Learning to Accelerate: Tuning Data Transfer Parameters

Benedikt Didrich
TU Berlin
Berlin, Germany
b.didrich@campus.tu-berlin.de

Haralampos Gavriilidis
BIFOLD & TU Berlin
Berlin, Germany
gavriilidis@tu-berlin.de

Vasilis Gkolemis
Athena Research Center
Athens, Greece
vgkolemis@athenarc.gr

Matthias Boehm
BIFOLD & TU Berlin
Berlin, Germany
matthias.boehm@tu-berlin.de

Volker Markl
BIFOLD, TU Berlin & DFKI
Berlin, Germany
volker.markl@tu-berlin.de

ABSTRACT

Efficient data transfer is crucial for modern distributed systems, but performance depends heavily on well-tuned transfer parameters. Optimizing these parameters is challenging due to the large search space and dynamic system conditions. Manual tuning is impractical, and existing heuristic methods lack sufficient adaptability. Suboptimal configurations can significantly degrade performance in data-intensive applications, highlighting the need for tuning strategies that adapt to their environment. In this paper, we introduce Adapt as a data-driven approach for automatically tuning data transfer parameters. Our framework employs an ensemble cost model with dynamic weights that combine prior knowledge and online observations, as well as an efficient two-phase exploration strategy for finding high-performing configurations. Our experiments show that Adapt outperforms both the existing heuristic optimizer and standard black-box baselines, achieving up to 34% higher throughput in 42% less time. Adapt also robustly adapts to changing environments, demonstrating the effectiveness of ML-based tuning in real-world data transfer scenarios.

VLDB Workshop Reference Format:
Benedikt Didrich, Haralampos Gavriilidis, Vasilis Gkolemis, Matthias Boehm, and Volker Markl. Learning to Accelerate: Tuning Data Transfer Parameters. VLDB 2025 Workshop: AIDB.

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/polydbms/xdbc-adapt.

1 INTRODUCTION

Modern data-driven applications depend on fast and scalable data movement. For example, ETL workflows, machine learning (ML) pipelines and federated query plans require moving data between OLTP and OLAP databases, data lakes, and data science runtimes [10, 18]. These transfers are executed across heterogeneous infrastructures and systems under different environmental conditions, e.g., fluctuating network bandwidth or competing CPU workloads. To address these challenges, we recently introduced XDBC [9, 11], a modular and scalable framework for efficient data transfer. XDBC exposes several tuning knobs such as parallelization strategies, memory configurations, and compressors, which can be configured for specific source and target systems as well as environments.

A Need for Data Transfer Optimization. Good configurations are important for efficient data transfers. Poor configurations can lead to slow runtime performance, underutilized CPU and network resources, or increased cloud costs due to excessive bandwidth usage. For example, employing compression may pay off in scenarios where sufficient CPU resources are available on source and target system and network bandwidth is limited. However, under sudden CPU spikes, the compression operation may degrade the overall throughput, in which case compression may be disabled. Overall, optimizing data transfer configurations and adjusting to environment changes is crucial for achieving both high throughput and resource efficiency across systems and environments.

Data Transfer Tuning Challenges. Optimizing data transfer is challenging due to the large and complex configuration space. In particular, XDBC exposes 12 parameters related to parallelism, buffer sizes, compression algorithms and serialization formats. Those parameters interact in non-trivial ways and their impact depends heavily on source and target systems, data, and environment characteristics. Transfers may occur between arbitrary systems, e.g., a cloud-hosted DBMS and a local Python pandas runtime. Additionally, topologies are dynamic, i.e., network bandwidth and CPU availability may change during execution. Moreover, evaluating a single configuration requires executing a full transfer, limiting the number of configurations that can be explored. As a result, an effective tuner must explore the configuration space efficiently, i.e., balance exploration and exploitation, while also adapting to changing system and network environments.

A Case for Adaptive Tuning. Most existing ODBC/JDBC-based data transfer frameworks rely on manual tuning and expose only a small set of configuration knobs, i.e., batch (or fetch) size and read parallelism, which limits their ability for fully utilizing available resources and adapting to changing environments. Our existing XDBC framework comprises multiple tuning knobs and a practical heuristic optimizer. The existing optimizer relies on a small number of profiling runs and greedily assigns worker threads based on an analytical cost model, and selects compression, memory, and format choices based on the characteristics of participating systems. However, this heuristic optimizer makes simplifying assumptions about the environment and workload, and lacks the ability to generalize across data transfers and to adapt to changing

Proceedings of the VLDB Endowment. ISSN 2150-8097.

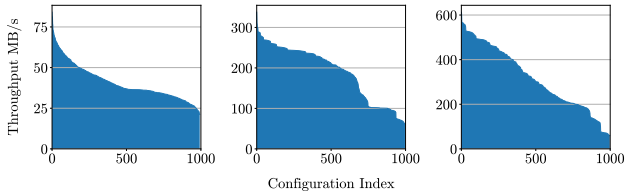


Figure 1: Distribution of Throughput of different configurations for Environments (2,2,50), (8,8,150) and (16,16,1000).

workloads dynamically. While data-driven auto-tuning methods have shown promising results in other domains, such as DBMS configuration tuning or index selection, data transfer is a special case that requires an tailor-made solution.

Contributions. To address the limitations of manual tuning and the heuristic optimizer, we introduce Adapt, an adaptive data transfer parameter tuner for the XDBC framework. Adapt learns from prior transfers and runtime profiling for quickly finding good (high throughput) data transfer configurations. We propose a dynamic ensemble of XGBoost regression models that uses weights that combine prior environment knowledge and live feedback, enabling fast performance prediction across different transfer environments. Unlike the previous heuristic optimizer, Adapt generalizes across source-target system pairs and adapts to changing environment characteristics such as CPU load or network fluctuations. In detail, our technical contributions are:

- **Adaptive Tuning Framework:** A modular tuner that supports cold-start optimization, transfer learning, and online adaptation across dynamic data transfer environments.
- **Ensemble Cost Model:** A dynamic ensemble of XGBoost models that combine prior environment knowledge and live feedback for fast performance prediction.
- **Exploration Strategy:** An efficient two-phase exploration of the parameter search space using batch prediction and dedicated search space pruning techniques.
- **Experiments:** We evaluate Adapt against other optimization algorithms, including the XDBC heuristic optimizer, and show improved transfer throughput and adaptability.

The rest of this paper is structured as follows: In Section 2, we motivate the need for data transfer optimization and introduce our XDBC framework. Then, in Section 3, we introduce our approach for a data and machine learning (ML) driven parameter tuner, and describe our cost model and exploration strategy. In Section 4, we share and discuss our experimental evaluation, which shows that Adapt achieves higher throughput and better robustness than baselines, especially in dynamic environments. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 MOTIVATION AND BACKGROUND

In this section, we first motivate the need for adaptive parameter tuning in the context of data transfer across heterogeneous systems and environments. Subsequently, we provide an overview of our *XDBC* framework for modular and scalable data transfer, and motivate the need for adaptive, data-driven tuning.

2.1 Motivation

Data transfer is a common task in data-driven applications, but different use cases require different configurations depending on system properties and runtime environments. Transfers may occur between cloud databases, local runtimes, data lakes, and analytics engines, often across changing and heterogeneous infrastructure. We illustrate this challenge with a running example.

Example 1 (Feature Engineering Pipeline). *A data scientist periodically extracts training data from a cloud-hosted PostgreSQL DBMS into a Python pandas runtime for feature engineering and model training. Depending on the execution setup, the pandas environment may run on the user's local laptop with limited resources and slower wide-area-network (WAN) connection, or on a cloud-hosted notebook with better hardware and network throughput.*

Impact of Configurations. Even for this simple workflow, data transfer performance can vary drastically depending on the environment. Large batch sizes, and highly-parallelized operations may benefit data transfers in cloud-hosted environments, but become counterproductive in resource-constrained local setups. Therefore, selecting effective transfer parameters is crucial for runtime performance, but non-trivial given the variety of execution contexts. Figure 1 shows the distribution of observed throughput values for 1,000 randomly sampled parameter configurations in three distinct transfer environments. Only a small subset of configurations reach high throughput values, with the true optima being up to 20% higher then the maximum values shown in the plots, highlighting the importance of selecting an effective configuration instead of relying on default configurations. To support tuning for such scenarios, the XDBC framework exposes fine-grained control over key data transfer parameters, including compression settings, parallelism, buffer sizes, and serialization formats. XDBC further includes a heuristic optimizer that uses a small number of profiling runs and an analytical cost model to find configurations tailored to a given source-target pair as well as data and system characteristics.

Example 2 (Recurring Transfers in Volatile Topologies). *The same pipeline may be scheduled to run daily on cloud infrastructure with elastic compute and shared network links. Depending on external factors, such as concurrent jobs, resource scaling events, or background traffic, the available CPU or bandwidth may fluctuate substantially between and during runs.*

Limitations of Heuristic Optimization. In such volatile environments, static configurations quickly become suboptimal. The heuristic optimizer in XDBC provides a good starting point, but this optimizer relies on simplifying assumptions and cannot adapt to dynamic changes in real time. As a result, transfer performance may degrade under changing conditions, leading to slower pipeline execution or inefficient resource usage. The above examples motivate the need for a system that not only exposes rich tuning knobs, but also adapts its configuration based on prior knowledge and live system feedback. These limitations and goals directly motivate the design of our data-driven tuner Adapt.

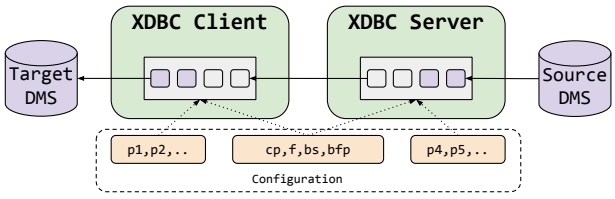


Figure 2: XDBC's Configurable Component Architecture.

2.2 The XDBC Data Transfer Framework

XDBC is a holistic framework for fast and scalable data transfer across heterogeneous systems, such as from relational databases (e.g., PostgreSQL) to pandas, or from Parquet to CSV files. This framework addresses the limitations of generic JDBC/ODBC-like connectors—which offer limited tuning capabilities—by achieving performance competitive with specialized solutions such as ConnectorX [27], while also providing good generality.

XDBC Overview. As shown in Figure 2, XDBC models the data transfer pipeline as a modular pipeline of decoupled logical components, including *read*, *serialize*, *compress*, *send*, *receive*, *decompress*, *deserialize*, and *write*. Each component has different physical implementations and can be individually tuned. The framework operates in a client-server model, using a ring-buffer mechanism for memory management, where components communicate through in-memory queues and pass fixed-size buffers through the pipeline stages. Key strengths of XDBC are its flexibility and configurability. For each pipeline component, we can tune the degree of parallelism independently to match the available compute resources on the source and target systems. The size of the buffers in flight and the total buffer pool on both client and server sides can also be controlled to fit memory constraints. In addition, the system supports multiple intermediate transfer formats and compression algorithms, which can be selected to optimize for CPU, I/O, or network bandwidth. Table 1 summarizes the main configuration parameters of XDBC. The combination of parallelism, memory management, format, and compression options results in a large and high-dimensional search space with complex parameter interactions. This complexity makes manual tuning difficult and motivates automated optimization.

The Heuristic Optimizer. To improve out-of-the-box performance, XDBC includes a rule-based heuristic optimizer that automatically selects data transfer configurations. The optimizer controls the following four key parameter groups:

- **Parallelism:** Configures the number of worker threads assigned to each component in the pipeline.
- **Compression:** Configures the choice of compression algorithm (or none)—and thus, also the implied decompression algorithm—to balance CPU cost and data reduction.
- **Memory Management:** Configures the size of in-flight buffers and the size of the buffer pool (i.e., number of buffers in the pool) on both client and server sides.
- **Component Skipping:** Applies rules for bypassing optional pipeline components depending on system compatibility and efficiency (e.g., serialization formats).

Table 1: Configuration Search Space C.

Name	Type	Domain
Compression Library	categorical	{nocomp, zstd, lz4, lzo, snappy}
Intermediate Format	categorical	{1, 2}
Buffer Size	discrete	{64, 256, 512, 1024}
Client Bufferpool Size	integer	$[1, \ldots, 8]$
Server Bufferpool Size	integer	$[1, \ldots, 8]$
$*$ Parallelism	integer	$[1, \ldots, 16]$

The heuristic optimizer begins with a short sampling phase to profile the throughput of individual pipeline stages, or utilizes pre-materialized offline profiling information. Based on these profiles, the optimizer constructs a simple analytical cost model and iteratively assigns worker threads to the slowest stages to mitigate bottlenecks. If the network is identified as a limiting factor, the optimizer enables compression and rebalances the parallelism configuration. Format selection and component-skipping decisions are made based on characteristics of the source and target systems. Finally, the buffer sizes are selected based on available memory and CPU cache sizes, conditioned on the configured parallelism.

Towards Data-driven Tuning. While this heuristic is capable of producing better-than-default configurations quickly, it has limitations due to its static and rule-based design. First, the embedded rules rely on expert knowledge and are difficult to generalize across new systems or workloads. Second, the optimizer assumes a fixed execution environment and depends on an initial profiling phase, which can be expensive or infeasible in some deployments. Third, the pre-defined scaling assumptions used by the cost model can break down in volatile environments with fluctuating resource availability. Most importantly, the heuristic optimizer does neither learn from past transfers nor adjust its configuration over time. Once selected, the configuration remains fixed, making it unsuitable for transfers that experience dynamic runtime conditions such as CPU contention or variable network throughput.

3 ADAPTIVE PARAMETER TUNING

XDBC [9, 11] exposes a rich set of configurations that control parallelism, memory management, and compression. Finding high-throughput configurations is difficult though due to the exponential search space of configurations and sensitivity to system conditions such as CPU, memory, and network bandwidth. This challenge represents a broader class of tuning problems in distributed systems, where static heuristics often fall short under dynamic workloads. In this section, we introduce Adapt, a data-driven tuner designed to automatically optimize XDBC parameters using machine learning, transfer learning, and runtime feedback. Our tuning framework is designed for fast convergence, strong generalization across environments, and adaptability to real-time changes.

3.1 Adapt Overview

Adapt is designed to efficiently—with minimal tuning overhead and few transfer attempts—identify effective parameter configurations with high throughput. We leverage a set of predictive models that are trained offline to generalize across a wide range of environments. Adapt is fully integrate in the XDBC system to guide

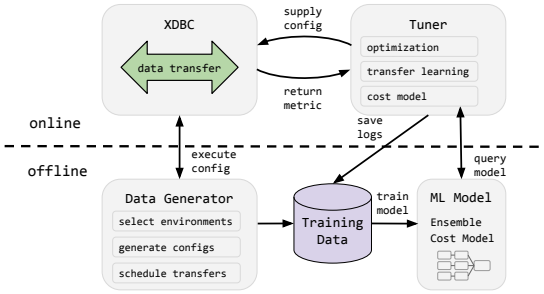
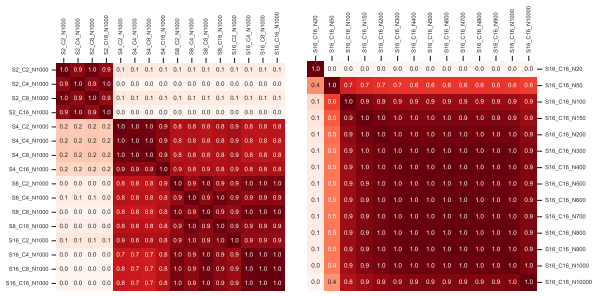


Figure 3: ADAPT Tuning Components.



(a) Scaling Compute **(b) Scaling Network**

Figure 4: Similarities Between Environments.

configuration selection during cross-environment transfers. Figure 3 shows the overall tuning process. When XDBC initiates a transfer, ADAPT selects a promising configuration based on historical data and recently observed performance. After execution, the transfer outcome is logged and fed back to ADAPT in order to update its internal predictive models. This feedback loop enables increasingly accurate tuning over time[1]. To realize this functionality, ADAPT operates in three conceptual phases, each supported by a corresponding system component:

- **Training Phase** (offline): Implemented by the *Data Generator*, this phase generates synthetic configuration- environment performance data. We cluster similar environments and train a separate regression model for each cluster.
- **Ensemble Prediction Phase** (offline): Implemented by the *Cost Model*, this phase constructs a weighted ensemble over the trained models. We assign weights based on prior environment knowledge and historical predictive accuracy.
- **Tuning Phase** (online): Handled by the *Tuner*, this phase uses the ensemble to suggest configurations and updates model weights based on observed transfer performance.

Algorithm 1 summarizes the full ADAPT workflow across these phases, including model training, ensemble prediction, and online adaptation. In the following, we first formalize the problem setting (Section 3.2), then describe our approach to synthetic data generation and model training (Section 3.3), followed by our ensemble prediction method (Section 3.4), and finally present the online adaptation mechanism used during tuning (Section 3.5).

3.2 Problem Definition

Our problem consists of selecting the optimal system configuration for a data transfer in a specific computing environment.

Optimization Objective. Let C denote the configuration space, and let E represent the data transfer environment. Our optimization objective is to find the optimal configuration $C^* \in C$ that maximizes the data transfer throughput $T(C, E)$:

$$C^* = \arg\max_{C \in C} T(C, E) \tag{1}$$

Notation. The transfer environment E is defined as a tuple (S_E, C_E, N_E), where S_E and C_E denote the number of available CPU cores on the server and client, respectively, and N_E denotes the network bandwidth in megabits per second. The configuration space C consists of 12 tunable parameters, including numerical (e.g., parallelism, buffer size) and categorical (e.g., compression) parameters. A detailed description of these parameters is provided in Table 1. In practice, the target environment E is often unknown at prediction time. Accordingly, our method must address the optimization problem in (1) without always assuming knowledge of E.

[1]We also support a lightweight online mechanism for intra-transfer reconfiguration but this fine-grained adaptation is limited to a subset of reconfigurable XDBC parameters.

3.3 Training Phase

To train accurate cost models for evaluating transfer configurations, we need a large and diverse set of labeled performance data. However, collecting such data through real-world transfers is expensive and time-consuming. Accordingly, it is impractical to exhaustively run transfers for every combination of configuration parameters and deployment environments. To overcome this challenge, we generate training data in a controlled setting of simulated docker environments but real performance measurements.

Environment Simulation. We emulate deployment environments using Docker-based container virtualization. Our goal is to create synthetic topologies with controlled variations in compute and network characteristics. We use the `docker-tc` utility to emulate a range of bandwidth conditions by applying traffic control rules inside containers. In addition, we use CPU quotas to constrain the compute resources on both the server and client sides. This setup allows us to run XDBC transfers under realistic resource limitations, while maintaining reproducibility and low overhead.

Environment Sampling. Evaluating the full cross-product of potential (S_E, C_E, N_E) values (introduced in Section 3.2) would be prohibitively expensive. Instead, we identify a compact set of representative environments that induce distinct performance behaviors. To this end, we run a fixed set of configurations across candidate environments, ranking them by throughput, and measuring similarity using Spearman's rank correlation. Figures 4a and 4b show the similarity patterns across compute and network dimensions, respectively. Based on these results and insights, we select a minimal yet diverse set of values that yield environments with different optimal parameter configurations (Table 2).

Configuration Sampling. The XDBC framework exposes a rich set of tunable parameters, including per-component parallelism, compression algorithms, intermediate formats, and buffer

Table 2: Sample Environments.

Server CPU		Client CPU		Network Speed
{2, 8, 16}	×	{2, 8, 16}	×	{50, 150, 1000, 100000}

management. We defined value ranges for each parameter, such as setting parallelism between 1 and 16, and constraining buffer pool sizes as a multiple of thread counts. The resulting configuration space comprises 6.87×10^{11} distinct configurations (Table 1). To sample from this space efficiently, we used Latin Hypercube Sampling (LHS), which ensures uniform coverage across parameter values while minimizing the number of samples. LHS allows us to generate well-distributed points that cover the full parameter space, including boundary values and rare combinations. We generate such a minimal set of configurations (i.e., 400) for each of the representative environments. For each configuration, we evaluate an XDBC data transfer in the corresponding environment, and record the resulting throughput. These data form the labeled dataset for training our ensemble cost model, enabling both environment-specific predictions and transfer learning across workloads.

Environment Clustering and Model Training. Using the collected throughput data for each configuration-environment pair, we represent every environment by a performance signature vector. To enable consistent comparison, we apply the same normalization used later in the ensemble weighting mechanism, transforming all signatures into a shared representation. We compute normalized pairwise Euclidean distances between transformed environment signatures, yielding a similarity matrix. This similarity matrix serves as input to hierarchical agglomerative clustering, where we apply a distance threshold of 0.15 to form environment clusters. Let $\{\mathcal{K}_1, \ldots, \mathcal{K}_K\}$ denote the resulting set of K clusters. Each cluster $\mathcal{K}_k$ groups environments with similar responses to configuration changes. We also compute a signature per cluster as a weighted average of its assigned environment signatures. For each cluster $\mathcal{K}_k$, we train a separate predictive model $m_k : C \to \mathbb{R}$ that estimates the transfer throughput $T(C, E)$ for any configuration $C \in \mathcal{C}$, assuming the environment E belongs to that cluster. We use XGBoost [6] as the predictive model due to its robustness as well as ability to handle high-dimensional, mixed-type feature spaces, and missing values. As shown in Lines 1–8 of Algorithm 1, the training process involves clustering the environments, computing signatures, and fitting one model per cluster. This approach reduces model complexity, lowers compute costs, and improves the convergence of our ensemble weighting mechanism, as the clusters are more homogeneous and less noisy than individual environments.

Trained Model. The result of the training phase is a set of environment clusters, each associated with a labeled training dataset and a corresponding XGBoost regression model. These models specialize in predicting transfer performance for configurations within their respective environment groups. Together, these models form the basis of our ensemble prediction model, which combines their outputs to guide configuration selection in new environments.

3.4 Ensemble Prediction Phase

After the training phase, we have a set of clusters $\{\mathcal{K}_1, \ldots, \mathcal{K}_K\}$, each associated with a predictive model $m_k : C \to \mathbb{R}$ trained as described in Section 3.3. These models form the basis for our ensemble prediction mechanism.

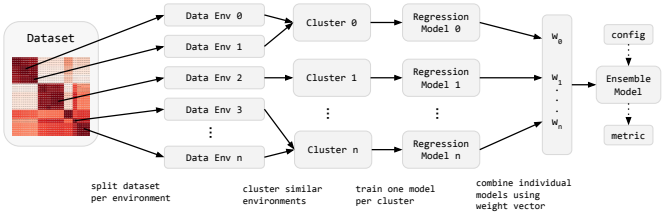


Figure 5: Cost model architecture used in ADAPT. Each base model is trained for a cluster of environments and dynamically weighted during inference.

Ensemble Model. The ensemble prediction phase construct a single cost model $m : C \to \mathbb{R}$ that will be later used to find the optimal configuration C^*. To this end, we combine the predictions of all base models $m_1, \ldots, m_K$ into an ensemble model. Each base model contributes in a weighted manner to the final prediction:

$$m(C) = \sum_{k=1}^{K} w_k m_k(C), \quad \text{where } |w| = 1 \tag{2}$$

with w_k denoting the weight assigned to the k-th model, and $m_k(c)$ its throughput prediction. Figure 5 illustrates this ensemble architecture, including clustering, per-cluster model training, and inference-time weight combination.

Ensemble Weights. The weights $w = (w_1, \ldots, w_K)$ reflect confidence in each model's ability to accurately predict throughput in the current (possibly unknown) environment. We compute them by combining two complementary criteria. The first is a similarity-based vector, denoted $w^E = (w_1^E, \ldots, w_K^E)$, which captures how closely the current environment signature E' (if available) resembles each cluster. This is computed using a distance metric between E' and the centroid of each cluster. If E' is not known, a uniform vector is used instead. The second is a performance-based score, $w^u = (w_1^u, \ldots, w_K^u)$, derived from historical observations of each model performance. Model with lower prediction errors on historical data receive a higher weight. The final ensemble weights are then obtained by combining w^e and w^u via a linear combination:

$$w_k = \alpha \cdot w_k^E + (1 - \alpha) \cdot w_k^u \tag{3}$$

where $\alpha \in [0, 1]$ is a balancing parameter that evolves over time, gradually shifting emphasis from prior similarity to empirical accuracy as more observations become available. As shown in Lines 10–21 of Algorithm 1, we compute similarity weights based on environment distance, combine them with update weights using a time-dependent interpolation factor $\alpha(t)$, and form a weighted ensemble prediction. Initially, when no or little prediction history is available, the ensemble relies more on the similarity-based weights w^E. As more data transfers are observed and model performance can be assessed, the influence gradually shifts toward the accuracy-based weights w^u. This interpolation is controlled by a time-dependent parameter $\alpha(t) \in [0, 1]$, where t denotes the number of data transfers. The function $\alpha(t)$ uses an exponential decay, ensuring that the ensemble becomes increasingly guided by empirical evidence.

Algorithm 1 Dynamic Regression Environment Adaptive Model

1: *// Initialize:*
2: **Input:** Labeled data $(C, E, T(C, E))$, decay factor λ
3: Initialize: $U \leftarrow \mathbf{1}/K$, $models \leftarrow \{\}$, $signatures \leftarrow \{\}$
4: Cluster environments $\{E\}$ into $\mathcal{K}_1, \ldots, \mathcal{K}_K$
5: **for** each cluster $\mathcal{K}_k$ **do**
6: Train model m_k on cluster data
7: Store cluster centroid signature s_k
8: **end for**
9: *// Prediction Phase:*
10: **Input:** Configuration C, Target environment E'
11: Normalize E' to obtain signature s'
12: **for** each model m_k **do**
13: Predict $p_k \leftarrow m_k(C)$
14: Compute similarity $w_k^E \leftarrow 1/d(s', s_k)$
15: **end for**
16: Normalize w^E
17: Compute $\alpha(t) \leftarrow 1/(1 + \beta \cdot t)$ {β controls weight shift rate}
18: Combine weights: $w_k \leftarrow \alpha(t) \cdot w_k^E + (1 - \alpha(t)) \cdot w_k^u$
19: Normalize w
20: Predict: $m(C) \leftarrow \sum_k w_k \cdot p_k$
21: **return** $m(C)$
22: *// Update Phase:*
23: **Input:** Executed config C^*, observed throughput $T(C^*, E')$
24: **for** each model m_k **do**
25: Compute error: $e_k \leftarrow (m_k(C^*) - T(C^*, E'))^2$
26: $w_k^{\text{new}} \leftarrow 1/e_k$
27: **end for**
28: Normalize w^{new}
29: Update: $w^u \leftarrow \lambda \cdot w^u + (1 - \lambda) \cdot w^{\text{new}}$

3.5 Tuning Phase

In order to find the optimal (or at least good) configurations for a new data transfer efficiently, we use the ensemble cost model

$$m(C) = \sum_{k=1}^{K} w_k m_k(C)$$

defined in Section 3.4 to guide exploration of the configuration space C. Since executing data transfers is costly, we aim to minimize the number of real-world trials required to find an effective configuration. Rather than relying on iterative black-box optimizers such as Bayesian Optimization—which incur overhead per evaluation—we adopt a batch-oriented exploration strategy that enables fast, parallelizable decision-making.

Two-Phase Sampling Strategy. To balance efficiency and prediction quality, we employ a two-phase exploration approach based on Latin Hypercube Sampling (LHS) [14]. In the first phase, we generate an initial candidate set $C_{\text{init}} \subset C$ using LHS, ensuring uniform coverage of the full configuration space. We evaluate each candidate $C \in C_{\text{init}}$ using the ensemble model $m(C)$, and retain the top 20% of configurations based on predicted throughput.

Search Space Pruning. We then construct a reduced search space $C_{\text{sub}} \subset C$, which bounds the parameter ranges observed among the top configurations from before. This pruned space focuses the search on high-potential regions while discarding areas unlikely to yield strong performance. Subsequently, we generate a second set of candidates C_{refined} via LHS within C_{sub}, and again score each configuration using the ensemble model. The optimal configuration is selected as

$$C^* = \arg \max_{C \in C_{\text{refined}}} m(C)$$

for execution on the target environment.

Feedback and Adaptation. Once the selected configuration C^* is executed, we observe the actual throughput $T(C^*, E')$. This measurement is used to update the model's internal accuracy-based weight vector w^u, as described in Section 3.4. Specifically, models that yield lower prediction error for C^* are rewarded with increased future weights. As more feedback becomes available, the weighting mechanism gradually shifts from relying on prior similarity (w^E) to observed performance (w^u), according to the dynamic mixing parameter $\alpha(t)$ (Algorithm 1, Lines 23–29).

Summary. Each tuning iteration (i.e., the tuning of a new data transfer) begins by generating candidate configurations, scoring them using the ensemble cost model, and selecting a configuration for execution. The observed throughput is then incorporated as feedback, allowing the model to adapt and improve over time. By combining structure-aware sampling, efficient batch prediction, and adaptive weight adjustment, this tuning loop supports effective configuration selection with low overhead.

4 EXPERIMENTAL EVALUATION

Our experiments study the effectiveness and efficiency of our tuning approach in different data transfer scenarios. We first describe our experimental setup, and then present our end-to-end evaluation comparing our approach to baseline algorithms, and finally explore specific aspects through micro-benchmarks.

4.1 Experimental Setup

In the following, we describe the used hard- and software environments, baselines, metrics, as well as the used transfer environments.

Hardware/Software. We perform our transfers on compute nodes with 2 Intel(R) Xeon(R) Silver 4216 CPU @ 2.10 GHz each having 16 cores and 32 threads, 512 GB of main memory and a 2 TB SSD disk drive. The tuning algorithms run on a machine with an Intel(R) Core(TM) i5-12600K @ 3.70 GHz with 10 cores and 16 threads, 32 GB of main memory and a 2 TB SSD disk drive. We run XDBC on Ubuntu v20.04.2 LTS with Docker v27.1.1 and use Python v3.8 on Windows 11 v23H2 to run our data generation and tuning algorithms. For the implementation of Bayesian Optimization and RGPE we use openbox v0.8.4 [13], and Syne Tune v0.13.0 [19] for the implementation of Quantile and Random Search. We use scikit-learn v1.3.2 [17] for the RandomForestRegressor and Hierarchical Clustering, scipy v1.10.1 [26] for Latin Hypercube Sampling and xgboost v2.1.1 [6] for the implementation of XGBRegressor.

Baselines. As baselines, we use the existing Heuristic Optimizer, Random Search and Bayesian Optimization. Random search is a simple but effective approach for finding optimal parameter configurations, which makes it a common baseline for comparing

Table 3: Evaluation Environments (S_E, C_E, N_E).

Server \ Client	Edge	Fog	Cloud
Edge	(2, 2, 50)	(2, 8, 50)	(2, 16, 50)
Fog	(8, 2, 150)	(8, 8, 150)	(8, 16, 150)
Cloud	(16, 2, 1000)	(16, 8, 1000)	(16, 16, 1000)

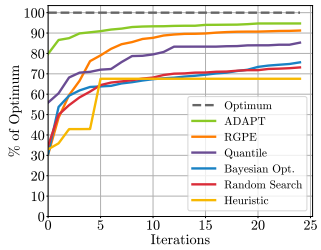


Figure 6: Best Throughput.

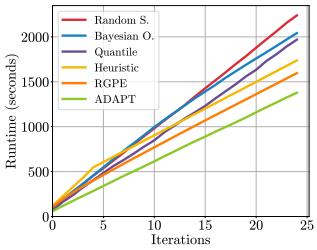


Figure 7: Accumulated Time.

different optimization algorithms [5]. We use Random Search to show the relative improvement of our approach compared to randomly picking parameter configurations. Bayesian Optimization is regarded as the go-to algorithm for tuning expensive black-box functions such as complex data systems [22, 24]. Including Bayesian Optimization gives us a strong, sample-efficient baseline that reflects current optimization practices. Additionally, we evaluated two transfer learning optimization algorithms: Ranking Gaussian Process Ensemble (RGPE) [8] and Quantile [20]. RGPE builds a single surrogate model per previous task, and then tries to match these previous tasks to the current one based on rank correlation. Quantile maps the performance data of each individual tasks into normalized distributions to find similarities in high performing configurations across tasks.

Metrics. To measure the effectiveness, we use the Best Found Throughput. This metric shows the best throughput achieved so far, rewarding good exploration of the search space. Since the absolute scale of the achieved throughput varies across environments, we normalized the throughput with respect to the optimum. To this end, we divided all throughputs by the optimal throughput for the specific environment before averaging across environments. We determined this optimal throughput by running Bayesian Optimization for 2000 iterations, with convergence typically observed after 1000 iterations. The result is a normalized throughput value in percent of the optimum. In the following sections, we refer to the best found throughput as the performance of a tuning algorithm.

Data Transfer Setup. We evaluate each tuning algorithm by executing data transfers on the simulated topology. Each transfer involved transferring a 7.7 GB CSV file of the TPC-H lineitem table [3] (scale factor 10). We observed similar throughput performance for larger datasets, which is why our trained model and evaluation can be generalized for data transfers of larger size. A tuning run consisted of 25 transfer executions. Unless stated otherwise, each algorithm is evaluated three times per environment, and we report the mean. For methods using historical training data, all environments except the current one were included to simulate transfer to an unseen setting. We supplied no data of previous transfers to Bayesian Optimization's surrogate model as we operate under the assumption of having no data for the current environment. Alternatively combining data of multiple environments in a single surrogate model showed very poor performance due to similar parameter choices having conflicting performance impacts.

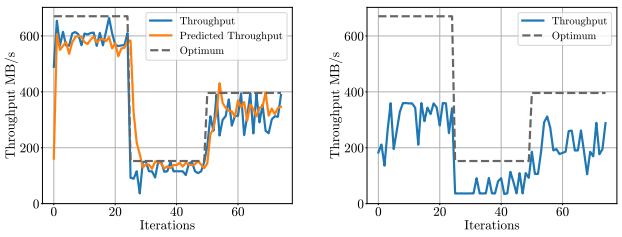
Transfer Environments. To evaluate our algorithms, we choose a subset of the sampling environments which we introduced in Table 2. We characterize these environments as edge, fog and cloud connections, and select one environment per connection pair, which we show in Table 3. This way our evaluation covers a broad range of transfer environments, similar to CPU characteristics on general-purpose aws-ec2 instances [1], enabling the ability to generalize our experimental results. We simulate these environments by restricting docker CPU usage and using docker traffic control [2].

4.2 End-to-End Tuning Performance

Best Found Throughput. The primary goal of our tuning approach is to identify high-throughput configurations efficiently. To measure this aspect, we evaluate the *Best Found Throughput*, the highest throughput observed during a tuning run of 25 iterations. This metric is relevant in practical scenarios where tuning is performed before exploiting the best configuration for many future data transfers. The faster a method finds a good configuration, the earlier data transfers can start. Figure 6 shows the performance curves for all tuning algorithms over the 25 iterations. ADAPT consistently outperforms all other algorithms throughout the entire tuning run. After just 5 iterations, we achieve 90.9% of the optimum, while RGPE reaches 76.5% and Quantile 71.3%. Bayesian Optimization and Random Search lag behind at 65% and 65.0% respectively, and the Heuristic Optimizer reaches 67.5%. By the end of the 25-iteration tuning run, ADAPT achieves 94.7% of the optimal throughput. This result is higher than RGPE (89.5%), Quantile (85.8%), Bayesian Optimization (75.9%), and Random Search (73.0%). The Heuristic Optimizer remains static at 67.5%, as it does not adapt to feedback. These results highlight the value of transfer learning and adaptive tuning. ADAPT identifies effective configurations—by leveraging historic knowledge and runtime feedback—faster than algorithms that start from scratch.

KEY INSIGHT 1. *History-aware and adaptive algorithms, such as ADAPT, are substantially more effective than static or uninformed strategies for quickly finding high-throughput configurations in high-dimensional search spaces.*

Accumulated Time. Beyond finding high-performing configurations, a tuning algorithm must do so efficiently in terms of overall time. To evaluate this aspect, we measure the *Accumulated Time* as the cumulative wall-clock time taken across 25 iterations, including initialization, finding new configurations, executing transfers, and updating models. As shown in Figure 7, ADAPT yields the lowest overall accumulated time. The next best baselines are RGPE and Quantile. Initially, Quantile performs slightly better, but is



(a) ADAPT (b) RGPE

Figure 8: Adaptability Across Changing Environments.

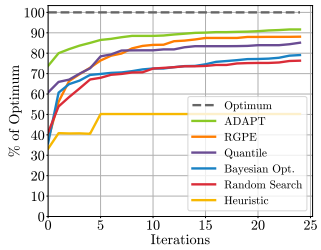


Figure 9: Postgres - CSV.

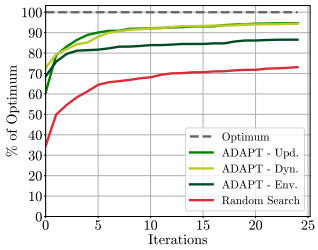


Figure 10: Weight Strategies.

overtaken by the Heuristic Optimizer in the second half of the tuning runs. Random Search has the highest accumulated time, with Bayesian Optimization only marginally ahead. This behavior reflects the trade-off between exploration and exploitation inherent in all algorithms. Since ADAPT incorporates environment knowledge, it can immediately exploit prior knowledge, reducing both search overhead and data transfer runtime. RGPE and Quantile also benefit from historical data, but must first assess the similarity between the current environment and past tuning tasks. RGPE is particularly effective in this regard, maintaining strong overall performance with moderate overhead, while Quantile spends more time on exploration. The Heuristic Optimizer follows a fixed strategy: after an initial exploration phase of 5 iterations, it switches to exploiting a single static configuration. This approach leads to relatively low accumulated time despite poor throughput performance. Bayesian Optimization is able to exploit to a limited extent, but its iterative search and model update steps introduce non-trivial computational costs, resulting in accumulated time only slightly better than Random Search (which continues with pure exploration without any exploitation).

KEY INSIGHT 2. *ADAPT finds high performance configurations, and does so with the lowest runtime overhead. Its ability to immediately exploit prior knowledge and efficiently adapt sets it apart from both static and black-box tuning approaches.*

4.3 Adapting to Environments and Systems

Robust tuning methods must remain effective despite two challenges of real-world deployment: dynamically changing environments and heterogeneous system architectures. Transfer conditions can fluctuate at runtime due to network congestion or variable compute availability, requiring models to adapt quickly. Similarly, tuning must generalize across systems with different data sources, formats, or physical operator implementations. We evaluate both aspects: adaptation to shifting runtime conditions, as well as transfer performance across distinct system backends.

Adapting to Dynamic Environments. To test adaptability during runtime changes, we evaluate ADAPT and RGPE under dynamic conditions. We choose RGPE as baseline because it is the closest in performance. We construct a three-phase environment schedule, beginning with a high-bandwidth setting (16, 16, 1000), simulating a bottleneck by reducing bandwidth to 50 MBit/s, and then partially recovering to 150 MBit/s. We execute each algorithm for 25 iterations in each environment before transitioning to the next, without prior notice. Importantly, ADAPT is run without any prior knowledge of the target environment. It had to infer changes solely based on observed transfer results, similar to how other transfer-learning algorithms like RGPE operate. To avoid information leakage, we excluded the three tested environments from the training data (instead of just one), ensuring that the evaluation was performed on entirely unseen conditions. Figure 8 shows the throughput achieved in each phase. ADAPT quickly adapts after each environment change, recovering performance within a few iterations and consistently suggesting high-performing configurations. In contrast, RGPE fails to adapt effectively: it underperforms in the initial environment, struggles severely in the bottleneck phase, and only partially recovers in the final setting. This strong adaptability is driven by ADAPT's dynamic weighting mechanism. We monitor the prediction accuracy of each base model and updates its weights based on recent feedback. As the environment shifts, outdated models receive lower weight, allowing the ensemble to quickly focus on more relevant behavior. While RGPE does incorporate performance feedback, it struggles to adapt effectively to changing environments, suggesting its mechanism is less responsive and less capable of discarding outdated knowledge.

KEY INSIGHT 3. *ADAPT is able to adapt to changing conditions substantially faster and more reliably than RGPE. The ability to recognize environment shifts from performance feedback and update the weighting strategy accordingly enables ADAPT to effectively handle real-world, non-stationary workloads.*

Robustness Across Data Systems. We evaluate the generalization of our tuning approach across different data systems by applying it to PostgreSQL-to-CSV transfers. This setup reflects practical use cases such as data lake ingestion or backups. PostgreSQL has distinct performance characteristics due to query execution, disk I/O, transactions, and resource contention within the database. XDBC also uses a different physical implementation for the PostgreSQL connector, particularly for how data is read and deserialized, leading to different throughput patterns. These changes are a representative test for generalization across systems with different physical operators. To evaluate this setup, we generate a training dataset using 50% random samples and 50% simulated optimization runs. We determine the optimal throughput per environment using the same Bayesian Optimization procedure as

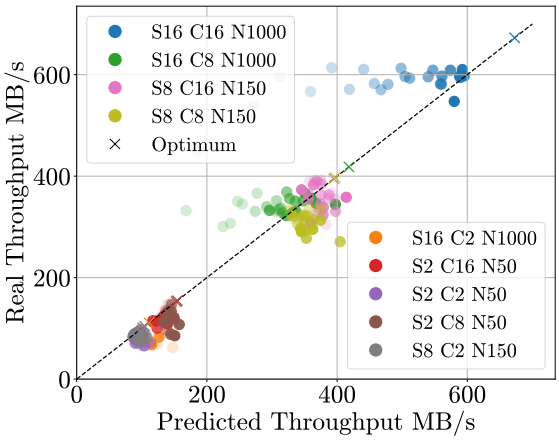


Figure 11: Prediction Accuracy.

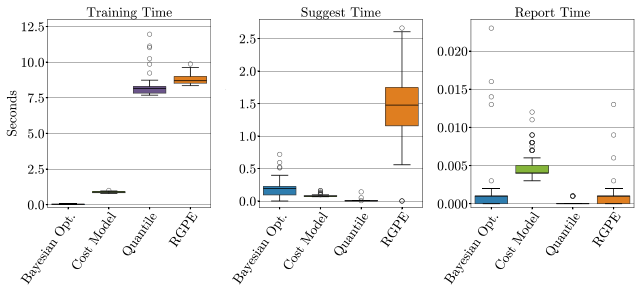


Figure 12: Algorithms Computation Times

before. Figure 9 shows the Best Found Throughput across algorithms for PostgreSQL-to-CSV transfers. The relative performance of the algorithms are consistent with our results in Figure 6.

Key Insight 4. *Despite major system-level differences, Adapt generalizes well when provided with representative training data, maintaining strong relative performance compared to the baselines across heterogeneous systems.*

4.4 Micro-Benchmarks

To understand the behavior and design trade-offs of our tuning approach, we present a series of targeted micro-benchmarks. These benchmarks evaluate key properties of Adapt, including prediction performance, efficiency, and generalizability.

Prediction Quality. As a first micro-benchmark, we evaluate Adapt's cost model's predictive behavior by comparing predicted and observed throughput values across environments. Figure 11 shows a scatter plot where each point corresponds to a single transfer configuration. Colors represent different environments, and X markers indicate the per-environment optimal throughput. Ideally, accurate predictions should fall along the diagonal (perfect prediction) and close to the corresponding X marker (high-quality configuration for that environment). We observe both patterns: later predictions (darker points) tend to align more closely with the diagonal, indicating improved accuracy as the model incorporates more feedback. Many points also lie near their environment's X, showing that the model can effectively identify strong-performing configurations across a range of conditions. Despite initially conservative estimations, the model improves rapidly after a few iterations and consistently converges toward optimal configurations. This bias is favorable in practice, as it avoids overconfident predictions while still discovering and evaluating high-performing configurations.

Key Insight 5. *The cost model produces conservative and increasingly accurate predictions that reliably guide the search toward high-throughput configurations.*

Weighting Strategies. We further compare three different weighting strategies in our ensemble cost model:

- **Environment-based Weights**, which favor models of similar environments and enable strong zero-shot performance;
- **Update-based Weights**, which adapt to recent prediction errors and enable responsiveness to new conditions; and
- **Dynamic Weighting** as a combination of environment- and update-based weights, which gradually shifts from environment similarity to feedback-driven reliability.

Figure 10 shows that environment-based weights perform best initially, providing strong early configuration candidates. Update-based weights improve steadily over time as feedback from measurements accumulate. The dynamic strategy combines both behaviors, starting with good performance and adapting quickly, and achieves the highest overall performance.

Key Insight 6. *The dynamic weighting strategy offers the best of both worlds: reliable zero-shot performance in early iterations and robust adaptation as tuning progresses.*

Tuning Trade-offs. During tuning, algorithms incur computational overhead at three key stages: initialization, configuration suggestion, and result update. Initialization can be costly for methods that rely on historical data, as they must ingest and train surrogate models. Suggesting a new configuration involves search and inference, where the runtime mostly depends on the model complexity. Finally, updating the model with new results adds additional overhead, which is typically very small though. Figure 12 reports the mean time of the individual phases of different baselines. Adapt is quick at initialization (under 1s on average), achieving a 10× speedup over RGPE and Quantile. Furthermore, Adapt also offers fast configuration suggestions (0.08s), which is close to Quantile (0.003s) and faster than Bayesian Optimization (0.17s) and RGPE (1.4s). Model update times are low across all methods, with negligible impact on overall overheads, especially compared to executing data transfers with the selected configurations.

Key Insight 7. *Adapt has low overhead across all tuning phases, making it a versatile method and efficient enough for time-constrained tuning algorithms and deployments.*

Data Efficiency. Since executing real data transfers is costly, it is important to understand how much training data is needed for Adapt to perform well. We evaluate this aspect by varying the number of training samples per environment: 100, 200, 400,

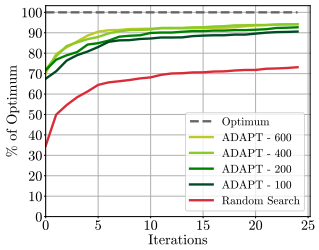


Figure 13: Data Efficiency.

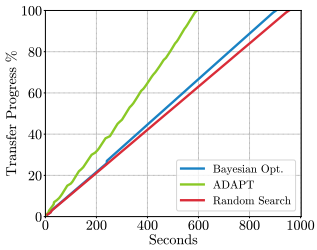


Figure 14: Online Tuning.

and 600. Figure 13 illustrates the resulting performance. These results show that more training data improves initial performance, with noticeable gains up to around 400 samples. Beyond that point, improvements are minimal, indicating diminishing returns. Notably, even with only 100 samples, the model quickly recovers after a few iterations as it incorporates online feedback.

Key Insight 8. *Adapt is data-efficient, achieving strong performance with moderate sizes of the training dataset, and quickly adapting when initial data is limited.*

4.5 Online Tuning

The previous experiments focused on inter-transfer optimization, where configurations are selected before a transfer begins. However, real-world scenarios such as streaming pipelines or large file transfers often benefit from adapting parameters during execution in response to fluctuating resources. This characteristic motivates intra-transfer (online) tuning, where parameter adjustments are made dynamically while the transfer is in progress. XDBC currently supports online reconfiguration for a limited set of parameters, including component parallelism and compression. Although work to support other parameters (e.g., network parallelism, buffer sizes) is ongoing, we evaluate our method in this constrained setting.

Intra-transfer Progress. For this experiment, we modified our tuning loop to initialize the transfer with a starting configuration, then update it every 3 seconds based on recent performance. Each update involves passing the latest metrics to the tuning algorithm, which updates its model and selects a new configuration within the restricted search space. We used the TPC-H lineitem table at scale factor 100 (77 GB) to ensure transfers were long enough for multiple updates, and averaged the results over threes transfers per environment across 9 environments. Figure 14 shows transfer progress over time, where steeper curves indicate faster completion. Adapt consistently outperforms both Bayesian Optimization and Random Search, completing transfers in less time across all environments. While the performance gap is smaller than in the inter-transfer setting (cf. Figure 6), our method remains the most effective overall, even with limited reconfiguration support.

Key Insight 9. *Adapt effectively adapts during transfer execution, making it suitable for online, intra-transfer tuning scenarios, even with limited reconfigurability.*

5 RELATED WORK

Our Adapt is related to general data transfer optimizations, ML-based knob tuning in databases, as well as learned cost models.

Data Transfer Optimization. Prior work has explored methods to automate data transfer parameter configurations in various settings. HARP [4] combines historic data analysis with real-time probing, using polynomial regression to predict throughput for application-layer parameters and their iterative refinement. Nine et al. [15] similarly separate the process into offline and online phases: the offline stage clusters historic data using K-means++ and fits polynomial surfaces, while the online stage leverages these surfaces for guided sampling and prediction. Both approaches focus on a narrow set of network-level parameters and use simple models, in contrast to Adapt's broader scope and learned ensemble model. Moreover, these existing approaches are not integrated into data connectors, preventing them from optimizing data loading or transformation stages. Sapkota et al. [21] propose a real-time genetic optimization algorithm across a broader parameter space, including read, write, and network threads—similar to our setting. However, the reliance of this approach on live exploration with small configuration populations limits the initial performance because strong starting configurations (informed by prior data) are lacking.

Machine Learning for Data Systems. The complexity and high dimensionality of modern data systems have motivated ML-based tuning approaches to replace the tedious manual knob configuration and tuning. OtterTune [25] uses runtime metrics to characterize workloads, applies lasso regression to rank impactful knobs, and leverages Gaussian processes to recommend DBMS configurations based on historical data from similar workloads. Peloton [16] proposes a broader autonomous DBMS architecture, using DBSCAN for workload classification and recurrent neural networks for workload forecasting, which feed into a planning component for optimizing system performance. Both systems use machine learning to automate DBMS tuning, relying either on workload similarity or internal feedback loops, but do not target the transfer-specific optimization problem Adapt addresses. ProteusTM [7] frames the problem of choosing a transactional-memory algorithm as a recommendation task to improve application performance. They create a workload-configuration utility matrix in offline training using collaborative filtering, and use it to guide the online search of a Bayesian Optimizer. While this approach allow to quickly arrive at high performing configurations with few online iterations, it is missing the cross-environment transfer-learning ability which is a crucial component of our Adapt approach.

Learned Cost Models. As more accurate and adaptive alternatives to traditional analytical cost models, existing work introduced learned cost models, particularly in query optimization. CLEO [23] uses an ensemble of Elastic Net regressors—some specialized for operator subgraphs, others generalized by operator type—and combines them using a FastTree Regression meta-model trained to choose the best predictor per case. CLEO continuously retrains its models on observed plan execution logs to remain adaptive. Hilprecht et al.'s Zero-Shot approach [12] tackles generalization to unseen databases using a GNN-based cost model trained across diverse databases. Their model takes database features and query plan encodings as input, with optional fine-tuning for specific deployment targets. Both works demonstrate the effectiveness of ensemble learning and feedback-driven adaptation, which we extend to the domain of transfer parameter tuning in Adapt.

6 CONCLUSIONS

Optimizing data transfer parameters in dynamic, distributed systems remains a major challenge, with a direct impact on the performance of data-intensive applications. The large configuration space and fluctuating system conditions render manual tuning infeasible, and existing heuristic methods lack the adaptability needed in practice. In this paper, we presented ADAPT, a data- and ML-driven approach for automatically tuning transfer parameters in the XDBC framework. Our contributions include an environment-aware ensemble cost model that combines prior knowledge and live observations through a dynamic weighting mechanism. To select the next configuration, we proposed an efficient two-phase search strategy that leverages Latin Hypercube Sampling, batch prediction, and search space pruning. Through extensive experiments, we showed that ADAPT consistently outperforms the existing heuristic optimizer, as well as standard black-box optimization methods like Bayesian Optimization, and transfer learning baselines such as RGPE. ADAPT achieves higher data transfer throughput and adapts substantially faster to environmental changes, including sudden bandwidth bottlenecks. We also analyzed the training runtime and data efficiency, demonstrating low overhead and strong performance even with limited training data. Future work includes extending the full support for online tuning of live transfers, which would further enhance responsiveness. Incorporating richer environment and data characteristics into the weighting mechanism may also improve adaptation. Overall, ADAPT offers a practical and effective solution for tuning data transfer parameters, demonstrating strong performance across diverse conditions and workloads.

ACKNOWLEDGMENTS

We gratefully acknowledge funding from the German Federal Ministry of Education and Research under the grants BIFOLD25B and 01IS17052 (as part of the Software Campus project PolyDB). We thank Midhun Kaippillil Venugopalan for his contributions to the XDBC prototype, and the anonymous reviewers for their suggestions on improving the paper.

REFERENCES

[1] 2025. AWS EC2-T2. https://aws.amazon.com/de/ec2/instance-types/t2/. Accessed: 2025-06-09.

[2] 2025. Docker Traffic Control. https://github.com/lukaszlach/docker-tc. Accessed: 2025-06-05.

[3] 2025. TPC BENCHMARK H. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf. Accessed: 2025-06-05.

[4] Engin Arslan and Tevfik Kosar. 2018. High-Speed Transfer Optimization Based on Historical Analysis and Real-Time Tuning. 29, 6 (2018), 1303–1316. https://doi.org/10.1109/TPDS.2018.2790948 Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[5] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *J. Mach. Learn. Res.* 13 (2012), 281–305. https://doi.org/10.5555/2503308.2188395

[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *SIGKDD*. 785–794. https://doi.org/10.1145/2939672.2939785

[7] Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. 2016. ProteusTM: Abstraction Meets Performance in Transactional Memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM, 757–771. https://doi.org/10.1145/2872362.2872385

[8] Matthias Feurer, Benjamin Letham, Frank Hutter, and Eytan Bakshy. 2022. Practical Transfer Learning for Bayesian Optimization. https://doi.org/10.48550/arXiv.1802.02219

[9] Haralampos Gavriilidis, Kaustubh Beedkar, Matthias Boehm, and Volker Markl. 2025. Fast and Scalable Data Transfer Across Data Systems. *Proc. ACM Manag. Data* 3, 3, Article 157 (June 2025). https://doi.org/10.1145/3725294

[10] Haralampos Gavriilidis, Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. In-Situ Cross-Database Query Processing. In *ICDE*. 2794–2807. https://doi.org/10.1109/ICDE55515.2023.00214

[11] Haralampos Gavriilidis, Joel Ziegler, Midhun Kaippillil Venugopalan, Benedikt Didrich, Matthias Boehm, and Volker Markl. 2025. Enter the Warp: Fast and Adaptive Data Transfer with XDBC. *Proc. VLDB Endow.* 18, 12 (2025).

[12] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. *CoRR* abs/2201.00561 (2022). arXiv:2201.00561 https://arxiv.org/abs/2201.00561

[13] Yang Li, Yu Shen, Wentao Zhang, Yuanwei Chen, Huaijun Jiang, Mingchao Liu, Jiawei Jiang, Jinyang Gao, Wentao Wu, Zhi Yang, Ce Zhang, and Bin Cui. 2021. OpenBox: A Generalized Black-box Optimization Service. In *SIGKDD*. 3209–3219. https://doi.org/10.1145/3447548.3467061

[14] M. D. McKay, R. J. Beckman, and W. J. Conover. 1979. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. 21, 2 (1979), 239–245. https://doi.org/10.2307/1268522 Publisher: [Taylor & Francis, Ltd., American Statistical Association, American Society for Quality].

[15] MD S Q Zulkar Nine and Tevfik Kosar. 2021. A Two-Phase Dynamic Throughput Optimization Model for Big Data Transfers. 32, 2 (2021), 269–280. https://doi.org/10.1109/TPDS.2020.3012929 Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[16] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR*. http://cidrdb.org/cidr2017/papers/p42-pavlo-cidr17.pdf

[17] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830. https://doi.org/10.5555/1953048.2078195

[18] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2017. Data Management Challenges in Production Machine Learning. In *SIGMOD* (New York, NY, USA, 2017). 1723–1726. https://doi.org/10.1145/3035918.3054782

[19] David Salinas, Matthias Seeger, Aaron Klein, Valerio Perrone, Martin Wistuba, and Cedric Archambeau. 2022. Syne Tune: A Library for Large Scale Hyperparameter Tuning and Reproducible Research. In *International Conference on Automated Machine Learning, AutoML 2022* (2022). https://proceedings.mlr.press/v188/salinas22a.html

[20] David Salinas, Huibin Shen, and Valerio Perrone. 2020. A Quantile-based Approach for Hyperparameter Transfer Learning. In *ICML*. PMLR, 8438–8448. https://proceedings.mlr.press/v119/salinas20a.html

[21] Hemanta Sapkota, Engin Arslan, and Sushil Louis. 2020. Real-time genetic optimization of large file transfers. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion* (New York, NY, USA, 2020-07-08) *(GECCO '20)*. Association for Computing Machinery, 283–284. https://doi.org/10.1145/3377929.3389917

[22] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. 2016. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proc. IEEE* 104, 1 (2016), 148–175. https://doi.org/10.1109/JPROC.2015.2494218

[23] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *SIGMOD* (New York, NY, USA, 2020) *(SIGMOD)*. 99–113. https://doi.org/10.1145/3318464.3380584

[24] Tinu Theckel Joy, Santu Rana, Sunil Gupta, and Svetha Venkatesh. 2016. Hyperparameter tuning for big data using Bayesian optimisation. 2574–2579. https://doi.org/10.1109/ICPR.2016.7900023

[25] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD* (2017). 1009–1024. https://doi.org/10.1145/3035918.3064029

[26] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[27] Xiaoying Wang, Weiyuan Wu, Jinze Wu, Yizhou Chen, Nick Zrymiak, Changbo Qu, Lampros Flokas, George Chow, Jiannan Wang, Tianzheng Wang, Eugene Wu, and Qingqing Zhou. 2022. ConnectorX: accelerating data loading from databases to dataframes. *PVLDB* 15, 11 (2022), 2994–3003. https://doi.org/10.14778/3551793.3551847