# CODA: Toward Automatically Identifying and Scheduling COflows in the DArk

Hong Zhang[1] Li Chen[1] Bairen Yi[1] Kai Chen[1] Mosharaf Chowdhury[2] Yanhui Geng[3]
[1]SING Group, Hong Kong University of Science and Technology
[2]University of Michigan [3]Huawei
{hzhangan,lchenad,biy,kaichen}@cse.ust.hk, mosharaf@umich.edu, geng.yanhui@huawei.com

## ABSTRACT

Leveraging application-level requirements using coflows has recently been shown to improve application-level communication performance in data-parallel clusters. However, existing coflow-based solutions rely on modifying applications to extract coflows, making them inapplicable to many practical scenarios.

In this paper, we present CODA, a first attempt at automatically identifying and scheduling coflows *without* any application modifications. We employ an incremental clustering algorithm to perform fast, *application-transparent coflow identification* and complement it by proposing an *error-tolerant coflow scheduler* to mitigate occasional identification errors. Testbed experiments and large-scale simulations with production workloads show that CODA can identify coflows with over 90% accuracy, and its scheduler is robust to inaccuracies, enabling communication stages to complete 2.4× (5.1×) faster on average (95-th percentile) compared to per-flow mechanisms. Overall, CODA's performance is comparable to that of solutions requiring application modifications.

## CCS Concepts

•**Networks** → **Cloud computing;**

## Keywords

Coflow; data-intensive applications; datacenter networks

## 1 Introduction

A growing body of recent work [21, 23, 24, 30, 38, 68] has demonstrated that leveraging application-level information using coflows [22] can significantly improve the communication performance of distributed data-parallel applications.[1] Unlike the traditional flow abstraction, a coflow captures a collection of flows between two groups of machines in successive computation stages, where the communication stage finishes only after *all* the flows have completed [23, 26]. A typical example of coflow is the shuffle between the mappers and the reducers in MapReduce [28]. By taking a holistic, application-level view, coflows avoid stragglers and yield benefits in terms of scheduling [21, 23, 24, 30], routing [68], and placement [38].

However, extracting these benefits in practice hinges on one major assumption: *all* distributed data-parallel applications in a shared cluster – be it a platform-as-a-service (PaaS) environment or a shared private cluster – have been modified to *correctly* use the *same* coflow API.

Unfortunately, enforcing this requirement is infeasible in many cases. As a first-hand exercise, we have attempted to update Apache Hadoop 2.7 [58] and Apache Spark 1.6 [65] to use Aalo's coflow API [24] and faced multiple roadblocks in three broad categories (§5): the need for intrusive refactoring, mismatch between blocking and non-blocking I/O APIs, and involvement of third-party communication libraries.

Given that users on a shared cluster run a wide variety of data analytics tools for SQL queries [3, 4, 7, 15, 41, 63], log analysis [2, 28, 65], machine learning [33, 43, 48], graph processing [34, 44, 46], approximation queries [7, 12], stream processing [9, 11, 13, 50, 66], or interactive analytics [7, 65], updating one application at a time is impractical. To make things worse, most coflow-based solutions propose their own API [23, 24, 30]. Porting applications back and forth between environments and keeping them up-to-date with evolving libraries is error-prone and infeasible [54, 57].

Therefore, we ponder a fundemantal question: *can we automatically identify and schedule coflows without manually updating any data-parallel applications?* It translates to three key design goals:

- **Application-Transparent Coflow Identification** We must be able to identify coflows without modifying applications.



*SIGCOMM '16, August 22–26, 2016, Florianopolis, Brazil*
 ISBN 978-1-4503-4193-6/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2934872.2934880

[1]We use the terms application and framework interchangeably in this paper. Users can submit multiple jobs to each framework.

- **Error-Tolerant Coflow Scheduling** Coflow identification cannot guarantee 100% accuracy. The coflow scheduler must be robust to some identification errors.
- **Readily Deployable** The solution must be compatible with existing technologies in datacenter environments.

In this paper, we provide a cautiously optimistic answer via CODA. At the heart of CODA is an application-transparent coflow identification mechanism and an error-tolerant coflow scheduling design.

For coflow identification, we apply machine learning techniques over multi-level attributes without manually modifying any applications (§3). Besides explicit attributes directly retrieved from flows (e.g., arrival times and packet headers), we further explore implicit attributes that reflect communication patterns and data-parallel framework designs. As to the identification algorithm, we find that traditional traffic classification methods [17, 19, 29, 40, 47, 49, 51, 55, 67] do not directly apply in our case. This is because coflows capture a one-off, mutual relationship among some flows that cannot be pre-labeled and need timely identification. To this end, we first identify DBSCAN [31] as the base algorithm that fits our requirements, and then we develop an incremental version of Rough-DBSCAN [59] that provides fast identification with high accuracy.

Despite its high accuracy, CODA's identifier is not perfect, and identification errors are unavoidable in practice. Such errors, if present, may greatly affect the performance of existing coflow schedulers. Consider Figure 1 as an example: a misclassified flow can significantly affect the coflow completion time (CCT) of its parent coflow.

The key to CODA's effectiveness lies in developing a robust scheduler that can tolerate such errors (§4). For error-tolerant coflow scheduling, we start by studying how identification errors would influence scheduling results. Our analysis reveals that stragglers significantly affect CCT, and recent coflow schedulers [23, 24] suffer performance degradation in the presence of errors. Thus, CODA employs *late binding* to delay the assignment of flows to particular coflows until they must be scheduled to minimize the impact of stragglers. Furthermore, we find that *intra-coflow prioritization* [14, 16, 37, 62] can play a crucial role in the presence of identification errors. Hence, unlike existing coflow schedulers [21, 23, 24, 30], CODA combines per-flow (intra-coflow) prioritization with inter-coflow scheduling.

We have implemented a CODA prototype (§5) and built a small-scale testbed with 40 servers to evaluate its performance (§6.2). Our implementation experience shows that CODA can be readily deployed in today's commodity datacenters with no modifications to switch hardware or application software. By replaying a coflow benchmark based on Facebook traces [5], we show that CODA achieves over 95% accuracy in identification, improves the average and 95-th percentile CCT by 2.4× and 5.1× compared to per-flow fairness, and performs almost as well as Aalo [24], which requires correct, manual coflow identification. Moreover, CODA can scale up to 40,000 agents with small performance loss.

We further perform large-scale trace-driven simulations to inspect CODA (§6.3 and §6.4). CODA's identifier achieves over 90% accuracy for Spark, Hadoop, and mixed workloads, and provides significant speedup over vanilla DBSCAN. In terms of error-tolerant scheduling, we show that CODA can effectively tolerate misidentifications over a wide range of scenarios. For example, in a challenging case with less than 60% identification accuracy, CODA's error-tolerant design brings up to 1.16× speedup in CCT, reducing the impact of errors by 40%. Overall, CODA achieves the performance comparable to that of prior solutions using manual annotations in many cases.

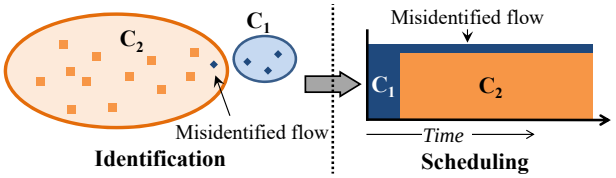



**Figure 1:** Motivating example: a coflow scheduler (e.g., Aalo [24]) tends to optimize the CCT by prioritizing the small coflow $C_1$ over the large coflow $C_2$. However, a misidentified flow of $C_1$ will be scheduled together with $C_2$, significantly affecting the CCT of its parent coflow $C_1$.

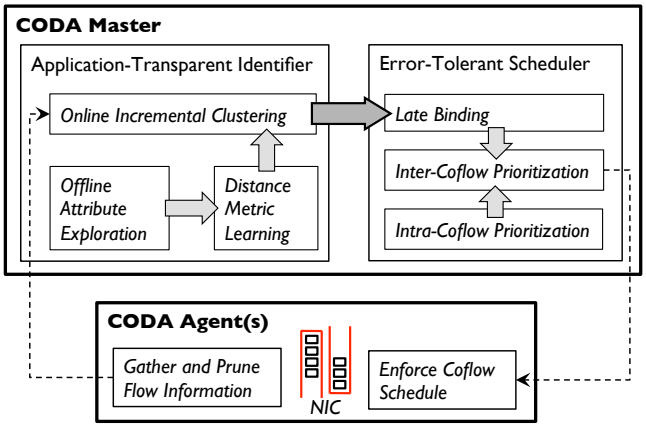



**Figure 2:** CODA architecture: CODA agents collect flow-level information, and CODA master periodically updates coflow schedules using application-transparent identification and error-tolerant scheduling mechanisms.

## 2 CODA Overview

The goal of CODA is to design *an identifier-scheduler joint solution that works "in the dark"*, relying only on externally observable coflow attributes that can be collected from the network without modifying applications/frameworks.

Figure 2 presents an overview of CODA system architecture. At a high level, it contains a central CODA *master* that performs the primary role of coflow identification and scheduling every $\triangle$ interval (e.g., $10 - 100ms$), as well as a CODA *agent* on each end host that collects aggregated flow-level information to feed the master and enforces scheduling decisions made by the master.

**Information Gathering and Pruning** Each CODA agent monitors flow-level attributes and IP/port information of all flows in the corresponding host and periodically forwards them to the master. Before sending out the records, each agent

prunes the records of all finished flows, non-TCP flows[2] and flows with sent size less than a threshold (e.g., 100KB). This reduces identification time and avoids extra traffic (e.g., control flows) not useful to the identification process.

**Application-Transparent Coflow Identification** Given periodic flow records, CODA master invokes a coflow identifier to identify coflow relationships using machine learning (§3). To achieve high accuracy, the identifier explores useful attributes on multiple levels and learns an appropriate distance metric to reflect coflow relations. For timely identification, it trades off a small amount of accuracy for significantly higher speed and relies on the coflow scheduler to amend the errors.

**Error-tolerant Coflow Scheduling** Next, the master runs a coflow scheduler on the identified coflows. The scheduler tries to minimize coflow completion times (CCT) in the presence of possible identification errors (§4). Specifically, the error-tolerant design integrates the following two design principles. First, we observe that stragglers may heavily affect CCTs. We apply *late binding* to the identification results – i.e., delaying the assignment of a flow to a particular coflow until we must schedule – to decrease the number of stragglers. Second, we notice that intra-coflow scheduling affects CCT under identification errors, and we introduce *intra-coflow prioritization* to reduce the impact of errors. Finally, the master sends out updated schedules to relevant end hosts to complete the identification-scheduling cycle.

CODA's centralized architecture is inspired by the success of many large-scale infrastructure deployments such as [28, 32, 35, 65] that employ a central controller at the scale of tens to hundreds of thousands of machines. Because CODA master must serve a large number of CODA agents, it must be scalable and fault-tolerant.

**Scalability** The faster CODA agents can coordinate, the better CODA performs. The number of messages is linear with the number of agents and independent of the number of flows or coflows, and it is not a bottleneck in our testbed. Our evaluation suggests that CODA can scale up to 40,000 machines with small performance loss (§6.2). Because many coflows are tiny [23] and can effectively be scheduled through local decisions [24], they do not face coordination overheads.

**Fault Tolerance** CODA fails silently from an application's perspective, as it is application-transparent by design. CODA handles master/ agent failures by restarting them. A restarted CODA master can rebuild its state from the next wave of updates from the agents. Restarted CODA agents remain inconsistent only until the next schedule arrives from the master.

# 3 Coflow Identification

CODA identifier aims to meet three practical objectives:

- *Transparency*: It should not require any modification to applications.
- *Accuracy*: It should identify accurate coflow relationships to enable correct scheduling.
- *Speed*: It should be fast enough for timely scheduling.

To achieve these goals, CODA identifier relies on the following three steps:

1. **Attribute Exploration** A flow can be characterized by a tuple of attributes, and searching for useful attributes is a key first step for coflow identification. Instead of taking a black-box approach, CODA explores explicit and implicit attributes and heuristics on multiple levels (§3.1).
2. **Distance Calculation** Given the attributes, CODA calculates distances between flows to capture coflow relationships – flows belonging to the same coflow will have smaller distances. The key here is having a good metric to reflect the importance of each attribute. CODA employs distance metric learning [64] to learn such a metric (§3.2).
3. **Identifying Coflows via Clustering** Finally, CODA employs *unsupervised clustering* to identify coflow relationships. We use unsupervised learning because coflows cannot be labeled by predefined categories – mutual relationships among flows captured by a particular coflow do not recur once its parent job is over. CODA leverages an incremental Rough-DBSCAN algorithm to achieve fast yet accurate coflow identification by clustering flows with small distances (§3.3).

## 3.1 Multi-Level Attributes

We first explore a set of flow, community, and application level attributes that might be useful in coflow identification. We prune this set in §3.2.

**Flow-level Attributes** First and foremost, we consider the widely-used flow-level attributes [52]: (i) $S_{time}$: flow start time; (ii) $M_{size}$: mean packet size inside a flow; (iii) $V_{size}$: variance of packet sizes inside a flow; (iv) $M_{int}$: average packet inter-arrival time inside a flow.

IPs and ports have application-specific meanings, which we exploit later when considering application structures and communication patterns. We ignore flow size and duration as they cannot be acquired until a flow finishes; at that time, they would be useless.

**Community-Level Attributes** Recent studies on datacenter traffic show that the traffic matrix is sparse and most bytes stay within a stable set of nodes [18, 56]. This suggests a community attribute; i.e., the datacenter can be separated into service groups where intra-group communication is frequent while inter-group communication is rare. With this, we can have a useful heuristic: *two flows belonging to different communities are less likely to be inside one coflow.* We define the community distance $D_{com}(f_i, f_j)$ to be 0 if flow $f_i, f_j$ are in the same community, and 1 otherwise. To calculate $D_{com}$, we develop a community detection module, which uses spectral clustering [60] to segment machines into communities while minimizing inter-community traffic.

Community-level attributes can be very helpful in differentiating coflows across services that show stable and isolated patterns, e.g., service groups within private datacenters or tenants in public clouds. However, it may not work under uniformly distributed traffic across the entire cluster [56].

[2]Currently most data-parallel computing frameworks leverage TCP for reliable data transfer.

**Application-Level Attributes** We seek more useful attributes by taking advantage of application designs. We investigate two use cases – Spark and Hadoop[3] – to observe their data transfer design by delving into the source code.

*Port assignment in Spark:* The port assignment rule in Spark reveals that data transmission to the same executor [8] will have the same destination IP and port (the port of the reducer's `ConnectionManager`). If we denote all flows to the same IP/port as a *flow aggregation*, then all flows within a flow aggregation are likely to be within a coflow. Hence, we define port distance $D_{prt}(f_i, f_j)$ for two flows $f_i$ and $f_j$ to be 0 if they are in one flow aggregation, and 1 otherwise.

*Port assignment in Hadoop:* Unlike Spark, shuffle traffic from different Hadoop jobs are likely to share the same source port of `ShuffleHandler` (13562 by default) and random destination ports. Consequently, port assignments do not provide distinctive information for Hadoop.

**OS-level Attributes** OS-level knowledge can also be helpful for coflow identification. For example, for each flow one can trace the corresponding process ID (PID) of the mapper, and flows sharing the same PID are likely to be in one coflow. Currently we have not included OS-level attributes due to their unavailability in public clouds.[4]

## 3.2 Distance Calculation

Given multiple attributes, a naive distance metric between two flows $f_i$ and $f_j$ can be defined as the Euclidean distance between them. However, equally weighing all attributes is not effective because different attributes may contribute differently – using irrelevant or redundant attributes may degrade identification accuracy.

Thus we need a good distance metric that can effectively reflect coflow relationships – one that assigns smaller distances between flows within the same coflow and larger distances between flows belonging to different coflows.

**Problem Formulation** Consider a flow set $\{\mathbf{f}\}$ and a distance metric $d(f_i, f_j) = ||f_i - f_j||_A = \sqrt{(f_i - f_j)^T A (f_i - f_j)}$. Suppose $(f_i, f_j) \in S$ if $f_i$ and $f_j$ belong to the same coflow, and $(f_i, f_j) \in D$ otherwise. Here, $A$ is the distance matrix reflecting the weight of different attributes, and setting $A = I$ gives Euclidean distance. We desire a metric where any pairs of flows in $S$ have small distances, while any pairs of flows in $D$ have distances larger than some threshold. This leads to the following optimization problem similar to [64]:

$$\begin{aligned} \min_{A} \quad & \sum_{(f_i,f_j)\in S} ||f_i - f_j||_A^2 \\ \text{s. t.} \quad & \sum_{(f_i,f_j)\in D} ||f_i - f_j||_A \geq 1, \quad A \succeq 0 \end{aligned} \tag{1}$$

We simplify the problem by restricting $A$ to be diagonal and solve it using Newton-Raphson method [64].

**Learning Results** We divide our testbed into two equal-sized communities with 10 servers each and run some typical benchmarks (e.g., Wikipedia-PageRank, WordCount) in Spark and Hadoop. We collect the trace, and the ground-truth coflow information is annotated by applications for metric learning. We use the attributes in §3.1, and run the above distance learning algorithm to see how they contribute to coflow identification. The resulting diagonal elements of matrices for Spark ($A_s$) and Hadoop ($A_h$) traffic are:

$$A_s = \begin{bmatrix} S_{time} & M_{size} & V_{size} & M_{int} & V_{int} & D_{com} & D_{prt} \\ 3.825 & 0.000 & 0.000 & 0.000 & 0.000 & 5.431 & 0.217 \end{bmatrix}$$

$$A_h = \begin{bmatrix} S_{time} & M_{size} & V_{size} & M_{int} & V_{int} & D_{com} & D_{prt} \\ 3.472 & 0.000 & 0.000 & 0.000 & 0.000 & 3.207 & 0.000 \end{bmatrix}$$

We observe three high-level characteristics:

1. Flow-level attributes other than the flow start time are not useful. This is because coflows may demonstrate similar packet-level characteristics regardless of their parent jobs;
2. Community-level attributes are distinctive; and
3. While port information is not useful for Hadoop as expected, it turns out to be of little use (with a small weight of only 0.217) for Spark as well, which is unexpected. One possible reason is that although flows within the same flow aggregation are likely to belong to one coflow in Spark, flows in one coflow may belong to different flow aggregations (and thus have $D_{prt} = 1$). This makes $D_{prt}$ less distinctive compared to $S_{time}$ and $D_{com}$.

We note that our procedure of identifying important attributes is critical for CODA's identification, especially under generic frameworks. Simulation results show that useless attributes greatly hamper identification accuracy, and distance metric learning brings significant improvement (§6.3). In our clustering algorithm below, we prune the useless attributes with near zero weights to simplify the distance calculation.

## 3.3 Identifying Coflows via Clustering

CODA leverages a fast and accurate unsupervised clustering algorithm to identify coflows. We choose DBSCAN [31] as the basis of our solution for two primary reasons. First, because the number of coflows changes dynamically over time, it is hard to timely and accurately estimate the number of clusters *a priori*. Unlike k-means [45] and many alternatives, DBSCAN can automatically determine the number of clusters given a radius parameter $\epsilon$. Second, a typical workload consists of a mix of small coflows (or single flows) with large coflow groups. Such imbalance prevents clustering algorithms that try to balance the size of clusters – e.g., spectral clustering [60] – from accurately identifying the singletons. DBCSAN does not impose such preference.

However, DBSCAN has one major drawback – its $O(n^2)$ worst-case time complexity, where $n$ is the number of flows. We address this drawback in two steps. First, we consider Rough-DBSCAN [59] (R-DBSCAN) – a variant of DBSCAN – instead, which trades off small accuracy for significantly faster speed. Second, we further improve R-DBSCAN to perform incremental classification, accounting for dynamic flow arrival/departure.

**R-DBSCAN for Clustering** The idea of R-DBSCAN is sim-

[3] We used Spark-1.6 and Hadoop-2.7.1 for this study.

[4] Cloud providers usually do not have access to customer VMs, and hence, cannot introduce any identification mechanism that hinges on OS-level information.

**Algorithm 1** Incremental R-DBSCAN

1: **procedure** CLUSTERING(Previous leader-follower structure $\mathbb{L}$ (initially $\emptyset$), New flows $\mathbb{F}_{new}$, Flows left $\mathbb{F}_{lft}$, range $\tau$)
2: **for** each Flow $f \in \mathbb{F}_{new}$ **do** ▷ Add new flows
3: Find a leader $l \in \mathbb{L}$ such that $d(f, l) < \tau$
4: **if** no such leader exists **then**
5: $\mathbb{L} = \mathbb{L} \bigcup \{f\}$ ▷ Create a new leader
6: $f.followers = \{f\}$
7: **else**
8: $l.followers = l.followers \bigcup \{f\}$ ▷ Add to an old leader
9: **end if**
10: **end for**
11: **for** each Flow $f \in \mathbb{F}_{lft}$ **do** ▷ Delete left flows
12: Find its leader $l$
13: **if** $f = l$ **then**
14: Delete $l$ from $\mathbb{L}$ if $l.followers = \{l\}$
▷ A leader is deleted only when it has no other followers
15: **else**
16: $l.followers = l.followers \setminus \{f\}$
17: **end if**
18: **end for**
19: Run $DBSCAN(\mathbb{L}, \epsilon, 1)$ and get $\mathbb{C}'$ (cluster of leaders)
20: Obtain $\mathbb{C}$ by replacing each leader by its followers
21: **return** cluster of flows $\mathbb{C}$
22: **end procedure**

ple – to perform DBSCAN only on a selected group of representative nodes (i.e., leaders). More specifically, a leader is a representative of flows within a distance range $\tau$ (i.e., followers of the leader). R-DBSCAN works in three steps:

1. Scan the dataset to derive leaders and their followers;
2. Run an algorithm similar to DBSCAN (with the same radius $\epsilon$ as in DBSCAN), but use only the set of leaders in deriving the clusters;
3. Derive the cluster of flows from the identified cluster of leaders, based on leader-follower relationships.

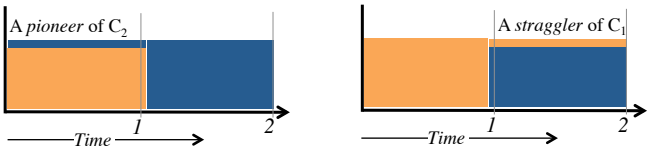
The complexity of R-DBSCAN is $O(nk + k^2)$, where $k$ is the number of leaders. In many cases, $k$ is much smaller than $n$, and it is proved that $k$ can be further bounded by a constant given the range $\tau$ [59]. More importantly, R-DBSCAN introduces very small accuracy loss compared to DBSCAN.

**Incremental R-DBSCAN** Recall that CODA master performs periodic identification and scheduling. When the set of active flows barely changes between intervals, a complete re-clustering over all flows is unnecessary. To this end, we develop an incremental R-DBSCAN (Algorithm 1) for further speedup, by considering dynamic flow arrival/departure. In each interval, it first updates the leader-follower relation based on last round information and flow dynamics (lines 1–18), and then applies R-DBSCAN on the updated leader-follower relations (lines 19–22). The incremental R-DBSCAN has a complexity of only $O(mk + k^2)$, where $m$ is the number of newly arrived/left flows. Since most intervals do not experience a big change in active flows, the incremental design can effectively improve the identification time (§6.3).

## 3.4 Discussion and Caveat

Our study in this paper currently centers around Spark and Hadoop – two of the most popular frameworks used in production datacenters today. While different frameworks may have different sets of useful attributes, we note that our approach toward attribute exploration, distance metric learning, and coflow identification via clustering is generally applicable. In future work, we are particularly interested in a comprehensive study on more attributes across more frameworks, their effectiveness and commonality.

Another observation is that, for a framework, the optimal weights of attributes may vary depending on workloads. However, such variations do not significantly affect identification accuracy as long as they clearly separate the distinctive attributes from the useless ones. As a result, to apply CODA to different frameworks and dynamic workloads, one possible way is to learn the weights of each framework offline and fix the setting for online identification. For example, we applied the weights learned above with our testbed workload (§3.2) to the Facebook workload (§6.1), achieving over 90% identification accuracy in many cases (§6.3). However, evaluating the robustness of this method and exploring the optimal weight settings of CODA under a variety of real-world workloads is another important future work beyond the scope of this paper.




**(a)** A pioneer increases the average CCT to (1.1+2)/2=1.55

**(b)** A straggler increases average CCT to (2+2)/2=2

**Figure 3:** Impact of misidentifications. $C_1$ in light/orange is scheduled before $C_2$ (dark/blue); without errors, each completes in one time unit for an average CCT of 1.5 time units.

# 4 Error-Tolerant Scheduling

Despite its accuracy, the proposed coflow identification procedure (§3) can sometimes misidentify flows from one coflow into another. Unfortunately, such mistakes can have a drastic impact on existing schedulers' performance. In this section, we categorize different types of errors and their impacts on performance (§4.1) and design an error-tolerant coflow scheduler that is robust to misidentifications (§4.2).

## 4.1 Identification Errors and Their Impacts

To assess how identification errors affect scheduling performance, we first divide misidentified flows into two categories based on when they are scheduled:

1. *Pioneers:* Flows that are misidentified into a coflow that is scheduled *earlier* than the parent coflow;
2. *Stragglers:* Flows that are misidentified into a coflow that is scheduled *later* than the parent coflow.

These two types of errors affect the average CCT differently. To illustrate this, we consider a simple scenario in Figure 3, where two identical coflows ($C_1$ and $C_2$) sharing the same bottleneck link arrive at the same time, and each contains 10 identical flows. We further assume that the scheduler assigns $C_1$ with higher priority, and each coflow takes one

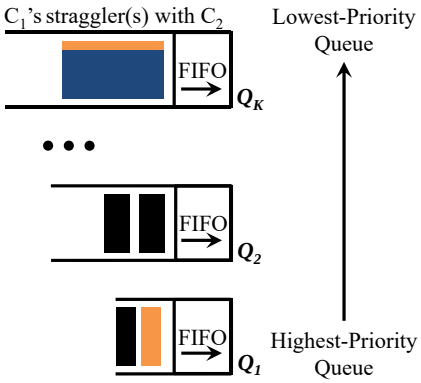



**Figure 4:** Impact of misidentifications on Aalo. Stragglers of high-priority $C_1$ (light/orange) can get stuck with $C_2$ (dark/blue) in a low-priority queue, while other lower-priority coflows (black) compared to $C_1$ complete earlier.

time unit to finish. When there is no identifications error, this schedule leads to an optimal CCT of 1.5 time units.

However, in Figure 3a, a pioneer increases both the CCT of $C_1$ (1.1×) and the average CCT (1.03×). A straggler hurts even more – in Figure 3b, it doubles the CCT of $C_1$ and increases the average CCT by 1.33×.

**Observation 1** *In the presence of misidentifications, stragglers are likely to more negatively affect the average CCT than pioneers.* □

### 4.1.1 Impacts of Identification Errors

Existing coflow schedulers assume prior coflow knowledge [21, 23, 24, 30] for efficient scheduling. However, they can be highly inefficient in the presence of identification errors.

**Clairvoyant Schedulers** Consider Minimum-Allocation-for-Desired-Duration (MADD), the optimal algorithm used in Varys [23] for intra-coflow scheduling when flow sizes are known a priori. MADD slows down all the flows in a coflow to match the completion time of the flow that will take the longest to finish. Because all flows finish together using MADD, a misidentified flow (especially for stragglers) can significantly impact the CCT (e.g., in Figure 3b).

**Non-Clairvoyant Schedulers** Unlike Varys, Aalo [24] uses Discretized Coflow-Aware Least-Attained Service (D-CLAS) – that divides coflows into multiple priority queues and schedules in the FIFO order within each queue – to minimize average CCT without any prior knowledge of flow sizes. However, Aalo can perform even worse in the presence of identification errors. This is because a misidentified flow can drop to a low-priority queue together with another large coflow and can become a "super" straggler.

Figure 4 illustrates such an example. This is not a corner case. Because only 17% coflows create 99% traffic [23], flows from the 83% small coflows can easily be misidentified into the larger ones and suffer performance loss.

**Possible Remedies** In both sets of solutions, intra-coflow scheduling – MADD or per-flow fairness – elongates flows until the end of the entire coflow. However, if we prioritize flows [14, 16, 37, 62] *within* each coflow, a misidentified flow might have a higher chance of finishing earlier. This can decrease the impact of identification errors. For example, in Figure 3b, the expected average CCT would have been 1.75 time units[5] instead of 2 if we performed per-flow prioritization within $C_1$.

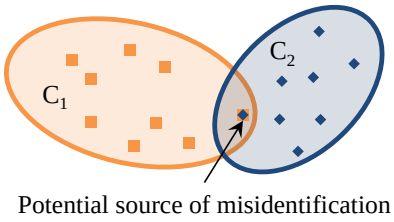



**Figure 5:** Flows falling within multiple coflow clusters during identification can become stragglers if misidentified.

**Observation 2** *Intra-coflow prioritization can matter in the presence of identification errors.* □

## 4.2 Error-Tolerant Coflow Scheduling

Based on the observations in §4.1, in this section, we present the key principles behind designing an error-tolerant coflow scheduler and discuss its components. Our proposed scheduler extends the general structure of a non-clairvoyant coflow scheduler described in Aalo [24].

### 4.2.1 Design Principles

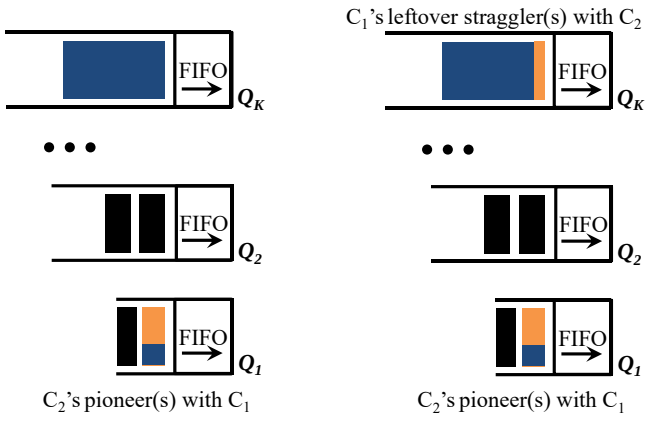
We rely on two key design principles to mitigate the impacts of stragglers and intra-coflow scheduling in the presence of identification errors:

1. *Late binding* errs on the side of caution to reduce the number of stragglers;
2. *Intra-coflow prioritization* leverages per-flow prioritization [14, 16, 37, 62] in the context of coflows.

**Design Principle 1: Late Binding** Observation 1 indicates that avoiding stragglers is key to error-tolerant scheduling. To this end, we take a *late binding* approach toward the coflow relationships identified in the clustering process. For example, consider a flow that can potentially belong to either coflow $C_1$ or coflow $C_2$ – i.e., it lies on the boundary between the two during clustering (Figure 5). Instead of arbitrarily assigning it to either $C_1$ or $C_2$, we delay the decision and consider it to be in both $C_1$ and $C_2$ for the time being. Only during scheduling, we assign it to the higher priority coflow in $C_1$ and $C_2$. Consequently, this flow does not become a straggler to its parent coflow, no matter whether it belongs to $C_1$ or $C_2$.

There can be two outcomes from our decision: (i) if the original classification is correct, we introduce one pioneer in the worst case; (ii) if the original classification is wrong, we effectively prevent this flow from becoming a straggler. Essentially, we try to reduce the number of stragglers at the risk of increasing the number of pioneers. To stop all flows from becoming pioneers, we restrict late binding only to flows that straddle classification boundaries. For example, in Figure 4, instead of creating stragglers for $C_1$, we would instead cause pioneers of $C_2$ that have lower impact on the average CCT (Figure 6a). Our evaluation (§6.4) suggests that this principle

[5] CCT of $C_1$ can be between 1 and 2 time units based on when its straggler flow is scheduled, with an expected CCT of 1.5 time units.




**(a)** Late binding **(b)** Intra-coflow prioritization

**Figure 6:** Impact of CODA's design principles: (a) minimize stragglers by increasing pioneers; and (b) complete individual flows fast to handle leftover stragglers.

contributes to more than 10% CCT improvements in the presence of identification errors, reducing the impact of errors by more than 30%.

**Design Principle 2: Intra-Coflow Prioritization** Although late binding helps, some flows may still be misidentified. An even more troublesome case is when $C_1$ and $C_2$ are clustered as one coflow. To reduce the impact of errors in such cases, we leverage Observation 2, which suggests that intra-coflow prioritization can be more effective in error-tolerant inter-coflow scheduling than in the absence of errors.

To this end, we use per-flow prioritization based on bytes sent (similar to [16]) within each identified coflow without any prior knowledge. This is especially effective for flows from small coflows that become stragglers with large coflows – the most likely case given the prevalence of small coflows [23]. For example, instead of the straggler of $C_1$ taking the same amount of time as longer flows in $C_2$ (Figure 4), it will finish much earlier due to its small size (Figure 6b).

This scheme also takes effect in the reverse scenario – i.e., when a flow from the larger coflow $C_2$ becomes a pioneer with smaller coflow $C_1$. By preferring smaller flows, $C_1$'s flows are likely to finish earlier than the pioneer from $C_2$. Evaluation in §6.4 suggests that this principle brings up to 30% speedup for small coflows under low identification accuracy.

### 4.2.2 *CODA Scheduler*

Putting everything together, Algorithm 2 describes CODA's error-tolerant scheduler, which has the following three components working cooperatively to minimize the impact of stragglers and pioneers during identification as well as to perform intra- and inter-coflow scheduling.

1. **Late Binding** In CoflowExtension($\cdot$), for each identified coflow $C$, we create a corresponding extended coflow $C^*$ by extending its boundary by a diameter $d$ (line 4). Meaning, $C^*$ further includes all flows whose distances to $C$ are smaller than $d$.[6] Note that a flow might belong to two or more extended coflows simultaneously after this step. Later, the flow belonging to multiple coflows will be bound into the coflow with the highest priority when it is scheduled for the first time (line 20).
2. **Inter-Coflow Prioritization** In InterCoflow($\cdot$), we adopt D-CLAS [24] to prioritize across these extended coflows. Basically, we dynamically place coflows into different coflow queues of $Q^C$, and among the queues we enforce prioritization (line 10). Within each queue, we use FIFO among coflows (line 11) so that a coflow will proceed until it reaches queue threshold or completes. Using FIFO minimizes interleaving between coflows in the same queue which minimizes CCTs.
3. **Intra-Coflow Prioritization** In IntraCoflow($\cdot$), we apply smallest-first heuristic [16] to prioritize flows within each coflow. For this purpose, we implement multi-level feedback queue scheduling (MLFQ) among flow queues of $Q^F$ with exponentially increasing thresholds. Such scheme prioritizes short flows over larger ones with no prior knowledge of flow sizes [16]. Flows within each flow queue use max-min fairness (line 19).

**Choice of Diameter $d$** Diameter $d$ reflects the tradeoff be-

**Algorithm 2** CODA's Error-Tolerant Scheduler

```
1:  procedure CoflowExtension((Identified) Coflows ℂ, diameter d)
2:      ℂ* = ∅                          ▷ Set of extended coflows to be returned
3:      for all Coflow C ∈ ℂ do
4:          G = {(Flows) f_i | d(f_i, C) ≤ d}
5:          ℂ* = ℂ* ⋃ {C ⋃ G}          ▷ Extend coflow and add
6:      end for
7:      return ℂ*
8:  end procedure

9:  procedure InterCoflow(Extended Coflows ℂ*, Coflow Queues Q^C)
10:     for all i ∈ [1, |Q^C|] do
11:         for all C* ∈ Q_i^C do       ▷ Q_i^C sorted by arrival time
12:             IntraCoflow(C*, Q^F)
13:         end for
14:     end for
15: end procedure

16: procedure IntraCoflow(Extended Coflow C*, Flow Queues Q^F)
17:     for all j ∈ [1, |Q^F|] do
18:         for all Flows f ∈ C* ⋂ Q_j^F and not yet scheduled do
19:             f.rate = Max-min fair share rate
20:             Mark f as scheduled     ▷ Binds f to the highest priority coflow among all it belongs to
21:             Update the residual bandwidth
22:         end for
23:     end for
24: end procedure

25: procedure CODAScheduler(ℂ, Q^C, Q^F, d)
26:     ℂ* = CoflowExtension(ℂ, d)
27:     InterCoflow(ℂ*, Q^C)
28: end procedure
```

[6] The distance between a flow $f$ and a coflow group $C$ is defined as the smallest distance between $f$ and flows in $C$.

tween stragglers and pioneers. The optimal value of $d$ in terms of average CCT is closely related to the choice of radius $\epsilon$ in identification, and it varies under different traffic patterns. There is no doubt that an extreme value of $d$ (e.g., infinity) will lead to poor CCT. However, as mentioned earlier (§4.1), the impact of stragglers is much bigger than that of pioneers, making late binding beneficial under a wide range of $d$ (§6.4).

# 5 Implementation

In this section, we discuss the difficulties we have faced implementing an existing coflow API in Hadoop 2.7 & Spark 1.6, and describe the implementation of CODA prototype.

## 5.1 Implementing Coflow API

In order to validate CODA, we implemented Aalo's coflow API in Hadoop 2.7 and Spark 1.6 to collect ground truth coflow information. We faced several challenges, including intrusive refactoring of framework code, interactions with third-party libraries to collect coflow information, and Java bytecode instrumentation to support non-blocking I/O APIs.

**Coflow Information Collection** Modern applications are built on top of high-level abstractions such as Remote Procedure Call (RPC) or message passing, rather than directly using low-level BSD socket APIs or equivalent coflow primitives. As a result, matching the high-level coflow information with the low-level flow information requires refactoring across multiple abstraction layers and third-party libraries.

In our implementation of collecting coflows in Hadoop, which implements its own RPC submodule, we: (i) changed the message formats of RPC requests and responses to embed coflow information; (ii) modified the networking library to associate individual TCP connections to the coflow information in the RPC messages; and (iii) added an extra parsing step to look up coflow information in binary messages, since RPC messages are often serialized into byte stream before being passed into the networking level.

To make things worse, there is no universal interface for messaging or RPC. For example, unlike Hadoop, Spark uses third-party libraries: Akka [1] and Netty [6]. Hence, collecting coflow information in Spark almost doubled our effort.

**Supporting Non-blocking I/O** Current coflow implementations [23, 24] emulate blocking behavior in the user space, effectively forcing threads sending unscheduled flows to sleep. As a result, each CPU thread can send at most one flow at any time, which does not scale. To let each thread serve multiple I/O operations, the common practice is to employ I/O multiplexing primitives provided by the OS (e.g., "select" and "poll" in POSIX, and "IOCP" in Windows). Both Hadoop and Spark uses "java.nio" for low-level non-blocking I/O.

Since many popular frameworks (including Hadoop and Spark) are compiled against JVM, we seek an implementation that can support "java.nio" as well as a variety of third party libraries on JVM. To this end, we employed Java bytecode instrumentation – partially inspired by Trickle [10] – to dynamically change the runtime behavior of these applications, collect coflow information, and intercept I/O operations based on scheduling results. Similar to the dynamic linker in Trickle, during the JVM boot, our instrumentation agent is pre-loaded. Upon the first I/O operation, the agent detects the loading of the original bytecode and modifies it to record job IDs in Hadoop and Spark shuffles at runtime, so that coflow information can be collected.

## 5.2 CODA Prototype

Our prototype implements the master-slave architecture shown in Figure 2. The error-tolerant scheduler runs in the master with the information collected from CODA agents. The decisions of the master are enforced by the agents. CODA agent thus has the following two main functions: collection (of flow information) and enforcement (of scheduling decisions).

Flow information collection can be done with a kernel module [16], which does not require any knowledge of how the application is constructed, complying with our goal of application transparency. In prototype implementation, we build upon our coflow API integration, and reuse the same technique (bytecode instrumentation). Instead of job ID, we collect the information for identification: source and destination IPs and ports, as well as the start time of flow.

To enforce the scheduling decisions in each agent, we leverage Hierarchical Token Bucket (HTB) in `tc` for rate limiting. More specifically, we use the two-level HTB: the leaf nodes enforce per-flow rates and the root node classifies outgoing packets to their corresponding leaf nodes.

**Implementation Overhead of CODA Agent** To measure the CPU overheads of CODA agents, we saturated the NIC of a Dell PowerEdge R320 server with 8GB of memory and a quad-core Intel E5-1410 2.8GHz CPU with more than 100 flows. The extra CPU overhead introduced is around $1\%$ compared with the case where CODA agent is not used. The throughput remained the same in both cases.

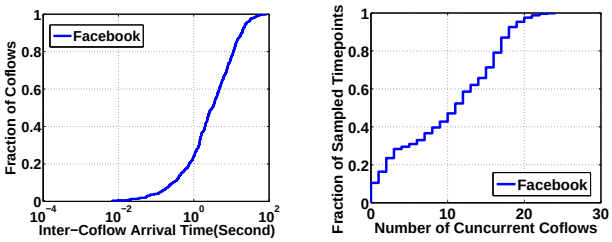
# 6 Evaluation

Our evaluation seeks to answer the following 3 questions:

**How does CODA Perform in Practice?** Testbed experiments (§6.2) with realistic workloads show that CODA achieves over $95\%$ accuracy in identification, improves the average and 95-th percentile CCT by $2.4\times$ and $5.1\times$ compared to per-flow fair sharing, and performs almost as well as Aalo with prior coflow knowledge. Furthermore, CODA can scale up to 40,000 agents with small performance loss.

**How Effective is CODA's Identification?** Large-scale trace-driven simulations show that CODA achieves over $90\%$ accuracy under normal production workloads, and degrades to around $60\%$ under contrived challenging workloads. Furthermore, CODA's distance metric learning (§3.2) is critical, contributing $40\%$ improvement on identification accuracy; CODA's identification speedup design (§3.3) is effective, providing $600\times$ and $5\times$ speedup over DBSCAN and R-DBSCAN respectively with negligible accuracy loss (§6.3).

**How Effective is CODA's Error-Tolerant Scheduling?** Under normal workloads with over $90\%$ identification accuracy, CODA effectively tolerates the errors and achieves comparable CCT to Aalo (with prior coflow information), while




**(a)** Inter coflow arrival time **(b)** No. of concurrent coflows

**Figure 7:** Time-related characteristics of the workload.

outperforming per-flow fair sharing by 2.7×. Under challenging scenarios, CODA degrades gradually from 1.3× to 1.8× compared to Aalo when accuracy decreases from 85% to 56%, but still maintaining over 1.5× better CCT over per-flow fair sharing. Moreover, CODA's error-tolerant design brings up to 1.16× speedup in CCT, reducing the impact of errors by 40%. Additionally, both late binding and intra-coflow prioritization are indispensable to CODA– the former brings 10% overall CCT improvement, while the latter one brings 30% improvement on CCT of small coflows (§6.4).

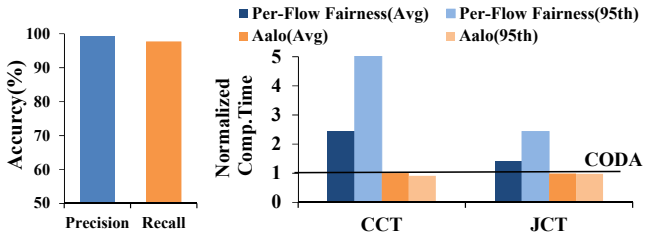
## 6.1 Evaluation Settings

**Testbed** We built a testbed that consists of 40 servers connected to a Pronto 3295 48-port Gigabit Ethernet switch. Each server is a Dell PowerEdge R320 with a 4-core Intel E5-1410 2.8GHz CPU, 8G memory, a 500GB hard disk, and a Broadcom BCM5719 NetXtreme Gigabit Ethernet NIC. Each server runs Debian 8.2-64bit with Linux 3.16.0.4 kernel. We adopted the same compute engine used in both Varys [23] and Aalo [24]. We set coordination interval $\triangle = 100ms$, and set $\epsilon = 100$ and $d = 150$ as default.

**Simulator** For large-scale simulations, we use a trace-driven flow-level simulator that performs a detailed task-level replay of the coflow traces. It preserves input-to-output ratios of tasks, locality constraints, and inter-arrival times between jobs, and it runs at $1s$ decision intervals.

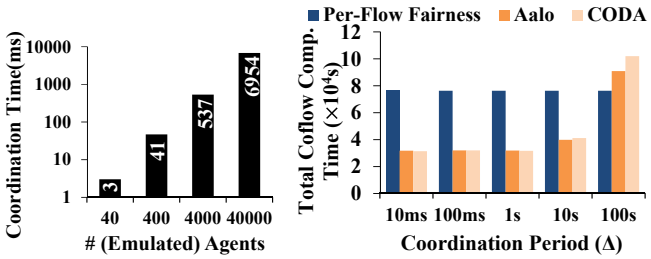
**Workload** We use a realistic workload based on a one-hour Hive/MapReduce trace collected from a 3000-machine, 150-rack Facebook production cluster [5]. The trace contains over 500 coflows ($7 \times 10^5$ flows). The coflow size (1MB−10TB) and the number of flows within one coflow ($1-2\times10^4$) follow a heavy-tailed distribution. Figure 7 plots the distribution of inter-coflow arrival time and the number of concurrent coflows. In our testbed experiments, we scale down jobs accordingly to match the maximum possible 40 Gbps bisection bandwidth of our deployment while preserving their communication characteristics.

However, the Facebook trace does not contain detailed flow-level information such as flow start times and port numbers. To perform a reasonable replay in our simulations, we first run typical benchmarks (e.g., WordCount and PageRank) on Spark and Hadoop in our testbed. Based on the flow arrival time pattern within one coflow we learned from our testbed, we add the start time information back to the Facebook workload to emulate Spark and Hadoop traffic:




**(a)** Accuracy **(b)** CCT and JCT

**Figure 8:** [Testbed] CODA's performance in terms of (a) identification accuracy, and (b) coflow and corresponding job completion times (JCT) compared to Aalo and per-flow fairness. The fraction of JCT jobs spent in communication follows the same distribution shown in Table 2 of Aalo [24].




**(a)** Overheads at scale **(b)** Impact of △

**Figure 9:** [Testbed] CODA scalability: (a) more agents require longer coordination periods (Y-axis is in log scale), and (b) delayed coordination hurts overall performance (measured as sum of CCT).

- **Spark Traffic:** Flows inside each coflow are generated within $100ms$ following a uniform distribution,
- **Hadoop Traffic:** Flows inside each coflow are generated within $1000ms$ following a uniform distribution, and we add an extra exponential delay with a mean of $100ms$.

As to port assignments, they follow the rules described in §3.1 for Spark and Hadoop respectively.

**Metrics** As for identification, we use *precision* and *recall* to measure CODA's accuracy: *precision* is the proportion of the flows which are truly in coflow $C_i$ among all flows classified as in $C_i$, and *recall* is the proportion of flows in $C_i$ which are correctly classified. Finally, the identification *accuracy* is defined as the average of recall and precision.

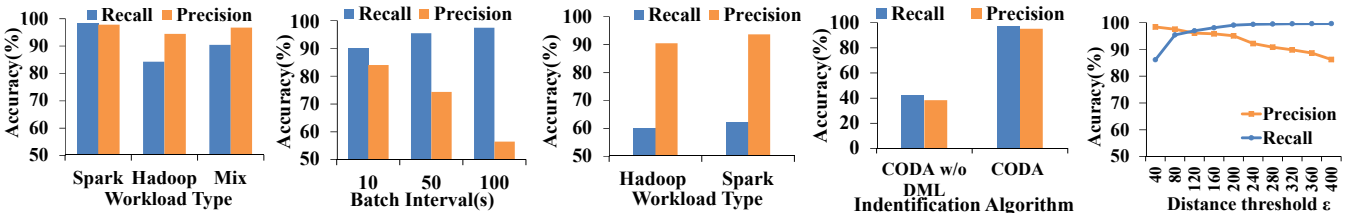
As for scheduling, we measure the coflow completion time (CCT), and compare CODA against Aalo [24] (the state-of-the-art coflow scheduler with manually annotated coflows) and per-flow fair sharing. For easy comparison, we normalize the results by CODA's CCT, i.e.,

$$\text{Normalized Comp. Time} = \frac{\text{Compared Duration}}{\text{CODA's Duration}}$$

Smaller values indicate better performance, and if the normalized completion time of a scheme is greater (smaller) than 1, CODA is faster (slower).

## 6.2 Testbed Experiments

**Performance** For identification, Figure 8a shows that we achieve 99% precision and 97% recall in testbed experiments with the Facebook workload. As for scheduling, Figure 8b




**(a)** Normal workloads **(b)** Batch arrival **(c)** Stretched arrival **(d)** Effectiveness of DML **(e)** Sensitivity to $\epsilon$

**Figure 10:** [Simulation] Inspecting CODA's identifier. Here DML refers to distance metric learning.

shows that CODA reduced the average and 95-th percentile coflow completion times by $2.4\times$ and $5.1\times$ respectively in comparison to TCP-based per-flow fairness. The corresponding improvements in the average job completion time are $1.4\times$ and $2.5\times$. Also, we can see that Aalo has the normalized job and coflow completion times close to 1, meaning that CODA performs almost as well as Aalo.

Additionally, in our testbed, we also ran the SWIM workloads [20] using CODA prototype in Hadoop. A 97% precision and 88% recall is observed, validating the effectiveness of our identifier design. However, due to the disk I/O-intensive nature of the workload and the inherent bottlenecks introduced in the current software implementation of Hadoop, the network is hardly saturated most of the time and scheduling does not provide obvious improvement to CCT.

**Scalability** To evaluate CODA's scalability, we emulated running up to 40,000 agents on our testbed. Figure 9a shows the time to complete a coordination round averaged over 500 rounds for varying number of emulated agents (e.g., 40,000 emulated agents refer to each machine emulating 1000 agents). During each experiment, the coordinator transferred scheduling information for 100 concurrent coflows on average to each of the emulated agents.

As expected, CODA's scalability is not as good as Aalo [24] because of its identification procedure, which does not exist in Aalo. However, we note that our identification speedup already brings big improvement – DBSCAN takes minutes with only 400 agents.

Even though we might be able to coordinate 40,000 agents in $6954ms$, the coordination period ($\triangle$) must be increased. To understand the impact of $\triangle$ on performance, we re-ran the earlier experiments with increasingly higher $\triangle$ (Figure 9b). Note that, to reduce the impact of the long coordination period for small flows, CODA adopts the same method as Aalo – the first 10MB of a flow will go without waiting for coordination. We observe that similar to Aalo, CODA worsens with increasing $\triangle$, and the performance plummeted at $\triangle > 100$s.

## 6.3 Inspecting Identification

**Results under Normal Workloads** We first look at the identification results of CODA under normal workloads. As shown in Figure 10a, we find that CODA achieves high accuracy overall – e.g., it achieves around 97% precision and 98% recall under the Spark traffic, 94% precision and 84% recall for Hadoop. In addition, we observe 97% precision and 92% recall under the mixed traffic (Hadoop/ Spark each accounts for 50%). Comparatively, CODA obtains a higher accuracy in recall for the Spark traffic than that for the Hadoop traffic, which is probably due to its closer inter-flow arrival times (inside one coflow).

**Results under Challenging Scenarios** We observe that time plays a key role in the high accuracy of CODA in the earlier experiment. Specifically, when flows within a coflow come in batches, which usually has a much smaller inter-flow arrival time than the inter-coflow arrival time, they are easier to differentiate. In order to stress CODA, we intentionally increase concurrency by overlapping coflows in two ways:

1. ***Batch arrival*** decreases inter-coflows arrival time. Basically, we create the case where coflows arrive in batch. We set the batch interval to be $10s$, $50s$ and $100s$, and all the coflows in one batch will be condensed with very close arrival times (100-$300ms$). In this way, coflows come in bursts with increased overlaps in each batch.
2. ***Stretched arrival*** increases inter-flow arrival times between flows in one coflow. Specifically, for both Spark and Hadoop traffic, flows are generated with a delay of $5000ms$ following a uniform distribution, and for Hadoop traffic we add an extra exponential delay with a mean of $1000ms$. In this way, flows inside one coflow will spread out over time and overlap more with other coflows. Such scenario represents cases where machines have poor coordination, or when some workers experience late start up.

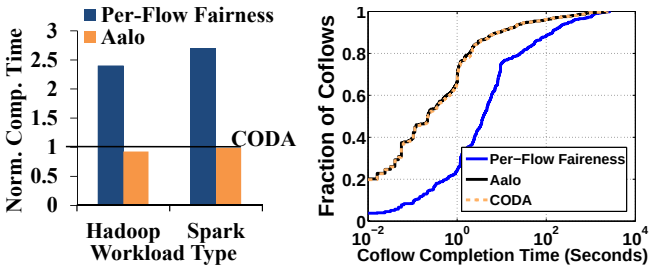
Figure 10b shows the identification results under batch arrival. Here we only focus on the Hadoop traffic, as the exponential delay makes identification more difficult. As expected, we observe an obvious degradation in precision as batch interval increases. For example, the precision decreases from 85% to 56% as the batch interval increases from $10s$ to $100s$. This is because when the traffic becomes more bursty, the number of concurrent coflows increases, making CODA more likely to misclassify unrelated flows into a coflow.

Figures 10c shows the identification results under stretched arrival. We observe that CODA's recall drops to around 60% for both Hadoop and Spark traffic. Due to the large delay added to inter-flow arrival times, flows inside one coflow may have inter-arrival times as large as tens of seconds, which makes it more difficult to classify them to the same coflow. The Hadoop traffic suffers from a lower accuracy due to the $1000ms$ exponential delay.

In addition, we find that the Facebook trace exhibits a unified community. As a result, *the community attribute has little*

| Algorithm | DBSCAN | R-DBSCAN | CODA |
|---|---|---|---|
| Average Identification Time (ms) | 3217.27 | 27.50 | 5.23 |
| Identification Accuracy (%) | 98.21% | 96.47% | 96.41% |

**Table 1:** [Simulation] Effectiveness of CODA's speedup design




**(a)** Normalized CCT **(b)** CCT distribution

**Figure 11:** [Simulation] CODA's scheduler under normal workload.

*effect*. But we envision that concurrent coflows generated by different applications or tenants may be effectively identified via communities even with very close start times.

**Effectiveness of Distance Metric Learning (DML)** To evaluate the effectiveness of DML in §3.2, we run the identification procedure with the same weight assigned to each of the attributes in §3.1. Figure 10d shows the average recall and precision in such case. Compared to CODA, over 40% degraded identification accuracy is clearly observed. This suggests that different attributes contribute differently to the final result, and our DML can effectively distinguish them.

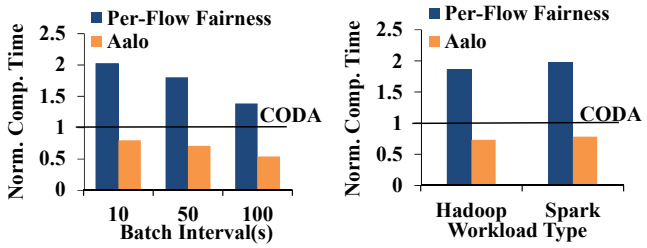
**Impact of Parameter $\epsilon$ (§3.3)** The radius parameter $\epsilon$ is key for CODA (i.e., incremental R-DBSCAN) to determine the number of clusters. Figure 10e shows CODA's performance under varying $\epsilon$.[7] While CODA maintains a high accuracy under a wide range of $\epsilon$, it is not perfect: too small a diameter can misidentify coflows into several small clusters, leading to low recall, while too large a diameter tends to misidentify many coflows into one big cluster, leading to low precision. As time plays a key role in identification, the best choice of $\epsilon$ is closely related to flow and coflow arrival patterns. In general, we believe that an ideal $\epsilon$ should be larger than the average inter-flow arrival time inside one coflow and smaller than the average inter-coflow arrival time.

**Effectiveness of Identification Speedup** We evaluate our design for identification speedup (§3.3) and show the results in Table 1. Compared to DBSCAN, with up to 30 concurrent coflows ($1 \times 10^5$ flows), CODA provides around $600\times$ speedup at the cost of 2% accuracy. Compared to R-DBSCAN, CODA achieves $5\times$ speedup with negligible accuracy loss.

## 6.4 Inspecting Scheduling

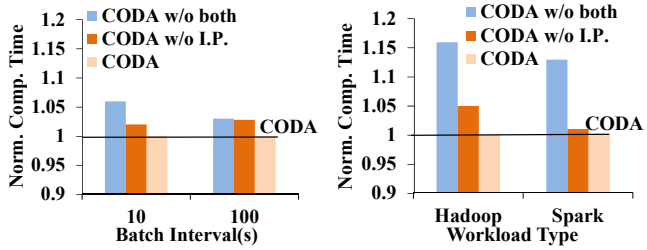
**Results under Normal Workloads** We first inspect CODA scheduler under normal workloads. Figure 11a shows the performance of different scheduling algorithms in terms of normalized CCT. It is evident that CODA effectively tolerates certain identification errors and performs as well as Aalo with correct coflow information, and significantly outperforms per-flow fair sharing. For example, for the Spark traffic, with around 95% identification accuracy (corresponding to Figure 10a), it is not a surprise that CODA performs almost the same as Aalo, and outperforms per-flow fair sharing by $2.7\times$. For the Hadoop traffic, with around 90% accuracy, CODA is slightly worse than Aalo (about $1.1\times$ worse), but still $2.3\times$ better than per-flow fair sharing. To better visualize the result, Figure 11b show the CDF of CCT for the Spark traffic, we can see that CODA almost coincides with Aalo.

[7] We normalized the value of $S_{time}$ to 1 in $A_s$ and $A_h$ (§3.2).




**(a)** Batch arrival case (Hadoop) **(b)** Stretched arrival case

**Figure 12:** [Simulation] CODA's scheduler under challenging scenarios.




**(a)** Batch arrival case (Hadoop) **(b)** Stretched arrival case

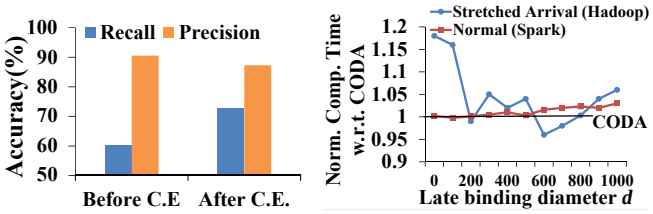
**Figure 13:** [Simulation] Effectiveness of CODA's error-tolerant scheduler. Here "both" refers to late binding (L.B.) and intra-coflow prioritization (I.P.).

**Results under Challenging Scenarios** We next check CODA scheduler under the two challenging scenarios – batch arrival and stretched arrival – described above, where the identification is not as accurate as the normal case.

Figure 12a compares different scheduling algorithms under the batch arrival case using Hadoop traffic. As expected, we observe that with more identification errors introduced, the performance of CODA scheduler degrades gradually. For example, we find that CODA performs around $1.3\times$ to $1.8\times$ worse compared to Aalo with correct information when batch interval equals $10s$ (85% precision in Figure 10b)) to $100s$ (56% precision) respectively. In the meanwhile, CODA is still around $2\times$ to $1.5\times$ better than fair sharing.

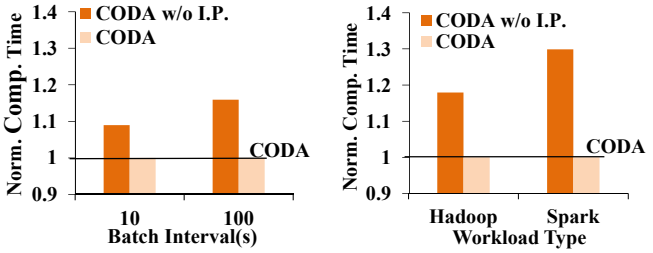
Figure 12b shows the performance of different scheduling algorithms under the stretched arrival case. We observe that for both Spark and Hadoop traffic, even under 60% recall (Figure 10c), CODA performs only around $1.3\times$ worse than Aalo, while outperforming fair sharing by $2\times$.

**Effectiveness of Error-Tolerant Design** We proceed to check the effectiveness of CODA's error-tolerant design in Figure 13. Note that CODA without both late binding and intra coflow prioritization is equivalent to directly adopting Aalo for scheduling with inaccurate identification input. Figure 13a shows that, for batch arrival, the error-tolerant design brings 3–5%




**(a)** L.B. increases recall accuracy **(b)** Sensitivity to parameter $d$

**Figure 14:** [Simulation] Understanding Late binding (L.B.) ).




**(a)** Batch arrival case (Hadoop) **(b)** Stretched arrival case

**Figure 15:** [Simulation] Intra-coflow prioritization brings obvious improvement on small & narrow (SN) coflows.

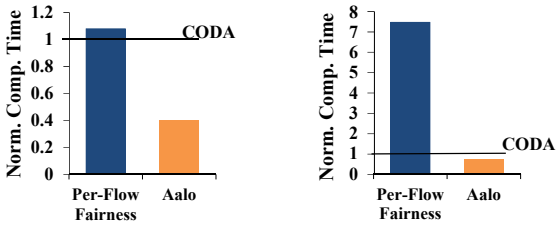
overall improvement in CCT, and especially, it brings 10–20% improvement in CCT for small coflows (shown later in Figure 15a). Furthermore, we observe a bigger improvement in Figure 13b for stretched arrival, where the error-tolerant design provides an overall 1.16× and 1.14× speedup for Hadoop and Spark traffic. Given that CODA is around 1.3× slower than Aalo in this case (Figure 12b), the 1.16× speedup means it reduces the impact of errors by 40%.[8]

Next, we look into independent benefits of late binding and intra-coflow prioritization. We observe that late binding brings non-trivial improvements under stretched arrival – more than 10% for both Hadoop and Spark. Comparatively, intra-coflow prioritization introduces less improvement – 7% for Hadoop under stretched arrival, and 1–5% under other cases. However, we show later that intra-coflow prioritization does bring up to 30% improvement on CCT of small coflows.

**Why does Late Binding Work?** To understand why late binding brings big improvement on CCT, we plot the identification accuracy before/after we extend the identified coflows by a diameter $d$ (i.e., the extended coflow $C^*$ in §4.2.2) in Figure 14a. We observe a 10% improvement in recall at the cost of 4% reduction in precision. Note that a higher recall indicates that more flows in a coflow are successfully classified into one group, which means that coflow extension successfully identifies some stragglers. These identified stragglers will no longer be stragglers after they are bound to the coflow with the highest priority. As a result, late binding can effectively reduce the number of stragglers, thus improving CCT.

**Impact of Parameter $d$ (§4.2.2)** We study how $d$ affects the performance of late binding. In Figure 14b, the blue line refers to the stretched arrival case (Hadoop), where late binding brings obvious improvement. We see that the normalized CCT improves with $d$ at the beginning. This indicates that more stragglers are successfully identified, thereby reducing the CCT of the corresponding coflows. However, as $d$ keeps increasing, late binding introduces too many pioneers, leading to a longer CCT. Moreover, the red line shows the results for the normal workloads (Spark). As the identification is already very accurate, late binding does not provide obvious improvement, and CODA's performance degrades slowly with an increasing $d$. In general, we observe that CODA is stable under a wide range of $d$, and we consider setting $d$ to be a multiple of $\epsilon$ (discussed in §6.3) is a feasible choice.

[8]Calculated as: $\frac{\text{CCT(CODA w/o both) - CCT(CODA)}}{\text{CCT(CODA w/o both) - CCT(Aalo)}} = \frac{1.3\times1.16-1.3}{1.3\times1.16-1} = 40\%$.




**(a)** Overall **(b)** SN coflows

**Figure 16:** [Simulation] CODA's performance w/o identification step.

**How does Intra-Coflow Prioritization Help?** To answer this question, we first categorize coflows based on their lengths and widths. Specifically, we consider a coflow to be *short* if its longest flow is less than 5MB and *narrow* if it has at most 50 flows. We find that around 50% of coflows are short & narrow (SN) coflows. However, their performance cannot be clearly reflected by the overall CCT, as they contribute to less than 0.1% of the total traffic load. Figure 15a shows the normalized CCT of SN coflows in batch arrival. We see that intra-coflow prioritization brings up to 16% improvement. One possible reason is that when many coflows come in batch, CODA is likely to misclassify many coflows as a "super" coflow. Intra-coflow prioritization can effectively speed up SN coflows in such misclassified coflows.

Figure 15b shows the normalized CCT of SN coflows in stretched arrival. The stretched arrival pattern tends to generate many stragglers, and intra-coflow prioritization can effectively speed up stragglers of SN coflows by up to 30%.

**What if Identification is Totally Unavailable?** Finally, we study one extreme case where the entire identification procedure is unavailable (Figure 16a). As we can no longer distinguish coflows, neither inter-coflow prioritization nor late binding takes effect. In such case, intra-coflow prioritization alone still improves coflow completion time by around 8%. Figure 16b further shows that for SN coflows, the improvement can be as large as 7.4×. One important reason for such big improvement is that the Facebook workload is heavy-tailed in terms of coflow sizes.[9] As a consequence, prioritizing small flows can effectively benefit average CCT (especially for SN coflows) as well.

**Remark** We note that our evaluation is restricted by the workload available to us. Thus, we synthesize start times and

[9]Less than 20% coflows contribute to about 99% of the traffic.

perturb arrival times to create different workloads to learn under which workloads CODA works well and under which it does not. First, CODA achieves high accuracy and near-optimal CCT under the normal case, which generally applies to workloads, where the average inter-coflow arrival time is much larger than the inter-flow arrival time inside one coflow. Second, the results under the stretch case indicate that CODA can still achieve comparable CCT to Aalo when workers have poor coordination or experience slow start up. Third, the results under the batch arrival case indicate that CODA does not perform well when coflows have very close start times. We hope that these observations could be helpful in bridging the gap between synthetic workloads and real-world workloads and in providing guidelines for further improvements.

# 7 Discussion

**CODA with DAG Information** The DAG representation of each job and information about the physical location of each task can be useful for coflow identification. However, such information may not always be available. For example, a public cloud operator typically does not communicate with the application masters of tenants. Furthermore, even with the DAG and location information, matching the high-level coflow information with the low-level flow information is non-trivial (§5.1).

However, we believe that coflow identification and scheduling in the presence of such information is an important problem. Particularly, the DAG information can be viewed as an extra attribute, and combining the DAG information with other attributes can potentially increase the identification accuracy, especially for the batched arrival case and for multi-stage jobs. Moreover, as many datacenter workloads are repetitive, it is possible to learn the DAG information instead of directly retrieving it from the master. We consider this as a promising future direction.

**CODA Speedup** Although we spent a lot of efforts in speeding up CODA (§3.3), CODA's scalability is not as good as Aalo due to its identification procedure. To deploy CODA in large datacenters with hundreds of thousands of machines, further speedup is important. We note that one possible way is to parallelize the identification procedure. For example, we would like to see if CODA can benefit from recent proposals on parallel DBSCAN algorithms [36, 53].

# 8 Related Work

**Coflow Scheduling** The coflow abstraction is gaining increasingly more attention in recent years. However, all existing coflow-aware solutions, e.g., [21, 23, 24, 25, 26, 30, 38, 68], require developers to make changes to their applications and manually annotate coflows. CODA challenges this assumption via a combination of application-transparent coflow identification and error-tolerant coflow scheduling.

**Internet Traffic Classification (ITC)** Despite the rich literature in ITC [17, 19, 29, 40, 47, 49, 51, 52, 55, 67], some intrinsic differences prevent us from directly adopting them for coflow identification. First, mutual relations among flows captured by a particular coflow do not recur once its parent job is over; hence, coflows cannot be labeled by predefined categories. In contrast, in traditional traffic classification, traffic typically correspond to stable categories [17, 40, 47, 47, 49, 55, 67]. Second, timeliness is paramount in coflow identification because its result is the input for scheduling. In contrast, belated identification is still useful in many traditional ITC tasks (e.g., intrusion detection).

**Robust Scheduling** We also notice that a similar topic, robust scheduling, has been explored in operations research [27, 39, 42, 61]. However, robust scheduling primarily deals with unexpected events happening during a pre-computed schedule, while error-tolerant scheduling in CODA attempts to schedule task with possibly erroneous input.

# 9 Concluding Remarks

We have presented CODA to automatically identify and schedule coflows *without* any application modifications. CODA employs an incremental clustering algorithm to perform fast, application-transparent coflow identification, and complements it by proposing an error-tolerant coflow scheduling to tolerate identification errors. Testbed experiments and trace-driven simulations show that CODA achieves over $90\%$ identification accuracy, and its scheduler effectively masks remaining identification errors. CODA's overall performance is comparable to Aalo and $2.4\times$ better than per-flow fairness.

In conclusion, this work takes a natural step toward making coflows more practical and usable by removing the need for manual annotations in applications. It also opens up exciting research challenges, including generalization of the identification mechanism beyond data-intensive workloads, decentralization for better scalability, online parameter tuning, handling coflow dependencies, and extending error-tolerant scheduling and allocation algorithms to other resources.

# Acknowledgments

This work is supported in part by the Hong Kong RGC ECS-26200014, GRF-16203715, GRF-613113, CRF- C703615G, and the China 973 Program No.2014CB340303. We thank our shepherd, Nandita Dukkipati, and the anonymous NSDI and SIGCOMM reviewers for their valuable feedback.

# References

[1] Akka. http://akka.io.
[2] Apache Hadoop. http://hadoop.apache.org.
[3] Apache Hive. http://hive.apache.org.
[4] Apache Tez. http://tez.apache.org.
[5] Coflow Benchmark Based on Facebook Traces. https://github.com/coflow/coflow-benchmark.
[6] Netty. http://netty.io.
[7] Presto. https://prestodb.io.
[8] Spark 1.4.1 cluster mode overview. https://spark.apache.org/docs/latest/cluster-overview.html.
[9] Storm: Distributed and fault-tolerant realtime computation. http://storm-project.net.
[10] Trickle. https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/eriksen/eriksen.pdf.

[11] Trident: Stateful stream processing on Storm. http://storm.apache.org/documentation/Trident-tutorial.html.

[12] S. Agarwal et al. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*. 2013.

[13] T. Akidau et al. MillWheel: Fault-tolerant stream processing at Internet scale. *VLDB*, 2013.

[14] M. Alizadeh et al. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM*. 2013.

[15] M. Armbrust et al. Spark SQL: Relational data processing in Spark. In *SIGMOD*. 2015.

[16] W. Bai et al. Information-agnostic flow scheduling for commodity data centers. In *NSDI*. 2015.

[17] L. Bernaille et al. Traffic classification on the fly. *SIGCOMM CCR*, 36(2):23–26, 2006.

[18] P. Bodík et al. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*. 2012.

[19] P. Cheeseman et al. Bayesian classification (AutoClass): Theory and results. 1996.

[20] Y. Chen et al. The case for evaluating mapreduce performance using workload suites. In *MASCOTS*, pages 390–399. 2011.

[21] M. Chowdhury et al. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*. 2011.

[22] M. Chowdhury et al. Coflow: An application layer abstraction for cluster networking. In *Hotnets*. 2012.

[23] M. Chowdhury et al. Efficient coflow scheduling with Varys. In *SIGCOMM*. 2014.

[24] M. Chowdhury et al. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*. 2015.

[25] M. Chowdhury et al. HUG: Multi-resource fairness for correlated and elastic demands. In *NSDI*. 2016.

[26] N. M. M. K. Chowdhury. *Coflow: A Networking Abstraction for Distributed Data-Parallel Applications*. Ph.D. thesis, University of California, Berkeley, 2015.

[27] R. L. Daniels et al. Robust scheduling to hedge against processing time uncertainty in single-stage production. *Management Science*, 41(2):363–376, 1995.

[28] J. Dean et al. Mapreduce: Simplified data processing on large clusters. In *OSDI*. 2004.

[29] A. P. Dempster et al. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 1–38, 1977.

[30] F. R. Dogar et al. Decentralized task-aware scheduling for data center networks. In *SIGCOMM*. 2014.

[31] M. Ester et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*. 1996.

[32] S. Ghemawat et al. The google file system. In *SOSP*. 2003.

[33] A. Ghoting et al. SystemML: Declarative machine learning on mapreduce. In *ICDE*. 2011.

[34] J. E. Gonzalez et al. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*. 2014.

[35] C. Guo et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM*. 2015.

[36] Y. He et al. Mr-dbscan: an efficient parallel density-based clustering algorithm using mapreduce. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 473–480. IEEE, 2011.

[37] C.-Y. Hong et al. Finishing flows quickly with preemptive scheduling. In *SIGCOMM*. 2012.

[38] V. Jalaparti et al. Network-aware scheduling for data-parallel jobs: Plan when you can. In *SIGCOMM*. 2015.

[39] V. Jorge Leon et al. Robustness measures and robust scheduling for job shops. *IIE transactions*, 26(5):32–43, 1994.

[40] T. Karagiannis et al. BLINC: multilevel traffic classification in the dark. In *SIGCOMM*. 2005.

[41] M. Kornacker et al. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*. 2015.

[42] P. Kouvelis et al. Robust scheduling of a two-machine flow shop with uncertain processing times. *Iie Transactions*, 32(5):421–432, 2000.

[43] T. Kraska et al. MLbase: A distributed machine-learning system. In *CIDR*. 2013.

[44] Y. Low et al. GraphLab: A new framework for parallel machine learning. In *UAI*. 2010.

[45] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297. 1967.

[46] G. Malewicz et al. Pregel: A system for large-scale graph processing. In *SIGMOD*. 2010.

[47] A. McGregor et al. Flow clustering using machine learning techniques. In *PAM*. 2004.

[48] X. Meng et al. MLlib: Machine learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[49] A. W. Moore et al. Internet traffic classification using Bayesian analysis techniques. 33(1):50–60, 2005.

[50] D. G. Murray et al. Naiad: A timely dataflow system. In *SOSP*. 2013.

[51] T. T. Nguyen et al. Training on multiple sub-flows to optimize the use of machine learning classifiers in real-world IP networks. In *LCN*. 2006.

[52] T. T. Nguyen et al. A survey of techniques for Internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, 10(4):56–76, 2008.

[53] M. Patwary et al. A new scalable parallel dbscan algorithm using the disjoint-set data structure. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[54] M. P. Robillard. What makes APIs hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[55] M. Roughan et al. Class-of-service mapping for QoS: a statistical signature-based approach to IP traffic classification. In *IMC*. 2004.

[56] A. Roy et al. Inside the social network's (datacenter) network. In *SIGCOMM*. 2015.

[57] C. Scaffidi. Why are APIs difficult to learn and use? *Crossroads*, 12(4):4–4, 2006.

[58] V. K. Vavilapalli et al. Apache Hadoop YARN: Yet another resource negotiator. In *SoCC*. 2013.

[59] P. Viswanath et al. Rough-DBSCAN: A fast hybrid density based clustering method for large data sets. *Pattern Recognition Letters*, 30(16):1477–1488, 2009.

[60] U. Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[61] J. Wang. A fuzzy robust scheduling approach for product development projects. *European Journal of Operational Research*, 152(1):180–194, 2004.

[62] C. Wilson et al. Better never than late: meeting deadlines in datacenter networks. In *SIGCOMM*. 2011.

[63] R. S. Xin et al. Shark: SQL and rich analytics at scale. In *SIGMOD*. 2013.

[64] E. P. Xing et al. Distance metric learning with application to clustering with side-information. In *NIPS*. 2002.

[65] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. 2012.

[66] M. Zaharia et al. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*. 2013.

[67] S. Zander et al. Automated traffic classification and application identification using machine learning. In *LCN*. 2005.

[68] Y. Zhao et al. RAPIER: Integrating routing and scheduling for coflow-aware data center networks. In *INFOCOM*. 2015.