Tech Report of Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches

Yuhao Zhang
University of California, San Diego
yuz870@eng.ucsd.edu

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

Frank McQuillan
VMware, Inc.
fmcquillan@vmware.com

Nandish Jayaram*
Intuit, Inc.
nandish_jayaram@intuit.com

Nikhil Kak
VMware, Inc.
nkak@vmware.com

Ekta Khanna
VMware, Inc.
ekhanna@vmware.com

Orhan Kislal
VMware, Inc.
okislal@vmware.com

Domino Valdano
VMware, Inc.
dvaldano@vmware.com

ABSTRACT

Deep learning (DL) is growing in popularity for many data analytics applications, including among enterprises. Large business-critical datasets in such settings typically reside in RDBMSs or other data systems. The DB community has long aimed to bring machine learning (ML) to DBMS-resident data. Given past lessons from in-DBMS ML and recent advances in scalable DL systems, DBMS and cloud vendors are increasingly interested in adding more DL support for DB-resident data. Recently, a new parallel DL model selection execution approach called Model Hopper Parallelism (MOP) was proposed. In this paper, we characterize the particular suitability of MOP for DL on data systems, but to bring MOP-based DL to DB-resident data, we show that there is no single "best" approach, and an interesting tradeoff space of approaches exists. We explain four canonical approaches and build prototypes upon Greenplum Database, compare them analytically on multiple criteria (e.g., runtime efficiency and ease of governance) and compare them empirically with large-scale DL workloads. Our experiments and analyses show that it is non-trivial to meet all practical desiderata well and there is a Pareto frontier; for instance, some approaches are 3x-6x faster but fare worse on governance and portability. Our results and insights can help DBMS and cloud vendors design better DL support for DB users. All of our source code, data, and other artifacts are available at https://github.com/makemebitter/cerebro-ds.

1 INTRODUCTION

Deep learning (DL) for data analytics continues to grow in popularity, leading to a growing demand for products that make it easier to adopt DL, especially among enterprises [57]. The DBMS community has long worked on bringing machine learning (ML) closer to the home of business-critical datasets in enterprises: DBMSs and other data systems. This paradigm of "In-DBMS ML" (or "In-data system ML") has waxed and waned over the last 20 years, with 3 general waves of work. It now merits a revisit in the era of DL.

One may wonder if DL is useful for DBMS users, since DL is primarily popular on unstructured data, while DBMSs mainly handle structured data [66]. Although much of DL's successes are on unstructured data that are commonly stored on filesystems or data lakes, DBMSs have long provided storage support for text, multimedia [94, 126], and other objects [23, 113]. Furthermore, due to the benefits of embedding learning and less feature engineering in DL [117, 124], many recent works in both research and enterprise applications show that DL is becoming increasingly usable and effective even on structured data [49, 76, 80, 118, 119]. Multimodal analytics combining structured and unstructured data are also popular and relevant for DB users [20, 78, 87, 122]. Finally, DL's "interpretability" pain, once a showstopper for some enterprise users, is being actively mitigated by ML researchers [32, 131]. In the reality of ML practice, data scientists do not think in an all-or-nothing manner; different model types, including DL, are popular for different use cases. A recent Kaggle survey [82] confirmed multiple model types, including DL, remain popular. This paper focuses specifically on DL because we believe it is an area that needs more attention from the database community.

1.1 Lessons from In-RDBMS ML

In the first wave of in-RDBMS ML, DB vendors built "data mining tools" that scaled a few ML algorithms to DB-resident data [14, 33, 93]. They enabled access to ML from the SQL console. But as ML algorithms grew in complexity, a second wave of unified implementation abstractions were devised for in-data system ML [34, 45]; MADlib [51] and Spark MLlib [83] are key examples. The third wave is seeing cloud DBMS vendors adding more in-RDBMS ML support, e.g., Google's BigQuery ML [3, 8, 13], as well as invoking DL from DBMSs [4, 9].

In this context, DBMS and cloud vendors are increasingly asking: *"How to enable seamless support for DL over DB-resident data?"*. The past waves of in-RDBMS ML offer at least four lessons.

(1) The main user base of in-RDBMS ML tools are not Python-oriented data scientists but SQL-oriented business analysts. Such users increasingly want access to DL training and inference *from within the SQL console*. As per estimates by the MADlib team [2], about 20-25% of Greenplum customers today use its in-RDBMS ML analytics capabilities alongside SQL analytics.

(2) Although *governance and provenance* were always important for enterprises in sensitive domains such as financial and health care, they now have renewed urgency for *all companies* including the tech giants, due to new laws such as GDPR [36] and CCPA [91].

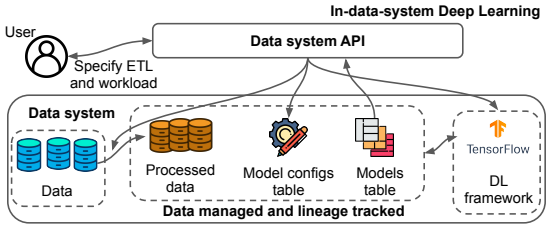

Figure 1: In-data-system DL. Data system invokes DL tool and helps mitigate data provenance/governance issues.

Companies will likely start frowning upon DL users manually exporting, copying, and moving business-critical data around in an ad hoc manner. Although one could program to automate such processes, and use services like MLFlow and Kubeflow [65, 85] for governance and provenance tools, it is still an extra burden for the enterprise users to learn, especially when they are already familiar with established DBMS support for governance/provenance.

(3) It is far too tedious for DBMS developers to reimplement DL algorithms. So, one must *preserve the usability of DL tools* such as TensorFlow for specifying complex DL workloads. This also allows analysts to just reuse DL training specification programs written by data scientists or others.

(4) Parallel RDBMSs already offer a mature execution engine on sharded large-scale data. But state-of-the-art distributed DL execution tools such as Horovod [108] are still notoriously painful to set up, operate, and debug [19]. This presents parallel RDBMSs/data systems an opportunity to *bridge the gap on scalable execution.*

Overall, we see two contrasting paradigms for how DL is brought to DB-resident data. The DL user can export the data to a file system, invoke a DL tool manually, and manage all derived data/metadata/artifacts on their own. Alternatively, in the "in-data system DL" approach, the DL workload and ETL are orchestrated by the data system itself, as Figure 1 illustrates. Crucially, this approach leaves room for *implementation flexibility* on how exactly the DL tool consumes the data; this flexibility opens up possibilities that we will explore later.

1.2 Toward In-Data System DL

Apache MADlib has recently pioneered in-DBMS DL support [11]. The DL workload is specified using Keras APIs, enabling business analysts to reuse DL configurations written by, say, data scientists. Under the hood, MADlib ships mini-batch data from the DB to a TensorFlow function invoked in a DBMS User Defined Function (UDF)/User Defined Aggregate Function (UDAF). For distributed execution, MADlib used the "model averaging" (MA)[1] heuristic for SGD [45, 134]. Alas, MA has poor convergence behavior for highly non-convex DL [89]. Thus, this approach is sub-optimal for bringing DL to DBs.

We observe that MA misses a major opportunity for parallelism in DL: *model selection.* ML theory teaches us that tuning hyperparameters is crucial, and this leads to the need for training many models [68, 109]. Often, DL users also compare alternate neural architectures, alter the base features, etc. Thus, model selection in practice often leads to dozens, if not hundreds, of models to train in one go [43, 89].

Exploiting the above observation, recent work proposed a new approach to distributed DL model selection called Model Hopper Parallelism (MOP) [69, 88, 89]. MOP is a *hybrid* of sharded data parallelism and task parallelism, the latter being common in the ML world [104]. MOP works as follows: train different models on different workers in parallel for one *sub-epoch* on their local shards, checkpoint and "hop" the models across workers, and restart training the *same epoch* on the next worker's shard; repeat all this across epochs. MOP is a form of bulk *asynchronous* parallelism since it imposes no barrier synchronization across workers, unlike Bulk Synchronous Parallel (BSP) data systems. Overall, MOP was shown to be the most resource-efficient approach to distributed DL model selection [89].

1.3 Focus of this Paper

Given the benefits of MOP we ask: *"How to bring MOP-based DL to DB-resident data?"* We find that there is no single "best" approach, and there is an interesting tradeoff space of alternative approaches. This paper explains these approaches, contrasts them analytically, and compares them empirically with large-scale DL workloads. We use Greenplum as the archetype but emphasize that the approaches compared are generic and applicable to any parallel RDBMS. Thus, *our results could be of wide interest to all DBMS and cloud vendors.*

We seek approaches that *do not change the code* of the data system. This eases practical adoption but restricts how MOP can be applied. For instance, Spark now supports flexible scheduling of workers [38]; this made it easy to integrate MOP with Spark in the Cerebro system [1]. But parallel RDBMSs such as Greenplum, AWS Redshift, etc., use BSP across workers, conflicting with MOP's asynchrony. We have *multiple axes of comparative evaluation*, including *runtime efficiency*, *ease of governance*, *implementation difficulty*, and *portability*. Section 3 explains all approaches and Section 4 compares them in detail, but as a preview, Figure 2 shows the approaches on the first two axes.

We compare 4 new approaches: (1) Fully in-DBMS MOP using UDAF; this approach has been adopted by MADlib [12] (2) Partially in-DBMS MOP using Concurrent Targeted Queries (CTQ) (to be introduced in Section 3.2); (3) In-DB but *not* in-DBMS (data is in DB but all operations are out-of-DBMS) MOP using Direct Access (DA); and (4) Regular out-of-DBMS approach using Cerebro-Spark. MA is largely dominated by the UDAF approach but all the other approaches fall on the Pareto frontier. For instance, the out-of-DBMS Cerebro-Spark approach and in-DB DA approaches are much more efficient than UDAF but may be harder to govern in a production environment. The CTQ approach offers a middle ground on these two axes.

Our comparative analyses of these approaches expose more interesting gaps. For instance, with theoretical and simulation analyses, we show that the efficiency gap between CTQ and UDAF grows wider when the models and hardware are more heterogeneous, even up to 6x in a realistic scenario. Finally, an extensive empirical comparison using the ML benchmark datasets ImageNet and Criteo shows that the real runtime gaps between UDAF and DA be as high as 3x. Overall, our experiments and analyses show that it is beneficial to bring MOP-based DL to DB-resident data, but it is

[1]In addition to MA, MADlib has adopted one of the approaches we will evaluate [12].

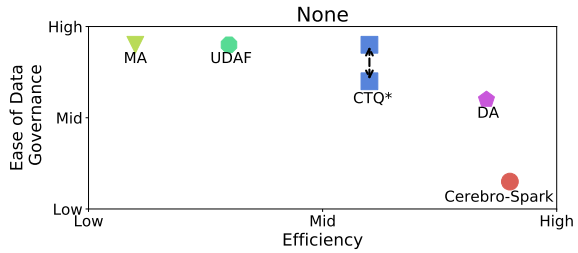

Figure 2: Tradeoffs of ease of data governance vs. efficiency for various approaches. *Depending on an implementation detail CTQ may have the same ease of governance as MA and UDAF; see Section 4.2 for details.

non-trivial to meet all practical desiderata. We hope our results spur more conversations in the DB and cloud industries on how best to support DL on DB-resident data.

In summary, this paper makes the following contributions:

- To the best of our knowledge, this is the first paper to analyze the tradeoffs and design alternatives of supporting large-scale DL model selection on DB-resident data.
- We show a spectrum of possible approaches on the Pareto frontier of efficiency, ease of governance, and other practical desiderata. In particular, we show a new approach that is in-DB but not in-DBMS, posing new accessibility questions for DB vendors.
- We perform a formal analysis of the limits of the efficiency gaps between the new approaches.
- We perform an extensive empirical comparison of the approaches using large ML benchmark datasets to evaluate their runtimes, scalability, and internal design tradeoffs.

2 BACKGROUND AND PRELIMINARIES

We start with a brief background on the most common DL training algorithm: mini-batch SGD, and the burden of DL model selection. Then we explain the challenges and constraints faced by in-DBMS approaches. Finally, we review and compare various paradigms on parallelizing DL model building on DBMSs and explain why MOP is perhaps the most desirable choice, given the DBMS constraints.

2.1 Background on ML Concepts

Mini-batch SGD. DL training is a non-convex optimization problem [48]. It is solvable by mini-batch SGD or its variants (e.g., Adam [61] or RMSprop [25]). SGD is iterative in nature and performs multiple passes over the data. Each pass is called an *epoch*. Within one epoch, it randomly samples a batch of examples (called a *mini-batch*) without replacement and uses that to estimate gradients and updates the model. It repeats this process until the epoch ends. Such sampling is achieved in practice by performing a *random shuffle* of the dataset beforehand or after each epoch. In SQL, this can be easily achieved using ORDER BY RANDOM() [45]. SGD is inherently sequential; deviating from sequential execution may worsen convergence behavior [89, 130], typically raising the number of epochs needed for a given accuracy.

Model selection. DL requires subtle tuning. Developing a DL model from scratch is challenging, for the model's performance depends non-linearly on the neural architectures and hyperparameter settings. Neural architecture defines a DL model, and different architectures could offer drastically different accuracies on the same task. Hyperparameters are the knobs that control how the model is trained and regularized. It is critical to tune these settings as they could greatly affect the model accuracy [68, 109]. Hence, the user needs to find the best combination of choices for neural architectures and hyperparameters; this process is called model selection [68]. We call the collection of model architecture and hyperparameters model configuration, or simply model config. Due to model selection burdens, DL routinely requires training of dozens to hundreds of models [43]. There is a line of research trying to guide model selection with meta-heuristics like grid/random search, PBT [54], ASHA [73], Hyperopt [26]. These works are orthogonal to our work, because they study the algorithmic issue of *what* configurations to explore, while our paper studies the systems-level issue of *how* to execute a given model selection workload efficiently and compares the alternatives on multiple fronts.

2.2 Constraints and Challenges in Bringing DL to DBMSs

We first consider fully in-DBMS DL that relies only on UDAFs/UDFs without modifying the internal code of the DBMS. That is, rely only on UDFs/UDAFs and use the data handling functionalities of the DBMS. It is challenging to implement because of constraints that many parallel DBMSs share. We summarize these constraints as follows:

Bulk synchronous parallelism (BSP). Each query executes in an all-or-nothing manner on a dataset that is sharded across workers. A synchronization barrier is injected at the end of every query. There is no trivial way at the SQL/UDF level to poll partial results.

No message-passing among workers. Some of the existing DL systems rely on protocols such as MPI or RPC for communication, but to enable these functionalities at the UDF level would require modifications to the DBMS and/or substantial efforts. Hence the preferable communication method among workers is the pipes provided by the DBMS. This constraint would make some distributed DL paradigms especially hard to implement.

One query at a time. For each database connection session, only one query is permitted at any time. Using multiple clients and DB sessions for the same query is a way to achieve parallelism by manually dissecting the query into subqueries and unifying the results on the client-side, it may be considered as an anti-pattern as now the query planning takes place out-of-DBMS.

Data access through DBMS. In a DBMS, data is usually compressed and stored on disk as pagefiles (physical files on disk that contain database pages.). To access data, one must go through the DBMS query stack. If the data is frequently and iteratively accessed as we see in DL training, such repeated accessing and decompression could bring serious overheads.

2.3 Paradigms for Distributed DL

There has been a lot of work on multi-node parallel DL training. However, most of these techniques do not have or assume a trivial data layer. Adjustments or re-implementation must be made to integrate them into an existing data system. The DBMS has constraints

Table 1: Summary of various parallel paradigms' fitness for in-data-system DL.

	Centralized communication	Coarse grained	Data-parallel	Fast convergence
Task Parallel	✓	✓	✗	✓
Model Avg.	✓	✓	✓	✗
Param. Server	✓	✗	✓	✓
Horovod	✗	✗	✓	✓
MOP	✓	✓	✓	✓

that render many of the approaches unsuitable or difficult to implement. We translate the constraints of Sec 2.2 into the following requirements for distributed DL paradigms for amenability to the in-DBMS setting.:

- **Centralized communication.** As mentioned in Section 2.2, we want the communication pattern to be as simple as possible. P2P communication is typically not allowed.
- **Coarse-grained parallelization.** The training would better be parallelized at epoch instead of mini-batch level. Since we will embed the training jobs as data system tasks/queries, fine-grained parallelization will lead to massive number of queries that can cause heavy overheads.
- **Data-parallelism.** The data is already partitioned in the data system. Fully replicating the entire data across workers is not desirable and may not even be feasible at large scales.
- **Fast convergence.** In order to save computational and resource costs of model selection, we want the models to converge fast in terms of number of epochs, ideally resembling the learning curves obtained by the gold-standard sequential SGD.

Next we explain the major distributed DL model selection approaches in the literature and explain how well they fit (or not) the above constraints. Table 1 summarizes our comparative analysis.

Task Parallel. In this paradigm, different model configs of the model selection workload run on different workers in a task-parallel manner. Example tools include Python Dask, Celery, Vizier [47], and Ray [86]. Workers locally run sequential SGD on the whole dataset. Thus, this approach provides the best convergence efficiency. There is no communication across workers during training. Still, it *requires full data replication on each worker*, which is inefficient, and may not even be feasible for large sharded datasets in DBMSs.

Model Averaging (MA). In BSP systems such as TensorFlow with model averaging [5], data is sharded. The model configs are trained in parallel one-by-one. Every model is broadcasted and trained on each worker's data shard independently. Then a merge step takes place on the master; it averages the weights (or gradients). This process repeats every epoch. This approach is a potential candidate and has been adopted by MADlib [11]. Alas, it converges poorly for DL models, which are highly non-convex [112]. Nevertheless, since it satisfies most of the constraints, we include it as a key baseline in our experiments.

Fine-grained Parallel. These paradigms are similar to BSP, but they work at a finer granularity at the mini-batch level. The communication pattern can be centralized or decentralized. The most prominent example of centralized paradigms is Parameter Server (PS) [74]. The best example for decentralized paradigms is

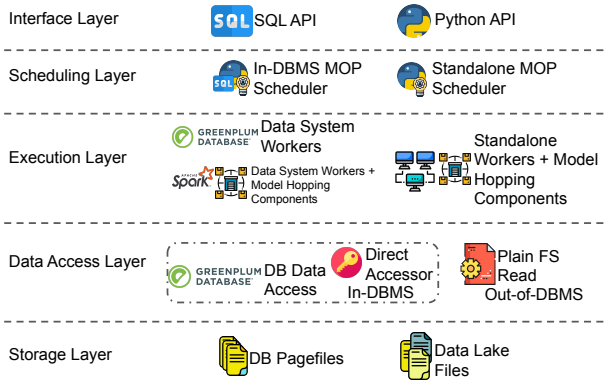

Figure 3: Design alternatives for MOP in DBMS.

Horovod [108]; it adopts HPC-style techniques to enable synchronous all-reduce SGD. These methods all have good convergence behavior but very high communication costs. They too are not good candidates because of the finer granularity. Horovod further requires P2P communication patterns that are not allowed in most data systems.

Model Hopper Parallelism (MOP). MOP used in system Cerebro [89] is recent progress towards resource-efficient DL. This is a hybrid of task- and data-parallelism. Each worker is assigned one model config from the model selection workload and trains the model with its local data shard; this process is called one sub-epoch. When one sub-epoch finishes, the model is passed to other data shards for further training. After several sub-epochs, every model finally has seen the entire dataset, and that is one epoch of training. Overall, a model *hops* from one worker to another in-between sub-epochs. The scheduling is done via an asynchronous random scheduler that works well on heterogeneous workloads and supports fault tolerance. MOP fits all our requirements because communication-wise, it has a centralized pattern and low cost, for it works at sub-epoch granularity. Data-wise, it works nicely with sharded data. Finally, it offers equivalency to sequential SGD, which has the highest convergence efficiency. Hence, we decided MOP would be a better choice for in-DBMS DL.

3 OVERVIEW OF ALTERNATE APPROACHES

Given the benefits of MOP, the question becomes how to bring MOP-based DL to DBMS-resident data. There are multiple possible approaches due to the implementation flexibility. To better explain these alternatives, we first divide the components of MOP execution into five layers of design decisions: Interface, Scheduling, Execution, Data Access, and Storage. Each layer can be implemented in flexible ways. Figure 3 summarizes the architectural alternatives.

(1) Interface layer: the high-level APIs that take in the user's DL model selection workload; it could be implemented in SQL, familiar to business analysts, or in Python, familiar to data scientists.

(2) Scheduling layer: the scheduler orchestrates and manages placements of training units. We could implement the scheduler as an in-DBMS procedure or use a standalone MOP scheduler.

(3) Execution layer: execution engine invokes DL tools and conducts model training via mini-batch SGD. We could use the data systems' execution engine or resort to standalone Cerebro for this

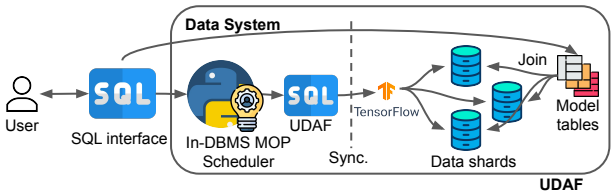

Figure 4: UDAF approach. Fully in-DBMS.

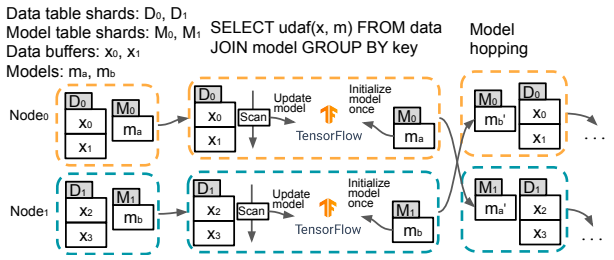

Figure 5: Conceptual illustration of one sub-epoch of UDAF.

Table 2: Summary of each approach's design for the 5 layers. IN: in-DBMS. OUT: out-of-DBMS.

Approach	Interface	Scheduling	Execution	Data Access	Storage
UDAF	SQL	IN	IN	IN	IN
CTQ	Python	OUT	IN	IN	IN
DA	Python	OUT	OUT	OUT	IN/OUT
Cerebro-Spark	Python	OUT	OUT	OUT	OUT

layer. Model hopping can be cast as SQL queries or be implemented as separate components with other communication methods.

(4) Data Access and Storage layer: we could leave the data in DBMS or export them. There are also multiple ways to access the data; if data is in a data lake, access is trivial. Otherwise, if the data is in DBMS, we can rely on DBMS's native data accessor or use a technique we call Direct Access to bypass the whole query stack and access the data from its physical storage directly.

Because of these flexibilities, there are various approaches for the end-to-end implementation of MOP. We find four interesting canonical approaches: (1) Fully in-DBMS MOP using User-Defined Aggregate Functions (UDAF); (2) Partially in-DBMS MOP using Concurrent Targeted Queries (CTQ); (3) In-DB but not in-DBMS MOP using Direct Access (DA); and (4) Regular out-of-data-system approach using Cerebro-Spark. We use Greenplum as the archetype but emphasize that the approaches compared are generic and applicable to any parallel DBMS. We do not claim these are the only possible approaches; rather, we find these are prototypical examples of feasible approaches based on the combinations of design decisions in Figure 3. We now introduce each approach and dive into the analysis and comparisons later in Section 4. We summarize the design choices for each approach in Table 2.

3.1 User-Defined Aggregate Functions (UDAF)

This approach implements MOP as DBMS extensions with UDFs and UDAFs. On the 5 design decisions of this approach and other approaches to be introduced, please refer to Table 2. UDAF approach is called fully in-DBMS because all functions are in-DBMS procedures, and both data and models are stored in DBMS.

Figure 4 illustrates the approach. It maintains a data table and a model table storing the DL model selection workload, with each row containing a model. Both tables are sharded based on distribution

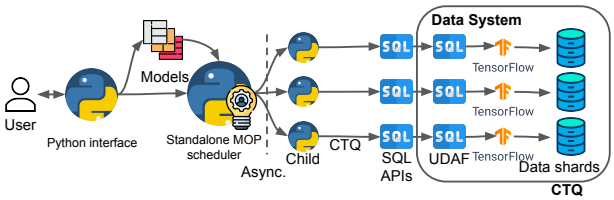

Figure 6: CTQ approach. Partially in-DBMS.

keys. These keys are used by the DBMS to determine where the rows are stored. The rows are distributed across worker nodes by the master node matching the values of a designated distribution key column. So all the rows with the same distribution key end up in the same worker node. By manipulating these keys, we can control the affinity of data/model. The user first defines the model architectures and workloads through a SQL interface. DBMS then invokes the MOP scheduler implemented in UDF.

This approach's scheduler is synchronous due to the BSP nature of in-DBMS execution, in contrast to the asynchronous random scheduler that standalone MOP adopts. It uses a simple round-robin heuristic for placing models on data shards. The scheduler translates the workload into UDAF queries dispatched and executed on the joined table of data and model. These UDAFs subsequently invoke DL tool (we use TensorFlow/Keras) for training. It schedules a batch of sub-epochs on the workers at a time and waits for completion. After several batches, one epoch is completed; it then repeats the process to train for multiple epochs.

Conceptually, each UDAF is a query of `SELECT udaf(...) FROM data JOIN model GROUP BY key`. Figure 5 illustrates the execution of one sub-epoch batch. Data is pre-packed into buffers and stored in a sharded table. Models are stored similarly in another sharded table. On each physical node of the DBMS, there are multiple rows of data buffers and only one row of model. During the execution, the model row is fed to DL tools to initialize the model, which will be stored as the aggregation state. The worker then scans the data shard and feeds each data buffer to the DL tool, which unpacks the buffers and generates mini-batches for training and updating the stored model. After scanning is done, the scheduler redistributes updated models to different physical nodes by manipulating their distribution keys. The data table, on the other hand, never redistributes.

3.2 Concurrent Targeted Queries (CTQ)

This approach is built upon CTQs, a DBMS feature we will explain shortly. In contrast to UDAF, it has a Python interface, uses a standalone MOP scheduler and out-of-DBMS model hopping components. We chose to store and hop models out of the DBMS for implementation simplicity; it is technically possible to keep models governed by DBMS, which can raise this approach's ease of governance (see Section 4.2). Since some core computations run outside the DBMS, we call this approach partially in-DBMS.

We now explain what a CTQ is. In a parallel DBMS, tables are sharded according to distribution keys. When a query only affects one shard, e.g., with a predicate that filters on the distribution key, the query processor will dispatch a query plan to that specific shard only. Such feature is sometimes called targeted query and commonly available [7, 21, 84]. Meanwhile, most DBMSs also allow concurrent queries. Therefore, we can assume more fine-grained

Table 3: Conceptual comparison of various architectural approaches of integrating MOP with DBMS. *CTQ can have highest or high ease of governance, depending on whether models are governed by DBMS. †If node RAM is insufficient, swap is needed for DA and the blowup could rise up to 2x.

	Efficiency	Governance	Storage Blowup	Implementation Difficulty	Portability	Design Anti-patterns
UDAF	Medium	Highest	None	Medium	Medium	No
CTQ	High	High-Highest*	None	Medium	Medium	Yes
DA	Highest	High	None - 2x†	Hard	Low	Yes
Cerebro-Spark	Highest	Low	2x	Easy	High	N/A

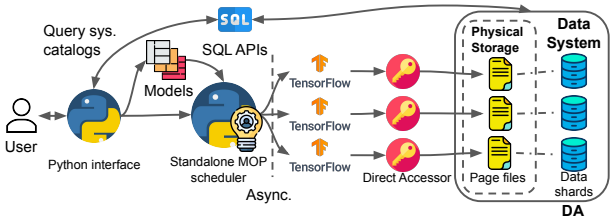

Figure 7: DA approach. In-DB but not in-DBMS.

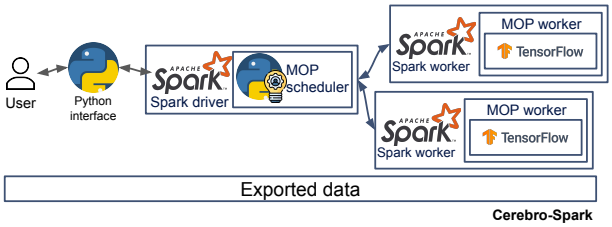

Figure 8: Cerebro-Spark approach. Fully out of DBMS.

control over the execution by issuing targeted queries concurrently. We name this trick Concurrent Targeted Queries (CTQ).

Figure 6 shows the CTQ approach. The user interacts with a Python interface to define models and workloads. It then invokes a standalone MOP scheduler, as described in [89]. This scheduler works differently from the one used in the UDAF approach; it orchestrates DL training by spawning children processes that contain DBMS connections and using them to issue CTQs. Meanwhile, models are hopped outside the DBMS; we use a shared filesystem for this task. Conceptually, each CTQ is a query of `SELECT udf(model) FROM data WHERE key=x`. Each DBMS node loads the assigned model from the shared filesystem and uses its local data shard to train the model, then checkpoints the updated model back to the filesystem. This concludes one sub-epoch; after every model has visited every data shard once, it is called one epoch.

3.3 Direct Access (DA)

This approach further deviates from UDAF and CTQ by employing a method we call Direct Access, bypassing the entire query processor of DBMS and accessing the on-disk pagefiles directly. This way, there is enough freedom to plug in and run standalone Cerebro [89] system but without exporting the data. This approach is called in-DB but not in-DBMS because although the DBMS still governs data, all executions are out of DBMS.

Figure 7 illustrates DA. The user talks to a Python interface to define workloads and query necessary system catalogs. DA then uses the standalone Cerebro for scheduling and execution. Workers perform training on the data table's sharded pagefiles directly through DAs. DAs first retrieve the pagefiles' location, mapping, layout, and compression information from system catalogs. Then they emulate DBMS's access methods to fetch the pages' contents and feed the data to Cerebro. The latter then consumes the data and runs MOP to train the workload. Notably, this approach is very generalizable and not limited to MOP execution; one can essentially plug in any data-parallel training frameworks like Pytorch or Horovod. To demonstrate the generality, in addition to MOP, we will also implement a fine-grained data-parallel approach with DA using Pytorch DDP. We will show the evaluations in Section 5.1.

3.4 Cerebro-Spark

Cerebro-Spark is a regular out-of-DBMS approach that exports data to filesystem, runs ETL processes, and feeds data to the DL tools. It uses the data system (Spark) workers and stores data as plain files. The DBMS does not participate in the training and loses the governance of data.

Figure 8 illustrates the architecture. The user defines workloads through Python APIs. The standalone MOP scheduler initializes MOP workers by embedding them as long-running Spark tasks. It then communicates with these workers and orchestrates the training just like in the standalone Cerebro system. In addition to Cerebro-Spark, we will also evaluate other frameworks such as Pytorch DDP and Hyperopt-Spark in Section 5.

4 COMPARATIVE ANALYSES OF APPROACHES

With all the approaches introduced, we now compare and analyze them on 6 major axes: runtime efficiency, ease of governance, storage blowup (defined as the actual storage usage divided by the original data size), implementation difficulty, portability, and design anti-patterns. Table 3 shows a conceptual comparison. These axes represent the desiderata and we find that no single approach can fulfill all of them. The more the approach is in-DBMS, the lower the runtime efficiency but the higher the ease of governance and vice versa. We will discuss the reasons in Section 4.1 and 4.2. In terms of storage, Cerebro-Spark has 2x blowup because of exporting; the other approaches have no such blowup. The situation is more complicated on the implementation difficulty and portability axes, and we will give a more rigorous analysis in Section 4.3 and 4.4. As for the last axis, CTQ and DA both introduce design anti-patterns since they are not fully in-DBMS: CTQ introduces a user-level anti-pattern by issuing multiple queries concurrently from outside of the SQL console, while MADlib and many other tools makes a single query inside a SQL console. This violates *One query at a time* mentioned in Section 2.2. DA has anti-patterns that violate *No message-passing among workers* and *data access through DBMS* from Section 2.2. UDAF is free of such issues. In the rest of

Table 4: Notation for discussion on scheduling makespans.

Notation	Description
$\mathbb{M}, M$	Set of models and the cardinality of it
$\mathbb{W}, W$	Set of workers and the cardinality of it
$\mathbb{L}$	Set of each model's per sub-epoch runtime
$\mathbb{L}_i$	For UDAF only. Batch of models scheduled for the i-th sub-epoch
m_x	The x-th model
l_x	The per sub-epoch runtime of the x-th model
l_s	A scale representing the runtimes of fast models
l_m	A scale representing the runtimes of slow models
p	Probability of a model being a fast model
T_u, T_c	End-to-end runtimes for sync. and async. MOP, respectively
η	Theoretical upper bound of the speedup T_u/T_c

this section, we pick 4 of the most interesting axes and analyze them in more detail.

4.1 Runtime Efficiency

This is one of the most important desiderata. Several factors affect runtime: DL tool invocation, data access, model hopping, and schedule makespans (end-to-end runtime of the generated schedule).

DL tool invocation. Both UDAF and CTQ invoke the DL tools through wrappers, whereas DA and Cerebro-Spark do not. Such wrappers may be a source of inefficiency.

Data access. Both UDAF and CTQ access data through DBMS. They could be bottlenecked by data transmission[2], especially when the data is compressed and tweaked by the DBMS, e.g., TOAST-ed [17]. DA can mitigate this issue; by accessing the physical pagefiles directly and caching data, it provides similar efficiency to Cerebro-Spark, which also reads from filesystem and caches data in memory.

Model hopping. Model hopping might be another source of inefficiency. CTQ, DA, and Cerebro-Spark all do model hopping outside of the DBMS and have similarly low overheads on this end, as [89] pointed out. On the other hand, the UDAF approach relies on the DBMS to hop models through `JOIN` between the data and model tables. This `JOIN` may bring some overheads, especially if the models are large. In later experiments (Section 5.2.3), we will indeed see UDAF is much slower than CTQ on model hopping. However, even for UDAF, model hopping still incurs negligible runtime compared to other components.

Scheduling makespans. CTQ, DA, and Cerebro-Spark all employ the same asynchronous random scheduler. This scheduler's robustness on heterogeneous workloads/workers has been tested in [89]. However, UDAF uses a synchronous round-robin scheduler, which may not work very well with heterogeneity. We show visualizations of potential scenarios in Appendix. How large is the gap between these two schedulers, and could it be a major performance bottleneck? We now analyze the differences theoretically between sync. and async. MOP and later verify it empirically in Section 5.2.2. Table 4 presents all notations used in this section.

Let there be a set of model configs $\mathbb{M}$ and a set of workers $\mathbb{W}$. $|\mathbb{M}| = M$, and $|\mathbb{W}| = W$. Assume the workers to be identical and each worker contains the same amount of data. Let l_x denote the per sub-epoch runtime of model config m_x and $\mathbb{L} = \{l_x\}$. For analysis simplicity, let $\mathbb{L}$ be a two-mode right-tailed distribution, i.e., most models are fast and have per sub-epoch runtime of l_s, while only some are slow and take l_m, $l_m \gg l_s$. Let p be the probability of l_x being fast: $p = Pr(l_x \sim l_s)$. We now analytically compute the per-epoch makespan T_u and T_c for sync. and async. MOP, respectively. We have the following two propositions, the proofs to them can be found in Appendix.

Proposition 1. Speedup of async. over sync. MOP is:

$$\frac{T_u}{T_c} = p^W \frac{l_s}{\bar{l}} + (1 - p^W)\frac{l_m}{\bar{l}}. \tag{1}$$

Proposition 2. Theoretical upper bound of the speedup is:

$$\eta = \frac{l_m}{pl_s + (1-p)l_m}. \tag{2}$$

Section 5.2.2 shows an experiment that verifies the analysis.

4.2 Ease of Governance

As we mentioned earlier, data governance/provenance now has renewed urgency for all enterprises and even the Web companies, because of the new regulations and laws like GDPR [36] and CCPA [91]. Among the four approaches, UDAF provides the best support for governance/provenance, as it keeps both the dataset and the models in DBMS, which already has built-in governance support. CTQ and DA both use DBMS to govern data. For CTQ, we chose to store models out of DBMS for simplicity, but it is technically possible to keep models in DBMS; this way, it can provide similar ease of governance as UDAF. DA, which relies on external Cerebro, does not manage models with DBMS and thus, loses some ease of governance. Cerebro-Spark does not come with existing governance support and may impose other security issues due to the ad hoc data export and copying. To regain governance, one has to maintain exporting scripts and seek help from external services like MLflow or Kubeflow [65, 85], and such external services are not under the DBMS vendor's control.

4.3 Implementation Difficulty

The out-of-DBMS approach (Cerebro-Spark) is generally the easy one to implement. One naive implementation would be a `SELECT * FROM ...` query followed by some pipelines that feed the exported data to DL tools. The UDAF approach requires more effort to implement the MOP scheduler, wrappers for invoking DL tools, and pipelines that feed data to DL tools and return results to the DBMS. CTQ requires similar efforts as UDAF does, except its scheduler is asynchronous and slightly harder to implement due to concurrency in queries. DA requires the most effort because it needs to implement/port the whole DBMS table scan method, including locating, unpacking, and reading the pagefiles. If the table is compressed and TOAST-ed [17], then one must also implement/port the decompression and de-TOAST methods. Such work is ad-hoc, DBMS-specific, and may not even be viable for proprietary DB and pagefile formats. Simultaneously, because its execution is outside the DBMS, unified memory management is difficult, and it could interfere with other queries. As a result, more careful tuning and setting of configurations are required to implement DA.

4.4 Portability

Portability indicates how much code can be reused if one wants to change the underlying DBMS. The out-of-DBMS approach again excels in this area because it is almost agnostic to the DBMS and can usually be ported easily. UDAF approach is also portable as

[2] Active development by the MADlib team is going on to mitigate this issue.

Table 5: Workloads.*architectures similar to VGG16 and ResNet50, respectively.

Dataset	Model arch.	Batch size	Learning rate	Regularization	Epochs
ImageNet	{VGG16*, ResNet50*}	{32, 256}	$\{10^{-4}, 10^{-6}\}$	$\{10^{-4}, 10^{-6}\}$	10
Criteo	3-layer NN, 1000+500 hidden units	{32, 64, 256, 512}	$\{10^{-3}, 10^{-4}\}$	$\{10^{-4}, 10^{-5}\}$	5

it requires only UDFs and UDAFs, which are supported in most DBMSs. Medium efforts are needed to export these functions to other DBMSs. CTQ is largely similar to UDAF, except it, in addition, requires the DBMS to support concurrent targeted queries. DA is the less portable option, as it is deeply coupled with the DBMS. Unless the target DBMS employs a similar physical storage layer, to port one existing DA implementation would be difficult.

5 EMPIRICAL COMPARISONS AND ANALYSES

We will first thoroughly compare the end-to-end performance of all the described approaches and study the tradeoff space. Then we will study the effects of factors such as heterogeneous and AutoML (Hyperopt) workloads and model sizes. We will also evaluate the scalability of each approach. All of our source code, data, and other artifacts are available at [6]. We will test on both GPU-enabled and CPU-only environments. One might wonder how GPUs will be available in practice for users that operate traditional DBMS clusters. As per Greenplum team estimates [2], at least 80% of its customers continue using on-premise clusters, largely due to privacy and security concerns, especially in the government, financial and health care sectors. Such users are increasingly purchasing GPUs and connecting them to their Greenplum clusters for in-house deployment of DL workloads. In cloud-native DBMSs such as AWS Redshift, one can easily spin up GPU instances and connect them with the DBMS instances. Use of hybrid cloud and public cloud is also increasing. It is not uncommon to run POCs and tests in public cloud with rented GPUs, before purchasing GPUs for in-house production deployment.

Compared approaches. We compare Cerebro-Spark, UDAF, CTQ, DA (renamed to DA-Cerebro), and MADlib MA, which is included as a key baseline. Only for the end-to-end test, we also include PytochDDP, a fine-grained out-of-DBMS data parallel DL training framework; it relies on NCCL and MPI for communications. We have also combined DA with PytorchDDP (named as DA-PytorchDDP) so that it can work with DB data directly. For the Hyperopt tests in Section 5.2.4, we also include a system called Hyperopt-Spark, which is an out-of-DBMS task parallel model selection system.

Datasets. We use two large benchmark datasets: *ImageNet* [39] and *Criteo* [37]. We use the processing scripts and versions released as part of Cerebro [1, 89]. ImageNet contains 1.2M images with 1000 classes; it has an on-disk size of 250GB. Criteo has 100M data points, binary classes, and an on-disk size of 400GB.

Workloads. We use various DL model selection workloads with different degrees of heterogeneity for different tests. Please refer to each corresponding section for details. We use Adam [61] as the mini-batch SGD method for all tests.

Experimental Setup. We use one cluster on CloudLab [103] with 8 worker nodes and 1 master node. Each node has two Intel Xeon 10-core 2.20 GHz CPUs, 192GB memory, 1TB HDD, and 10 Gbps network. Each worker node also has an Nvidia P100 GPU. For tests with MLP on the Criteo dataset, we disable the GPUs to demonstrate the system's performance under CPU-only setting. All nodes run Ubuntu 16.04. We use GPDB 5.27, Spark 2.4.5, Cerebro 1.0.0, TensorFlow 1.14.0, Pytorch 1.4.0, CUDA 10.0, and cuDNN 7.4. Both datasets are randomly shuffled and split into 8 equal-sized partitions.

5.1 End-to-end Performance Study

We first present the end-to-end results for both ImageNet and Criteo. For ImageNet, we use two different neural architectures and a hyperparameters grid, yielding 16 training configs. For Criteo, we conduct a hyperparameter-tuning-only workload with also 16 training configs. Table 5 offers the details. Such grid search-based model selection is standard in DL practice and still widely used by practitioners [31]. We compare our various architectural approaches with each other. MA is the baseline for this comparison.

For each different approach, separate ETL processes must be done beforehand. For UDAF, CTQ, and MA, ETL is in-DBMS preprocessing that packs the original data into byte arrays and buffers for the UDAFs to consume. For DA, ETL includes the above processing, plus accessing tables and TOAST pagefiles, de-TOAST, and loading into the main memory. For Cerebro-Spark, ETL consists of data exporting from DBMS and preprocessing to cast the data formats; we use a distributed Greenplum ETL tool `gpfdist` [116] for exporting and a customized program for preprocessing.

We examine the performance on multiple fronts: convergence, runtime, and resource utilization/cost including GPU/CPU, DRAM, network, and disk. Figure 9(A) demonstrates the convergence behaviors for ImageNet. All but MA converge to the same optima, as they are equivalent to sequential SGD. MA, on the other hand, has a convergence problem and learns much slower than the rest. We skip the convergence curves on Criteo for brevity's sake because all methods, including MA, have almost indistinguishable convergence behavior (reaching 99% accuracy quickly).

Table 6 summarizes the runtime performance and resource utilizations/costs.[3] The three in-DBMS approaches, MA, UDAF, and CTQ, show close speed. They have identical reads on each dataset, equal to the local on-disk pagefile size (12 GB for ImageNet and 1 GB for Criteo). After the first table scan the pagefile remains in the OS cache. MA is marginally the fastest among them, but note that it has poor convergence, as Figure 9(A) shows. CTQ is slightly faster than UDAF due to the removal of sub-epoch level synchronization barriers. The benefit is not obvious in this test, but we will drill deeper into the behavior in Section 5.2.2. DA-Cerebro and Cerebro-Spark show the best performance and are close in runtime. This shows that one can achieve the same high performance as a state-of-the-art out-of-DBMS DL approach while still operating on DB-resident data. The two data parallel methods PytorchDDP and DA-PytorchDDP are heavily bottlenecked by networking (over 1700x higher cost compared to other approaches) and had the worst

[3]Runtimes may not be directly comparable to figures in [89], as in this paper, we adopted newer model implementations and different on-disk file formats.

Table 6: Runtimes and resource utilizations of end-to-end tests. Execution time and all utilizations are measured excluding ETL. Per-epoch time equals Execution time divided by number of epochs. Total network means the total amount of data transmitted during execution. We report disk read/write as per worker average. *These methods showed little to no disk reads because data has been cached in memory during the ETL process.

	Approach	ETL time	Exec. time	Epoch time	GPU util.	GPU RAM util.	CPU util.	DRAM util.	Tol. network	Per w. disk R/W
ImageNet	MA	2.8 hr	42.6 hr	4.3 hr	56.8%	32.5%	2.3 %	3.1%	0.9 TB	12 GB/ 2 GB
	UDAF	2.8 hr	48.5 hr	4.9 hr	49.9%	28.6%	2.2%	5.6%	0.8 TB	12 GB / 279 GB
	CTQ	2.8 hr	45.1 hr	4.5 hr	56.2%	32.2%	2.5%	1.9%	0.6 TB	12 GB / 152 GB
	DA-Cerebro	5.4 hr	23.0 hr	2.3 hr	70.5%	42.5%	2.8%	20.2%	0.6 TB	0.6 GB* / 0.3 GB
	Cerebro-Spark	4.4 hr	23.9 hr	2.4 hr	65.1%	36.5%	11.2%	17.4%	1.1 TB	0.2 GB* / 2 GB
	PyTDDP	4.4 hr	77.3 hr	7.7 hr	97.1%	13.1%	8.1%	14.7%	1900 TB	None* / 11 GB
	DA-PyTDDP	5.4 hr	77.5 hr	7.8 hr	96.8%	13.2%	8.2 %	21.1%	1900 TB	None* / 1 GB
Criteo	MA	8.6 hr	38.5 hr	7.7 hr	N/A	N/A	44.1%	2.3%	0.1 TB	1 GB / 2 GB
	UDAF	8.6 hr	62.0 hr	12.4 hr	N/A	N/A	27.1%	2.3%	0.1 TB	1 GB / 38 GB
	CTQ	8.6 hr	40.0 hr	8.0 hr	N/A	N/A	41.0%	1.9%	0.08 TB	1 GB / 22 GB
	DA-Cerebro	10.5 hr	21.5 hr	4.3 hr	N/A	N/A	37.4%	28.5%	0.07 TB	0.2 GB* / 0.3 GB
	Cerebro-Spark	8.3 hr	22.5 hr	4.5 hr	N/A	N/A	35.2%	28.5%	0.2 TB	0.2 GB* / 1 GB

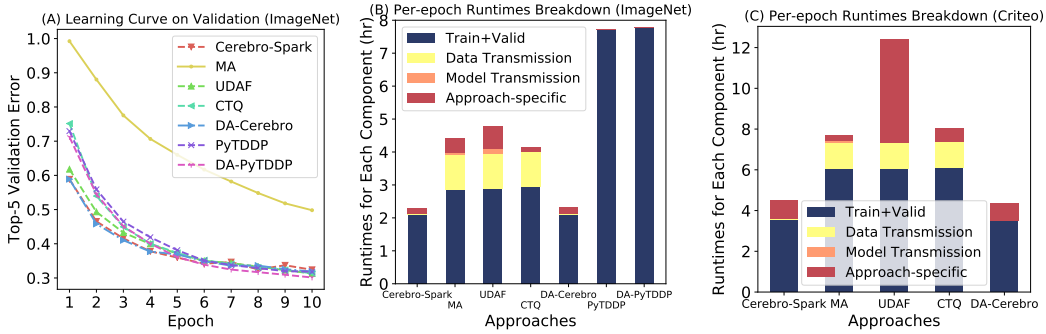

Figure 9: End-to-end tests results. (A): Convergence behavior on ImageNet. (B): Per-epoch breakdown of runtimes for each approach on ImageNet. (C): Per-epoch breakdown of runtimes for each approach on Criteo.

runtime. They both showed high GPU utilization only because they employ GPU for communication. They have less GPU memory consumption because how Pytorch differs from TensorFlow on memory management. They performed even slower on Criteo and were estimated to take over 16 days of runtime each, so we skipped these tests. DA-Cerebro, Cerebro-Spark, PytorchDDP, and DA-PytorchDDP showed higher DRAM usage because of caching. They also showed few disk reads due to preloading and caching during ETL; the writes are due to metadata management. Note disk R/W for all approaches are not significant, and none of them is bound by the disk IO speed.

Profiling and breakdowns. To further investigate the root cause of performance differences, we take both datasets and profile every approach by running several more breakdown tests and calculating each execution component's runtime. Excluding the ETL time, Figure 9(B) presents results for ImageNet, and Figure 9(C) presents results from Criteo tests. We record per-epoch machine time compositions for each worker and take the average among them. Hence, the summations of the runtime numbers are close to but may not be identical to the end-to-end runtimes in Table 6, which are determined by the slowest worker runtime instead of the mean runtime. We break down per-epoch runtimes into four different components:

(1) Train+Valid: time spent in the DL tools, including initialization and destruction of models, allocation and freeing of GPU memory, training and validation with GPU, etc. MA, UDAF, and CTQ are less efficient because these in-DBMS approaches invoke the DL tools through wrappers that cause extra overheads. For PytorchDDP and DA-PytorchDDP, since they overlap communication with computation, Train+Valid also includes time spent on model updating communications (we name these Model Transmission, introduced below). For this reason they showed very high Train+Valid time because they are bounded by networking.

(2) Data Transmission: time spent on transmitting data to the DL tool from storage. For the in-DBMS approaches, it also includes the data decompression time. This component is non-negligible for the 3 in-DBMS approaches due to data access overheads, while in the rest approaches, training data is cached in memory during ETL; this part costs little. Recall from Table 3 that Cerebro-Spark and PytorchDDP suffers a 2x storage blowup, while DA-based approaches do not.

(3) Model Transmission: time spent on transmitting serialized models/gradients between workers. For MA, Model Transmission means model collecting and broadcasting; it means model hopping for the rest MOP-based approaches. UDAF and MA are not so efficient on this end because they require database joins for redistributing models. We will further investigate this performance gap in Section 5.2.3. The two PytorchDDP approaches showed none on this front only because it is absorbed into Train+Valid.

(4) Approach-specific: for MA, it is the time spent on averaging model weights. For the rest, it means sub-optimal scheduling and/or

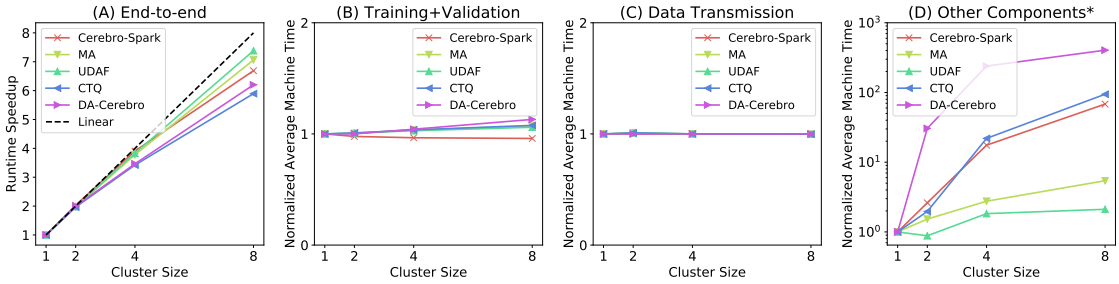

Figure 10: (A): End-to-end scalability plot, y-axis shows the speedups with respective to single-node runtime. (B - D): Per-epoch machine time for each component normalized against single-node. The machine time is averaged among all nodes. *Other Components: includes Model Transmission and Approach-specific as described in Section 5.1.

idling of some workers. Cerebro-Spark, CTQ, and DA-Cerebro use an asynchronous random scheduler and work better for heterogeneous workloads. As for UDAF, the performance is affected by its synchronous round-robin scheduler. We further discuss on these two schedulers in Section 5.2.2.

Comparing the Criteo tests to ImageNet tests, we notice two significant differences: (1). Model Transmission time drops for MA and UDAF. The MLP model used in Criteo tests is smaller than the CNNs used for ImageNet. (2). The UDAF approach suffers more from idling in Criteo because the workload is more heterogeneous due to highly disparate batch sizes.

Overall, the MA approach shows unfavorable convergence behavior and the fine-grained data parallel approaches (DA-PytorchDDP and PytorchDDP) are heavily bottlenecked. MOP-based approaches largely dominate these two parallelization models. As for MOP, the in-DBMS approaches suffer from various overheads and are, in general, less efficient than the DA-Cerebro approach and the Cerebro-Spark approach. However, recall that runtime efficiency is not the only criterion for such a system, as we showed earlier in Table 3. There exists a tradeoff space and perhaps no universal optima to the question.

5.2 Drill-down Experiments

5.2.1 Scalability (strong scaling).
In this test, we evaluate the strong scalability. We used clusters with 1, 2, 4, 8 workers with ImageNet. We use a workload of 8 homogenous configs (4 learning rates, 2 regularization values, and ResNet50 architecture) trained for one epoch. All runtimes exclude ETL. Figure 10(A) presents the results.

All approaches show close-to-linear scaling. To better understand these behaviors, we further drill down each runtime component and evaluate their scalability separately. For this purpose, we collect the average machine time spent on each component from all workers varying cluster size; we then report them against the single node time. Figures 10(B-D) summarize the results. Flat lines indicate that the component's machine time is constant regardless of cluster size, therefore perfectly scalable. An increasing curve means sub-linearity, and vice versa.

Figure 10(B) and Figure 10(C) show that the Training+Validation and Data Transmission component scale almost linearly for all approaches. The Model Transmission part is minuscule in the end-to-end time, thus we report it collectively with the Approach-specific components in Figure 10(D). For DA-Cerebro, CTQ, and Cerebro-Spark, workers may idle relatively more when the number of models approaches the number of workers. The random scheduler they use can yield sub-optimal scheduling under such circumstances. [89] Hence they all show sub-linear scalability, especially when cluster size grows from 4 to 8. On the other hand, UDAF adopts a round-robin scheduler to emulate MOP, which happens to be optimal for this specific homogenous workload; thus, it shows better scalability. MA utilizes all workers and shows no idle time, but the model averaging cost still rises a little when the cluster size grows.

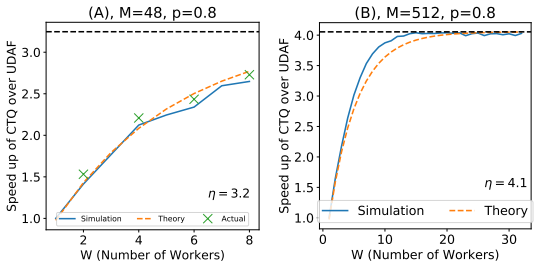

Figure 11: Heterogenous experiment. (A) Real experiments supplemented with simulation and theoretical results. $l_m/l_s = 8$ (B) Extreme scenario simulated. $l_m/l_s = 20$.

5.2.2 Async. MOP vs sync. MOP on Heterogeneous Workloads.
To verify Equation 1 and Equation 2 proposed in Section 4.1 and prove the benefit of async. MOP over sync. MOP for heterogeneous workloads, we conduct the following experiments. We have one sync. MOP approach: UDAF; and among the 3 async. MOP approaches we pick CTQ, as the main difference between UDAF and CTQ is only the synchronization model of scheduler. Following the analysis in Section 4.1, let $\mathbb{M}$ be drawn from a Bernoulli distribution: $Pr(l_x = l_s) = p, Pr(l_x = l_m) = 1 - p$.

We then test with real experiments. The fast model is MobileNetV2 with batch size 128, while the slow model is NASNetMobile with batch size 4. We down-sampled 6% of ImageNet, so that the experiments can finish in reasonable amount of time (same experiments on the full datasets are estimated to cost over two months). Sampling might alter the ratios between constant overheads and components that scale with dataset size. However, since the constant overheads are minuscule, the sampling proves not to affect our conclusion. In Figure 11(A) we see the actual runs fit nicely with our simulation and theory. Furthermore, Figure 11(B) shows one simulated extreme scenario with a large workload and 32 workers to demonstrate the theoretical upper bound of the speedup. We refer interested readers to Appendix for more simulations. Overall, these experiments verify that our theoretical bounds match with the actual runtime gaps. Meanwhile, we also show that the upper

bound of speedup is determined by η. This indicates that CTQ can be a more efficient choice than UDAF when working with highly heterogeneous workloads and/or hardware.

5.2.3 Effect of model size on UDAF and CTQ.

The size of models is typically orders of magnitude smaller than the size of training dataset. Thus, although model hopping time is proportional to model size, it is usually negligible in large-scale DL. However, this assumption may not hold for the UDAF approach because of the `JOIN` as explained in Section 4.1 Model hopping. We run a test to investigate model transmission cost with varying model sizes empirically. Our test shows that the CTQ approach imposes little to no bottleneck and is far less sensitive to the model size. However, the UDAF approach suffers more overheads on larger models. This confirms that the JOIN and storing models inside the DB can indeed cause some overheads, although this overhead is not too major (less than 10% in this case). The details of this test can be found in Appendix.

5.2.4 Experiments with Hyperopt Workloads.

In the end-to-end experiments we used a simple grid search workload. Now we evaluate the generality of workloads for the approaches. We use a model selection workload guided by Hyperopt (TPE algorithm) [26]. The parameter grid we use to sample model configs is as follows. Model: [ResNet18, ResNet34]; Learning rate: $[10^{-5}, \ldots, 10^{-1}]$; Weight decay: $[10^{-4}, 10^{-6}]$; Batch size: [16, …, 256]. We also include a comparison to Hyperopt-Spark [53], a standalone task parallel model selection system. We set the number of model configs to 32 and degree of parallelism to 8. Figure 12 plots the learning curves. CTQ has ~50% higher runtime than UDAF; This is because MOP's random scheduler has a decreased runtime performance when the degree of parallelism is close to the number of workers [89] and showed in Appendix. This issue can be largely mitigated by increasing the degree of parallelism. DA/Cerebro-Spark run similarly to Hyperopt-Spark, but the latter requires both data export and full data replication to each worker. Therefore it has a storage blowup of 9x, while Cerebro-Spark has 2x and DA has none in this case.

On GPU utilizations, the conclusion is consistent with those showed in Section 5.1. We have 32% (UDAF), 33% (CTQ), 44% (Cerebro-Spark), 44% (DA), and 45% (Hyperopt-Spark). The rest of the measurements are available in Appendix.

5.2.5 Implementation Difficulty.

Implementation difficulty is harder to measure quantitively. Following the discussion in Section 4.3, we now try to provide a more quantitive measurement in the form of lines of source code (LOC). The APPROACH (LOC) is as follows: UDAF (5866), CTQ (5939), DA-Cerebro (4230: 2764 for standalone Cerebro and 1466 for DA), and Cerebro-Spark (4338). Note these are counted for end-to-end implementation. UDAF and CTQ can largely share codebase: given UDAF, it requires only a few hundred lines to implement CTQ as well. Overall, DA and Cerebro-Spark take less code to implement than their counterparts UDAF and CTQ. However, this does not necessarily mean they are subjectively easier; as discussed in Section 4.3, the DA approach was much more time-consuming than the LOC number would otherwise suggest.

6 KEY TAKEAWAYS

For Cloud Vendors and Practitioners. We find that MOP is a good option for in-DBMS distributed DL model selection, compared to fine-grained data parallelism and task parallelism. It is possible to integrate MOP with existing DBMSs without modifying their internal code. Depending on governance/provenance requirements, we compared different approaches, ranging from fully in-DBMS with only UDAFs, to fully out-of-DBMS with data exports and external DL systems. Overall, the in-DBMS approaches of UDAF and CTQ may provide better ease of governance, while DA and Cerebro-Spark offer better efficiency. Furthermore, CTQ provides asynchronous scheduling over UDAF, which leads to better efficiency for heterogeneous workloads but sacrifices some governance. DA and Cerebro-Spark offer the same efficiency level. DA, being more difficult to implement and less portable, can avoid data exports that Cerebro-Spark requires. Finally, these MOP-based approaches are still subject to the inherent limits of Cerebro [89]; for instance, it only supports ML models trainable with SGD, and tree methods are currently not supported. We refer the reader to the above paper for details. All our code has been made available at [6]. The MADlib team is still actively studying and optimizing these approaches.

For Researchers. We realized that the current data warehouse architecture lacks optimizations for distributed DL. One of the major problems is the data format: data warehouses store data in proprietary pagefile formats. Figure 9 showed that accessing data via the DBMS can bring severe bottlenecks for DL workloads, which require frequent and rapid table scans. It is largely an open research question, but some proposals appear on the horizon, such as Lakehouse [128]. In this paper, we devised Direct Access with caching, and we hope it can serve as a candidate solution to the above problems. However, DA is coupled with the proprietary DBMS data formats. Standard pagefile formats such as Parquet would simplify the implementation and increase the portability drastically. Similarly, in-memory formats like Apache Arrow would vastly simplify the data transmission process between different runtimes and may also bring performance boosts.

7 RELATED WORK

ML in Data Systems. There is a long line of work on ML in data systems, both RDBMSs and dataflow systems. The general approach is to implement ML algorithms via UDFs or other APIs exposed by the data system. Apache MADlib [45, 51] is one of the most mature such tools, available on PostgreSQL and Greenplum. The UDAF approach we studied for integrating MOP is already a part of MADlib. Vertica-ML [44], Oracle Machine Learning [15], Microsoft SQL Server ML Services [13], and Google BigQuery [8] are other prominent examples of in-RDBMS ML tools. [97] brings ML to column stores. MLlib [83] and MLlib* [133] use Spark's APIs to implement various ML algorithms. Mahout [22] is a distributed ML system on top of dataflow systems. Increasingly, more data system builders want to integrate with DL via wrappers that invoke popular DL tools: Horovod on Spark [10], TensorFrames [16], and PS2 [132] are examples. More generally, the DBMS and cloud industry believe DBMSs will continue to play a key role in enterprise ML [18].

Some works also expand DBMS support for ML. Raven [58] deeply integrates ML runtimes into a DBMS. UDA-GIST [72] expands support for algorithms that are both data-parallel and state-parallel. [81] adds linear algebra support to RDBMS. [127] proposes a "tensor-relational" algebra towards declarative ML. TensorDB [60]

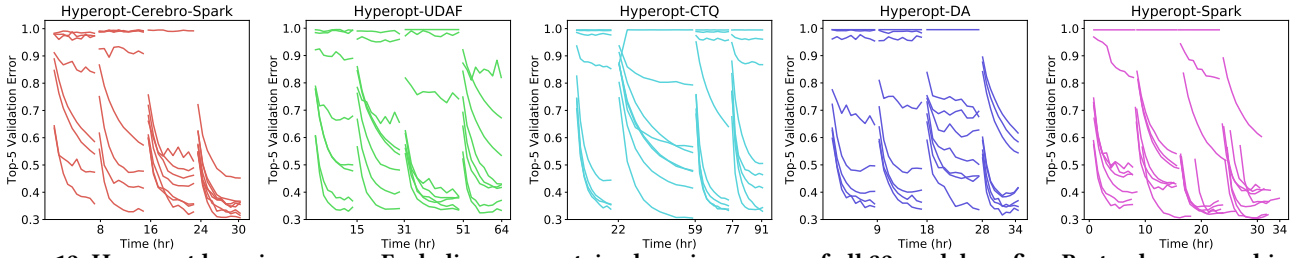

Figure 12: Hyperopt learning curves. Each diagram contains learning curves of all 32 model configs. Best val. errors achieved by each approach are within the margin: 0.31 (Cerebro-Spark), 0.33 (UDAF), 0.31 (CTQ), 0.33 (DA), 0.31 (Hyperopt-Spark).

is a system for in-DBMS tensor decomposition. [59] focuses on in-DBMS sparse tensors for ML. DB4ML [56] expedites iterative ML algorithms via asynchrony. [46] discusses declarative model weights distribution/aggregation for data-parallel ML. [55] adds better support for recursion to RDBMS for distributed ML. MLearn [107] is a declarative language for in-DBMS ML. AIDA [41] provides an abstraction for in-DBMS data analytics; it uses DBMS for relational operations and embeds Python for linear algebra.

All of the above works are complementary to ours. To the best of our knowledge, our paper is the first to study system design alternatives and tradeoffs for enabling DL workloads on DBMSs. Specifically, we focus on bringing a recently published hybrid parallel execution approach for DL model selection, MOP, to the traditionally bulk-synchronous parallel world of DBMSs.

Custom ML Systems. There is also a long line of work on custom systems for ML training/model selection. FlexPS [52] and Lapse [102] are both optimizations to Parameter Server [74]. Horovod [108] brings in decentralized communication to boost runtime efficiency. Vizier [47] and Rafiki [121] are systems for task-parallel model selection; Ray [77, 86] was initially designed for reinforcement learning but recently also supports task-parallel model selection. Singa [92, 120] and SystemML [28–30] are end-to-end platforms for ML that supports various distributed training. Visus [105] and Ease.ml [100, 101] are examples of AutoML systems that manage the whole ML lifecycle, including both data management and model selection. Crossbow [62] and Ako [123] are systems for better resource scheduling and utilization for ML. [40] handles collaborative working environments for ML development. Litz [96] focuses on the elasticity of distributed ML.

All these works are also complementary to ours because they study standalone ML/DL execution, not integration with data systems. While some of them may be faster than in-DBMS ML tools, as we explained in depth in this paper, ML practitioners, especially in enterprises, grapple with a more complex Pareto frontier beyond just runtimes. Our paper lays out these tradeoffs in bringing DL workloads closer to DB-resident data. That said, the CTQ approach we studied was in part inspired by the pervasive use of task parallelism in such custom ML systems, including in Cerebro as we explained earlier. More generally, we believe these historically distinct work lines–custom ML systems and ML on data systems–can learn a lot from each other.

Data Management for ML. More generally, data management for ML is a hot and pressing research topic [27, 67, 95, 106]. Such works aim to optimize or automate data management tasks in ML workflows to reduce user burden. Data Programming [99] and Snorkel [98] focus on ML training data creation through weak supervision and generative models. DeepDive [129] is a system for knowledge base construction. ModelDB [114, 115], TFX [24], Mlog [75], and MLFlow [85] all add data management and model management support for ML. ARDA [35] uses DBMS for data augmentation and feature selection tasks. Activeclean [64], boostclean [63] and [79] focus on data cleaning and debugging for ML. Vamsa [90] supports data lineage tracking for Python ML scripts. [50] proposes the concept of model materialization and reuse to speed up ML training.

All these works are also largely orthogonal to ours, since our focus is specifically on tradeoffs of in-DBMS execution of DL model selection, not auxiliary data/model management capabilities. Lessons from our work can be easily integrated with these other tools to enhance end-to-end support for ML applications for DB users.

Data Access and Pipeline Optimizations for ML. There is much prior work on optimizing ML+data processing pipelines. Lara [70], Alpine-Meadow [110], and KeystoneML [111] all allow the user to define pipelines with their APIs and perform pipeline-level optimizations. Helix [125] injects intelligent caching and reuse between training iterations to reduce redundant work. [42] proposes linear algebra that could work upon compressed data, thus saving decompression time. [71] introduces a tuple-oriented compression scheme for matrix and mini-batch SGD computations directly on compressed data.

The above works are largely orthogonal to our paper, since our goal is *not* to devise novel optimization schemes or systems but rather to analytically and empirically study the tradeoffs of alternative approaches to bring DL workloads to DB-resident data. That said, the DA approach we studied was in part inspired by such prior work on ML operating more directly on the raw stored data. It is interesting future work to integrate more such optimizations into systems that bring DL closer to DB-resident data.

REFERENCES

[1] Cerebro Documentation. https://adalabucsd.github.io/cerebro-system/.
[2] First hand knowledge from the authors.
[3] Create, Train, and Deploy Machine Learning Models in Amazon Redshift Using SQL with Amazon Redshift ML, Accessed December 13, 2020. https://aws.amazon.com/blogs/big-data/create-train-and-deploy-machine-learning-models-in-amazon-redshift-using-sql-with-amazon-redshift-ml/.
[4] The CREATE MODEL Statement for Deep Neural Network (DNN) Models, Accessed December 13, 2020. https://cloud.google.com/bigquery-ml/docs/reference/standard-sql/bigqueryml-syntax-create-dnn-models.
[5] Script for Tensorflow Model Averaging, Accessed January 31, 2020. https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/utils/avg_checkpoints.py.
[6] Code Release of This Work, Accessed November 19, 2020. https://github.com/makemebitter/cerebro-ds.
[7] About Greenplum Query Processing, Accessed October 31, 2020. https://gpdb.docs.pivotal.io/560/admin_guide/query/topics/parallel-proc.html.

[8] Google BigQuery ML, Accessed October 31, 2020. https://cloud.google.com/bigquery-ml/docs.

[9] Google BigQuery ML TensorFlow integration, Accessed October 31, 2020. https://cloud.google.com/bigquery-ml/docs/making-predictions-with-imported-tensorflow-models.

[10] Horovod on Spark, Accessed October 31, 2020. https://github.com/horovod/horovod/blob/master/docs/spark.rst.

[11] MADlib Deep Learning, Accessed October 31, 2020. https://madlib.apache.org/docs/latest/group__grp__dl.html.

[12] MADlib Model Selection, Accessed October 31, 2020. https://madlib.apache.org/docs/latest/group__grp__mdl.html.

[13] Microsoft SQL Server Machine Learning Services, Accessed October 31, 2020. https://docs.microsoft.com/en-us/sql/machine-learning/sql-server-machine-learning-services?view=sql-server-2017.

[14] Oracle Data Mining, Accessed October 31, 2020. https://www.oracle.com/database/technologies/advanced-analytics/odm.html.

[15] Oracle Machine Learning, Accessed October 31, 2020. https://www.oracle.com/data-science/machine-learning.html.

[16] TensorFrames, Accessed October 31, 2020. https://github.com/databricks/tensorframes.

[17] TOAST Tables in Postgres, Accessed October 31, 2020. https://wiki.postgresql.org/wiki/TOAST.

[18] A. Agrawal, R. Chatterjee, C. Curino, A. Floratou, N. Godwal, M. Interlandi, A. Jindal, K. Karanasos, S. Krishnan, B. Kroth, J. Leeka, K. Park, H. Patel, O. Poppe, F. Psallidas, R. Ramakrishnan, A. Roy, K. Saur, R. Sen, M. Weimer, T. Wright, and Y. Zhu. Cloudy with high chance of DBMS: a 10-year prediction for Enterprise-Grade ML. In *CIDR*. www.cidrdb.org, 2020.

[19] D. AI. AI Infrastructure for Everyone, Now Open Source, Accessed October 31, 2020. https://determined.ai/blog/ai-infrastructure-for-everyone/.

[20] R. Akita, A. Yoshihara, T. Matsubara, and K. Uehara. Deep learning for stock prediction using numerical and textual information. In *ICIS*, pages 1–6. IEEE Computer Society, 2016.

[21] Amazon. RedShift Query Planning and Execution Workflow, Accessed November 19, 2020. https://docs.aws.amazon.com/redshift/latest/dg/c-query-planning.html.

[22] R. Anil, G. Çapan, I. Drost-Fromm, T. Dunning, E. Friedman, T. Grant, S. Quinn, P. Ranjan, S. Schelter, and Ö. Yilmazel. Apache Mahout: Machine Learning on Distributed Dataflow Systems. *J. Mach. Learn. Res.*, 21:127:1–127:6, 2020.

[23] M. P. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. B. Zdonik. The Object-Oriented Database System Manifesto. In *DOOD*, pages 223–240. North-Holland/Elsevier Science Publishers, 1989.

[24] D. Baylor, E. Breck, H. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *KDD*, pages 1387–1395. ACM, 2017.

[25] Y. Bengio. Rmsprop and equilibrated adaptive learning rates for nonconvex optimization. *corr abs/1502.04390*, 2015.

[26] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. 2013.

[27] L. Berti-Équille, A. Bonifati, and T. Milo. Machine Learning to Data Management: A Round Trip. In *ICDE*, pages 1735–1738. IEEE Computer Society, 2018.

[28] M. Boehm, M. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. Reiss, P. Sen, A. Surve, and S. Tatikonda. SystemML: Declarative Machine Learning on Spark. *Proc. VLDB Endow.*, 9(13):1425–1436, 2016.

[29] M. Boehm, B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow.*, 11(12):1755–1768, Aug. 2018.

[30] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. R. Burdick, and S. Vaithyanathan. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow.*, 7(7):553–564, 2014.

[31] X. Bouthillier and G. Varoquaux. Survey of Machine-Learning Experimental Methods at NeurIPS2019 and ICLR2020. Research report, Inria Saclay Ile de France, Jan. 2020.

[32] S. Chakraborty, R. Tomsett, R. Raghavendra, D. Harborne, M. Alzantot, F. Cerutti, M. B. Srivastava, A. D. Preece, S. Julier, R. M. Rao, T. D. Kelley, D. Braines, M. Sensoy, C. J. Willis, and P. Gurram. Interpretability of Deep Learning Models: A Survey of Results. In *SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI*, pages 1–6. IEEE, 2017.

[33] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[34] Y. Cheng, C. Qin, and F. Rusu. GLADE: big data analytics made easy. In *SIGMOD Conference*, pages 697–700. ACM, 2012.

[35] N. Chepurko, R. Marcus, E. Zgraggen, R. C. Fernandez, T. Kraska, and D. Karger. ARDA: Automatic Relational Data Augmentation for Machine Learning. *Proc. VLDB Endow.*, 13(9):1373–1387, 2020.

[36] E. Commission. GDPR, Accessed October 31, 2020. https://ec.europa.eu/info/law/law-topic/data-protection/eu-data-protection-rules_en.

[37] CriteoLabs. Kaggle Contest Dataset Is Now Available for Academic Use!, Accessed January 31, 2020. https://ailab.criteo.com/category/dataset.

[38] Databricks. Introducing Apache Spark 2.4, Accessed October 31, 2020. https://databricks.com/blog/2018/11/08/introducing-apache-spark-2-4.html.

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A Large-scale Hierarchical Image Database. In *CVPR*, pages 248–255. IEEE, 2009.

[40] B. Derakhshan, A. R. Mahdiraji, Z. Abedjan, T. Rabl, and V. Markl. Optimizing Machine Learning Workloads in Collaborative Environments. In *SIGMOD Conference*, pages 1701–1716. ACM, 2020.

[41] J. V. D'silva, F. De Moor, and B. Kemme. AIDA - Abstraction for Advanced In-Database Analytics. *Proc. VLDB Endow.*, 11(11):1400–1413, 2018.

[42] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed Linear Algebra for Large-Scale Machine Learning. *Proc. VLDB Endow.*, 9(12):960–971, 2016.

[43] Facebook. Introducing FBLearner Flow: Facebook's AI backbone, Accessed January 31, 2020. https://engineering.fb.com/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/.

[44] A. Fard, A. Le, G. Larionov, W. Dhillon, and C. Bear. Vertica-ML: Distributed Machine Learning in Vertica Database. In *SIGMOD Conference*, pages 755–768. ACM, 2020.

[45] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD Conference*, pages 325–336. ACM, 2012.

[46] Z. J. Gao, N. Pansare, and C. M. Jermaine. Declarative Parameterizations of User-Defined Functions for Large-Scale Machine Learning and Optimization. *IEEE Trans. Knowl. Data Eng.*, 31(11):2079–2092, 2019.

[47] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google vizier: A Service for Black-box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495. ACM, 2017.

[48] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT press, 2016.

[49] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In *IJCAI*, pages 1725–1731. ijcai.org, 2017.

[50] S. Hasani, S. Thirumuruganathan, A. Asudeh, N. Koudas, and G. Das. Efficient Construction of Approximate Ad-Hoc ML models Through Materialization and Reuse. *Proc. VLDB Endow.*, 11(11):1468–1481, 2018.

[51] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, 2012.

[52] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, and J. Cheng. FlexPS: Flexible Parallelism Control in Parameter Server Architecture. *Proc. VLDB Endow.*, 11(5):566–579, 2018.

[53] hyperopt. Scaling out search with Apache Spark, Accessed January 31, 2020. http://hyperopt.github.io/hyperopt/scaleout/spark/.

[54] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando, and K. Kavukcuoglu. Population Based Training of Neural Networks. *arXiv preprint arXiv:1711.09846*, 2017.

[55] D. Jankov, S. Luo, B. Yuan, Z. Cai, J. Zou, C. Jermaine, and Z. J. Gao. Declarative recursive computation on an RDBMS: or, why you should use a database for distributed machine learning. *SIGMOD Rec.*, 49(1):43–50, 2020.

[56] M. Jasny, T. Ziegler, T. Kraska, U. Röhm, and C. Binnig. DB4ML - An In-Memory Database Kernel with Machine Learning Support. In *SIGMOD Conference*, pages 159–173. ACM, 2020.

[57] Kaggle. State of Data Science and Machine Learning 2019, Accessed October 31, 2020. https://www.kaggle.com/kaggle-survey-2019.

[58] K. Karanasos, M. Interlandi, F. Psallidas, R. Sen, K. Park, I. Popivanov, D. Xin, S. Nakandala, S. Krishnan, M. Weimer, Y. Yu, R. Ramakrishnan, and C. Curino. Extending Relational Query Processing with ML Inference. In *CIDR*. www.cidrdb.org, 2020.

[59] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. In-Database Learning with Sparse Tensors. In *PODS*, pages 325–340. ACM, 2018.

[60] M. Kim and K. S. Candan. Efficient Static and Dynamic In-Database Tensor Decompositions on Chunk-Based Array Stores. In *CIKM*, pages 969–978. ACM, 2014.

[61] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[62] A. Koliousis, P. Watcharapichat, M. Weidlich, L. Mai, P. Costa, and P. Pietzuch. Crossbow: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers. *Proc. VLDB Endow.*, 12(11):1399–1412, 2019.

[63] S. Krishnan, M. J. Franklin, K. Goldberg, and E. Wu. BoostClean: Automated Error Detection and Repair for Machine Learning. *CoRR*, abs/1711.01299, 2017.

[64] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg. ActiveClean: Interactive Data Cleaning For Statistical Modeling. *Proc. VLDB Endow.*, 9(12):948–959, 2016.

[65] Kubeflow. Kubeflow, Accessed November 26, 2020. https://www.kubeflow.org/.

[66] A. Kumar. ML/AI Systems and Applications: Is the SIGMOD/VLDB community losing relevance?, Accessed November 19, 2020. https://wp.sigmod.org/?p=2454.

[67] A. Kumar, M. Boehm, and J. Yang. Data Management in Machine Learning: Challenges, Techniques, and Systems. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1717–1722. ACM, 2017.

[68] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model Selection Management Systems: the Next Frontier of Advanced Analytics. *SIGMOD Record*, 2016.

[69] A. Kumar, S. Nakandala, Y. Zhang, S. Li, A. Gemawat, and K. Nagrecha. Cerebro: A Layered Data Platform for Scalable Deep Learning. In *CIDR*. www.cidrdb.org, 2021.

[70] A. Kunft, A. Katsifodimos, S. Schelter, S. Breß, T. Rabl, and V. Markl. An Intermediate Representation for Optimizing Machine Learning Pipelines. *Proc. VLDB Endow.*, 12(11):1553–1567, 2019.

[71] F. Li, L. Chen, Y. Zeng, A. Kumar, X. Wu, J. F. Naughton, and J. M. Patel. Tuple-oriented Compression for Large-scale Mini-batch Stochastic Gradient Descent. In *SIGMOD Conference*, pages 1517–1534. ACM, 2019.

[72] K. Li, D. Z. Wang, A. Dobra, and C. Dudley. UDA-GIST: An In-database Framework to Unify Data-Parallel and State-Parallel Analytics. *Proc. VLDB Endow.*, 8(5):557–568, 2015.

[73] L. Li, K. G. Jamieson, A. Rostamizadeh, E. Gonina, J. Ben-tzur, M. Hardt, B. Recht, and A. Talwalkar. A System for Massively Parallel Hyperparameter Tuning. In *MLSys*. mlsys.org, 2020.

[74] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*, 2014.

[75] X. Li, B. Cui, Y. Chen, W. Wu, and C. Zhang. MLog: Towards Declarative In-Database Machine Learning. *Proc. VLDB Endow.*, 10(12):1933–1936, 2017.

[76] J. Lian, X. Zhou, F. Zhang, Z. Chen, X. Xie, and G. Sun. xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems. In *KDD*, pages 1754–1763. ACM, 2018.

[77] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[78] Z. Liu, P. Luo, S. Qiu, X. Wang, and X. Tang. DeepFashion: Powering Robust Clothes Recognition and Retrieval with Rich Annotations. In *CVPR*, pages 1096–1104. IEEE Computer Society, 2016.

[79] R. Lourenço, J. Freire, and D. E. Shasha. Debugging Machine Learning Pipelines. *CoRR*, abs/2002.04640, 2020.

[80] J. Lu, C. Lin, J. Wang, and C. Li. Synergy of Database Techniques and Machine Learning Models for String Similarity Search and Join. In *CIKM*, pages 2975–2976. ACM, 2019.

[81] S. Luo, Z. J. Gao, M. N. Gubanov, L. L. Perez, and C. M. Jermaine. Scalable Linear Algebra on a Relational Database System. In *ICDE*, pages 523–534. IEEE Computer Society, 2017.

[82] MADlib. Kaggle Survey 2020, Accessed March 13, 2021. https://www.kaggle.com/kaggle-survey-2020.

[83] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.*, 17:34:1–34:7, 2016.

[84] Microsoft. Azure SQL Query Processing Architecture Guide, Accessed November 19, 2020. https://docs.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide?view=sql-server-ver15#distributed-query-architecture.

[85] MLflow. MLflow, Accessed November 26, 2020. https://mlflow.org/.

[86] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*, 2018.

[87] S. Nakandala and A. Kumar. Vista: Optimized System for Declarative Feature Transfer from Deep CNNs at Scale. In *SIGMOD Conference*, pages 1685–1700. ACM, 2020.

[88] S. Nakandala, Y. Zhang, and A. Kumar. 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*, pages 1–4, 2019.

[89] S. Nakandala, Y. Zhang, and A. Kumar. Cerebro: A Data System for Optimized Deep Learning Model Selection. *Proc. VLDB Endow.*, 13(11):2159–2173, 2020.

[90] M. H. Namaki, A. Floratou, F. Psallidas, S. Krishnan, A. Agrawal, Y. Wu, Y. Zhu, and M. Weimer. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *KDD*, pages 1542–1551. ACM, 2020.

[91] S. of California Department of Justice. CCPA, Accessed October 31, 2020. https://oag.ca.gov/privacy/ccpa.

[92] B. C. Ooi, K. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K. H. Tung, Y. Wang, Z. Xie, M. Zhang, and K. Zheng. SINGA: A Distributed Deep Learning Platform. In *ACM Multimedia*, pages 685–688. ACM, 2015.

[93] C. Ordonez. Integrating K-Means Clustering with a Relational DBMS Using SQL. *IEEE Trans. Knowl. Data Eng.*, 18(2):188–201, 2006.

[94] V. Oria, M. T. Özsu, P. Iglinski, S. Lin, and B. B. Yao. DISIMA: A Distributed and Interoperable Image Database System. In *SIGMOD Conference*, page 600. ACM, 2000.

[95] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich. Data Management Challenges in Production Machine Learning. In *SIGMOD Conference*, pages 1723–1726. ACM, 2017.

[96] A. Qiao, A. Aghayev, W. Yu, H. Chen, Q. Ho, G. A. Gibson, and E. P. Xing. Litz: Elastic Framework for High-Performance Distributed Machine Learning. In *USENIX Annual Technical Conference*, pages 631–644. USENIX Association, 2018.

[97] M. Raasveldt, P. Holanda, H. Mühleisen, and S. Manegold. Deep Integration of Machine Learning Into Column Stores. In *EDBT*, pages 473–476. OpenProceedings.org, 2018.

[98] A. Ratner, S. H. Bach, H. R. Ehrenberg, J. A. Fries, S. Wu, and C. Ré. Snorkel: Rapid Training Data Creation with Weak Supervision. *Proc. VLDB Endow.*, 11(3):269–282, 2017.

[99] A. J. Ratner, C. D. Sa, S. Wu, D. Selsam, and C. Ré. Data Programming: Creating Large Training Sets, Quickly. In *NIPS*, pages 3567–3575, 2016.

[100] C. Renggli, F. A. Hubis, B. Karlas, K. Schawinski, W. Wu, and C. Zhang. Ease.ml/ci and Ease.ml/meter in Action: Towards Data Management for Statistical Generalization. *Proc. VLDB Endow.*, 12(12):1962–1965, 2019.

[101] C. Renggli, B. Karlas, B. Ding, F. Liu, K. Schawinski, W. Wu, and C. Zhang. Continuous Integration of Machine Learning Models with ease.ml/ci: Towards a Rigorous Yet Practical Treatment. In *MLSys*. mlsys.org, 2019.

[102] A. Renz-Wieland, R. Gemulla, S. Zeuch, and V. Markl. Dynamic Parameter Allocation in Parameter Servers. *Proc. VLDB Endow.*, 13(11):1877–1890, 2020.

[103] R. Ricci, E. Eide, and CloudLabTeam. Introducing Cloudlab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[104] M. Rocklin. Dask: Parallel Computation with Blocked Algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, 2015.

[105] A. S. R. Santos, S. Castelo, C. Felix, J. P. Ono, B. Yu, S. R. Hong, C. T. Silva, E. Bertini, and J. Freire. Visus: An Interactive System for Automatic Machine Learning Model Building and Curation. In *HILDA@SIGMOD*, pages 6:1–6:7. ACM, 2019.

[106] S. Schelter, F. Bießmann, T. Januschowski, D. Salinas, S. Seufert, and G. Szarvas. On Challenges in Machine Learning Model Management. *IEEE Data Eng. Bull.*, 41(4):5–15, 2018.

[107] M. E. Schüle, M. Bungeroth, A. Kemper, S. Günnemann, and T. Neumann. MLearn: A Declarative Machine Learning Language for Database Systems. In *DEEM@SIGMOD*, pages 7:1–7:4. ACM, 2019.

[108] A. Sergeev and M. D. Balso. Horovod: Fast and Easy Distributed Deep Learning in TF. *arXiv preprint arXiv:1802.05799*, 2018.

[109] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: from Theory to Algorithms*. Cambridge university press, 2014.

[110] Z. Shang, E. Zgraggen, B. Buratti, F. Kossmann, P. Eichmann, Y. Chung, C. Binnig, E. Upfal, and T. Kraska. Democratizing Data Science through Interactive Curation of ML Pipelines. In *SIGMOD Conference*, pages 1171–1188. ACM, 2019.

[111] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *ICDE*, pages 535–546. IEEE Computer Society, 2017.

[112] H. Su and H. Chen. Experiments on Parallel Training of Deep Neural Network using Model Averaging. *CoRR*, abs/1507.01239, 2015.

[113] K. Tsuda, K. Yamamoto, M. Hirakawa, M. Tanaka, and T. Ichikawa. MORE: An Object-Oriented Data Model with a Facility for Changing Object Structures. *IEEE Trans. Knowl. Data Eng.*, 3(4):444–460, 1991.

[114] M. Vartak and S. Madden. MODELDB: Opportunities and Challenges in Managing Machine Learning Models. *IEEE Data Eng. Bull.*, 41(4):16–25, 2018.

[115] M. Vartak, H. Subramanyam, W. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. ModelDB: A System for Machine Learning Model Management. In *HILDA@SIGMOD*, page 14. ACM, 2016.

[116] VMware Tanzu/Pivotal. gpfdist, Accessed November 19, 2020. https://gpdb.docs.pivotal.io/510/utility_guide/admin_utilities/gpfdist.html.

[117] D. Wang, P. Cui, and W. Zhu. Structural Deep Network Embedding. In *KDD*, pages 1225–1234. ACM, 2016.

[118] H. Wang, N. Wang, and D. Yeung. Collaborative Deep Learning for Recommender Systems. In *KDD*, pages 1235–1244. ACM, 2015.

[119] R. Wang, B. Fu, G. Fu, and M. Wang. Deep & Cross Network for Ad Click Predictions. In *ADKDD@KDD*, pages 12:1–12:7. ACM, 2017.

[120] W. Wang, G. Chen, T. T. A. Dinh, J. Gao, B. C. Ooi, K. Tan, and S. Wang. SINGA: Putting Deep Learning in the Hands of Multimedia Users. In *ACM Multimedia*, pages 25–34. ACM, 2015.

[121] W. Wang, J. Gao, M. Zhang, S. Wang, G. Chen, T. K. Ng, B. C. Ooi, J. Shao, and M. Reyad. Rafiki: Machine Learning as an Analytics Service System. *Proc. VLDB Endow.*, 12(2):128–140, 2018.

[122] W. Wang, X. Yang, B. C. Ooi, D. Zhang, and Y. Zhuang. Effective deep learning-based multi-modal retrieval. *VLDB J.*, 25(1):79–101, 2016.

[123] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 84–97, 2016.
[124] J. Weston, F. Ratle, and R. Collobert. Deep learning via semi-supervised embedding. In *ICML*, volume 307 of *ACM International Conference Proceeding Series*, pages 1168–1175. ACM, 2008.
[125] D. Xin, S. Macke, L. Ma, J. Liu, S. Song, and A. G. Parameswaran. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *Proc. VLDB Endow.*, 12(4):446–460, 2018.
[126] A. Yoshitaka and T. Ichikawa. A Survey on Content-Based Retrieval for Multimedia Databases. *IEEE Trans. Knowl. Data Eng.*, 11(1):81–93, 1999.
[127] B. Yuan, D. Jankov, J. Zou, Y. Tang, D. Bourgeois, and C. Jermaine. Tensor Relational Algebra for Machine Learning System Design. *CoRR*, abs/2009.00524, 2020.
[128] M. Zaharia, A. Ghodsi, R. Xin, and M. Armbrust. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR*. www.cidrdb.org, 2021.
[129] C. Zhang, J. Shin, C. Ré, M. J. Cafarella, and F. Niu. Extracting Databases from Dark Data with DeepDive. In *SIGMOD Conference*, pages 847–859. ACM, 2016.
[130] J. Zhang, C. D. Sa, I. Mitliagkas, and C. Ré. Parallel SGD: When does averaging help? *CoRR*, abs/1606.07365, 2016.
[131] Q. Zhang and S. Zhu. Visual Interpretability for Deep Learning: A Survey. *Frontiers Inf. Technol. Electron. Eng.*, 19(1):27–39, 2018.
[132] Z. Zhang, B. Cui, Y. Shao, L. Yu, J. Jiang, and X. Miao. PS2: Parameter Server on Spark. In *SIGMOD Conference*, pages 376–388. ACM, 2019.
[133] Z. Zhang, J. Jiang, W. Wu, C. Zhang, L. Yu, and B. Cui. MLlib*: Fast Training of GLMs Using Spark MLlib. In *ICDE*, pages 1778–1789. IEEE, 2019.
[134] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li. Parallelized Stochastic Gradient Descent. In *NIPS*, pages 2595–2603. Curran Associates, Inc., 2010.

A SCENARIOS THAT COULD AFFECT SCHEDULER PERFORMANCE

Various scenarios could affect MOP schedulers' performance. One such case is heterogeneous workload, where the sync. and async. schedulers may yield schedules that deviate in makespan. One such scenario is shown in Figure 13.

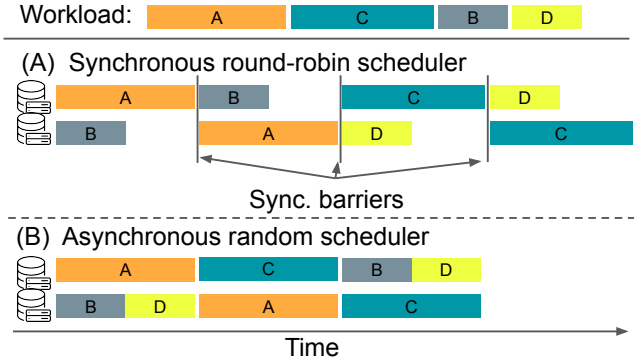

Figure 13: Gantt chart for possible schedules generated by (A) Synchronous round-robin scheduler and (B) Asynchronous random scheduler. The two workers are homogenous, but the workload, containing four models, is heterogenous.

The other notable example is when the number of model configs approximates the available degree of parallelism as shown in Figure 14; in this scenario, the async. random scheduler could suffer from straggler issues while the sync. scheduler is free of such problems.

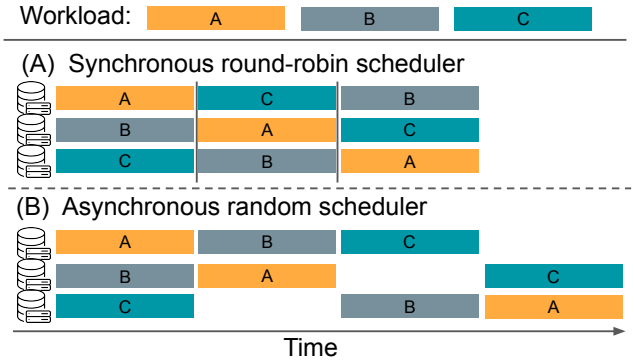

Figure 14: Gantt chart for possible schedules generated by (A) Synchronous round-robin scheduler and (B) Asynchronous random scheduler.

B PROOFS TO PROPOSITIONS

Proof to Proposition 1. For sync. MOP, there will be in total M sub-epoch batches, and each batch's runtime will be dominated by the longest running model within this batch, therefore we have in expectation:

$$T_u = \sum_{i}^{M} \max(\mathbb{L}_i), \tag{3}$$

where $\mathbb{L}_i = \text{rand}(\mathbb{L}, W)$. $\text{rand}(\mathbb{X}, N)$ means randomly sampling N elements from set $\mathbb{X}$. Assuming that $\max(\mathbb{L}_i) \sim l_s$, meaning every sub-epoch of UDAF run is dominated by l_s, we obtain:

$$T_u = p^W l_s M + (1 - p^W) l_m M. \tag{4}$$

On the other hand with async. MOP, assuming M is large enough ($M \gg W$) and the scheduler was capable of load-balancing, we have:

$$T_c = \frac{|\mathbb{L}|}{W} W = W \frac{M\bar{l}}{W} = M\bar{l}. \tag{5}$$

Immediately we have the speedup of async. MOP over sync. MOP:

$$\frac{T_u}{T_c} = \frac{p^W l_s M + (1 - p^W) l_m M}{M\bar{l}} = p^W \frac{l_s}{\bar{l}} + (1 - p^W) \frac{l_m}{\bar{l}}. \tag{6}$$

This indicates the speedup (weak scaling) of async. MOP over sync. MOP is related to the number of workers W and the skewness (represented by $\frac{l_s}{\bar{l}}$ and $\frac{l_m}{\bar{l}}$). Interestingly, note that this speedup is independent of the number of model configs.

Proof to Proposition 2. Asymptotically, when W goes up, we can see $\frac{T_u}{T_c} \to \frac{l_m}{\bar{l}}$. Define $\eta =: \frac{l_m}{\bar{l}}$. $\eta > 1$ as long as the underlying distribution of $\mathbb{L}$ is right-tailed. This means async. MOP will always be faster than sync. MOP under such circumstance, given sufficient number of workers.

Furthermore, since $\bar{l} = \frac{p l_s M + (1-p) l_m M}{M} = p l_s + (1-p) l_m$, if we expand η, there is:

$$\eta = \frac{l_m}{\bar{l}} = \frac{l_m}{p l_s + (1-p) l_m}. \tag{7}$$

When $p \to 1$ we obtain

$$\eta \to \frac{l_m}{l_s}, \tag{8}$$

which is unbounded and can potentially go to a very high number under extreme circumstances. This indicates when there are only a few outlier models that are time-consuming, the speed-up of async. MOP over sync. MOP is determined by the relative runtime difference of the outlier models and common models.

C EFFECT OF MODEL SIZE ON UDAF AND CTQ

The size of models is typically orders of magnitude smaller than the size of the training dataset. Thus, although model hopping time is proportional to model size, it is usually negligible in large-scale DL. However, this assumption may not hold for the UDAF approach because of the `JOIN` as explained in Section 4.1 Model hopping. We run the following test to investigate model transmission cost with varying model sizes empirically.

We choose 3 hyperparameter tuning workloads on ImageNet. Each workload features 8 model configs with one single model architecture and a fixed batch size of 32. Model architectures are: MobileNet (52 MB), ResNet50 (294 MB), and ResNet152 (693 MB). Model size is reported as the on-disk serialized size. We run each workload for 3 epochs and take the average to get per epoch and per worker machine time breakdown.

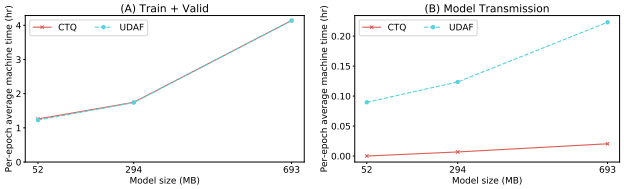

Figure 15: Per-epoch runtime for model size test. (A): Train+Valid time. (B): Model Transmission time.

Figure 15 presents the results. We profile and focus on two components: Train+Valid and Model Transmission. Figure 15(A) shows no difference in terms of Train+Valid, which is to be expected. Figure 15(B) shows that the CTQ approach imposes little to no bottleneck and is far less sensitive to the model size. However, the UDAF approach suffers more overheads on larger models. This confirms that the JOIN and storing models inside the DB can indeed cause some overheads, although this overhead is not too major (less than 10% in this case).

D SIMULATED EXTREME SCENARIOS OF ASYNC. MOP VS SYNC. MOP ON HETEROGENEOUS WORKLOADS

To supplement the experiments shown in Section 5.2.2, we now add more simulated tests to show both theoretical and simulated performance gain of Async. MOP over Sync. MOP on heterogeneous workloads. We assume $l_m/l_s = 20$. We evaluate the speedups of CTQ over UDAF for different numbers of workers W (up to 32). Figure 16 presents the results. Comparing Figure 16(A) with (B), we see that when M is not large enough, the theoretical upper bound is too loose as it assumes a sufficient number of models for the MOP scheduler to load-balance. Comparing Figure 16(B) with (C), we see when p goes up, which means higher heterogeneity and higher right-skewness, CTQ offers drastically higher speedups over UDAF. Furthermore, as Figure 16(E) shows, CTQ's performance gain can continue to increase with even higher skewness η.

E HYPEROPT EXPERIMENT RESOURCE UTILIZATIONS

Table 7 summarizes the detailed resource utilization figures for experiments shown in Section 5.2.4. Hyperopt-Spark is task parallel. Therefore it has little network usage compared to other approaches. It requires full data replication; data is not partitioned, and each worker keeps an entire copy of the whole dataset. Because the full dataset does not fit in the DRAM of a single node, it does not cache the data and resorts to frequent disk reads, resulting in orders of magnitude higher disk reads. Despite the MOP-based methods have higher network usage and Hyperopt-Spark has higher disk reads, these are still minor overheads, and none of the approaches is bounded by network nor disk R/W.

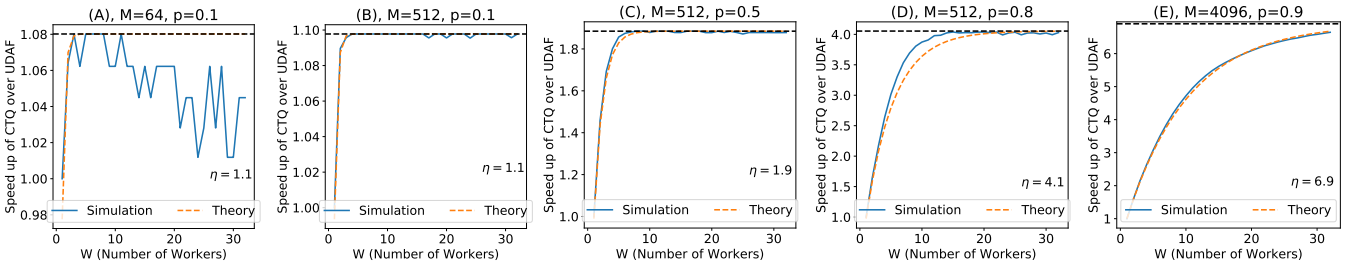

Figure 16: Runtimes of heterogeneous workloads. (A-E) represent different workload configs. Both theoretical bounds and simulated runtime gaps are shown. The upper bounds of speedup η are calculated for each workload. NB: Note the different ranges of the Y axes across plots.

Table 7: Resource utilizations from Hyperopt experiments shown in Section 5.2.4.

Approach	GPU util.	GPU RAM util.	CPU util.	DRAM util.	Total Network	Per Worker Disk R/W
UDAF	32.5%	16.2%	2.4%	7.2%	600 GB	12 GB / 173 GB
CTQ	33.2%	16.5%	2.5%	1.5%	600 GB	12 GB / 92 GB
DA-Cerebro	45.3%	24.1%	2.0%	20.2%	500 GB	0.7 GB / 6.7 GB
Cerebro-Spark	43.6%	24.2%	13.4%	15.8%	1000 GB	0.3 GB / 2 GB
Hyperopt-Spark	44.6%	24.0%	4.2%	3.6%	20 GB	3000 GB / 4 GB