian Li, Jie Zhong, Ji Liu, Wentao Wu, and Ce Zhang. 2018. Ease.Ml: Towards Multi-Tenant Resource Sharing for Machine Learning Workloads. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 607âĂŞ620. https://doi.org/10.1145/3187009.3177737

[45] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. arXiv:1807.05118 [cs.LG]

[46] Robert McNaughton. 1959. Scheduling with Deadlines and Loss Functions. *Manage. Sci.* 6, 1 (Oct. 1959), 1âĂŞ12. https://doi.org/10.1287/mnsc.6.1.1

[47] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB 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. *Journal of Machine Learning Research* 17, 34 (2016), 1–7. http://jmlr.org/papers/v17/15-237.html

[48] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[49] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2019. Cerebro: Efficient and Reproducible Model Selection on Deep Learning Systems. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning* (Amsterdam, Netherlands) *(DEEM'19)*. Association for Computing Machinery, New York, NY, USA, Article 6, 4 pages. https://doi.org/10.1145/3329486.3329496

[50] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A Data System for Optimized Deep Learning Model Selection. *Proc. VLDB Endow.* 13, 12 (July 2020), 2159âĂŞ2173. https://doi.org/10.14778/3407790.3407816

[51] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Resource-Efficient Deep Learning Model Selection on Apache Spark. https://databricks.com/session_na20/resource-efficient-deep-learning-model-selection-on-apache-spark.

[52] Jorge Nocedal and Stephen J. Wright. 2006. *Numerical Optimization*. Springer.

[53] Dan Olteanu and Maximilian Schleich. 2016. F: Regression Models over Factorized Views. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1573âĂŞ1576. https://doi.org/10.14778/3007263.3007312

[54] Steffen Rendle. 2013. Scaling Factorization Machines to Relational Data. *Proc. VLDB Endow.* 6, 5 (March 2013), 337âĂŞ348. https://doi.org/10.14778/2535573.2488340

[55] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. arXiv:1505.04597 [cs.CV]

[56] Sebastian Schelter, Andrew Palumbo, Shannon Quinn, Suneel Marthi, and Andrew Musselman. 2016. Samsara: Declarative Machine Learning on Distributed Dataflow Systems.

[57] 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* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 3âĂŞ18. https://doi.org/10.1145/2882903.2882939

[58] Ethan L. Schreiber, Richard E. Korf, and Michael D. Moffitt. 2018. Optimal Multi-Way Number Partitioning. *J. ACM* 65, 4, Article 24 (July 2018), 61 pages. https://doi.org/10.1145/3184400

[59] Maximilian E. Schüle, Matthias Bungeroth, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. MLearn: A Declarative Machine Learning Language for Database Systems. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning* (Amsterdam, Netherlands) *(DEEM'19)*. Association for Computing Machinery, New York, NY, USA, Article 7, 4 pages. https://doi.org/10.1145/3329486.3329494

[60] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). arXiv:1802.05799 http://arxiv.org/abs/1802.05799

[61] S. Shalev-Shwartz. 2012. . https://doi.org/10.1561/2200000018

[62] S. Sievert, T. Augspurger, and M. Rocklin. 2019. Better and faster hyperparameter optimization with Dask.

[63] Umar Syed and Sergei Vassilvitskii. 2017. SQML: Large-Scale in-Database Machine Learning with Pure SQL. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 659. https://doi.org/10.1145/3127479.3132746

[64] K. Yang, Y. Gao, L. Liang, B. Yao, S. Wen, and G. Chen. 2020. Towards Factorized SVM with Gaussian Kernels over Normalized Data. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1453–1464. https://doi.org/10.1109/ICDE48307.2020.00129

[65] Zhipeng Zhang, Bin Cui, Yingxia Shao, Lele Yu, Jiawei Jiang, and Xupeng Miao. 2019. PS2: Parameter Server on Spark. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 376âĂŞ388. https://doi.org/10.1145/3299869.3314038

[66] Z. Zhang, J. Jiang, W. Wu, C. Zhang, L. Yu, and B. Cui. 2019. MLlib*: Fast Training of GLMs Using Spark MLlib. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1778–1789. https://doi.org/10.1109/ICDE.2019.00194

A VERIFICATION OF THE HEURISTICS

We use some real-world metadata to verify the schema and expected makespans yielded by the constrained wrap-around algorithm. We perform an empirical verification of the constrained wrap-around algorithm by random sampling from the Cityscapes dataset in Figure 6. In the first experiment, we fix it to four workers and change

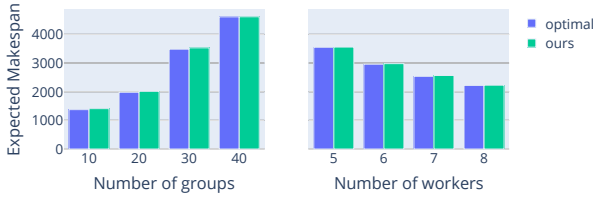

Figure 10: Simulation of the constrained wrap-around.

the number of groups. In the second one, we set the number of groups to 40 and change the number of workers. We compare the expected makespan returned by equation 2 and returned by our heuristic algorithm. The result is depicted in Figure 10. Overall, we find that the constrained performs only slightly worse than the optimal.

B STORAGE USAGE

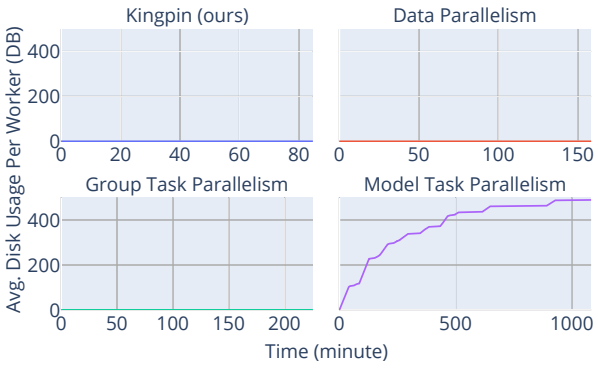

Figure 11: Disk usage of algebraic on Criteo (Country)

Figure 11 displays disk usage of all approaches. *Kingpin*, DP, and GTP all directly read data into memory, and therefore no disk space is used at all. MTP caches training data on local disk to avoid reading from remote storage repeatedly. However, We can see from the plot that each worker ends up caching almost a full copy of the dataset on the local disk.

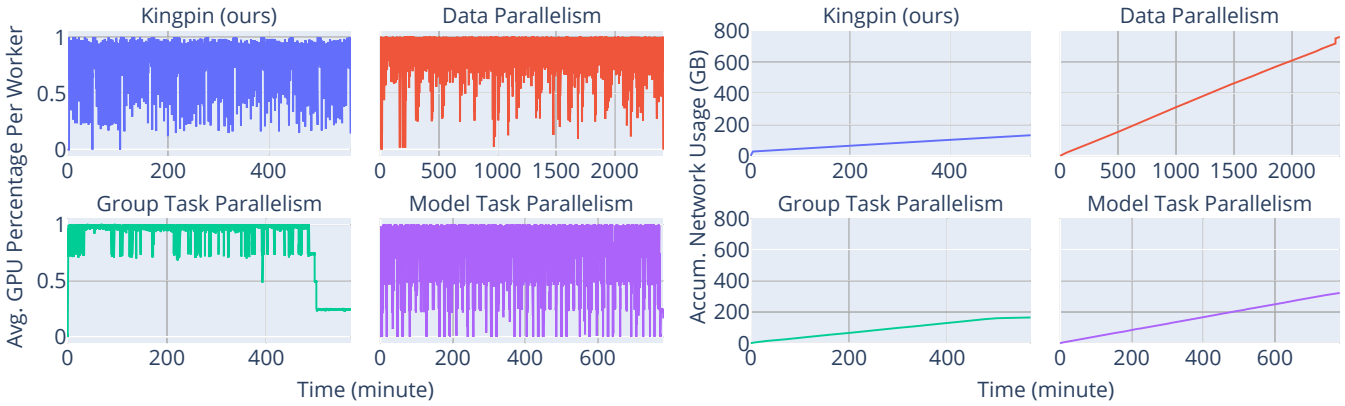

Figure 12: GPU and Network Usage of Sequential on Cityscapes

C RESOURCE USAGE OF UNET WORKLOAD

We discuss the resource usages in the experiment of running UNet on Cityscapes. We only cover compute (GPU) and network, as other factors are similar to those of algebraic workloads. Figure 12 shows the detailed usages.

Compute (GPU). *Kingpin* keeps GPU busy most of the time while having lots of falloffs in-between. These falloffs come from the overhead of switching models to train in GAP, leading to *Kingpin* making worse use of GPU than GTP. Alas, GTP leaves some workers idle towards the end. Thus, we see the GPU usage of GTP falls to 25% after about 500 min. MTP also has lots of falloffs when loading data repeatedly to memory. DP surprisingly saturates the GPU most of the time. We figure the reason being that NVIDIA's GPU utility library (NCCL) is in charge of communicating gradients.

Network. Overall, *Kingpin* and GTP have the minimum network usages. Both do not explicitly trigger network traffic except loading data and hopping models, though we still see about 100GB of additional network usage started by the underlying Ray infrastructure. MTP essentially copies the whole dataset to all workers, topping about 200GB of network usage. DP is the only approach that requires constant network communication within one epoch, as it uses all-reduce to exchange gradients for each mini-batch. Hence, we see its network usage grow over time to 800 GB to the end.

D MORE DISCUSSION ON SCALABILITY

In section 6.3, we cover worker and group scalability of all approaches. We now expand to compare them over other axes.

Data Scalability. The cost model plots in Figure 3 show that all approaches' runtimes increase as the size of data increases. While *Kingpin* and DP are reasonably scalable, MTP and GTP's runtimes shoot up for large data. For MTP, it is primarily because the full dataset will end up being copied to each worker over the network. GTP is less scalable because the largest group will always bind the end-to-end runtime. We can also find the memory usage of all approaches in Figure 8. DP is the least memory-hungry approach, as it only loads one group at a time and distributes data across workers. *Kingpin*, GTP, and MTP all require each worker to have at least 150GB of memory space. If we do not have big memory machines, a backup strategy for *Kingpin* will be to train a subset of groups at a time. For GTP and MTP, the only choice is to swap data to disk, which can be slow and flaky. Plus, MTP already caches data on local disks, and this is an unnecessary extra cost.

Skew Scalability. *Kingpin*, DP, and MTP are agnostic to skew as shown in the cost model plots in Figure 3, though MTP may break when the largest group's data cannot fit in the memory. GTP is expected to step into trouble if skew increases in the dataset, as it will place the largest group onto only one worker, whose runtime will also be the end-to-end runtime. Though not shown on the plots, MTP actually faces the same issue when the number of configurations is greater than the number of workers. Both *Kingpin* and DP try to partition data across workers, so skew is less of a concern.

E ANALYTICAL COST MODELS PARAMETERS

Using our analytical cost models, we contrast all approaches (*Kingpin*, DP, GTP and MTP) to expose trade-off spaces in Figure 3. The plots use cost parameters calibrated based on real empirical runs on Criteo dataset. The table below summarizes the parameters applied. We also have m=1e6, n=1.4e7, s=12, h=10, $|G|$=18 and n_{max}/n=0.25 as the basic setting. For each plot, we vary one variable,while fixing all other workload properties.

Approach (ours)	α	β	γ	δ
Kingpin	1	1	1	1e6
Group Task Parallelism	1	1	1	3
Model Task Parallelism	1	1	50	1
Data Parallelism	1	1	2	$\frac{2e6}{g \cdot p}$