Bulletin of the Technical Committee on

Data Engineering

December 2017 Vol. 40 No. 4

IEEE Computer Society

Letters

Special Issue on Global-scale Data Management

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

ICDE 2018

The IEEE International Conference on Data Engineering will be held in April 14-19, 2018 in Paris, France. This is the flagship conference of the Computer Society's Technical Committee on Data Engineering. It is a great conference, at a great location. What could possibly be better than April in Paris at ICDE! I am attending and hope to see you there.

The Current Issue

In the olden days (the past millenium), the notion that data management could scale across the globe was considered far-fetched. There were no known application scenarios in which it arose. And further, it was widely considered to be a technological pipe dream. The current issue then serves as a reminder of how rapidly our world is changing, and how rapidly our technology capabilities are growing. Social networks, travel services, and e-commerce are simply the tip of the iceberg (the iceberg being perhaps a bit smaller due to global warming) of a growing number of global-scale applications, including machine learning, which was not even considered an application in the previous millenium.

In addition to the number of global-scale applications growing, our technology to deal with them is advancing by "leaps and bounds". Not only can we scale out our data management infrastructure, but we can enforce strong invariants on the data being managed. The critical consistency requirement here is frequently atomic transactions. Transactions at global scale were nowhere in sight before the year 2000 but are now supported by several web scale vendors.

This is where the current issue of the Bulletin comes in. Tim Kraska has assembled an issue that captures some of the exciting research activity on global-scale data management. This area has become a "hot topic" for both researchers and practitioners. It has become part of our online "life", though usually behind the scenes, as so much of data management is. But being a technical member of our data management community means it is essential that you understand this technology. I can strongly recommend this issue to you. It is a real "eye opener" into some of the critical directions our community is and will be exploring. I want to thank Tim for exposing this hugely important technology to us.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

Globalization has significantly altered the way in which the world operates. More than ever before are businesses required to operate at a global scale with customers expecting fast and seamless access to applications independent of where they are located. Similar, also the availability requirements for applications have changed over the last years. Whereas a decade ago, it was still sufficient to be able to sustain a single machine failure, nowadays mission-critical applications are expected to tolerate an entire data-center outage. Thus, it is no longer sufficient to think about data management at the scale of a few servers in some basement. Rather, we need to think about data management at the global-scale; from how is data replicated across data centers, over how bandwidth and latency of slow wide-area replication can be avoided or hidden to not negatively impact the application latency, up to how local regularization, e.g., privacy regulations in Europe, impacting what data can be stored at which location.

In this issue, we brought together an exciting collection of recent work in the space of global-scale data management addressing issues from reducing latency to how to execute machine-learning algorithms across data-center boundaries. The first paper, "A System Infrastructure for Strongly Consistent Transactions on Globally-Replicated Data" by Faisal Nawab and others, describes a first system infrastructure for global-scale transaction processing with serializability guarantees. The authors analyze the fundamental limits of executing transactions across data-center boundaries and develop a theoretical framework of the optimality of strongly-consistent transaction latency. Based on those findings, the authors present a transaction-processing protocol which takes advantage of the previously determined lower bounds on the transaction latency.

The second paper, "Writes: the dirty secret of causal consistency" by Lorenzo Alvisi and others, discusses in detail an alternative geo-replication strategy, which is based on causal consistency. Causal consistency has several advantages over other (stronger) consistency guarantees as it is able to provide high availability even in the presence of network partitioning. At the same time, causal consistency is still not widely adopted. The authors argue that one of the main reasons for this is, that current implementations do not handle write operations, especially write-skew, efficiently. Finally, by proposing two system, TARDiS and Occult, the authors aim to overcome this issue and make causal consistency more broadly applicable.

The third paper, "Cost-Effective Geo-Distributed Storage for Low-Latency Web Services" by Zhe Wu and Harsha V. Madhyastha, addresses the issue of the cost-latency trade-off by spreading the data across data centers of multiple cloud providers. Furthermore, the authors propose additional techniques, such as augmenting every GET/PUT request with a set of redundant requests to mitigate the effect of isolated latency spikes, among other techniques to reduce the high cost of geo-replication.

The final paper, "Towards Geo-Distributed Machine Learning" by Ignacio Cano and others, makes the case for Geo-Distributed Machine Learning (GDML). Regulatory requirements often prohibit to move data out of a country, but at the same time, many Machine-Learning applications require a global view of the geo-distributed data in order to achieve the best results. Thus, the authors propose a system architecture for geo-distributed training and show by means of an empirical evaluation on three real datasets that GDML is not only possible but also advisable in many scenarios.

I would like to thank all authors for their insightful contributions to this special issue. Happy reading!

Tim Kraska
Brown University
Rhode Island, US

A System Infrastructure for Strongly Consistent Transactions on Globally-Replicated Data

Faisal Nawab Vaibhav Arora Victor Zakhary Divyakant Agrawal Amr El Abbadi

Department of Computer Science, University of California, Santa Barbara
Santa Barbara, CA 93106
{nawab,vaibhavarora,victorzakhary,agrawal,amr}@cs.ucsb.edu

Abstract

Global-scale data management (GSDM) empowers systems by providing higher levels of fault-tolerance, read availability, and efficiency in utilizing cloud resources. This has led to the emergence of global-scale data management and event processing. However, the Wide-Area Network (WAN) latency separating datacenters is orders of magnitude larger than typical network latencies, and this requires a reevaluation of many of the traditional design trade-offs of data management systems. Therefore, data management problems must be revisited to account for the new design space. In this paper, we revisit the problem of supporting strongly-consistent transaction processing for GSDM. This includes providing an understanding of the limits imposed by the WAN latency on transaction latency in addition to a design of a system framework that aims to reduce response time and increase scalability. This infrastructure includes a transaction processing component, a fault-tolerance component, a communication component, and a placement component. Finally, we discuss the current challenges and future directions of transaction processing in GSDM.

1 Introduction

The cloud computing paradigm promises high-performance 24/7 service to users dispersed around the world for cloud applications. Achieving this is threatened by complete datacenter outages and the physical limitations of both the datacenter infrastructure and wide-area communication. To overcome these challenges, systems are increasingly being deployed on multiple datacenters spanning large geographic regions. The replication of data across datacenters (geo-replication) allows requests to be served even in the event of complete datacenter-scale outages. Likewise, distributing the processing and storage across datacenters brings the application closer to users and sources of data, enabling higher levels of availability and performance. Moving to global-scale data management (GSDM), despite its benefits, raises many novel challenges. One of the main sources of these challenges is the large WAN communication latency, which is orders of magnitude larger than traditional communication latency (See Figure 1). This invalidates the traditional space of design trade-offs and makes the communication latency the dominating bottleneck.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

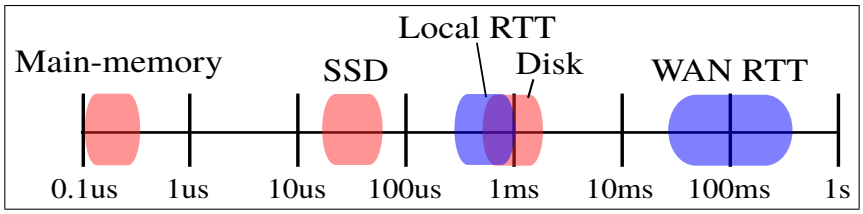


Figure 1: Latency of the Wide-Area Network Round-Trip Time communication (WAN RTT) compared to memory access latency [27] and network latency within the datacenter (local RTT)

When coordination is involved in the data management task, the effect of WAN communication latency is amplified. This makes transaction processing—a coordination-intensive task—a victim of geo-replication's large communication latency. The significance of transaction processing to data management systems and the seriousness of the WAN communication challenge have led many researchers from both academia and industry to design new and improved transaction processing protocols specifically for geo-replication [2, 3, 6–10, 13, 15, 16, 18–20, 22–25, 28, 31, 35]. Many of these efforts explored weakening consistency guarantees to lower the coordination demands [7–9, 13, 15, 16, 18, 19, 22, 24, 31] or explored coordination-free approaches to avoid the cost of wide-area coordination [2, 28, 35]. However, a large-body of work tackled the problem of global-scale transaction processing while maintaining the stringent consistency guarantees of traditional data management systems [3, 6, 10, 20, 23, 25]. The argument for maintaining stringent guarantees is that they manifest easy-to-use abstractions for concurrent access to data. Serializability, for example, ensures an outcome equivalent to a serial execution [4], ridding the program developers from worrying about concurrency anomalies.

In this paper, we present a system infrastructure design for global-scale transaction processing with serializability guarantees. We adopt a holistic approach where in addition to the transaction processing engine, we improve the design of the surrounding system components that affects the performance of transactions. We begin the paper by discussing the fundamental limits of transaction processing in geo-replication and develop a theoretical framework of the optimality of transaction latency. Then, we present transaction processing protocols that leverage the newfound knowledge about transaction latency optimality to achieve better performance in geo-replication. Then, we present two components outside of the transaction processing engine that are essential for transactions processing efficiency in geo-replicated systems. The first is global-scale communication and the second is data placement. We conclude the paper with an outlook on the future of geo-replicated transaction processing in the context of the advancements in global-scale data management systems.

2 Transaction Processing

In this section, we present the development of transaction processing protocols that target reducing transaction latency on geo-replicated data. This begins by understanding the source of the latency inefficiency in traditional transaction processing systems (Section 2.1). Then, we develop a lower-bound formulation for transaction latency on geo-replicated data (Section 2.2) and use this newfound understanding to develop Helios [23], a transaction processing protocol that achieves the lower-bound (Section 2.3).

2.1 The Latency Limit of the Request-Response Paradigm

There is a fundamental coordination latency limit due to the polling nature of traditional protocols that we call the Request-Response paradigm. In the Request-Response paradigm, the coordination for a request starts *after* the request is made, where the node making the request polls other nodes to inquire about their state and detect conflicts. The request is served only after receiving a *response* from other replicas. This makes a Round-Trip

Time (RTT) of communication inevitable—an expensive cost in GSDM. This applies to both centralized and quorum-based protocols.

The limit of the Request-Response paradigm leads to the question: *Is it possible to avoid the Request-Response paradigm limit on coordination or is it a fundamental limit on performance?* Our work Message Futures [21] demonstrates the possibility of breaking the limit of the Request-Response paradigm by an observation that coordination of future requests can start before they arrive. As requests arrive, they are assigned to a predetermined future coordination point. We call this approach *Proactive Coordination*. Coordination points are judiciously calculated to ensure conflicts are detected. A coordination point still needs an RTT for coordination. However, because a request is assigned to a coordination point that already started, the request's observable latency is less than RTT. Message Futures is the first protocol that shows the possibility of faster-than-RTT coordination for all the replicas of a distributed system. Also, it introduces Proactive Coordination, a new approach to coordination that overcomes the limitations of the Request-Response paradigm.

2.2 Theoretical Lower-Bound on Coordination Latency

Breaking the RTT latency barrier via the Proactive Coordination paradigm invalidates the previously held convention that coordination cannot be performed faster than the RTT latency. Thus, it opens the question: *What is the lower-bound on coordination latency?* This is a fundamental question in understanding the extent of the effect of the wide-area latency limit on coordination latency. Such a lower-bound, if proven, will provide system designers and researchers with a theoretical foundation on what is achievable by current and future systems.

Our objective now is to develop a lower-bound on commit latency of transactions on replicated data stores *while maintaining serializability* [4]. Maintaining serializability requires coordination between replicas (datacenters in our case). The communication latency necessary for this coordination imposes a limit on *commit latency*, which is the time duration to decide whether a transaction commits or aborts. Achieving low commit latency is the focus of this study. Consider two datacenters A and B with unique commit latencies L_A and L_B, respectively. We show in this section that the *summation* of L_A and L_B must be at least the Round-Trip Time (RTT) between A and B. Note that this is a summation which means that the commit latency of a datacenter can be lower than RTT.

The lower-bound result extends to larger groups of datacenters by applying the lower-bound to all pairs in the group. This will allow us to judge whether the group of datacenters can commit with a certain set of commit latency values. We are particularly interested in minimizing the average commit latency of all datacenters. We call the minimum average latency a *Minimum Average Optimal* (MAO) latency or optimal latency for short.

2.2.1 Theoretical model and assumptions

We consider a theoretical model that consists of datacenters with communication links connecting them. Each transaction undergoes two phases. First, the transaction is issued and it becomes visible to the datacenter. At that stage it is called a *preparing* transaction. Then, at a later time the datacenter decides whether it commits or aborts and it becomes a *finished* transaction. The time spent as a preparing transaction is the *commit latency*.

The following are the assumptions on communication and computation for this model.

- *Compute power:* Infinite compute power is assumed in the model. The datacenter does not experience any overhead in processing and storing transactions. We make this assumption to focus our attention on communication overhead.

- *Communication links:* Sending a message through a link takes a specific latency to be delivered to the other end. Links are symmetric and take the same amount of time in both directions. Note that different links could have different latencies. However, triangle inequality must hold.

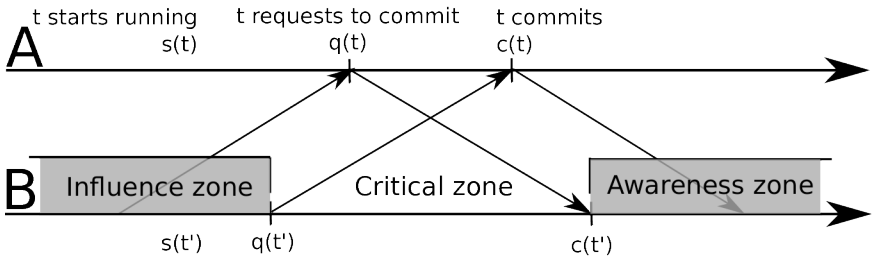


Figure 2: Two transactions, t and t', executing in a scenario with two datacenters

- *Arbitrary read-write transactions:* All datacenters have no restrictions on their choice or order of objects to be read or written in a transaction. Additionally, each transaction must have at least a single write operation. Thus, the model does not apply to optimizations for read-only transactions and disjoint data manipulation techniques. Also, transactions must try to commit, hence aborting all transactions is not allowed.

- *Knowledge:* Each datacenter A knows precisely every preparing and finished transaction that exists at another datacenter B up to the current time minus half the RTT between them, *i.e.*, now - $\frac{RTT(A,B)}{2}$. This reflects the fastest time a datacenter knows about any event in another datacenter. In a realistic setting this is a lower bound of such knowledge.

- *Commit latency:* We assume that the commit latency at each datacenter is fixed. This assumption simplifies the presentation. The discussions can be extended to the general case by taking each point in time in isolation.

2.2.2 Lower-bound proof

Intuition. For any two concurrent conflicting transactions, at least one of them must be able to detect the other before committing. Otherwise, both transactions will commit, which could result in incorrect executions. Here, we show that there is a lower-bound on commit latency. If the commit latency is lower than the lower-bound, then two conflicting transactions could commit without detecting each other, thus possibly violating correctness.

Formulation. Consider two datacenters A and B and a transaction t executing at datacenter A and transaction t' executing at datacenter B that could be conflicting with t. Figure 2 shows these transactions. In the figure, $s(t)$ is the transaction's start time, $q(t)$ is the commit request time, and $c(t)$ is the commit time. Transaction t's read and write-set are visible at the commit request time. Given the knowledge assumption, B knows about t starting from time $q(t) + \frac{RTT(A,B)}{2}$. Transaction t is preparing from time $q(t)$ until time $c(t)$ when it is committed. Three *zones* are defined at datacenter B with respect to t: (1) The *awareness zone* where B can possibly know about t, (2) The *influence zone* where B's transactions can be known to t, and (3) the *critical zone* where B is neither in the awareness nor influence zone.

Lemma 1: The sum of the commit latencies of two datacenters is greater than or equal to the RTT between them, *i.e.*, $L_A + L_B \geq RTT(A, B)$, where L_X is the commit latency at datacenter X.

Now, we present a description of the intuition behind this lemma (based on the proof in the original paper [23]). Consider the scenario in Figure 2 . The time when t requests to commit is $q(t)$ and the time it commits is $c(t)$, *i.e.*, $c(t) - q(t) = L(t)$. Based on transaction t, we can divide the timeline in B to three regions: (1) *Awareness zone:* this zone contains the events at B that can be affected by t. This zone starts from the earliest point of time

Protocol	L_A	L_B	L_C	Average
Leader-based (A leader)	0	30	20	16.67
Leader-based (C leader)	20	40	0	20
Majority	20	30	20	23.33
Optimal (MAO)	5	25	15	15

Table 1: Possible commit latencies, L_A, L_B and L_C, for three datacenters with Round-Trip Times $RTT(A, B) = 30$, $RTT(A, C) = 20$, and $RTT(B, C) = 40$.

when B can receive t, which is $q(t) + \frac{RTT(A,B)}{2}$. (2) *Influence zone:* this zone contains the events that can affect the outcome of t. An event can affect the outcome of t if it is received before the commit point, $c(t)$. Events at B that can arrive by the commit point are ones with a timestamp lower than $c(t) - \frac{RTT(A,B)}{2}$. Events at B with a higher timestamp cannot be received at A by time $c(t)$ because half an RTT is required to communicate. (3) *Critical zone:* this zone represents the time duration that is neither in the awareness zone nor in the influence zone. We postulate that it is impossible for a transaction to have requested to commit and commits in the critical zone. To show this, consider a transaction t' at B that requests to commit and commits in the critical zone. Transaction t' will not affect the outcome of t, since t' is not in the influence zone. Also, t will not affect t', since t' is not in the awareness zone. However, t' can conflict with t. Since t' is not aware of t and t is, likewise, not aware of t', both transactions successfully commit. This potentially results in an inconsistency, and is thus not allowed. To summarize, a transaction that starts at the beginning of the critical zone at B cannot commit with a commit latency smaller than the duration of the critical zone. This duration is equal to $RTT(A, B) - L(t)$. Thus, *the sum of L_A and L_B must be greater than or equal to $RTT(A, B)$.*

The lower-bound shows a direct trade-off between the commit latencies of two datacenters. Given this lower-bound we are now able to judge whether a set of commit latencies are *achievable* or violates the lower-bound for scenarios with more than two datacenters by applying the lower-bound to each pair of datacenters.

Example. Consider an example of three datacenters, A, B, and C. The RTTs between the datacenters are: $RTT(A, B) = 30$, $RTT(A, C) = 20$, and $RTT(B, C) = 40$. Table 1 shows four achievable commit latencies and the average commit latency of the datacenters. The first two represent a leader-based replication approach, where a single leader is responsible for committing transactions. In this approach, the leader commits immediately, and the other datacenters commit latencies are the RTT to the leader. Note how each pair of datacenters satisfies the lower-bound, *e.g.*, when A is the leader $L_A + L_B = 30 = RTT(A, B)$. The third row represents a majority replication approach. For the case of three datacenters, the commit latency of a datacenter is the RTT to the nearest datacenter. These replication protocols experience different average commit latencies: 16.67, 20, and 23.33. However, the minimum average commit latency (MAO) that is achievable for this scenario is 15. The fourth row in the figure show the commit latencies, L_A, L_B, and L_C, that achieve an average commit latency of 15 while not violating the lower-bound.

MAO solutions, such as the one in the previous example, can be derived using the following linear programming formulation:

Definition 2: (Minimum Average Optimal)
The Minimum Average Optimal commit latencies for n datacenters is derived using a linear program with the following objective and constraints:

$$\begin{array}{lll} \text{Minimize} & \sum_{A \in R} & L_A \\ \text{subject to} & \forall_{A,B \in R} & L_A + L_B \geq RTT(A, B) \\ \text{and} & \forall_{A \in R} & L_A \geq 0 \end{array}$$

where R is the set of datacenters. This formulation follows directly from Lemma 1. Minimizing the latency

is our objective and the constraints are the correctness conditions that commit latencies are not negative and Lemma 1 is satisfied. We will use this methodology to derive the commit latency values used with the Helios commit protocol. This linear program can be adapted to objectives other than average latency [23].

2.2.3 Summary

The lower-bound result shows that the coordination latency can be faster than what is previously achieved by traditional protocols and even faster than what is achieved by Message Futures. The model of coordination, in addition to being essential for deriving the lower-bound, advances our understanding of the cost of global-scale coordination. It also brings a newfound understanding of the latency characteristics of traditional and Proactive Coordination protocols.

2.3 Achieving the Lower-Bound Transaction Latency

The lower-bound model inspired a protocol based on Proactive Coordination, called **Helios** [23] that theoretically achieves the lower-bound, thus proving that the lower-bound is tight. Experimental evaluation shows that Helios approaches the lower-bound in a real global-scale deployment. Next, we provide an intuition of how Helios achieves the lower-bound and refer the interested reader to the full paper [23].

To provide an intuition of the Helios commit protocol, consider the scenario in Figure 2. The figure shows the timeline of two datacenters, A and B. At A, a transaction t is issued at time $q(t)$ and committed at time $c(t)$. Transaction t commits immediately after receiving a log of events from B that is shown as an arrow going from B to A. This log carries transactions that were issued up to the time of sending the log, including transaction t' (assume t' is issued at the time of log transmission). The time the log was sent from B is $q(t')$. $q(t')$ is also the commit request time of t'. Helios receives the log in order (via a FIFO channel), meaning that all transactions, preparing or finished, at B prior to or at time $q(t')$ are known to A at time $c(t)$.

Transactions at B must not conflict with t. The approach to avoid conflicts is influenced by the way the lower-bound latency was developed in Section 2.2. However, here we do not make any assumptions regarding clock synchronization or communication. Rather, we rely on the exchanged event logs and received transaction timestamps. A transaction, t', at B is either issued during the influence zone, critical zone, or awareness zone. If t' starts during the influence zone, then transaction t will detect it because the log will contain a record of t'. If t' starts in the awareness zone, then it will detect t. Thus, for these two cases, conflicts will be detected. An undetected conflict can arise only if t' starts *and* commits within the critical zone. Thus, if t' is issued in the critical zone, Helios must ensure that it does not commit until it is in the awareness zone, which means that B will detect the conflict between t and t'.

3 Global-Scale Data Communication

In the previous section, we have tackled the problem of coordination latency and building transaction commit protocols that ameliorate the impact of the large WAN communication latency. However, large applications that receive large amounts of requests may face the problem of *scaling* inter-datacenter communication. Typical communication protocols used by data management systems are not built for WAN environments and large communication demands. This leads to the increase in the demand and stress on the network I/O, which translates into significant communication latency overhead. To combat this challenge, we developed Chariots [22], a scalable inter-datacenter communication platform. Chariots implement a communication layer to be used by transaction processing engines such as Message Futures and Helios that we presented earlier. Chariots maintains the causal order of the communicated events. This is because Message Futures and Helios rely on a causally consistent communication solution called the Replicated Dictionary [33]. However, these communication solutions cannot scale, and Chariots offer a scalable alternative for causally-consistent communication systems.

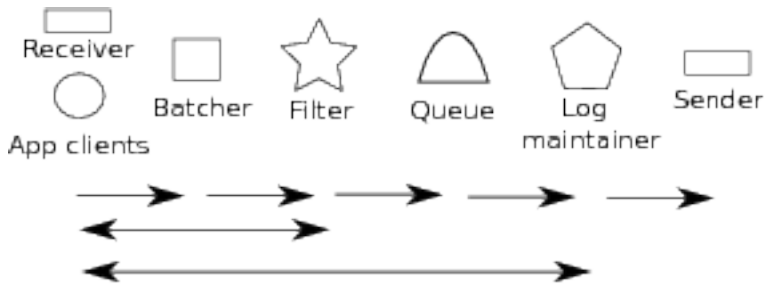


Figure 3: The components of the multi-data center shared log. Arrows denote communication pathways in the pipeline.

To enable higher levels of scalability, Chariots adapts the following design features:

- (1) Highly-available stateless control. This approach has been identified as the most suitable to scale out in cloud environments [11]. Communicated data is represented in the form of immutable records in a distributed shared log. The architecture of Chariots consists of three types of machines: *Log maintainers* are responsible for persisting the log's records and serving read requests. *Indexers* are responsible of access to log maintainers. Finally, control and meta-data management is the responsibility of a highly-available cluster called the *Controller*.

- (2) Fast data ingestion. To achieve this, Chariots implements a distributed log that allows inserting records arbitrarily to any log node without coordination with other log nodes. All coordination is done lazily in the background, in summarized batches, to reduce the impact on ingesting new records.

- (3) Efficient causal-dependency enforcement. To achieve this, a pipeline design is adopted to process inter-datacenter communication. Chariots pipeline consists of six stages depicted in Figure 3. The first stage contains nodes that are generating records. These are Application clients and machines receiving the records sent from other datacenters. These records are sent to the next stage in the pipeline, *Batchers*, to batch records to be sent collectively to the next stage. *Filters* receive the batches and ensure the uniqueness of records. Records are then forwarded to Queues where they are ordered. After it is ordered, a record is forwarded to a log node that constitutes the Log maintainers stage. The local records in the log are read from the log nodes and sent to other datacenters via the *Senders*.

  The arrows in Figure 3 represent the flow of records. Generally, records are passed from one stage to the next. However, there is an exception. Application clients can request to read records from the Log maintainers. Chariots support elastic expansion of each stage to accommodate increasing demand. Thus, each stage can consist of more than one machine, *e.g.*, five machines acting as Queues and four acting as Batchers.

4 Global-Scale Data Placement

The success of the cloud model has led to the continuing increase of the number of public cloud providers in addition to the increase in the amount of resources offered by these providers. This includes an increase in the number of available physical datacenters, which raises the question: *at which datacenters should data be placed?* This is called the Global-Scale Data placement problem, or placement problem for short. The challenging aspect of the placement problem, is that developers want to optimize a diverse set of objectives that are sometimes in conflict, such as monetary cost and performance. This problem has been tackled by providing frameworks to reason about placement decisions [1, 17, 26, 30, 32]. However, these solutions are not suitable for

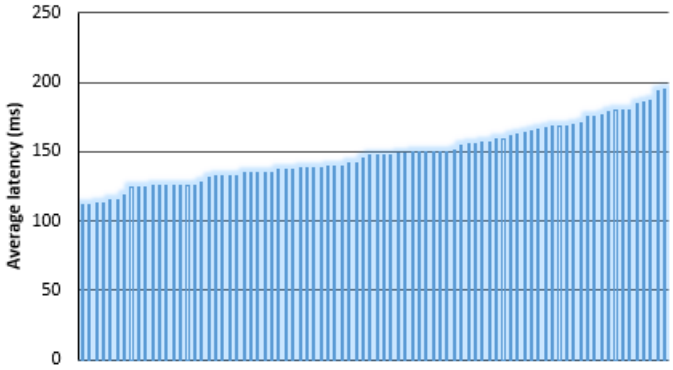


Figure 4: The average latency, of all the clients in 9 datacenters, to reach the closest quorum (2 out 3) for all the possible $\binom{9}{3} = 84$ different placements sorted by latency.

our system infrastructure because they target systems with relaxed access abstractions [1, 17, 26, 32] or systems with different coordination patterns [30].

To fill the gap in global-scale placement systems, we present DB-Risk [34], a data placement framework that targets multi-leader strongly-consistent transactional systems. DB-Risk embeds the commit protocol constraints into an optimization to derive both the data placement and the commit protocol configurations that minimize the overall transaction latency. Figure 4 shows the effect of only changing the placement on the average obtained latency for all the clients in a multi-leader protocol (*i.e.*, Paxos). Clients are equally distributed at nine of Amazon's datacenters in California, Oregon, Virginia, Sao Paulo, Ireland, Sydney, Singapore, Tokyo, and Seoul. As shown in Figure 4, changing only the placement while fixing all the other parameters (the protocol, the workload distribution, etc.) can lead to a significant change of 1.75x between the minimum and the maximum reported average latency.

In developing DB-Risk, we discovered counter-intuitive lessons about data placement and transaction execution practices. The most notable lesson is what we call *Request Handoff*, which is choosing the datacenter that will drive the execution of the transaction (the coordinator). The transaction latency is mainly affected by the distance between the client and the coordinator, the distance between the coordinator and the other participants, and finally the number of communication rounds required between the coordinator and the participants to serve the request. Different transaction management protocols choose the coordinator based on some intuitive heuristics, such as choosing the closest replica to the client. However, the choice of the coordinator can, in fact, drastically affect the request latency.

Request Handoff, in addition to other lessons, can be used to achieve better performance by being aware of the latency diversity and asymmetry of the WAN links. They are also applicable to both Paxos-based commitment protocols [3, 20, 25] and leader-based commitment protocols [6, 10]. DB-Risk incorporates these lessons, the commitment protocol constraints, and the application requirements in an optimization to find the placement that minimizes the average transaction latency.

5 The Future of Global-Scale Transaction Processing

In this section, we discuss some of the open challenges and future directions in the area of global-scale transaction processing.

Edge Resources. Emerging edge datacenter technologies, such as micro datacenters and cloudlets, have the potential to bring data even closer to end users. This has motivated many mobility and Internet of Things (IoT) systems to adopt the *edge computing model*, also called by other names such as fog computing [5]. However, *for*

data management tasks that require updating data, many edge computing systems still rely on storage solutions that do not utilize edge datacenters. We envision extending GSDM system beyond traditional datacenters to cover edge datacenters. Such a direction will complement and enhance existing edge computing frameworks by providing a unified storage abstraction at the edge that is globally synchronized and supports transaction processing. Edge data management enjoys the benefits—and faces the challenges—of edge computing [29]. For example, in this new model, the data management system can leverage edge datacenters for tasks such as maintaining copies for fault-tolerance and to offload smaller instances to serve requests at edge location close to users. The incorporation of edge datacenters changes the model of geo-replication. The number of replicas is no longer confined to a small number of datacenters, rather to a potentially large number of edge datacenters. Also, the Round Trip Time (RTT) between a datacenter and nearby edge datacenters is an order of magnitude lower than typical inter-datacenter RTT. Significant research and practical effort is needed to accommodate GSDM systems to this new environment. However, utilizing edge datacenters will improve the performance of GSDM systems and will enable emerging edge and mobile applications.

Flexible Fault-tolerance. Even with GSDM and efficient coordination, achieving a low end-to-end latency is a challenge. This is due to the needed latency to coordinate access between different replicas to maintain consistency, and the latency to replicate data across datacenters for fault-tolerance. Reducing and controlling the latency of coordination by relaxing consistency has been the topic of many research efforts including ours. *The remaining frontier is investigating relaxing fault-tolerance to reduce and control the need for synchronous WAN communication.* We argue that analogous to how some applications may have relaxed consistency requirements, some applications may have relaxed fault-tolerance requirements. We envision an exploration of the trade-off between fault-tolerance and latency in the context of edge data management, while preserving strongly consistent abstractions. Such exploration may lead to methods to control the fault-tolerance level in a way that result in achieving higher performance for weaker fault-tolerance levels. Also, there may be methods to relax the requirements of fault-tolerance and find a spectrum of durability guarantees with various performance characteristics.

Emerging WAN Techniques. Leveraging advances in WAN research from the networking community is an important step towards building efficient GSDM systems. Networking techniques, like Software-defined Networking (SDN), are now being applied to the context of WANs (*e.g.*, BwE [14] and B4 [12]). A promising opportunity is to develop GSDM systems that integrate these advances in networking.

6 Concluding Remarks

We believe that to make GSDM a reality, it is essential to provide intuitive and easy-to-use abstractions for application developers. Our goal is to enable web and cloud programmers to build their applications to benefit from the opportunities of GSDM. To achieve this, we focused on providing abstractions at the database layer that are intuitive by providing strong consistency, thus ridding the programmer from thinking about concurrency, replication, and other complexities related to the GSDM environment.

Each one of our proposed protocols treats wide-area coordination as the main bottleneck. This approach has turned to be rewarding in terms of novel designs that achieve a much higher performance than traditional counterparts in GSDM environments. We believe that the principles we propose in these protocols will impact the design of a wide-range of problems that share the aspect of wide-area coordination.

We handle strongly consistent transaction processing in GSDM. We have proposed a theoretical formulation of the performance limit imposed by wide-area latency, in addition to a transaction management paradigm called proactive coordination. The theoretical formulation enables finding a lower-bound transaction latency in global-scale environments. This lower-bound result provides a guide to system designers and researchers to reason about latency limits in multi-datacenter environments. Proactive coordination is a paradigm for transaction commit protocols where coordination for future transactions start before they are issued. We have shown how

this general concept can be used to implement two GSDM transaction commit protocols: (1) Message Futures, that breaks the RTT barrier for transaction latency, and (2) Helios, that is able to achieve transaction latency close to the theoretical lower bound. Message Futures and Helios combine traditional concurrency control approaches such as the use of timestamp ordering, log propagation, loose-time synchronization, and certification with the concepts that we propose in the paper such as proactive coordination and lower-bound latency.

In the course of developing global-scale systems, we encountered a common challenge in managing communication between globally-distributed nodes. This motivated Chariots, our work to provide a communication platform for GSDM systems. Chariots maintains a shared log abstraction between nodes in various datacenters. Chariots targets both scalability within the datacenter and across datacenters. Within the datacenter, Chariots includes a distributed shared log system, which removes coordination from the path of appending operations. Chariots then provides a framework to replicate distributed, shared logs across datacenters. Chariots guarantees causal order of events in the shared log, which is sufficient to implement strongly consistent protocols on top of it, including Message Futures and Helios. Also, we addressed the data placement problem of geo-replicated databases and find that a special treatment for strongly-consistent systems is needed. We envision that the presented designs will aid and inspire future protocols in global-scale data management.

References

[1] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 2–2, Berkeley, CA, USA, 2010. USENIX Association.

[2] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, Nov. 2014.

[3] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 223–234, 2011.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. pages 251–264, 2012.

[7] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 715–726. VLDB Endowment, 2006.

[8] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[9] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 173–184, Washington, DC, USA, 2013. IEEE Computer Society.

[10] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 15–28, New York, NY, USA, 2011. ACM.

[11] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. *Proc. VLDB Endow.*, 7(12):1259–1270, Aug. 2014.

[12] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM.

[13] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[14] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 1–14, New York, NY, USA, 2015. ACM.

[15] Y. Lin, B. Kemme, M. Patiño Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 419–430, New York, NY, USA, 2005. ACM.

[16] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris. Enhancing edge computing with database replication. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, SRDS '07, pages 45–54, Washington, DC, USA, 2007. IEEE Computer Society.

[17] G. Liu and H. Shen. Minimum-cost cloud storage service across multiple cloud providers. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 129–138. IEEE, 2016.

[18] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[19] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[20] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9):661–672, July 2013.

[21] F. Nawab, D. Agrawal, and A. El Abbadi. Message futures: Fast commitment of transactions in multi-datacenter environments. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[22] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi. Chariots: A scalable shared log for data management in multi-datacenter cloud environments. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 13–24, 2015.

[23] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi. Minimizing commit latency of transactions in geo-replicated data stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1279–1294, New York, NY, USA, 2015. ACM.

[24] G. Pang, T. Kraska, M. J. Franklin, and A. Fekete. Planet: Making progress with commit processing in unpredictable environments. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 3–14, New York, NY, USA, 2014. ACM.

[25] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *Proc. VLDB Endow.*, 5(11):1459–1470, July 2012.

[26] F. Ping, J.-H. Hwang, X. Li, C. McConnell, and R. Vabbalareddy. Wide area placement of data replicas for fast and highly available data access. In *Proceedings of the fourth international workshop on Data-intensive distributed computing*, pages 1–8. ACM, 2011.

[27] M. K. Qureshi, S. Gurumurthi, and B. Rajendran. *Phase Change Memory: From Devices to Systems*. Morgan & Claypool Publishers, 1st edition, 2011.

[28] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1311–1326, New York, NY, USA, 2015. ACM.

[29] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[30] A. Sharov, A. Shraer, A. Merchant, and M. Stokely. Take me to your leader!: Online optimization of distributed storage configurations. *Proc. VLDB Endow.*, 8(12):1490–1501, Aug. 2015.

[31] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[32] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 292–308, New York, NY, USA, 2013. ACM.

[33] G. T. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 233–242, New York, NY, USA, 1984. ACM.

[34] V. Zakhary, F. Nawab, D. Agrawal, and A. El Abbadi. Db-risk: The game of global database placement. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 2185–2188, New York, NY, USA, 2016. ACM.

[35] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM.

Writes: the dirty secret of causal consistency

Lorenzo Alvisi[‡] Natacha Crooks[†‡] Syed Akbar Mehdi[†]
[†]The University of Texas at Austin [‡]Cornell University

Abstract

Causal consistency offers geo-distributed systems what ought to be a sweet option between the poor performance of strong consistency and the weak guarantees of eventual consistency. Yet, despite its appealing properties, causal consistency has seen limited adoption in industry, where systems have instead been clustering around the two extremes of eventual and strong consistency. We argue that this reluctance stems primarily from how causal consistency handles writes—both in how they are propagated, and in how conflicting writes are applied. We present our experience in designing and building two recent systems, Occult [25] and TARDiS [11], that try to address or mitigate these problems, and highlight some of the open challenges that remain in this space.

1 Introduction

Causal consistency appears to be ideally positioned to respond to the needs of geographically replicated datastores that support today's large-scale web applications. It strikes a sweet point between the high latency of stronger consistency guarantees [10], and the programming complexity of eventual consistency [4, 7, 27]. Indeed, unlike eventual consistency, causal consistency preserves operation ordering and gives Alice assurance that Bob, whom she has defriended before posting her Spring-break photos, will not be able to access her pictures, even though Alice and Bob access the photo-sharing application using different replicas [5, 9, 20]. Crucially, causal consistency provides these guarantees while continuing to provide applications the ALPS properties [20] of availability, low latency, partition tolerance and high scalability.

These appealing properties have generated much interest in the research community. In the last few years, we have learned that no guarantee stronger than real-time causal consistency can be provided in a replicated datastore that combines high-availability with convergence [23], and that, conversely, it is possible to build causally-consistent datastores that can efficiently handle a large number of shards [3, 6, 15, 16, 20, 21]. Likewise, we have established increasingly sophisticated techniques [11, 14, 20, 23, 24, 32, 34] to ensure the convergence of objects to which conflicting updates have been applied updates.

Perhaps surprisingly, the interest has been rather more tepid in industry: to this day, causal consistency is rarely used in production, with companies preferring either strictly weaker guarantees like those provided in Riak [7], Redis Cluster [1], and Cassandra [4], or, at the other extreme, stronger guarantees like those offered by Google Spanner [10], FaunaDB [17], or CockroachDB [19]. As of the time of writing, the only commercially

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

available systems that report supporting causal consistency are AntidoteDB [30][1], Neo4j [28][2], MongoDB [27][3] and CosmosDB [26]. This, despite Facebook suggesting that causal consistency is a feature that they would like to support [2, 22]! We submit that industry's reluctance to deploy causal consistency is in part explained by the inability of its current implementations to handle *write operations* efficiently, while aligning with the semantics of the applications that run on top of these systems. Unlike reads, which can be served from any "up-to-date" replica [20,21,33] and return a state that includes the effects of all operations that could have influenced that state, writes come with few promises: existing causally consistent systems neither guarantee when a given write will eventually become visible, nor how merging conflicting writes on individual objects will affect the semantics of the global system state.

Write visibility In existing causal systems, such as COPS [20] or Eiger [21], a datacenter performs a write operation only after applying all writes that causally precede it. This approach guarantees that reads never block, as all replicas are always in a causally consistent state, but, in the presence of slow or failed shards, may cause writes to be buffered for arbitrarily long periods of time. These failures, common in large-scale clusters, can lead to the *slowdown cascade* phenomenon, where a single slow or failed shard negatively impacts the entire system, delaying the visibility of updates across many shards and leading to growing queues of delayed updates [2, 25]. Slowdown cascades thus violate a basic commandment for scalability: do not let your performance be determined by the slowest component in your system.

Merging of write-write conflicts Geographically distinct replicas can issue conflicting operations, and the write-write conflicts that result from these operations may cause replicas' states to diverge. To insulate applications from this complexity, systems like COPS [20] or Dynamo [14] attempt to preserve the familiar abstraction that an application evolves sequentially through a linear sequence of updates: they aggressively enforce per-object convergence, either through simple deterministic resolution policies, or by asking the application to resolve the state of objects with conflicting updates as soon as conflicts arise. These techniques, however, provide no support for meaningfully resolving conflicts between concurrent sequences of updates that involve *multiple* objects: in fact, they often destroy information that could have helped the application in resolving those conflicts. For example, as we describe further in §2.2, deterministic writer-wins [35], a common technique to achieve convergence, hides write-skew from applications, preventing applications from correcting for the anomaly. Similarly, exposing multivalued objects without context as in Dynamo [14] obscures cross-object semantic dependencies.

In the rest of this paper, we discuss in greater detail the effect of slowdown cascades on existing causal systems (§2.1), and the semantic challenges that are left unaddressed by current techniques for merging concurrent conflicting write operations (§2.2). We then sketch the outline of Occult (§3.1) and TARDiS (§3.2), two systems designed to mitigate the aforementioned issues, and conclude (§4) by highlighting the challenges that arise when attempting to combine insights from these two systems.

2 The challenges

2.1 Propagating writes

Causal consistency derives its performance advantage over stronger consistency guarantees from its ability to asynchronously propagate and apply writes to replicas. The flip side of this flexibility is the unpredictability of the relative timing at which causally dependent updates to different shards are ultimately applied at a remote replica. Not only causally dependent updates may reach distinct remote shards out of order, but updates may be arbitrarily delayed when shards experience performance anomalies (because of abnormally-high read or write

[1] Currently in alpha release.

[2] As of release 3.1.

[3] As of the release 3.6.

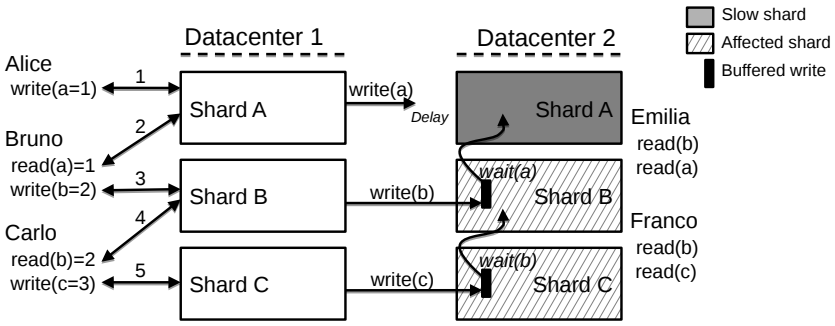


Figure 1: Example of a slowdown cascade in traditional causal consistency. Delayed replicated write(a) delays causally dependent replicated write(b) and write(c)

traffic, partially malfunctioning hardware, a congested top-of-rack switch, etc.) that are not only inevitable but indeed a routine occurrence in any system of sufficient size [12, 13].

Ideally, any drawback caused by such delays would be limited to the affected shard, and not spill over to the rest of the system. Unfortunately, to the best of our knowledge, all existing causally consistent systems [6, 16, 20, 21, 38] are *inherently* susceptible to spill overs, which ultimately can snowball into a system-wide *slowdown cascade*.

This vulnerability is fundamental to the design of these systems: they delay applying a write until after all writes that causally precede it have been applied. For example, Eiger [21] asks each replicated write w to carry metadata that explicitly identifies all writes that happened before w and have a single-hop distance from w in the causal dependency graph. When a replica receives w, it delays applying it until after all of w's dependencies have been applied; these dependencies in turn are delayed until *their* single-hop dependencies are applied; and so on). The visibility of a write within a shard can then become dependent on the timeliness of other shards in applying their own writes. Figure 1 illustrates this with an example. Shard A of DC_2 lags behind in applying the write propagating from DC_1, all shards in DC_2 must also wait before they make their writes visible. Shard A's limping inevitably affects Emilia's query, but also unnecessarily affects Franco's, which accesses exclusively shards B and C.

Industry has long identified the spectre of slowdown cascades as one of the leading reasons behind its reluctance to build strongly consistent systems [2, 8], pointing out how the slowdown of a single shard, compounded by *query amplification* (e.g., a single user request in Facebook can generate thousands of, possibly dependent, internal queries to many services), can quickly cascade to affect the entire system.

Figure 2 shows how a single slow shard affects the size of the queues kept by Eiger [21] to buffer replicated writes. Our setup is geo-replicated across two datacenters in Wisconsin and Utah, each running Eiger sharded across 10 physical machines. We run a workload consisting of 95% reads and 5% writes from 10 clients in Wisconsin and a read-only workload from 10 clients in Utah. We measure the average length of the queues buffering replicated writes in Utah. Larger queues mean that newer replicated writes take longer to be applied. If all shards proceed at approximately the same speed, the average queue length remains stable. However, if *any* shard cannot keep up with the arrival rate of replicated writes, then the average queue length across *all* shards grows indefinitely.

2.2 Merging conflicting writes

Propagating writes is not the only challenge that causal consistency faces: in multi-master systems (like COPS [20], Dynamo [14], Cassandra [4], Riak [7], or Voldemort [36]), conflicting writes can execute concurrently at different sites. The challenge is then not simply how to propagate these writes efficiently, but how to resolve

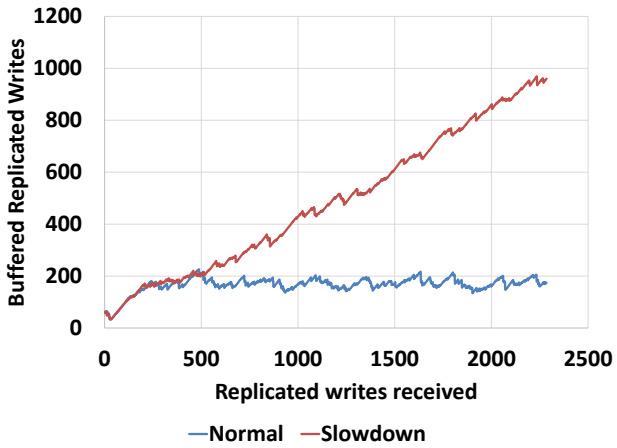


Figure 2: Average queue length of buffered replicated writes in Eiger under normal conditions and when a single shard is delayed by 100 ms.

the write-write conflict that arises from these operations. Though causal consistency elegantly guarantees that Emilia will observe a consistent state, causal consistency does not specify how to handle conflicting write operations. Write-write conflicts define a class of conflicts that causal consistency cannot prevent, detect, or repair.

To illustrate, consider the process of updating a Wikipedia page consisting of multiple HTML objects (Figure 3(a)). The page in our example, about a controversial politician, Mr. Banditoni, is frequently modified, and is thus replicated on two sites, A and B. Assume, for simplicity, that the page consists of just three objects—the content, references, and an image. Alice and Bruno, who respectively strongly support and strongly oppose Mr. Banditoni, concurrently modify the content section of the webpage on sites A and B to match their political views (Figure 3(b)). Carlo reads the content section on site A, which now favors Mr. Banditoni, and updates the reference section accordingly by adding links to articles that praise the politician. Similarly, Davide reads the update made by Bruno on site B and chooses to strengthen the case made by the content section by updating the image to a derogatory picture of Mr. Banditoni (Figure 3(c)). Eventually, the operations reach the other site and, although nothing in the preceding sequence of events violates causal consistency, produce the inconsistent state shown in Figure 3(d): a content section that exhibits a write-write conflict; a reference section in favor of Mr. Banditoni; and an image that is against him. Worse, there is no straightforward way for the application to detect the full extent of the inconsistency: unlike the explicit conflict in the content sections, the discrepancy between image and references is purely semantic, and would not trigger an automatic resolution procedure. To the best of our knowledge, this scenario presents an open challenge to current causally consistent systems. We conjecture that these systems struggle to handle such write-write conflicts for two reasons: they choose to syntactically resolve conflicts, and do not consider cross-object semantics.

Syntactic conflict resolution. We observe that the majority of weakly consistent systems use fixed, syntactic conflict resolution policies to reconcile write-write conflicts [4, 20] to maintain the abstraction of sequential storage. Returning to our previous example of the popular merging policy of deterministic writer-wins (DWW): DWW resolves write-write conflicts identically at all sites and ensures that applications never see conflicting writes, which guarantees eventual convergence. In our example, however, guaranteeing eventual convergence would not be sufficient to restore consistency: this policy would choose Bruno's update, and ignore the relationship between the content, references, and images of the webpage. Such greedy attempt at syntactic conflict resolution is not only inadequate to bridge this semantic gap, but in fact can also lose valuable information for reconciliation (here, Alice's update).

Lack of cross-object semantics. Some systems, like Dynamo [14] or Bayou [34], allow for more expressive

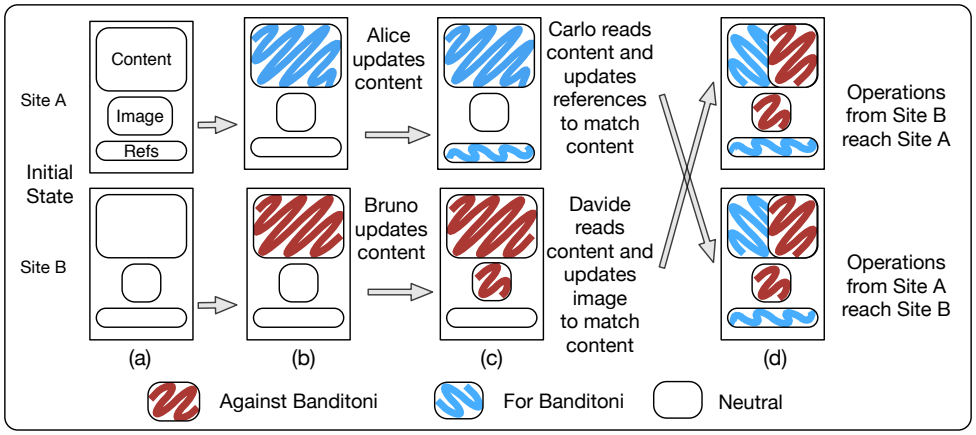


Figure 3: Weakly-consistent Wikipedia scenario - A webpage replicated on two sites with asynchronous replications. The end state is a write-write conflict on the content and an inconsistent web-page

conflict resolution policies by pushing to conflict resolution to the application [14, 34], but on a per-object basis only. This approach allows for more flexible policies than a purely syntactic solution, but reduces conflict resolution to the merging of explicitly conflicting writes. As a result, it is often still overly narrow. For example, it would not detect or resolve inconsistencies that are not write-write conflicts, but instead result indirectly from conflicts between two writes, such as the one between the references and the image. Per-object reconciliation policies fail to consider that the effects of a write-write conflict on an object do not end with that object: Carlo and Davide update references and images as they do because they have *read* the conflicting updates to the original content section. Indeed, any operation that depends on one of two conflicting updates is potentially incompatible with all the operations that depend on the other: the shockwaves from even a single write-write conflict may spread to affect the state of the entire database.

To the best of our knowledge, there is currently no straightforward way for applications to resolve consistently the kind of multi-object, indirect conflicts that our example illustrates. Transactions [20, 21], an obvious candidate, are powerless when the objects that directly or indirectly reflect a write-write conflict are updated, as in our example, by different users. After Bruno's update, the application has no way to know that Davide's update is forthcoming: it must therefore commit Bruno's transaction, forcing Bruno's and Davide's updates into separate transactions. Nor would it help to change the granularity of the object that defines the write-write conflict—in our example, by making that object be the entire page. It would be easy to correspondingly scale up the example, using distinct pages that link each other. Short of treating the entire database as the "object", it is futile to try to define away these inconsistencies by redrawing the objects' semantic boundaries.

In essence, conflicting operations fork the entire state of the system, creating distinct *branches*, each tracking the linear evolution of the datastore according to a separate thread of execution or site. The Wikipedia for example, consists of two branches, one in support of Banditoni at Site A, and one against the politician (Site B).

3 Towards a solution?

3.1 Preventing slowdown cascades through client-centric causal consistency

The example (Figure 1 in §2.1) illustrates why causal consistency can be subject to slowdown cascades despite replicating writes asynchronously: writes share the fate of all other writes on which they causally depend. If one write is slow to replicate, all subsequent writes will incur that delay. Though this delay may sometimes be necessary - Emilia must wait for the delayed write to observe a causally consistent snapshot of the system -

inheriting the delays of causally preceding writes can also introduce gratuitous blocking. Franco, for instance, never reads Alice's write (a): delaying writes (b) and (c) until (a) is replicated is thus unnecessary. Otherwise said, observing writes (b) and (c) while write (a) is in flight does not lead to a consistency violation.

This observation, though seemingly fairly innocent, is in fact key to alleviating the problem of slowdown cascades in causal consistency, as it precisely captures what causal consistency *requires*. Causal consistency defines a contract between the datastore and its users that specifies, for a given set of updates, which values the datastore is allowed to return in response to user queries. In particular, it guarantees that each client observes a monotonically non-decreasing set of updates (including its own), in an order that respects potential causality between operations. Causal consistency thus mandates that Franco, upon observing write (c), also observes write (b), but remains silent on the fate of write (a). Existing causally consistent systems, however, enforce internally a stronger invariant than what causal consistency requires: to ensure that clients observe a monotonically non-decreasing set of updates, they evolve their data store *only* through monotonically non-decreasing updates. It is this strengthening that leaves current implementations of causal consistency vulnerable to slowdown cascades.

To resolve this issue, we propose to revisit what implementing causal consistency actually requires: a system is causally consistent from the point of view of a client if all executed queries return results that are consistent with a causal snapshot of the system. The underlying system itself need never store a causally consistent state, as long as it appears *indistinguishable* from a system that does!

To this effect, we designed a system, Occult (**O**bservable **C**ausal **C**onsistency Using **L**ossy **T**imestamps) that shifts the responsibility for the enforcement of causal consistency from the datastore to those who actually perceive consistency anomalies—the clients. Moving the output commit to clients allows Occult to make its updates available as soon as it receives them, without having to first apply all causally preceding writes. Causal consistency is then enforced by clients on reads, but only for those updates that they are actually interested in observing. In our example, Occult empowers Franco to independently determine that the result of its query is causally consistent: in general, Occult clients can access states of replicas that may not yet reflect some of the (unrelated) writes that were already reflected in a replica they had previously accessed.

Taking this read-centric approach to causal consistency may seem like a small step, but pays big dividends: as Occult is no longer compelled to delay writes to enforce consistency, Occult is impervious to slowdown cascades. This approach also conveys other benefits: Occult, for instance, no longer needs its clients to be sticky, a flexibility that is useful in real-world systems like Facebook, where clients sometimes must bounce between datacenters because of failures, load balancing, and/or load testing [2]).

At first blush, moving the enforcement of causal consistency to clients may appear fairly straightforward. Each client c in Occult maintains metadata to encode the most recent state of the datastore that it has observed. On reading an object o, c determines whether the version of o that the datastore currently holds is safe to read (i.e., if it reflects all the updates encoded in c's metadata). The datastore keeps, together with o, metadata of its own to encode the most recent state known to the client that created that version of o. If the version is deemed safe to read, then c needs to update its metadata to reflect any new dependency; if it is not, then c needs to decide how to proceed (among its options: try again; contact a master replica guaranteed to have the latest version of o; or trade safety for availability by accepting a stale version of o).

In reality, however, implementing client-centric causal consistency is non-trivial: the size of this metadata can quickly grow to have prohibitive cost. Vector clocks, the traditional way of capturing causal dependencies, require in principle an entry per each tracked object, a clearly unacceptable proposition in the large-scale systems that Occult targets. It is to sidestep this issue that current causally consistent datastores prefer the pessimistic approach to causal consistency: if a write can never be applied to a replica until all previous causally related writes have been replicated, it is sufficient to track the *nearest-writes* on which an operation depends [5, 20]. The core logic of Occult is thus geared towards minimizing metadata size while retaining the flexibility and resiliency to slowdown cascades that the read-centric approach conveys.

Occult makes this possible through a core technique: *compressed causal timestamps*. Causal timestamps are, essentially, vector clocks that, rather than tracking dependencies at the granularity of individual objects, do

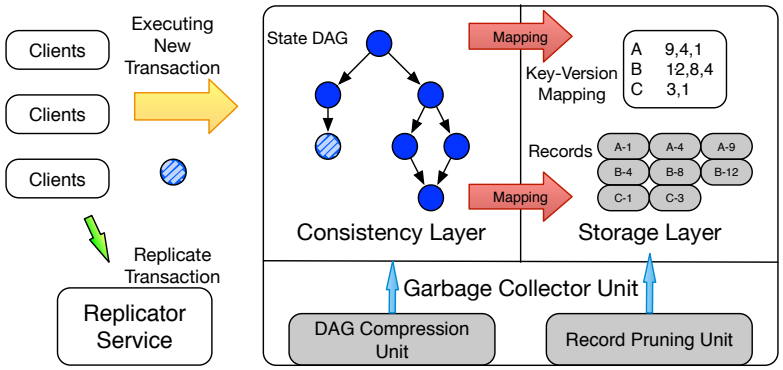


Figure 4: TARDiS architecture

so at the granularity of datacenter shards: each timestamp entry identifies the number of known writes from the corresponding shard. Occult uses causal timestamps to (i) encode the most recent state of the datastore observed by a client and (ii) capture the set of causal dependencies for write operations. Though conflating dependency tracking for all objects stored in a shard to to a single entry reduces overhead, they are still far from practical in large scale systems, which can have tens of thousands of shards. The challenge that Occult takes on is to devise a way to *compress* causal timestamps without significantly reducing their ability to track causal dependencies accurately.

To this end, Occult synthesizes a number of techniques that leverage structural and temporal properties to strike a sweet spot between metadata overhead and accuracy: for instance, using real-time rather than counters as entries in the causal timestamps allows Occult to eliminate timestamp inaccuracies that result from different shards seeing different write throughput. Likewise, Occult observes that more recently updated entries in the causal timestamp are more likely to generate spurious dependencies than older ones. Rather than using compression techniques that apply uniformly to all entries of a causal timestamp, Occult focuses the majority of its metadata budget to accurately resolve dependencies on the shards with the most recently updated vector entry. Specifically, clients assign an individual entry in their causal timestamp to the $n-1$ shards with the most recently updated vector entry they have observed; all other shards are mapped to the vectors "catch-all" last entry. Though very coarse, conflating to a single timestamp entry the tracking of all the shards that have not been recently updated is likely to cause few consistency check failures: with high likelihood, that timestamp entry naturally reflects updates that have already had enough time to be accepted by every replica. The conference paper describing Occult [25] offers more details about these techniques and discusses the system's transactional capabilities.

Our experience suggests that Occult performs well: we find that our prototype of Occult, when compared with the eventually-consistent system (Redis Cluster) it is derived from, increases the median latency by only $50\mu s$, the 99th percentile latency by only $400\mu s$ for a read-heavy workload ($4ms$ for a write-heavy workload), and reduces throughput by less than 10%. More importantly, we find that a four-entry causal timestamp suffices to achieve an accuracy of 99.6% for a cluster with over 16,000 logical shards.

3.2 Improving merging through branches

As suggested in §2.2, conflicting operations fork the entire state of the system, creating distinct *branches*. These branches have traditionally been hidden or greedily merged by systems' syntactic and per-object resolution strategies, that try, at all cost, to maintain the abstraction of a sequential view of the world. The lack of support for enforcing the cross-object consistency demands expressed in many application invariants thus makes conflict resolution more difficult and error-prone, as our Wikipedia example highlights.

Attempting to isolate applications from the reality of conflicting write operations is therefore, we believe, a well-intentioned fallacy. Conflicting writes "contaminate" data in a way that the storage system cannot understand: application logic is often indispensable to resolve those conflicts. Replicas, however, only see a sequence of read/write operations and are unaware of the application-logic and invariants that relate these operations. We consequently argue that the storage system should avoid deterministic quick fixes, and instead give applications the information they need to decide what is best. The question becomes: how can one provide applications with the best possible system support when merging conflicting states.

To answer this question, we designed TARDiS [11], an asynchronously replicated, multi-master key-value store designed for applications built above weakly-consistent systems. TARDiS renounces the one-size-fits-all abstraction of sequential storage and instead exposes applications, when appropriate, to concurrency and distribution. TARDiS' design is predicated on a simple notion: *to help developers resolve the anomalies that arise in such applications, each replica should faithfully store the full context necessary to understand how the anomalies arose in the first place, but only expose that context to applications when needed.* By default in TARDiS, applications execute on a branch, and hence perceive storage as sequential. But when anomalies arise, TARDiS reveals to the users the intricate details of distribution. TARDiS hence gives applications the flexibility of deciding if, when, and how divergent branches should be merged.

TARDiS provides two novel features that simplify reconciliation. First, it exposes applications to the resulting independent branches, and to the states at which the branches are created (fork points) and merged (merge points). Second, it supports atomic merging of conflicting branches and lets applications choose when and how to reconcile them. Branches, together with their fork and merge points, naturally encapsulate the information necessary for semantically meaningful merging: they make it easy to identify all the objects to be considered during merging and pinpoint when and how the conflict developed. This context can reduce the complexity and improve the efficiency of automated merging procedures, as well as help system administrators when user involvement is required. Returning to our Wikipedia example, a Wikipedia moderator presented with the two conflicting branches would be able to reconstruct the events that led to them and handle the conflicting sources according to Wikipedias guidelines [37]. Note that merging need not simply involve deleting one branch. Indeed, branching and merging states enables merging strategies with richer semantics than aborts or rollbacks [32]. It is TARDiS's richer interface that gives applications access to content that is essential to reasoning about concurrent updates, reducing the complexity of programming weakly consistent applications. In many ways, TARDiS is similar to Git [18]: users operate on their own branch and explicitly request (when convenient) to see concurrent modifications, using the history recorded by the underlying branching storage to help them resolve conflicts.

Unlike Git, however, branching in TARDiS does not rely on specific user commands but occurs implicitly, to preserve availability in the presence of conflicts. Two core principles underpin TARDiS: branch-on-conflict and inter-branch isolation. Branch-on-conflict lets TARDiS logically fork its state whenever it detects conflicting operations and store the conflicting branches explicitly. Inter-branch isolation guarantees that the storage will appear as sequential to any thread of execution that extends a branch, keeping application logic simple. The challenge is then to develop a datastore that can keep track of independent execution branches, record fork and merge points, facilitate reasoning about branches and, as appropriate, atomically merge them – while keeping performance and resource overheads comparable to those of weakly consistent systems.

TARDiS uses multi-master asynchronous replication: transactions first execute locally at a specific site, and are then asynchronously propagated to all other replicas (§4). Each TARDiS site is divided into two components: a *consistency layer* and a *storage layer*:

- *Consistency layer* The consistency layer records all branches generated during an execution with the help of a directed acyclic graph, the *state DAG*. Each vertex in the graph corresponds to a logical state of the datastore; each transaction that updates a record generates a new state.

- *Storage Layer* The storage layer stores records in a disk-based B-tree. TARDiS is a multiversioned system: every update operation creates a new record version.

Much of TARDiS logic is geared towards efficiently mapping the consistency layer to the storage layer, and maintaining low metadata overhead. Two techniques are central to the system's design: *DAG compression* and *conflict tracking*:

- *DAG compression* TARDiS tracks the minimal information needed to support branch merges under finite storage. It relies on a core observation: most merging policies only require the fork points and the leaf states of a given execution. All intermediate states can be safely removed from the state DAG, along with the corresponding records in the storage layer.
- *Conflict tracking* To efficiently construct and maintain branches, TARDiS introduces the notion of conflict tracking. TARDiS summarizes branches as a set of fork points and merge points only (a *fork path*) and relies on a new technique, *fork point checking* to determine whether two states belong to the same branch. Fork paths capture *conflicts* rather than dependencies. As such, they remain of small size (conflicts represent a small percentage of the total number of operations), and allow TARDiS to track concurrent branches efficiently while limiting memory overhead. This is in contrast to the traditional dependency checking approach [16,20,24], which quickly becomes a bottleneck in causally consistent systems [5,16].

We do not attempt to further discuss the details of these techniques, and defer the reader to the corresponding SIGMOD paper [11]. Generally though, the system is promising: we find for example, that using TARDiS rather than BerkeleyDB [29] to implement CRDTs [31] – a library of scalable, weakly-consistent datatypes – cuts code size by half, and improves performance by four to eight times.

4 Future Work and Conclusion

This article highlighted one of the thornier aspects of causal consistency: the handling of writes. It focused on two problems, the merging of conflicting writes, and the implications of asynchronous write-propagation in the presence of failures. TARDiS and Occult are two systems that attempt to mitigate these issues. However, several open challenges remain: though Occult prevents slowdown cascades, it currently adopts a single-master architecture that prevents applications from executing writes at every datacenter, increasing latency and limiting availability. A multi-master Occult would be faced with the same challenge of merging conflicting writes that we previously outlined. Adding TARDiS's branches to Occult without reintroducing the danger of slowdown cascades is not straightforward: branching requires knowledge of a per-datacenter state DAG that is currently centralized. Naive approaches like sharding this datastructure may reintroduce the dangers of slowdown cascades. We believe that combining the approach of these systems is a promising avenue of future work.

Acknowledgements Occult and TARDiS are the result of a collaboration with many other co-authors: Nathan Bronson, Allen Clement, Nancy Estrada, Trinabh Gupta, Cody Littley, Wyatt Lloyd and Youer Pu. These projects were generously supported by grants from the National Science Foundation under grant number CNS-1409555, a Google Cloud Platform credit awards and a Google Faculty Research Award. Natacha Crooks was partially supported by a Google Doctoral Fellowship in Distributed Computing.

References

[1] Redis Cluster Specification. `http://redis.io/topics/cluster-spec`.

[2] AJOUX, P., BRONSON, N., KUMAR, S., LLOYD, W., AND VEERARAGHAVAN, K. Challenges to Adopting Stronger Consistency at Scale. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Switzerland, 2015), HOTOS'15, USENIX Association.

[3] ALMEIDA, S., LEITÃO, J. A., AND RODRIGUES, L. Chainreaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic, 2013), EuroSys '13, ACM, pp. 85–98.

[4] APACHE. Cassandra. http://cassandra.apache.org/.

[5] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SOCC '12, pp. 22:1–22:7.

[6] BAILIS, P., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Bolt-On Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, 2013), SIGMOD '13, ACM, pp. 761–772.

[7] BASHO. Riak. http://basho.com/products/.

[8] BIRMAN, K., CHOCKLER, G., AND VAN RENESSE, R. Toward a Cloud Computing Research Agenda. *SIGACT News 40*, 2 (June 2009), 68–80.

[9] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1277–1288.

[10] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pp. 251–264.

[11] CROOKS, N., PU, Y., ESTRADA, N., GUPTA, T., ALVISI, L., AND CLEMENT, A. Tardis: A branch-and-merge approach to weak consistency. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 1615–1628.

[12] DEAN, J., AND BARROSO, L. A. The Tail at Scale. *Communications of the ACM 56*, 2 (Feb. 2013), 74–80.

[13] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, 2004), OSDI'04, USENIX Association, pp. 137–149.

[14] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM Symposium on Operating Systems Principles*, SOSP '07, pp. 205–220.

[15] DU, J., ELNIKETY, S., ROY, A., AND ZWAENEPOEL, W. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *Proceedings of the 4th ACM Symposium on Cloud Computing* (Santa Clara, California, 2013), SOCC '13, ACM, pp. 11:1–11:14.

[16] DU, J., IORGULESCU, C., ROY, A., AND ZWAENEPOEL, W. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pp. 4:1–4:13.

[17] FAUNADB. Faunadb. https://fauna.com/.

[18] GIT. Git: the fast version control system. http://git-scm.com.

[19] LABS, C. Cockroachdb. https://www.cockroachlabs.com/.

[20] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 401–416.

[21] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL, 2013), NSDI '13, USENIX Association, pp. 313–328.

[22] LU, H., VEERARAGHAVAN, K., AJOUX, P., HUNT, J., SONG, Y. J., TOBAGUS, W., KUMAR, S., AND LLOYD, W. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 295–310.

[23] MAHAJAN, P., ALVISI, L., AND DAHLIN, M. Consistency, availability, convergence. Tech. Rep. TR-11-22, Computer Science Department, University of Texas at Austin, May 2011.

[24] MAHAJAN, P., SETTY, S., LEE, S., CLEMENT, A., ALVISI, L., DAHLIN, M., AND WALFISH, M. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems 29*, 4 (2011), 12.

[25] MEHDI, S. A., LITTLEY, C., CROOKS, N., ALVISI, L., BRONSON, N., AND LLOYD, W. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 453–468.

[26] MICROSOFT. Cosmosdb. `https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels`.

[27] MONGODB. Agility, Performance, Scalibility. Pick three. `https://www.mongodb.org/`.

[28] NEO4J. Neo4j. `https://neo4j.com/docs/operations-manual/current/clustering/causal-clustering/introduction/`.

[29] OLSON, M. A., BOSTIC, K., AND SELTZER, M. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99.

[30] REGAL, I. Antidote.

[31] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, Jan. 2011.

[32] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 385–400.

[33] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABU-LIBDEH, H. Consistency-based service level agreements for cloud storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 309–324.

[34] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP '95, pp. 172–182.

[35] THOMAS, R. H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (June 1979), 180–209.

[36] VOLDEMORT, P. Voldemort, a distributed database. `http://www.project-voldemort.com`.

[37] WIKIPEDIA. Wikipedia: Conflicting Sources. `http://en.wikipedia.org/wiki/Wikipedia:Conflicting_sources`.

[38] ZAWIRSKI, M., PREGUIÇA, N., DUARTE, S., BIENIUSA, A., BALEGAS, V., AND SHAPIRO, M. Write Fast, Read in the Past: Causal Consistency for Client-Side Applications. In *Proceedings of the 16th Annual Middleware Conference* (Vancouver, BC, Canada, 2015), Middleware '15, ACM, pp. 75–87.

Cost-Effective Geo-Distributed Storage for Low-Latency Web Services

Zhe Wu and Harsha V. Madhyastha
University of Michigan

Abstract

We present two systems that we have developed to aid geo-distributed web services deployed in the cloud to serve their users with low latency. First, CosTLO [23] helps mitigate the high latency variance observed when accessing data stored in multi-tenant cloud storage services. To cost-effectively satisfy a web service's tail latency goals, CosTLO uses measurement-driven inferences about replication and load balancing within cloud storage to combine the use of multiple forms of redundancy. Second, to support web services that enable users to share data, SPANStore [22] leverages application hints about access patterns and knowledge about cloud latencies and pricing to cost-effectively replicate data across data centers. Key to SPANStore's design is to spread data across the data centers of multiple cloud providers, in order to present better cost vs. latency tradeoffs than possible with the use of any single cloud provider. In this paper, we summarize the key takeaways from our work in developing both systems.

1 Introduction

Minimizing user-perceived latency is a critical goal for web services, because even a few 100 milliseconds of additional delay can significantly reduce revenue [6]. Key to achieving this goal is to deploy web servers in multiple locations so that every user can be served from a nearby server. Therefore, cloud services such as Azure and Amazon Web Services are an attractive option for web service deployments as these platforms offer data centers in tens of locations spread across the globe [3, 4].

However, web services deployed in the cloud are faced with two challenges in ensuring that users have low latency access to their data.

- **High latency variance.** Since cloud storage services are used concurrently by multiple tenants, both fetching and storing content on these services is associated with high latency variance. For example, when we had 120 PlanetLab nodes across the globe each download 1 KB file from their respective closest Microsoft Azure data center, for over 70% of these nodes, the 99^{th} percentile and median download latencies differ by 100ms or more. These high tail latencies are problematic both for popular applications where even 1% of traffic corresponds to a significant volume of requests [16], and for applications where a single request issued by an end-host requires the application to fetch several objects (e.g., web page loads) and user-perceived latency is constrained by the object fetched last. For example, our measurements show that latency variance in Amazon S3 more than doubles the *median* page load time for 50% of PlanetLab nodes when fetching a web page containing 50 objects.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

- **Cost vs. latency tradeoff.** Collaborative services that enable users to share data with each other must also determine which data centers must store replicas of each object. While replicating all objects to all data centers can ensure low latency access [19], that approach is costly and may be inefficient. Some applications may value lower costs over the most stringent latency bounds, different applications may demand different degrees of data consistency, some objects may only be popular in some regions, and some clients may be near to multiple data centers, any of which can serve them quickly. All these parameters mean that no single deployment provides the best fit for all applications and all objects.

In this paper, we describe two systems that we have developed to address these challenges. First, CosTLO (Cost-effective Tail Latency Optimizer) enables application providers to avail of the cost benefits enabled by cloud services, without having latency variance degrade user experience. Second, SPANStore (Storage Provider Aggregating Networked Store) is a key-value store that presents a unified view of storage services present in several geographically distributed data centers.

CosTLO's design is based on three principles. First, since our measurements reveal that the high latency variance in cloud storage is caused predominantly by isolated latency spikes, CosTLO augments every GET/PUT request with a set of redundant requests [15,21], so that it suffices to wait for responses to a subset of the requests. Second, to tackle the variance in all components of end-to-end latency—latency over the Internet, latency over the cloud service's data center network, and latency within the storage service—CosTLO combines the use of multiple forms of redundancy, such as issuing redundant requests to the same object, to multiple copies of an object, to multiple front-ends, or even to copies of an object in multiple data centers. Lastly, since the number of configurations in which CosTLO can implement redundancy is unbounded, to add redundancy in a manner that satisfies an application's goals for tail latency cost-effectively, we model the load balancing and replication within cloud storage services in order to accurately capture the dependencies between concurrent requests.

The key premise underlying SPANStore is that, when geo-replicating data, spreading data across the data centers of multiple cloud providers offers significantly better latency versus cost tradeoffs than possible with any single cloud provider. This is not only because the union of data centers across cloud providers results in a geographically denser set of data centers than any single provider's data centers, but also because the cost of storage and networking resources can significantly differ across cloud providers. To exploit these factors to drive down the cost incurred in satisfying application providers' latency, consistency, and fault tolerance goals, SPANStore judiciously determines where to replicate every object and how to perform this replication. For every object that it stores, SPANStore makes this determination by taking into consideration several factors: the anticipated workload for the object (i.e., how often different clients access it), the latency guarantees specified by the application that stored the object in SPANStore, the number of failures that the application wishes to tolerate, the level of data consistency desired by the application (e.g., strong versus eventual), and the pricing models of storage services that SPANStore builds upon.

Our goal in this paper is not to provide a detailed description of the design and implementation of either system and our results from evaluating them; we refer the interested reader to [23] and [22] for those details. Here, we instead summarize the main takeaways from both CosTLO and SPANStore in enabling low latency geo-distributed storage in the cloud.

2 Motivation

We begin by presenting measurement data that motivates our design of CosTLO and SPANStore.[1]

[1] Though the data presented here was gathered between 2013 and 2015, similar trends exist today.

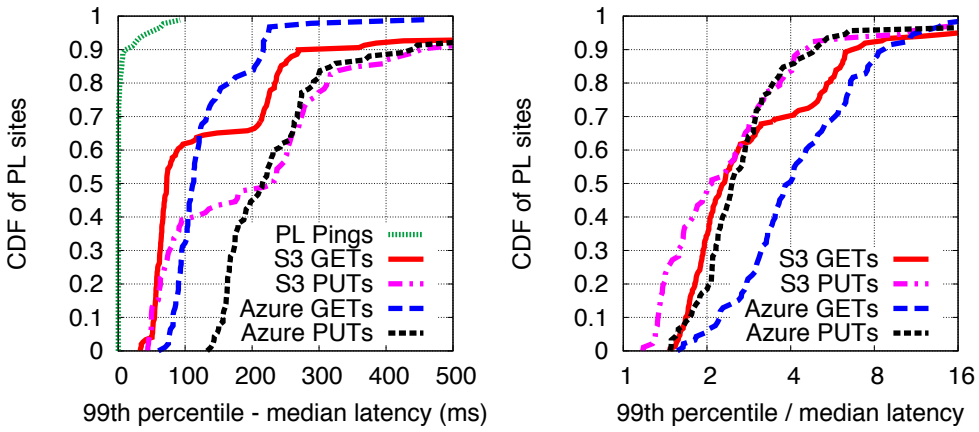


Figure 1: (a) Absolute and (b) relative inflation in 99^{th} percentile latency with respect to median. Note logscale x-axis in (b).

2.1 Characterizing latency variance

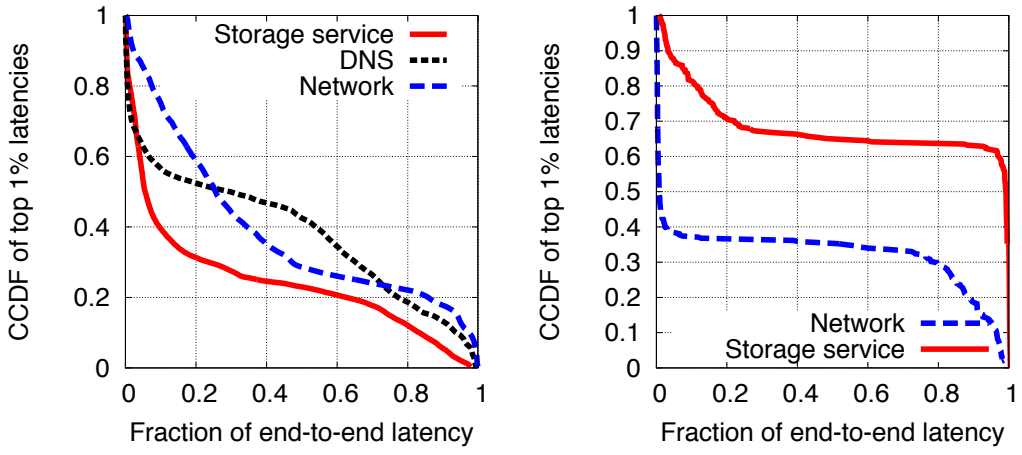
Overview of measurements. To analyze client-perceived latencies when downloading from and uploading to cloud storage services, we gather two types of measurements for a week. First, we use 120 PlanetLab nodes across the world as representative end-hosts. Once every 3 seconds, every node uploaded a new object to and downloaded a previously stored object from the S3 and Azure data centers to which the node has the lowest median RTT. Second, from "small instance" VMs in every S3 and every Azure data center, we issued one GET and one PUT per second to the local storage service. In all cases, every GET from a data center was for a 1 KB object selected at random from 1M objects of that size stored at that data center, and every PUT was for a new 1 KB object.

To minimize the impact of client-side overheads, we measure GET and PUT latencies on PlanetLab nodes as well as on VMs using timings from tcpdump. In addition, we leverage logs exported by S3 [1] and Azure [7] to break down end-to-end latency minus DNS resolution time into its two components: 1) latency within the storage service (i.e., duration between when a request was received at one of the storage service's front-ends and when the response left the storage service), and 2) latency over the network (i.e., for the request to travel from the end-host/VM to a front-end of the storage service and for the response to travel back). We extract storage service latency directly from the storage service logs, and we can infer network latency by subtracting storage service latency from end-to-end request latency.

Quantifying latency variance. Figure 1 shows the distribution across nodes of the spread in latencies; for every node, we plot the absolute and relative difference between the 99^{th} percentile and median latencies. In both Azure and S3, the median PlanetLab node sees an absolute inflation greater than 200ms (70ms) in the 99^{th} percentile PUT (GET) latency as compared to the median latency; the median relative inflation is greater than 2x in both PUTs and GETs. Figure 1 shows that this high latency variance is not due to high load or slow access links of PlanetLab nodes; for every node, the figure plots the difference between 99^{th} percentile and median latency to the node closest to it among all PlanetLab nodes.

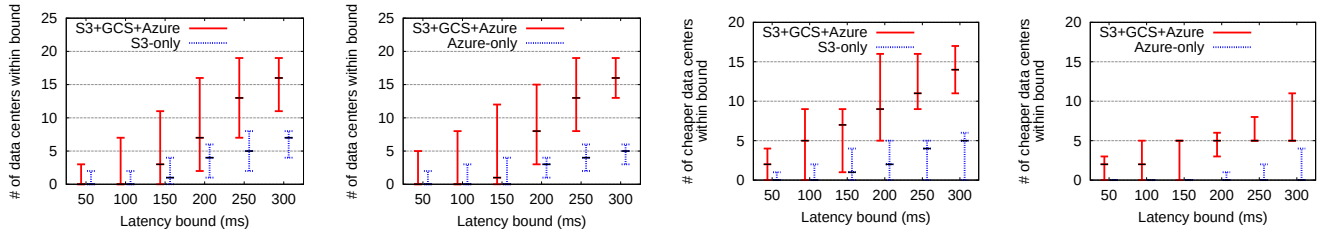
Causes for tail latencies. We observe two characteristics that dictate which solutions can potentially reduce the tail of these latency distributions.

First, we find that neither are the top 1% of latency samples clustered together in time nor are they correlated with time of day. Thus, the tail of the latency distribution is dominated by isolated spikes, rather than sustained periods of high latencies. *Therefore, a solution that monitors load and reacts to latency spikes will be ineffective.*



(a) End-host perspective (b) VM perspective

Figure 2: Breakdown of components of tail latencies.



(a) EC2 (b) Azure (c) EC2 (d) Azure

Figure 3: For applications deployed on a single cloud service (EC2 or Azure), a storage service that spans multiple cloud services offers a larger number of data centers (a and b) and more cheaper data centers (c and d) within a latency bound.

Second, Figure 2(a) shows that all three components of end-to-end latency significantly influence tail latency values. DNS latency, network latency, and latency within the storage service account for over half the end-to-end latency on more than 40%, 25%, and 20% of tail latency samples. Since network latencies as measured from PlanetLab nodes conflate latencies over the Internet and within the cloud service's data center network, we also study the composition of tail latencies as seen in our measurements from VMs to the local storage service. In this case too, Figure 2(b) shows that both components of end-to-end latency—latency within the storage service, and latency over the data center network—contribute significantly to a large fraction of tail latency samples. *Thus, any solution that reduces latency variance will have to address all of these sources of latency spikes.*

2.2 Why multi-cloud?

Next, we motivate the utility of spread data across multiple cloud providers by presenting data which shows that doing so can potentially lead to reduced latencies for clients and reduced cost for applications.

Lower latencies. We first show that using multiple cloud providers can enable lower GET/PUT latencies. For this, we instantiated VMs in each of the data centers in EC2, Azure, and Google Cloud. From every VM, we measured GET latencies to the storage service in every other data center once every 5 minutes for a week. We consider the latency between a pair of data centers as the median of the measurements for that pair.

Figure 3 shows how many other data centers are within a given latency bound of each EC2 [3(a)] and

Azure [3(b)] data center. These graphs compare the number of nearby data centers if we only consider the single provider to the number if we consider all three providers—Amazon, Google, and Microsoft. For a number of latency bounds, either graph depicts the minimum, median, and maximum (across data centers) of the number of options within the latency bound.

For nearly all latency bounds and data centers, we find that deploying across multiple cloud providers increases the number of nearby options. An application can use this greater choice of nearby storage options to meet tighter latency SLOs, or to meet a fixed latency SLO using fewer storage replicas (by picking locations nearby to multiple front-ends). Intuitively, this benefit occurs because different providers have data centers in different locations, resulting in a variation in latencies to other data centers and to clients.

Lower cost. Replicating data across multiple cloud providers also enables support for latency SLOs at potentially lower cost due to the discrepancies in pricing across providers. Figures 3(c) and 3(d) show, for each EC2 and Azure data center, the number of other data centers within a given latency bound that are cheaper than the local data center along some dimension (storage, PUT/GET requests, or network bandwidth). For example, nearby Azure data centers have similar pricing, and so, no cheaper options than local storage exist within 150ms for Azure-based services. However, for the majority of Azure-based front-ends, deploying across all three providers yields multiple storage options that are cheaper for at least some operations. Thus, by judiciously combining resources from multiple providers, an application can use these cheaper options to reduce costs.

3 Cost-effective support for lower latency variance with CosTLO

Goal and Approach. We design CosTLO to meet any application's service-level objectives (SLOs) for the extent to which it seeks to reduce latency variance for its users. Though there are several ways in which such SLOs can be specified, we do not consider SLOs that bound the absolute value of, say, 99^{th} percentile GET/PUT latency; due to the non-uniform geographic distribution of data centers, a single bound on tail latencies for all end-hosts will not help reduce latency variance for end-hosts with proximate data centers. Instead, we focus on SLOs that limit the tail latencies for any end-host *relative* to the latency distribution experienced by *that* end-host. Specifically, we consider SLOs which bound the difference, for any end-host, between 99^{th} percentile latency and its baseline median latency (i.e., the median latency that it experiences without CosTLO). Every application specifies such a bound separately for GETs and PUTs.

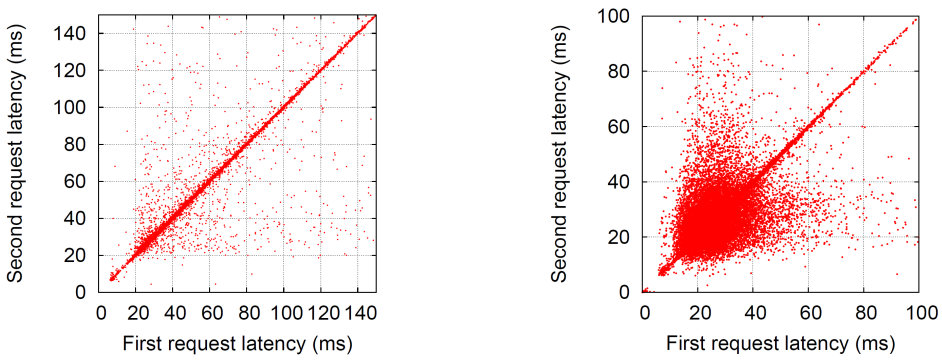
Since tail latency samples are dominated by isolated spikes, our high-level approach is to augment any GET/PUT request with a set of redundant requests, so that either the first response or a subset of early responses can be considered. Though this is a well-known approach for reducing tail latencies [9, 15, 21], CosTLO is unique in exploiting several ways of issuing redundant requests in combination.

3.1 Effective redundancy techniques

A wide range of redundancy approaches are feasible to tackle variance in each component of end-to-end latency. Here, we summarize what our measurements reveal to be the most effective techniques.

Latency over the Internet. To examine the utility of different approaches on reducing Internet tail latencies, we issued pairs of concurrent GET requests from each PlanetLab node in three different ways and then compared the measured tail latencies with those seen with single requests.

First, relying on load balancing in the Internet [12], when every end-host concurrently issues multiple requests to the storage service in the data center closest to it, relative latency inflation seen at the median PlanetLab node remains close to 2x. Second, in addition to a GET request to its closest S3 region, having every node issue a GET request in parallel to its second closest S3 region offers little benefit in reducing latency variance. This is because, for most PlanetLab nodes, the second closest region within a cloud service is too far to help tame latency spikes to the region closest to the node. Therefore, to reduce variance in latencies over the Internet, it



(a) Azure GETs (b) S3 GETs

Figure 4: Scatter plot of first vs. second request GET latency when issuing two concurrent requests to the same object stored in a cloud storage service.

is crucial for redundancy to exploit multiple cloud providers. Doing so reduces the inflation in 99^{th} percentile GET latency to be less than 1.5x the baseline median at 70% of PlanetLab nodes.

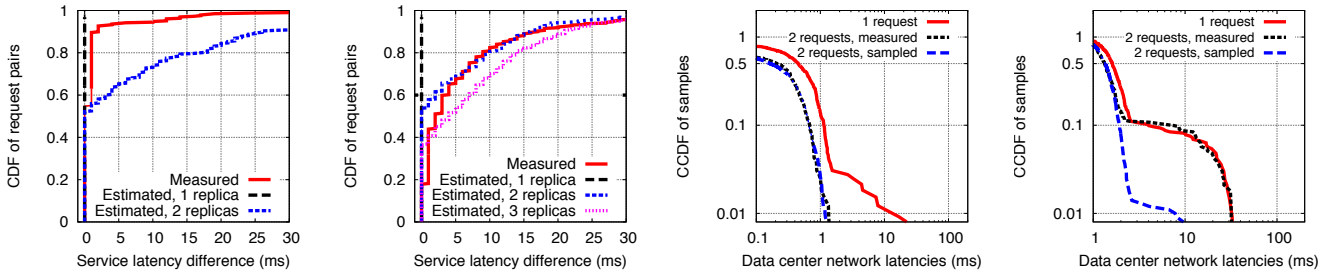
Data center network latencies. Latency spikes within a cloud service's data center network can either be addressed implicitly by issuing the same PUT/GET request multiple times in parallel to exploit path diversity or addressed explicitly by relaying each request through a different VM. In both S3 and Azure, we find that both implicit and explicit exploitation of path diversity significantly reduce tail latencies, with higher levels of parallelism offering greater reduction. However, using VMs as relays adds some overhead, likely due to requests traversing longer routes.

Storage service latencies. Lastly, we considered two approaches for reducing latency spikes within the storage service, i.e., latency between when a request is received at a front-end and when it sends back the response. When issuing n concurrent requests to a storage service, we can either issue all n requests for the same object or to n different objects. The former attempts to implicitly leverage the replication of objects within the storage service, whereas the latter explicitly creates and utilizes copies of objects. In either case, if concurrent requests are served by different storage servers, latency spikes at any one server can be overridden by other servers that are lightly loaded.

The takeaways differ between Azure and S3. On S3, irrespective of whether we issue multiple requests to the same object or to different objects, the reduction in 99^{th} percentile latency tails off with increasing parallelism. This is because, in S3, concurrent requests from a VM incur the same latency over the network, which becomes the bottleneck in the tail. In contrast, on Azure, 99^{th} percentile GET latencies do not reduce further when more than 2 concurrent requests are issued to the same object, but tail GET latencies continue to drop significantly with increasing parallelism when concurrent requests are issued to different objects. In the case of PUTs, the benefits of redundancy tail off at parallelism levels greater than 2 due to Azure's serialization of PUTs issued by the same tenant [14].

3.2 Selecting cost-effective configuration

The primary challenge in combining these various forms of redundancy in CosTLO is to select for each edge network a redundancy configuration that helps meet the application's tail latency goals at minimum cost. For this, CosTLO 1) takes as input the pricing policies at every data center, 2) uses logs exported by cloud providers



(a) (b) (c) (d)

Figure 5: Distribution of service latency difference between concurrent GET requests offers evidence for GETs to an object (a) being served by one replica in Azure, and (b) being spread across two replicas in S3. Data center network latencies for concurrent requests are (c) uncorrelated on Azure, and (d) correlated on S3. Note logscale on both axes of (c) and (d).

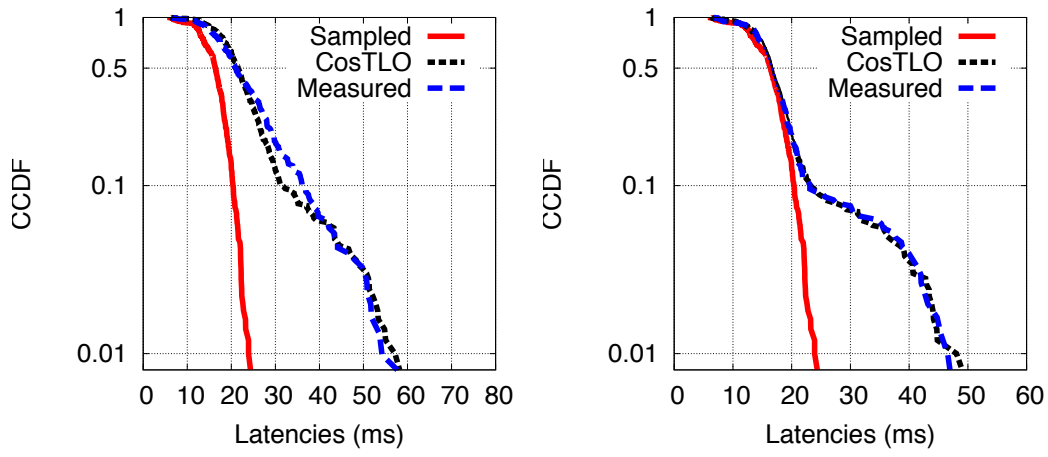
to characterize the workload imposed by clients in every prefix, and 3) employs a measurement agent at every data center. Every agent gathers three types of measurements: 1) pings to a representative end-host in every prefix, 2) pairs of concurrent GETs and pairs of concurrent PUTs to the local storage service, and 3) the rates at which VMs can relay PUTs and GETs between end-hosts and the local storage service without any queueing.

Given these measurements, CosTLO identifies a cost-effective configuration for end-hosts in each IP address prefix. We refer the reader to [23] for a description of the algorithm CosTLO uses to search through the configuration space, and focus here on the key question in doing so: for any particular configuration for an IP prefix, how do we estimate the latency distribution that clients in that prefix will experience when served in that configuration?

The reason it is hard to estimate the latency distribution for any particular redundancy configuration is due to dependencies between concurrent requests. While Figure 4 shows the correlation in latencies between two concurrent GET requests to an object at one of Azure's and one of S3's data centers, we also see similar correlations for PUTs and even when the concurrent requests are for different objects. Attempting to model these correlations between concurrent requests by treating the cloud service as a black box did not work well. Therefore, we explicitly model the sources of correlations: concurrent requests may incur the same latency within the storage service if they are served by the same storage server, or incur the same data center network latency if they traverse the same network path.

Modeling replication in storage service. First, at every data center, we use CosTLO's measurements to infer the number of replicas across which the storage service spreads requests to an object. For every pair of concurrent requests issued during CosTLO's measurements, we compute the difference in service latency (i.e., latency within the storage service) between the two requests. We then consider the distribution of this difference across all pairs of concurrent requests to infer the number of replicas in use per object. For example, if the storage service load balances GET requests to an object across 2 replicas, there should be a 50% chance that two concurrent GETs fetch from the same replica, therefore the service latency difference is expected to be 0 half the time. We compare this measured distribution with the expected distribution when the storage service spreads requests across n replicas, where we vary the value of n. We infer the number of replicas used by the service as the value of n for which the estimated and measured distributions most closely match. For example, though both Azure [5] and S3 [2] are known to store 3 replicas of every object, Figures 5(a) and 5(b) show that the measured service latency difference distributions closely match GETs being served from 1 replica on Azure and from 2 replicas on S3.

On the other hand, for concurrent GETs or PUTs issued to different objects, on both Azure and S3, we see that the latency within the storage service is uncorrelated across requests. This is likely because cloud



(a) Requests to same object

(b) Requests to different objects

Figure 6: Accuracy of estimating GET latency distribution for 8 concurrent GET requests from VM to local storage service in one S3 region. Note logscale on y-axis.

storage services store every object on a randomly chosen server (e.g., by hashing the object's name for load balancing [16]), and hence, requests to different objects are likely to be served by different storage servers.

Modeling load balancing in network. Next, we identify whether concurrent requests issued to the storage service incur the same latency over the data center network, or are their network latencies independent of each other. At any data center, we compute the distribution obtained from the minimum of two independent samples of the measured data center network latency distribution for a single request. We then compare this distribution to the measured value of the minimum data center network latency seen across two concurrent requests.

Figure 5(c) shows that, on Azure, the distribution obtained by independent sampling closely matches the measured distribution, thus showing that network latencies for concurrent requests are uncorrelated. Whereas, on S3, Figure 5(d) shows that the measured distribution for the minimum across two requests is almost identical to the data center network latency component of any single request; this shows that concurrent requests on S3 incur the same network latency.

By leveraging these models of replication and load balancing, CosTLO is able to accurately estimate the latency distribution when sets of requests are concurrently issued to a storage service. Figures 6(a) and 6(b) compare the measured and estimated latency distributions when issuing 8 concurrent GETs from a VM to the local storage service; all concurrent requests are for the same object in the former and to different objects in the latter. In both cases, our estimated latency distribution closely matches the measured distribution, even in the tail. In contrast, if we estimate the latency distribution for 8 concurrent GETs by independently sampling the latency distribution for a single request 8 times and considering the minimum, we significantly under-estimate the tail of the distribution.

4 Cost-effective geo-replication of data with SPANStore

Thus far, we have assumed that every object stored by an application is stored in a single data center, and CosTLO reduces tail latencies for executing GETs and PUTs on such objects. Now, we turn our attention to addressing the storage needs of web services in which sharing of data among users is an intrinsic part of the service (e.g., collaborative applications like Google Docs, and file sharing services like Dropbox), and geo-replication of shared data is a must to ensure low latency. Instead of every user having to access a centrally located copy of an object, replication of the object across data centers permits every user to access a nearby

subset of the object's copies.

Our overarching goal in developing SPANStore is to enable applications to interact with a single storage service, which underneath the covers uses several geographically distributed storage services.[2] The key benefit here is that the onus of navigating the space of replication strategies on an object-by-object basis is offloaded to SPANStore, rather than individual application developers having to deal with this problem, as is the case today. We are guided by four objectives: 1) minimize cost, 2) respect latency SLOs, 3) tolerate failures, and 4) provide support for both eventual and strong consistency.

While we refer the reader to [22] for a detailed description of SPANStore's design, implementation, and evaluation, here we summarize the key problem at the heart of its design: for any particular object, how to determine the most cost-effective replication policy that satisfies the application's latency, consistency, and fault-tolerance requirements? We first describe the inputs required by SPANStore and the format of the replication policies it identifies. We then separately present the algorithms used by SPANStore in two different data consistency scenarios.

4.1 Inputs and output

SPANStore requires three types of inputs: 1) a characterization of the cloud services on which it is deployed, 2) the application's latency, fault tolerance, and consistency requirements, and 3) a specification of the application's workload.

Characterization of SPANStore deployment. We require two pieces of information about SPANStore's deployment. First, we measure the distribution of latencies between every pair of data centers on which SPANStore is deployed. These measurements includes PUTs, GETs, and pings issued from a VM in one data center to the storage service or a VM in another data center. Second, we need the pricing policy for the resources used by SPANStore. For each data center, we specify the price per byte of storage, per PUT request, per GET request, and per hour of usage for the type of virtual machine used by SPANStore in that data center. We also specify, for each pair of data centers, the price per byte of network transfer from one to the other, which is determined by the upload bandwidth pricing at the source data center.

Application requirements. We account for an application's latency goals in terms of separate SLOs for latencies incurred by PUT and GET operations. Either SLO is specified by a latency bound and the fraction of requests that should incur a latency less than the specified bound, e.g., 90^{th} percentile latency should be less than 100ms.

To capture consistency needs, we ask the application developer to choose between strong and eventual consistency. In the strong consistency case, we provide linearizability, i.e., all PUTs for a particular object are totally ordered and any GET returns the data written by the last committed PUT for the object. In contrast, if an application can make do with eventual consistency, SPANStore can satisfy lower latency SLOs. Our algorithms for the eventual consistency scenario are extensible to other consistency models such as causal consistency [19] by augmenting data transfers with additional metadata.

In both the eventual consistency and strong consistency scenarios, the application developer can specify the number of failures—either of data centers or of Internet paths between data centers—that SPANStore should tolerate. As long as the number of failures is less than the specified number, SPANStore ensures the availability of all GET and PUT operations while also satisfying the application's consistency and latency requirements. When the number of failures exceeds the specified number, SPANStore may make certain operations unavailable or violate latency goals in order to ensure that consistency requirements are preserved.

Workload characterization. Lastly, we account for the application's workload in two ways.

[2]We assume an application employing SPANStore for data storage uses only the data centers of a single cloud service to host its computing instances (due to the differences across cloud providers), even though (via SPANStore) it will use multiple cloud providers for data storage.

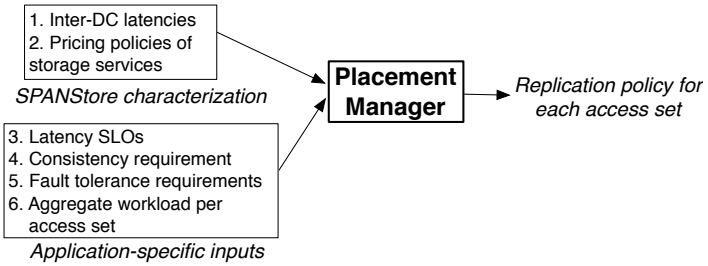


Figure 7: Overview of inputs and output in identifying cost-effective replication policies.

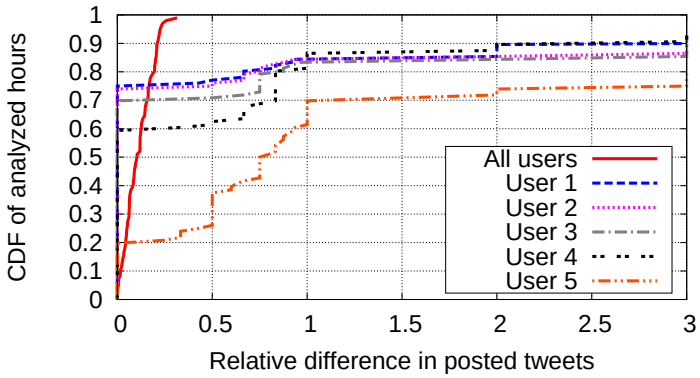
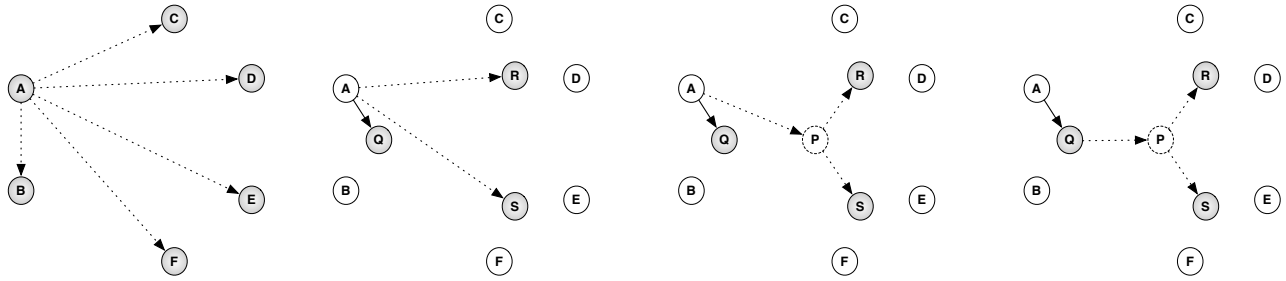


Figure 8: Comparison at different granularities of the stationarity in the number of posted tweets.

First, for every object stored by an application, we ask the application to specify the set of data centers from which it will issue PUTs and GETs for the object. We refer to this as the *access set* for the object. An application can determine the access set for an object based on the sharing pattern of that object across users. For example, a collaborative online document editing webservice knows the set of users with whom a particular document has been shared. The access set for the document is then the set of data centers from which the web service serves these users. In cases where the application itself is unsure which users will access a particular object (e.g., in a file hosting service like Rapidshare), it can specify the access set of an object as comprising all data centers on which the application is deployed; this uncertainty will translate to higher costs. While SPANStore considers every object as having a fixed access set over its lifetime, subsequent work [10] addresses this limitation.

Second, SPANStore's VMs track the GET and PUT requests received from an application to characterize its workload. Since the GET/PUT rates for individual objects can exhibit bursty patterns (e.g., due to flash crowds), it is hard to predict the workload of a particular object in the next epoch based on the GETs and PUTs issued for that object in previous epochs. Therefore, SPANStore instead leverages the stationarity that typically exists in an application's aggregate workload, e.g., many applications exhibit diurnal and weekly patterns in their workload [8, 13]. Specifically, at every data center, SPANStore VMs group an application's objects based on their access sets. In every epoch, for every access set, the VMs at a data center report to a central Placement Manager 1) the number of objects associated with that access set and the sum of the sizes of these objects, and 2) the aggregate number of PUTs and GETs issued by the application at that data center for all objects with that access set.

To demonstrate the utility of considering aggregate workloads in this manner, we analyze a Twitter dataset that lists the times at which 120K users in the US posted on Twitter over a month [18]. We consider a scenario in which every user is served from the EC2 data center closest to the user, and consider every user's Twitter timeline to represent an object. When a user posts a tweet, this translates to one PUT operation on the user's timeline and one PUT each on the timelines of each of the user's followers. Thus, the access set for a particular



(a) Full replication minimizes read latency

(b) Partial replication reduces costs

(c) Relayed update propagation reduces bandwidth costs

(d) Two-hop relaying may reduce costs further

Figure 9: When eventual consistency suffices, illustration of different replication policies for access set {A, B, C, D, E, F}. In all cases, we show how PUTs from data center A are propagated. Shaded circles are data centers that host replicas, and dotted circles represent data centers that propagate updates. Solid arrows correspond to transfers that impact PUT latencies, and dotted arrows represent asynchronous propagation.

user's timeline includes the data centers from which the user's followers are served.

Here, we consider those users whose timelines have their access set as all EC2 data centers in the US. Figure 8 presents the stationarity in the number of PUTs when considering the timelines of all of these users in aggregate and when considering five popular individual users. In either case, we compare across two weeks the number of PUTs issued in the same hour on the same day of the week, i.e., for every hour, we compute the difference between the number of tweets in that hour and the number of tweets in the same hour the previous week, normalized by the latter value. When we aggregate across all users, the count for every hour is within 50% of the count for that hour the previous week, whereas individual users often exhibit 2x and greater variability. The greater stationarity in the aggregate workload thus enables more accurate prediction based on historical workload measurements.

Replication policy. Given these inputs, at the beginning of every epoch, the central Placement Manager determines the replication policy to be used in the next epoch. Since we capture workload in aggregate across all objects with the same access set, the Placement Manager determines the replication policy separately for every access set, and SPANStore employs the same replication strategy for all objects with the same access set. For any particular access set, the replication policy output by the Placement Manager specifies 1) the set of data centers that maintain copies of all objects with that access set, and 2) at each data center in the access set, which of these copies SPANStore should read from and write to when an application VM at that data center issues a GET or PUT on an object with that access set.

Thus, the crux of SPANStore's design boils down to: 1) in each epoch, how does the Placement Manager determine the replication policy for each access set, and 2) how does SPANStore enforce Placement Manager-mandated replication policies during its operation, accounting for failures and changes in replication policies across epochs? We refer the reader to [22] for the answer to the latter question, and we describe here separately how SPANStore addresses the first question in the eventual consistency and strong consistency cases.

4.2 Eventual consistency

When the application can make do with eventual consistency, SPANStore can trade-off costs for storage, PUT/GET requests, and network transfers. To see why this is the case, let us first consider the simple replication policy where SPANStore maintains a copy of every object at each data center in that object's access set (as shown in Figure 9(a)). In this case, a GET for any object can be served from the local storage service. Similarly, PUTs can be committed to the local storage service and updates to an object can be propagated to other data centers in the background; SPANStore considers a PUT as complete after writing the object to the local storage service

because of the durability guarantees offered by cloud storage services. By serving PUTs and GETs from the storage service in the same data center, this replication policy minimizes GET/PUT latencies, the primary benefit of settling for eventual consistency. In addition, serving GETs from local storage ensures that GETs do not incur any network transfer costs.

However, as the size of the access set increases, replicating every object at every data center in the access set can result in high storage costs. Furthermore, as the fraction of PUTs in the workload increase, the costs associated with PUT requests and network transfers increase as more copies need to be kept up-to-date.

To reduce storage costs and PUT request costs, SPANStore can store replicas of an object at fewer data centers, such that every data center in the object's access set has a nearby replica that can serve GETs/PUTs from this data center within the application-specified latency SLOs. For example, as shown in Figure 9(b), instead of storing a local copy, data center A can issue PUTs and GETs to the nearby replica at Q.

However, SPANStore may incur unnecessary networking costs if it propagates a PUT at data center A by having A directly issue a PUT to every replica. Instead, we can capitalize on the discrepancies in pricing across different cloud services (see Section 2) and relay updates to the replicas via another data center that has cheaper pricing for upload bandwidth. For example, in Figure 9(c), SPANStore reduces networking costs by having A send each of its updates to P, which in turn issues a PUT for this update to all the replicas that A has not written to directly. In some cases, it may be even more cost-effective to have the replica to which A commits its PUTs relay updates to a data center that has cheap network pricing, which in turn PUTs the update to all other replicas, e.g., as shown in Figure 9(d).

We address this trade-off between storage, networking, and PUT/GET request costs by formulating the problem of determining the replication policy for a given access set AS as a mixed integer program. For every data center $i \in AS$, the integer program chooses $f + 1$ data centers (out of all those on which SPANStore is deployed) which will serve as the replicas to which i issues PUTs and GETs. SPANStore then stores copies of all objects with access set AS at all data centers in the union of PUT/GET replica sets.

The integer program we use imposes several constraints on the selection of replicas and how updates made by PUT operations propagate. First, whenever an application VM in data center i issues a PUT, SPANStore synchronously propagates the update to all the data centers in the replica set for i and asynchronously propagates the PUT to all other replicas of the object. Second, to minimize networking costs, the integer program allows for both synchronous and asynchronous propagation of updates to be relayed via other data centers. Synchronous relaying of updates must satisfy the latency SLOs, whereas in the case of asynchronous propagation of updates, relaying can optionally be over two hops, as in the example in Figure 9(d). Finally, for every data center i in the access set, the integer program identifies the paths from data centers j to k along which PUTs from i are transmitted during either synchronous or asynchronous propagation.

SPANStore's Placement Manager solves this integer program with the objective of minimizing total cost, which is the sum of storage cost and the cost incurred for serving GETs and PUTs. The storage cost is simply the cost of storing one copy of every object with access set AS at each of the replicas chosen for that access set. For every GET operation at data center i, SPANStore incurs the price of one GET request at each of i's replicas and the cost of transferring the object over the network from those replicas. In contrast, every PUT operation at any data center i incurs the price of one PUT request each at all the replicas chosen for access set AS, and network transfer costs are incurred on every path along which i's PUTs are propagated.

4.3 Strong consistency

When the application using SPANStore for geo-replicated storage requires strong consistency of data, we rely on quorum consistency [17]. Quorum consistency imposes two requirements to ensure linearizability. For every data center i in an access set, 1) the subset of data centers to which i commits each of its PUTs—the PUT replica set for i—should intersect with the PUT replica set for every other data center in the access set, and 2) the GET replica set for i should intersect with the PUT replica set for every data center in the access set. The cardinality

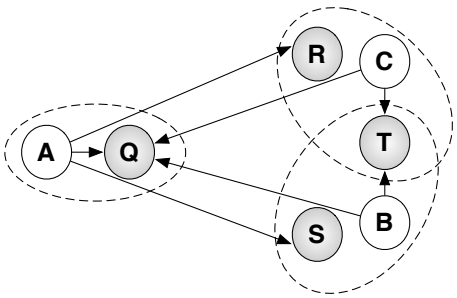


Figure 10: Example use of asymmetric quorum sets. Solid unshaded circles represent data centers in the access set, and shaded circles are data centers that host replicas. Directed edges represent transfers to PUT replica sets, and dashed ovals represent GET replica sets.

of these intersections should be greater than the number of failures that the application wants SPANStore to tolerate.

In our design, we use asymmetric quorum sets [20] to instantiate quorum consistency as above. With asymmetric quorum sets, the PUT and GET replica sets for any particular data center can differ. We choose to use asymmetric quorum sets due to the non-uniform geographic distribution of data centers. For example, as seen in Figure 3(a), EC2 data centers have between 2 and 16 other data centers within 200ms of them. Figure 10 shows an example where, due to this non-uniform geographic distribution of data centers, asymmetric quorum sets reduce cost and help meet lower latency SLOs.

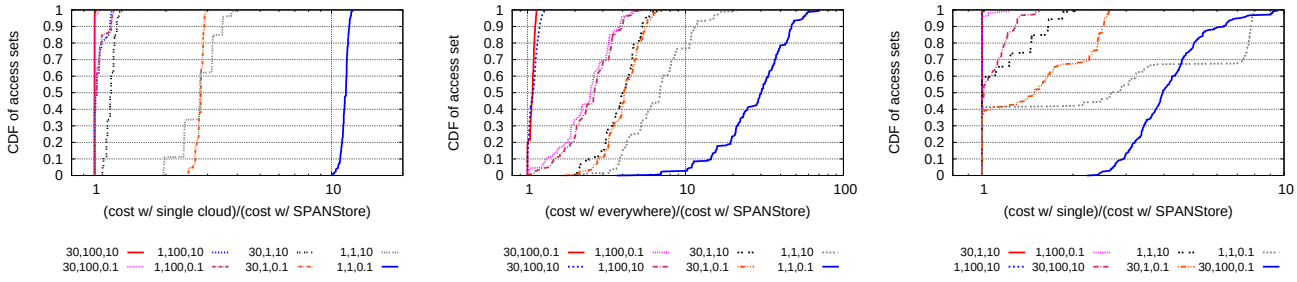
The integer program that the Placement Manager uses for choosing replication policies in the strong consistency setting mirrors the program for the eventual consistency case in several ways: 1) PUTs can be relayed via other data centers to reduce networking costs, 2) storage costs are incurred for maintaining a copy of every object at every data center that is in the union of the GET and PUT replica sets of all data centers in the access set, and 3) for every GET operation at data center i, one GET request's price and the price for transferring a copy of the object over the network is incurred at every data center in i's GET replica set.

However, the integer program for the strong consistency setting does differ from the program used in the eventual consistency case in three significant ways. First, for every data center in the access set, the PUT and GET replica sets for that data center may differ. Second, we constrain these replica sets so that every data center's PUT and GET replica sets have an intersection of at least $2f + 1$ data centers with the PUT replica set of every other data center in the access set. Finally, PUT operations at any data center i are propagated only to the data centers in i's PUT replica set, and these updates are propagated via at most one hop.

4.4 Cost savings

As an example of the cost savings that SPANStore can enable, Figure 11 compares cost with SPANStore against three other replication strategies. We consider a) GET:PUT ratios of 1 and 30, b) average object sizes of 1 KB and 100 KB, and c) aggregate data size of 0.1 TB and 10 TB. Our choice of these workload parameters is informed by the GET:PUT ratio of 30:1 and objects typically smaller than 1 KB seen in Facebook's workload [11]. In all workload settings, we fix the number of GETs at 100M and compute cost over a 30 day period. The results shown here are for the strong consistency case with SLOs for the 90^{th} percentile GET and PUT latencies set at 250ms and 830ms; 830 ms is the minimum PUT latency SLO if every object was replicated at all data centers in its access set.

Comparison with single-cloud deployment. First, Figure 11(a) compares the cost with SPANStore with the minimum cost required if we used only Amazon S3's data centers for storage. When the objects are small (i.e., average object size of 1KB), SPANStore's cost savings predominantly stem from differences in pricing



(a) (b) (c)

Figure 11: Cost savings enabled by SPANStore compared to (a) when data is replicated across the data centers of a single cloud service, and when using the (b) *Everywhere* and (c) *Single* replication policies. Legend indicates GET:PUT ratio, average object size (in KB), and overall data size (in TB). Note log-scale on x-axis.

per GET and PUT request across cloud providers. When the average object size is 100KB, SPANStore still offers cost benefits for a sizeable fraction of access sets when the PUT/GET ratio is 1 and overall data size is small. In this case, since half of the workload (i.e., all PUT operations) require propagation of updates to all replicas, SPANStore enables cost savings by exploiting discrepancies in network bandwidth pricing across cloud services. Furthermore, when the total data size is 10 TB, SPANStore reduces storage costs by exploiting the greater density of data centers and storing fewer copies of every object.

Comparison with fixed replication policies. Figure 11 also compares the cost incurred when using SPANStore with that imposed by two fixed replication policies: *Everywhere* and *Single*. With the *Everywhere* policy, every object is replicated at every data center in the object's access set. With the *Single* replication policy, any object is stored at one data center that minimizes cost among all single replica alternatives that satisfy the PUT and GET latency SLOs.

In Figure 11(b), we see that SPANStore significantly outdoes *Everywhere* in all cases except when GET:PUT ratio is 30 and average object size is 100KB. On the other hand, in Figure 11(c), we observe a bi-modal distribution in the cost savings as compared to *Single* when the object size is small. We find that this is because, for all access sets that do not include EC2's Sydney data center, using a single replica (at some data center on Azure) proves to be cost-optimal; this is again because the lower PUT/GET costs on Azure compensate for the increased network bandwidth costs. When the GET:PUT ratio is 1 and the average object size is 100KB, SPANStore saves cost compared to *Single* by judiciously combining the use of multiple replicas.

5 Conclusion

In summary, high latency variance and the need for judicious data replication are two challenges for any geo-distributed web service deployed in the cloud. To address these challenges, we presented two systems. First, CosTLO combines the use of multiple forms of redundancy by inferring models of replication and load balancing within cloud storage services. Second, SPANStore replicates data across the data centers of multiple cloud providers in order to exploit pricing discrepancies and to benefit from the greater geographical density of data centers. Together, these systems enable cost-effective support for low latency data access in the cloud.

Acknowledgments

Many others contributed to the design, implementation, and evaluation of the systems presented in this paper: Michael Butkiewicz, Dorian Perkins, and Ethan Katz-Bassett for SPANStore, and Curtis Yu for CosTLO. This

work was supported in part by a NetApp Faculty Fellowship and by the National Science Foundation under grants CNS-1150219 and CNS-1463126.

References

[1] Amazon simple storage service - server access log format. http://docs.aws.amazon.com/AmazonS3/latest/dev/LogFormat.html.

[2] Announcing Amazon S3 reduced redundancy storage. http://aws.amazon.com/about-aws/whats-new/2010/05/19/announcing-amazon-s3-reduced-redundancy-storage/.

[3] AWS global infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/.

[4] Azure regions. https://azure.microsoft.com/en-us/regions/.

[5] Azure storage pricing. http://azure.microsoft.com/en-us/pricing/details/storage/.

[6] Latency is everywhere and it costs you sales - how to crush it. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[7] Windows Azure storage logging: Using logs to track storage requests. http://blogs.msdn.com/b/windowsazurestorage/archive/2011/08/03/windows-azure-storage-logging-using-logs-to-track-storage-requests.aspx.

[8] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, 2010.

[9] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Rao. Improving web availability for clients with MONET. In *NSDI*, 2005.

[10] M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *OSDI*, 2014.

[11] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[12] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *IMC*, 2006.

[13] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *SoCC*, 2010.

[14] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure storage: A highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[15] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[17] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, 1979.

[18] R. Li, S. Wang, H. Deng, R. Wang, and K. C.-C. Chang. Towards social user profiling: Unified and discriminative influence model for inferring home locations. In *KDD*, 2012.

[19] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[20] J.-P. Martin, L. Alvisi, and M. Dahlin. Small byzantine quorum systems. In *DSN*, 2002.

[21] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *CoNEXT*, 2013.

[22] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, 2013.

[23] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-effective redundancy for lower latency variance on cloud storage services. In *NSDI*, 2015.

Towards Geo-Distributed Machine Learning

Ignacio Cano[w*] Markus Weimer[m] Dhruv Mahajan[f†] Carlo Curino[m]
Giovanni Matteo Fumarola[m] Arvind Krishnamurthy[w]

[w]University of Washington
{icano,arvind}@cs.washington.edu
[m]Microsoft
{mweimer,ccurino,gifuma}@microsoft.com
[f]Facebook
dhruvm@fb.com

Abstract

Latency to end-users and regulatory requirements push large companies to build data centers all around the world. The resulting data is "born" geographically distributed. On the other hand, many Machine Learning applications require a global view of such data in order to achieve the best results. These types of applications form a new class of learning problems, which we call Geo-Distributed Machine Learning (GDML). Such applications need to cope with: 1) scarce and expensive cross-data center bandwidth, and 2) growing privacy concerns that are pushing for stricter data sovereignty regulations.

Current solutions to learning from geo-distributed data sources revolve around the idea of first centralizing the data in one data center, and then training locally. As Machine Learning algorithms are communication-intensive, the cost of centralizing the data is thought to be offset by the lower cost of intra-data center communication during training.

In this work, we show that the current centralized practice can be far from optimal, and propose a system architecture for doing geo-distributed training. Furthermore, we argue that the geo-distributed approach is structurally more amenable to dealing with regulatory constraints, as raw data never leaves the source data center. Our empirical evaluation on three real datasets confirms the general validity of our approach, and shows that GDML is not only possible but also advisable in many scenarios.

1 Introduction

Modern organizations have a planetary footprint. Data is created where users and systems are located, *all around the globe*. The reason for this is two-fold: 1) minimizing latency between serving infrastructure and end-users, and 2) respecting regulatory constraints, that might require data about citizens of a nation to reside within the

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*]Most of this work was done while interning at Microsoft.
[†]Most of this work was done while at Microsoft.

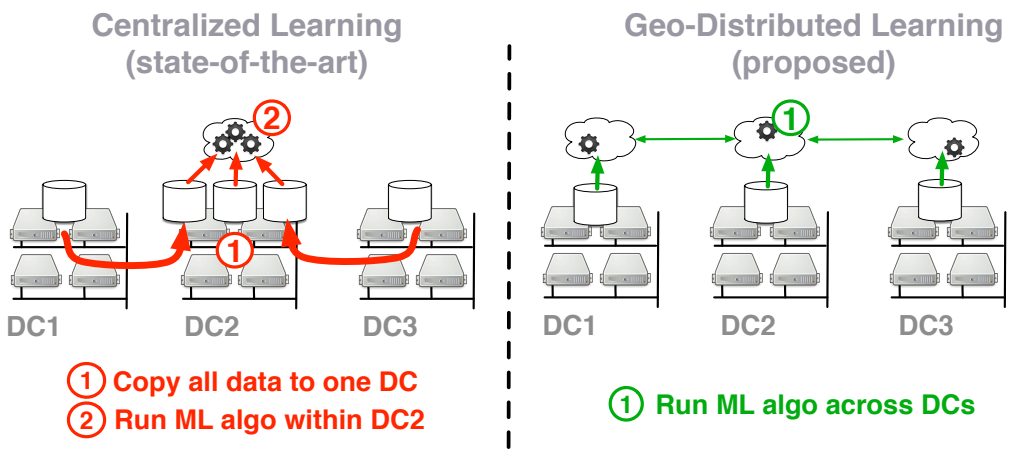


Figure 1: Centralized vs Geo-distributed Learning.

nation's borders. On the other hand, many machine learning applications require access to all that data at once to build accurate models. For example, fraud prevention systems benefit tremendously from the global picture in both finance and communication networks, recommendation systems rely on the maximum breadth of data to overcome cold start problems, and the predictive maintenance revolution is only possible because of data from all markets. These types of applications that deal with geo-distributed datasets belong to a new class of learning problems, which we call Geo-Distributed Machine Learning (GDML).

The state-of-the-art approach to Machine Learning from decentralized datasets is to centralize them. As shown in the left-side of Figure 1, this involves a two-step process: 1) the various partitions of data are copied into a single data center (DC)—thus recreating the overall dataset in a central location, and 2) learning takes place there, using existing intra-data center technologies. Based on conversations with practitioners at Microsoft, we gather that this *centralized* approach is predominant in most practical settings. This is consistent with reports on the infrastructures of other large organizations, such as Facebook [1], Twitter [2], and LinkedIn [3]. The reason for its popularity is two-fold, on the one hand, centralizing the data is the easiest way to reuse existing ML frameworks [4–6]), and on the other hand, Machine Learning algorithms are notoriously communication-intensive, and thus assumed to be non-amenable to cross-data center execution—as we will see, in many practical settings, we can challenge this assumption.

The centralized approach has two key shortcomings:

1. It consumes large amounts of cross-data center (X-DC) bandwidth (in order to copy the raw data to a single location). X-DC bandwidth has been shown to be scarce, expensive, and growing at a slower pace than most other intra-data center (in-DC) resources [7–10].

2. It requires raw data to be copied across data centers, thus potentially across national borders. While international regulations are quickly evolving, the authors of this paper speculate that the growing concerns regarding privacy and data sovereignty [11, 12] might become a key limiting factor to the applicability of centralized learning approaches.

We hypothesize that both challenges will persist or grow in the future [13, 14].

In this paper, we propose the geo-distributed learning approach (right-side of Figure 1), where raw data is kept in place, and learning tasks are executed in a X-DC fashion. We show that by leveraging communication-efficient algorithmic solutions [15] together with a distributed resource management fabric, this approach can achieve orders of magnitude lower X-DC bandwidth consumption in practical settings. Moreover, as the geo-distributed learning approach does not require to copy raw data outside their native data center (only statistics

and estimates are copied), it is structurally better positioned to deal with evolving regulatory constraints. A detailed study of this legal aspect of the geo-distributed approach is beyond the scope of this paper.

The solution we propose serves as an initial stand-in for a new generation of geo-distributed learning systems and algorithms, and allows us to present the first study on the relative efficiency of centralized vs. distributed approaches. In this paper, we concentrate on two key metrics: X-DC bandwidth consumption and learning runtime. We show experimentally that properly designed centralized solutions can achieve faster learning times (when the data copy latency is hidden by streaming data as it becomes available), but that distributed solutions can achieve much lower X-DC bandwidth utilization, and thus substantially lower cost for large-scale learning.

Note that while the above two metrics are of great practical relevance, many other dimensions (e.g., resilience to catastrophic DC failures) are worth considering when comparing alternative approaches. In §6, we briefly list these and other open problems that emerge when considering this new class of learning tasks: Geo-Distributed Machine Learning (GDML).

Summarizing, our main contributions are:

- We introduce GDML, an important new class of learning system problems that deals with geo-distributed datasets, and provide a study of the relative merits of state-of-the-art centralized solutions vs. novel geo-distributed alternatives.
- We propose a system that builds upon Apache Hadoop YARN [16] and Apache REEF [17], and extends their functionality to support multi-data center ML applications. We adopt a communication-sparse learning algorithm [15], originally designed to accelerate learning, and leverage it to lower the bandwidth consumption (i.e., cost) for geo-distributed learning.
- We present empirical results from both simulations and a real deployment across continents, which show that under common conditions distributed approaches can trade manageable penalties in training latency (less than $5\times$) for massive bandwidth reductions (multiple orders of magnitude).
- Finally, we highlight that GDML is a tough new challenge, and that many problems, such as WAN fault-tolerance, regulatory compliance, privacy preservation, X-DC scheduling coordination, latency minimization, and support for broader learning tasks remain open.

The remainder of this paper presents these findings as follows: §2 formalizes the problem setting, §3 introduces our approach, §3.1 introduces the algorithmic solution, and §3.2 describes our system implementation. We explain the evaluation setup and show the experimental results in §4. Finally, we discuss related work in §5, open problems in §6 and conclusions in §7.

2 Problem Formulation

In order to facilitate a study of the state-of-the-art centralized approach in contrast to potential geo-distributed alternatives, we formalize the problem below in two dimensions: 1) we specify assumptions about the data, its size and partitioning, and 2) we restrict the set of learning problems to the well known Statistical Query Model (SQM [18]) class.

2.1 Data distribution

We assume the dataset D of N examples (x_i, y_i), where $x_i \in \mathbb{R}^d$ denotes the feature vector and $y_i \in \{-1, 1\}$ denotes the label of example i, to exist in $p \in \{1, \ldots, P\}$ partitions D_p, each of which is generated in one of P data centers. Those P partitions consist of n_p examples each, with $N = \sum_p n_p$. Let d be the dimensionality of the feature vectors and $\bar{d}$ the average sparsity (number of non-zeros) per example. The total size of each partition

can be (roughly) estimated as $s_p = n_p \cdot \bar{d}$. Although this approximation serves our purposes, in order to have a more precise estimate of the partition sizes, we should not rely on a single $\bar{d}$ value (average instance sparsity across the entire data). This is because the sparsity of instance vectors may depend on the data center location. For example, in the case of a recommendation application, the US data center might have more dense feature vectors (user profiles) than those in a data center in South America.

Further, let p^* be the index of the largest partition, i.e., $p^* = \arg\max_p n_p$. Then, the total X-DC transfer needed to centralize the dataset is:

$$T_C = (N - n_{p^*})\, \bar{d} \tag{1}$$

The goal here is to transfer all instances to the data center that holds the largest subset of instances. Data compression is commonly applied to reduce this size, but only by a constant factor.

2.2 Learning Task

For any meaningful discussion of the relative merits of the centralized approach when compared to alternatives, we need to restrict the set of learning problems we consider. Here, we choose those that fit the Statistical Query Model (SQM) [18]. This model covers a wide variety of important practical machine learning models, including K-Means clustering, linear and logistic regression, and support vector machines.

The *beauty* of algorithms that fit into the SQM model is that they can be written in a summation form, which allows them to be easily parallelized [19]. In SQM, the learning algorithm is allowed to obtain estimates of statistical properties of the examples (e.g., sufficient statistics, gradients) but cannot see the examples themselves [20]. In other words, the algorithm can be phrased purely in terms of statistical queries over the data, and those statistical queries decompose into the sum of a function applied to each example [19].

Let that function be denoted by $f_q \in \{f_1, f_2, \ldots f_Q\}$. A query result F_q is then computed as $F_q = \sum_{i=1}^{N} f_q(x_i, y_i)$. With the data partitioning, this can be rephrased as

$$F_q = \sum_{p=1}^{P} \sum_{i=1}^{n_p} f_q(x_i, y_i) \tag{2}$$

In other words, the X-DC transfer per statistical query is the size of the output of its query function f_q, which we denote as s_q. The total X-DC transfer then depends on the queries and the number of such queries issued, n_q, as part of the learning task, both of which depend on the algorithm executed and the dataset. Let us assume we know these for a given algorithm and dataset combination. Then, we can estimate the total X-DC transfer cost of a fully distributed execution as:

$$T_D = (P - 1) \sum_{q=1}^{Q} n_q\, s_q \tag{3}$$

Note that this relies on the cumulative and associative properties of the query aggregation: we only need to communicate one result of size s_q per data center. The data center that aggregates the results does not need to communicate any data over the X-DC network, thus the $P - 1$ term in (3).

With this formalization, the current state-of-the-art approach of centralizing the data relies on the assumption that $T_C \ll T_D$. However, it is not obvious why this should always be the case: the X-DC transfer of the centralized approach T_C grows linearly with the dataset size, whereas the X-DC transfer of a distributed approach T_D grows linearly with the size and number of the queries. Additionally, relatively large partitions per DC typically yield more meaningful statistics per DC. This in turn means that the learning algorithm needs *fewer* queries to converge, lowering T_D as the dataset size grows given a fixed number of partitions. Hence, it is apparent that the assumption that $T_C \ll T_D$ holds for some, but not all regimes. All things being equal, it seems

that larger datasets would favor the distributed approach. Similarly, all things being equal, larger query results and algorithms issuing more queries seem to favor the centralized approach.

In order to study this more precisely, we need to restrict the discussion to a concrete learning problem for which the queries q, their functions f_q and their output sizes s_q are known. Further, the number of such queries can be bounded by invoking the convergence theorems for the chosen learning algorithm. Here, we choose linear modeling to be the learning problem for its simplicity and rich theory. In particular, we consider the l_2 regularized linear learning problem.

Let $l(w\, x_i, y_i)$ be a continuously differentiable loss function with Lipschitz continuous gradient, where $w \in \mathbb{R}^d$ is the weight vector. Let $L_p(w) = \sum_{i \in D_p} l(w\, x_i, y_i)$ be the loss associated with data center p, and $L(w) = \sum_p L_p(w)$ be the total loss over all data centers. Our goal is to find w that minimizes the following objective function, which decomposes per data center:

$$f(w) = \frac{\lambda}{2}||w||^2 + L(w) = \frac{\lambda}{2}||w||^2 + \sum_p L_p(w) \tag{4}$$

where $\lambda > 0$ is the regularization constant. Depending on the loss chosen, this objective function covers important cases such as linear support vector machines (hinge loss) and logistic regression (logistic loss). Learning such model amounts to optimizing (4). Many optimization algorithms are available for the task, and in §3.1 we describe the one we choose.

It is important to note that one common statistical query of all those algorithms is the computation of the gradient of the model in (4) with respect to w. The size of that gradient (per partition and globally) is d. Hence, s_q for this class of models can be approximated by d. This allows us to rephrase the trade-off mentioned above. All things being equal, datasets with more examples (x_i, y_i) would tend to favor the distributed approach. Similarly, all things being equal, datasets with higher dimensionality d would generally lean towards the centralized setting.

3 Approach

In order to study the problem described above, we need two major artifacts: 1) a communication-efficient algorithm to optimize (4), and 2) an actual geo-distributed implementation of that algorithm. In this section, we describe both of these items in detail.

3.1 Algorithm

We focus on the X-DC communication costs. Hence, we need a communication-efficient algorithm capable of minimizing the communication between the data centers. It is clear from (3) that such an algorithm should try to minimize the number of queries whose output size is very large. In the case of (4), this means that the number of X-DC gradient computations should be reduced.

Recently, many communication-efficient algorithms have been proposed that trade-off local computation with communication [18, 21–24]. Here, we use the algorithm proposed by Mahajan et al. [25] to optimize (4), shown in Algorithm 1. We choose this algorithm because experiments show that it performs better than the aforementioned algorithms, both in terms of communication and running time [25]. The algorithm was initially designed for running in a traditional distributed Machine Learning setting, i.e., single data center.[1] In this work, we adapt it for X-DC training, a novel application that was not originally intended for.

The main idea of the algorithm is to trade-off in-DC computation and communication with X-DC communication. The minimization of the objective function $f(w)$ is performed using an iterative descent method in

[1] We confirmed this with the first author as per 11/2017.

which the r-th iteration starts from a point w^r, computes a direction d^r, and then performs a line search along that direction to find the next point $w^{r+1} = w^r + t\, d^r$.

We adapt the algorithm to support GDML in the following manner. Each node in the algorithm now becomes a data center. All the local computations like gradients and loss function on local data now involves both computation as well as in-DC communication among the nodes in the same data center. On the other hand, all global computations like gradient aggregation involves X-DC communication. This introduces the need for two levels of communication and control, described in more detail in §3.2.

In a departure from communication-heavy methods, this algorithm uses distributed computation for generating a good search direction d^r in addition to the gradient g^r. At iteration r, each data center has the current global weight vector w^r and the gradient g^r. Using its local data D_p, each data center can form an approximation $\hat{f}_p$ of f. To ensure convergence, $\hat{f}_p$ should satisfy a gradient consistency condition, $\nabla \hat{f}_p(w^r) = g^r$. The function $\hat{f}_p$ is approximately[2] optimized using a method M to get the local weight vector w_p, which enables the computation of the local direction $d_p = w_p - w^r$. The global update direction is chosen to be $d^r = \frac{1}{P}\sum_p d_p$, followed by a line search to find w^{r+1}.

In each iteration, the computation of the gradient g^r and the direction d^r requires communication across data centers. Since each data center has the global approximate view of the full objective function, the number of iterations required are significantly less than traditional methods, resulting in orders of magnitude improvements in terms of X-DC communication.

The algorithm offers great flexibility in choosing $\hat{f}_p$ and the method M used to optimize it. A general form of $\hat{f}_p$ for (4) is given by:

$$\hat{f}_p(w) = \frac{\lambda}{2}||w||^2 + \tilde{L}_p(w) + \hat{L}_p(w) \tag{5}$$

where $\tilde{L}_p$ is an approximation of the total loss L_p associated with data center p, and $\hat{L}_p(w)$ is an approximation of the loss across all data centers except p, i.e., $L(w) - L_p(w) = \sum_{q \neq p} L_q(w)$. Among the possible choices suggested in [25], we consider the following quadratic approximations[3] in this work:

$$\tilde{L}_p(w) = \nabla L_p(w^r)(w - w^r) + \frac{1}{2}(w - w^r)^T H_p^r (w - w^r) \tag{6}$$

$$\hat{L}_p(w) = (\nabla L(w^r) - \nabla L_p(w^r))(w - w^r) + \frac{P-1}{2}(w - w^r)^T H_p^r (w - w^r) \tag{7}$$

where H_p^r is the Hessian of L_p at w^r. Replacing (6) + (7) in (5) we have the following objective function:

$$\hat{f}_p(w) = \frac{\lambda}{2}||w||^2 + g^r \cdot (w - w^r) + \frac{P}{2}(w - w^r)^T H_p^r (w - w^r) \tag{8}$$

We use the conjugate gradient (CG) algorithm [26] to optimize (8). Note that each iteration of CG involves a statistical query with output size d to do a hessian-vector computation. However, this query involves only in-DC communication and computation, whereas for traditional second order methods like TRON [27], it will involve X-DC communication.

Discussion Let T_{outer} be the number of iterations required by the algorithm to converge. Each iteration requires two queries with output size $s_q = d$ for the gradient and direction computation, and few queries of output

[2]Mahajan et al. [25] proved linear convergence of the algorithm even when the local problems are optimized approximately.

[3]One can simply use $\tilde{L}_p = L_p$, i.e., the exact loss function for data center p. However, Mahajan et. al [25] showed better results if the local loss function is also approximated.

Algorithm 1 Functional Approximation based Distributed Learning Algorithm (FADL)

Choose w^0
for $r = 0, 1...$ **do**
  Compute g^r (X-DC communication)
  Exit if $||g^r|| \leq \epsilon_g ||g^0||$
  for $p = 1, ..., P$ (in parallel) **do**
    Construct $\hat{f}_p(w)$ ((8))
    $w_p \leftarrow$ Optimize $\hat{f}_p(w)$ (in-DC communication)
  end for
  $d^r \leftarrow \frac{1}{P} \sum_p w_p - w^r$ (X-DC communication)
  Line Search to find t (negligible X-DC communication)
  $w^{r+1} \leftarrow w^r + t\, d^r$
end for

size $s_q = 1$ for the objective function computation in the line search. Since $d \gg 1$, we can ignore the X-DC communication cost for the objective function computation. Hence, we can rewrite (3) as $T_D = 2\,(P-1)\,d\,T_{outer}$. Hence, for T_D to be less than T_C the following must hold:

$$2\,(P-1)\,d\,T_{outer} \quad < \quad (N - n_{p^*})\,\bar{d} \tag{9}$$

In practice, the typical value of P (data centers) is relatively small (in the 10s). Since there are few data centers (i.e., few partitions of the data), the above algorithm will take only few (5-20) outer iterations to converge. In fact, in all our experiments in §4, the algorithm converges in less than 7 iterations. This means that as long as the total size of the data is roughly more than $2-3$ orders of magnitude greater than the dimensionality d,[4] doing geo-distributed learning would reduce the X-DC transfers compared to the centralized approach.

3.2 Distributed Implementation

We need a flexible system that can run in two regimes, i.e., in-DC and X-DC, without requiring two separate implementations. Here, we describe such system. Note that our system is *not tied* to the specific algorithm described above, rather, it exposes a generic framework suitable for geo-distributed and centralized implementations of, at least, the algorithms expressible in the Statistical Query Model.

SQM can be implemented using only Broadcast and Reduce operators (including the algorithm described in §3.1). As part of this work, we add X-DC versions of those to Apache REEF, which provides the basic control flow for our implementation. Moreover, our system needs to obtain resources (CPU cores, RAM) across different data centers in a uniformly basis. We manage those resources using Apache Hadoop YARN. Finally, our system leverages YARN's new federation feature (released as part of Apache Hadoop YARN 2.9) to *view* multiple data centers as a single one. We extend Apache REEF with support for that. Below, we provide more details on our three-layer architecture, from bottom to top.[5]

3.2.1 Resource Manager: Apache Hadoop YARN

A resource manager (RM) is a platform that dynamically leases resources, known as containers, to various competing applications in a cluster. It acts as a central authority and negotiates with potentially many Application Masters (AM) the access to those containers [17]. The most well known implementations are Apache Hadoop

[4] Note that for large datasets this is typically the case.

[5] All changes to Apache REEF and Apache Hadoop YARN have been contributed back to the projects where appropriate.

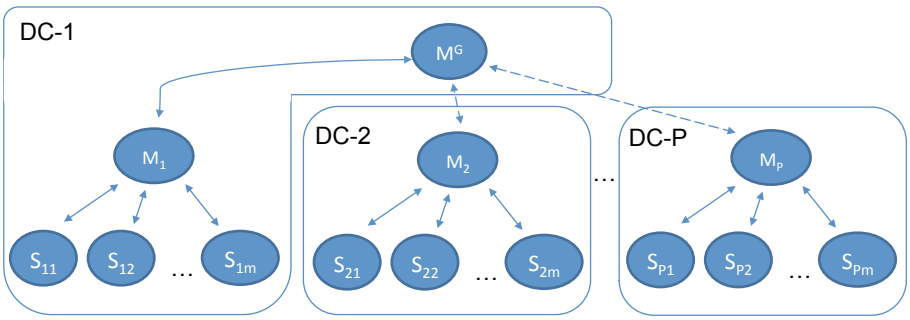


Figure 2: Multi-Level Master/Slave Communication Tree with P data centers, each with its own data center master (M_i) and slaves (S_{ij}). The global master M^G is physically located in DC-1. The solid and dashed lines refer to in-DC and X-DC links respectively. Our current implementation supports recovery from slave failures. Fault-tolerance at the master levels is left for future work.

YARN [16], Apache Mesos [28] and Google Omega [29]. All of these systems are designed to operate *within* one DC and multiplex applications on a collection of shared machines.

In our setting, we need a similar abstraction, but it must *span* multiple DCs. We build our solution on top of Apache Hadoop YARN. As part of our effort to scale-out YARN to Microsoft-scale clusters (tens of thousands of nodes), we have been contributing to Apache a new architecture that allows to *federate* multiple clusters [30]. This effort was not originally intended to operate in a X-DC setting, and as such, was focused on *hiding* from the application layer the different sub-clusters. It is worth mentioning that single DC federation is deployed in production at scale at Microsoft. As part of this work, we have been experimenting and extending this system, leveraging its transparency yet providing enough visibility of the network topology to our application layer (Apache REEF). As a result, we can run a *single* application that spans different data centers in an efficient manner.

3.2.2 Control Flow: Apache REEF

Apache REEF [17] provides a generalized control plane to ease the development of applications on resource managers. REEF provides a control flow master called Driver to applications, and an execution environment for tasks on containers called Evaluator. Applications are expressed as event handlers for the Driver to perform task scheduling (including fault handling) and the task code to be executed in the Evaluators. As part of this work, we extend REEF to support geo-federated YARN, including scheduling of resources to particular data centers.

REEF provides a group communications library that exposes Broadcast and Reduce operators similar to Message Passing Interface (MPI) [31]. Like MPI, REEF's group communications library is designed for the single data center case. We expand it to cover the X-DC case we study here.

3.2.3 GDML Framework

Statistical Query Model algorithms, such as the one introduced in §3.1, can be implemented using nothing more than Broadcast and Reduce operators [19], where data partitions reside in each machine, and the statistical query is Broadcast to those, while its result is computed on each machine and then aggregated via Reduce.

Both Broadcast and Reduce operations are usually implemented via communication trees in order to maximize the overall throughput of the operation. Traditional systems, such as MPI [31] implementations, derive much of their efficiency from intelligent (and fast) ways to establish these trees. Different from the in-DC environment where those are typically used, our system needs to work with network links of vastly different characteristics. X-DC links have higher latency than in-DC ones, whereas the latter have usually higher band-

widths [32]. Further, X-DC links are much more expensive than in-DC links, as they are frequently rented or charged-for separately, in the case of the public cloud.

In our system, we address these challenges with a heterogeneous, multi-level communication tree.

Figure 2 shows an example of the multi-level communication tree we use. A global Broadcast originates from M^G to the data center masters M_i, which in turn do a local Broadcast to the slave nodes S_{ij} in their own data centers. Conversely, local Reduce operations originate on those slave nodes, while the data center masters M_i aggregate the data prior to sending it to M^G for global aggregation.

To make this happen, the underlying implementation creates multiple communication groups. The global master M^G together with the data centers masters M_i, form the global communication group (GCG), where the global Broadcast / Reduce operations are performed, and used in the outer loop of Algorithm 1. Likewise, the slave nodes within each data center and their masters M_i form the local communication groups (LCG), where the local Broadcast / Reduce operations execute, and are used to optimize the local approximations $\hat{f}_p$ of (8).

Summary Overall, our system enables popular resource managing frameworks (Apache Hadoop YARN and Apache REEF) to be used effectively in the management of algorithms running in a geo-distributed fashion. Note that our framework *can work* with any algorithm that can be implemented using Broadcast and Reduce primitives. The specific changes our system embraces, perhaps surprising to software engineers but not to algorithm developers, make the geo-distribution *less* transparent in order for the algorithm author to explicitly control what to run in-DC and what to run across data centers.

4 Evaluation

The algorithm and system presented above allow us to evaluate the state-of-the-art of centralizing the data before learning in comparison with truly distributed approaches. In this section, we describe our findings, starting with the setup and definition of the different approaches used, followed by experimental results from both simulated and real deployments.

4.1 Experimental Setup

We report experiments on two deployments: a distributed deployment on Microsoft Azure across two data centers, and a large centralized cluster on which we simulate a multi-data center setup (2, 4, and 8 data centers). This simulated environment is our main test bench, and we mainly use it for multi-terabyte scale experiments, which are not cost-effective on public clouds. We use 256 slave nodes divided into 2-8 simulated datacenters in all our simulations. Further, all the experiments are done with the logistic loss function.

We ground and validate the findings from the simulations on a real cross-continental deployment on Microsoft Azure. We establish two clusters, one in a data center in Europe and the other on the U.S. west coast. We deploy two DS12 VMs into each of these clusters. Each of those VMs has 4 CPU cores and 28GB of RAM. We establish the site-to-site connectivity through a VPN tunnel using a High Performance VPN Gateway.[6]

4.2 Data

We use three datasets of user behavior data in web sites for our evaluation, two of which are publicly available. All of them are derived from click logs. Table 1 summarizes their statistics. CRITEO and KAGGLE are publicly available [33, 34]. The latter is a small subset of the former, and we use it for the smaller scale experiments in Azure. WBCTR is an internal Microsoft dataset. We vary the number of features in our experiments using hashing kernels as suggested in [35].

[6] `https://azure.microsoft.com/en-us/documentation/articles/vpn-gateway-about-vpngateways/`

Dataset	Examples (N)	Features (d)	Size	
			Model	Dataset
CRITEO	4B	5M	20MB	1.5TB
		10M	40MB	1.5TB
		50M	200MB	1.6TB
		100M	400MB	1.7TB
WBCTR	730M	8M	32MB	347GB
		16M	64MB	362GB
		80M	320MB	364GB
		160M	640MB	472GB
KAGGLE	46M	0.5M	2MB	8.5GB
		1M	4MB	8.5GB
		5M	20MB	9GB

Table 1: Datasets statistics. Dataset sizes reported are *after* compression. Weights in the models are represented in single-precision floating-point format (32 bits) with no further compression. Note that the average sparsity (number of non-zeros) of each of the dataset versions are very similar, thus we do not observe a significant size change after increasing the number of features.

The dataset sizes reported in Table 1 refer to compressed data. The compression/decompression is done using Snappy,[7] which enables high-speed compression and decompression with reasonable compression size. In particular, we achieve compression ratios of around 62-65% for the CRITEO and WBCTR datasets, and 50% for KAGGLE. Following current practice in large scale Machine Learning, our system performs all computations using double precision arithmetic, but communicates single precision floats. Hence, model sizes in Table 1 are reported based on single-precision floating point numbers.

In our experiments, we assume the datasets are randomly partitioned across the data centers, i.e., we assume each data center keeps an approximately equal number of instances. Note that although this is a strong assumption, as data in different data centers can be distributed differently, it holds true in some important production use cases we observe. In such cases, load balancing across data centers forces data to be "randomly" spread across them. However, this is not fully general, as other important GDML workloads require data to be close to the users (to achieve low latency interactions), thus strong geographically biases emerge. Besides the dataset sizes, in practice, data centers can also vary significantly in terms of their bandwidth and computational resources. We plan on addressing all these issues in future work.

4.3 Methods

We contrast the state-of-the-art approach of centralizing the data prior to learning with several alternatives, both within the regime requiring data copies and truly distributed:

centralized denotes the current state-of-the art, where we copy the data to one data center prior to training. Based on the data shipping model used, two variants of this approach arise:

> ***centralized-stream*** refers to a streaming copy model where the data is replicated as it arrives. When the learning job is triggered in a particular data center, the data has already been transferred there, therefore, no copy time is included in the job running time, and

[7] `https://github.com/xerial/snappy-java`

centralized-bulk refers to a batch replication scheme where the data still needs to be copied by the time the learning process starts, therefore, the copy time has an impact on the job running time, i.e., the job needs to wait until the transfer is made to begin the optimization.

We observe both flavors occur in practice. We simply refer to *centralized* when no distinction between its variants is required. This approach (and its variants) only performs *compressed* data transfers, and uses the algorithm described in §3.1 for solving the l_2 regularized linear classification problem mentioned in §2.

distributed builds the multi-level master/slave tree for X-DC learning, but does not use the communication-efficient algorithm in §3.1 to optimize (4), instead, it optimizes using TRON [27].

distributed-fadl uses the algorithm introduced in §3.1 to optimize (4), and similarly to *distributed*, it performs the optimization in a geo-distributed fashion, i.e., it leaves the data in place and runs a single job that spans training across data centers.

Both *distributed* and *distributed-fadl* methods represent the furthest departure from the current state-of-the-art as their execution is truly geo-distributed. Studying results from both allows us to draw conclusions about the relative merits of the system enabling truly geo-distributed training (*distributed*) as well as the system together with a communication-sparse algorithm (*distributed-fadl*).

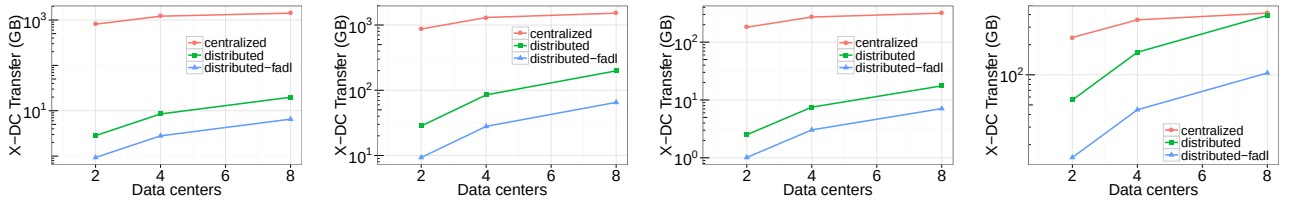
4.4 Results

In this section we present measurements from the methods introduced above, using the datasets described in §4.2. We focus on two key metrics: 1) total X-DC transfer size, and 2) latency to model.

4.4.1 Simulation

X-DC Transfer Figure 3 illustrates the total X-DC transfer of the different methods for different numbers of data centers. We only show two versions of CRITEO and WBCTR for space limitations, though the others follow the same patterns. In general, X-DC transfers increase with the number of data centers as there are more X-DC communication paths. As expected, increasing the model dimensionality also impacts the transfers in the distributed versions. In Figure 3b, the efficient distributed approach (*distributed-fadl*) performs at least one order of magnitude better than *centralized* in every scenario, achieving the biggest difference (2 orders of magnitude) for 2 data centers. In this setting, *centralized* (any variant) transfers half of the compressed data (870 GB) through the X-DC link before training, whereas *distributed-fadl* just needs 9 GBs worth of transfers to train the model. Likewise, in the WBCTR dataset (Figure 3d), we see the biggest difference in the 2 data centers scenario (1 order of magnitude). When the data is spread across 8 data centers, *centralized* transfers almost the same as *distributed*. In general, even the non communication-efficient *distributed* baseline also outperforms the current practice, *centralized*, on both datasets.

Objective / X-DC Transfer Trade-off Commercial deployments of Machine Learning systems impose deadlines and resource boundaries on the training process. This can make it impossible to run the algorithm till convergence. Hence, it is interesting to study the performance of the centralized and distributed approaches in relationship to their resource consumption. Figure 4 shows the relative objective function over time as a function of X-DC transfers for 2 and 8 data centers on the CRITEO and WBCTR datasets. We use the relative difference to the optimal function value, calculated as $(f - f^*)/f^*$, where f^* is the minimum value obtained across methods. X-DC transfers remain constant in the *centralized* (any variant) method as it starts the optimization after the data is copied, i.e., no X-DC transfers are made while training. In general, *distributed-fadl* achieves lower objective values much sooner in terms of X-DC transfers, which means that this method can get some meaningful



(a) CRITEO 10M (b) CRITEO 100M (c) WBCTR 16M (d) WBCTR 160M

Figure 3: X-DC transfer (in GB) versus number of data centers for two versions of CRITEO and WBCTR datasets (y-axis is in log scale). The method *distributed-fadl* consumes orders of magnitude less X-DC bandwidth than any variant (*stream* or *bulk*) of the compressed *centralized* approach. Moreover, a naive algorithm that does not economize X-DC communication, as is the case of the *distributed* method, also reduces transfers with respect to the current *centralized* state-of-the-art.

results faster. If an accurate model is not needed (e.g., 10^{-2} relative objective function value), *distributed-fadl* gives a quicker response. As we increase the number of data centers, X-DC communication naturally increases, which explains the right shift trend in the plots (e.g., Figures 4a and 4b).

Storage As the number of data centers increases, *centralized* (any variant) requires more space on disk. In particular, assuming the data is randomly partitioned across data centers, *centralized* stores at least $1.5\times$ more data than the distributed versions, with a maximum difference of almost $2\times$ when considering 8 data centers. On the other hand, both *distributed* and *distributed-fadl* only need to store the original dataset ($1\times$) throughout the different configurations.

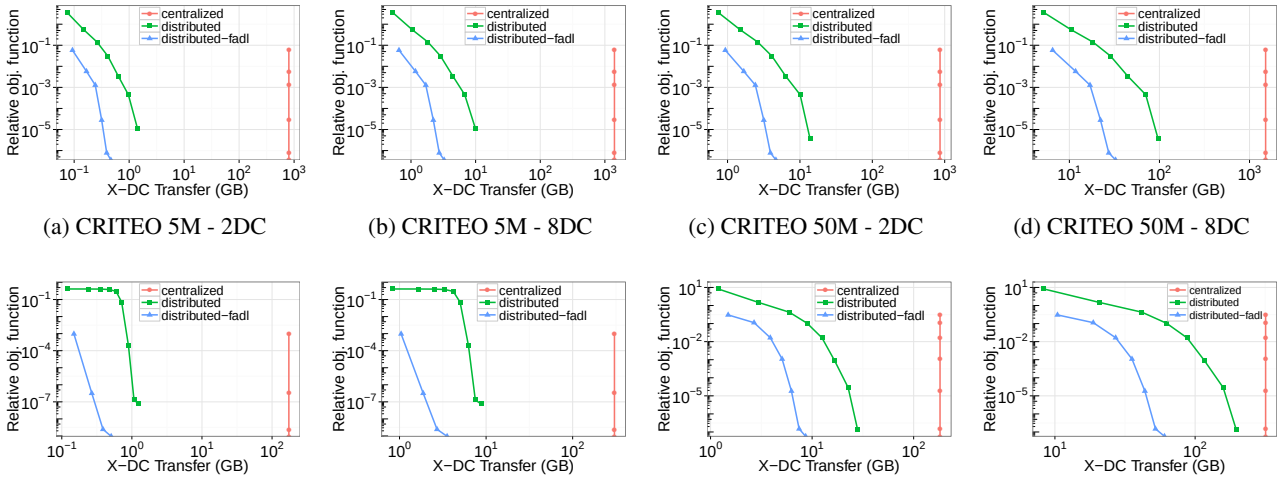
4.4.2 Real Deployment

X-DC Transfer To validate our simulation, we include Figure 5, which shows the relative objective function with respect to the X-DC bandwidth for the KAGGLE dataset in 2 Azure DCs (Western US and Western Europe). These experiments completely match our findings in the simulated environment. For the *centralized* approach, we transfer the data from EU to US, and run the optimization in the latter data center. Similar to Figure 4, the increase in the number of features expectedly causes more X-DC transfers (right shift trend in the plots). The efficient geo-distributed method *distributed-fadl* still communicates the least amount of data, almost 2 orders of magnitude less than the *centralized* (any variant) approach for the 500K model (Figure 5a).

Runtime Figure 6 shows the relative objective function over time for the 2 Azure data centers using the KAGGLE dataset. We normalize the time to the *centralized-stream* approach, calculated as t/t^*, where t^* is the overall time taken by *centralized-stream*. This method performs the fastest in every version of the dataset (500K, 1M, and 5M features) as the data has already been copied by the time it starts, i.e., no copy time overhead is added, and represents the lower bound in terms of running time.

Although the *centralized* approach always transfers compressed data, we do not take into account the compression/decompression time for computing the *centralized-bulk* runtime, which would have otherwise tied the results to the choice of the compression library. Figures 6a, 6b, and 6c show that *centralized-bulk* pays a high penalty for copying the data, it runs in approximately $8\times$ or more of its *stream* counterpart.

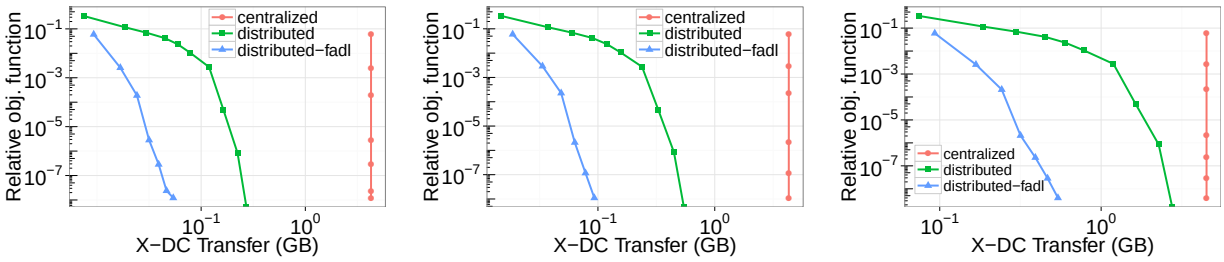
The communication-efficient *distributed-fadl* approach executes in $1.3\times$, $2.4\times$, and $7.4\times$ of the *centralized-stream* baseline for 500K, 1M, and 5M models respectively, which is a remarkable result given that it transfers orders of magnitude less data (Figure 5), and executes in a truly geo-distributed manner, respecting potentially strict regulatory constraints. Moreover, if we consider the relative objective function values commonly used



(a) CRITEO 5M - 2DC (b) CRITEO 5M - 8DC (c) CRITEO 50M - 2DC (d) CRITEO 50M - 8DC

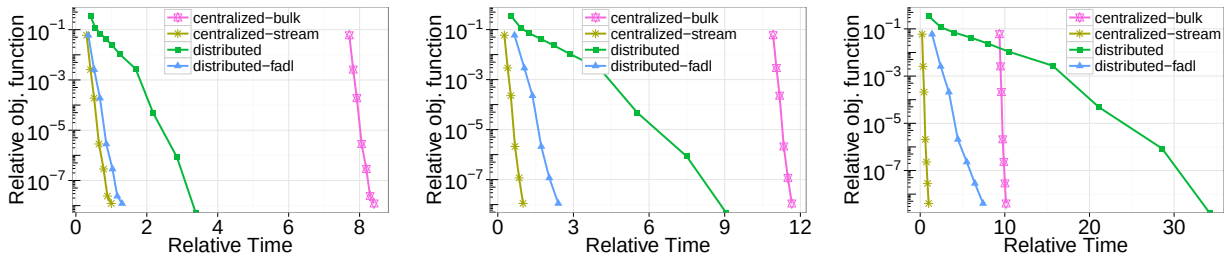
(e) WBCTR 8M - 2DC (f) WBCTR 8M - 8DC (g) WBCTR 80M - 2DC (h) WBCTR 80M - 8DC

Figure 4: Relative objective function (compared to the best) versus X-DC transfer (in GB) for 2 and 8 DCs for two versions of CRITEO and WBCTR datasets (both axis are in log scale). The method *distributed-fadl* achieves lower objective values much sooner in terms of X-DC transfers than the other methods. The *centralized* objective remains constant with respect to X-DC transfers throughout the optimization as it starts once the data has been transferred. The *distributed* method does incur in more transfers than *distributed-fadl*, although it also reduces the overhead of the *centralized* approach. Increasing the models dimensionality, naturally increases the X-DC transfers. Note that *centralized* refers to both of its variants (*stream* and *bulk*), and we only report compressed data transfers for this method.



(a) KAGGLE 500K (b) KAGGLE 1M (c) KAGGLE 5M

Figure 5: Relative objective function (compared to the best) versus X-DC transfer (in GB) for the KAGGLE dataset in 2 Azure data centers (both axis are in log scale). The increase in the model size explains the right shift trend in the plots. The method *distributed-fadl* consumes the least amount of X-DC bandwidth, at least 1 order less in every scenario, and 2 when using the 500K model. The objective/transfer pattern is similar to Figure 4. Both distributed methods transfer much less X-DC data than the *centralized* state-of-the-art. Note that *centralized* refers to both of its flavors (*stream* and *bulk*), and only transfers compressed data.



(a) KAGGLE 500K (b) KAGGLE 1M (c) KAGGLE 5M

Figure 6: Relative objective function (compared to the best) over time (relative to the *centralized-stream* method) for the KAGGLE dataset in 2 Azure data centers (y-axis is in log scale). The method *distributed-fadl* beats every approach but *centralized-stream*. This latter method is the best case scenario, where the data has already been copied and is available in a single data center when the job is executed. The *distributed-fadl* method lies very close to the optimum (*centralized-stream*), especially in low-dimensional models and when considering commonly accepted objective function values (10^{-4}, 10^{-5}). Both *distributed* and *distributed-fadl* performance degrades when the model size increases (as expected), but *distributed* does so much worse (6c), which further shows that in order to do Geo-Distributed Machine Learning, a communication-efficient algorithm is needed.

in practice to achieve accurate models (10^{-4}, 10^{-5}), this method's convergence time lies in the same ballpark as the lower bound *centralized-stream* in terms of running time. Still, *distributed-fadl* is way ahead in terms of X-DC transfers (orders of magnitude of savings in X-DC bandwidth), while at the same time it potentially complies with data sovereignty regulations.

Figure 6a shows that *distributed-fadl* performs very close to the best scenario, matching the intuition built in §2 that this method does very well on tasks with (relatively) small models and (relatively) large number of examples. Furthermore, this efficient method also runs faster than *distributed* in every setting, which further highlights the importance and benefits of the algorithm introduced in §3.1.

Finally, *distributed* performance degrades considerably as the model size increases. In particular, this method does a poor job when running with 5M features (Figure 6c), which concurs with the intuition behind the state-of-the-art *centralized* approach: copying the data offsets the communication-intensive nature of (naive) Machine Learning algorithms. We see that this intuition does not hold true for the efficient algorithm described in §3.1.

5 Related Work

Prior work on systems that deal with geographically distributed datasets exists in the literature.

The work done by Vulimiri et al. [8, 13] poses the thesis that increasing global data and scarce WAN bandwidth, coupled with regulatory concerns, will derail large companies from executing centralized analytics processes. They propose a system that supports SQL queries for doing X-DC analytics. Unlike our work, they do not target iterative machine learning workflows, neither do they focus on jobs latency. They mainly discuss reducing X-DC data transfer volume.

Pu et al. [36] proposes a low-latency distributed analytics system called Iridium. Similar to Vulimiri et al., they focus on pure data analytics and not on Machine Learning tasks. Another key difference is that Iridium optimizes task and data placement across sites to minimize query response time, while our system respects stricter sovereignty constraints and does not move raw data around.

JetStream [7] is a system for wide-area streaming data analysis that performs efficient queries on data stored "near the edge". They provide different approximation techniques to reduce the data size transfers at the ex-

pense of accuracy. One of such techniques is dropping some fraction of the data via sampling. Similar to our system, they only move *important* data to a centralized location for global aggregation (in our case, we only move statistics and models), and they compute local aggregations per site prior to sending (in our case, we perform local optimizations per data center using the algorithm described in §3.1). Another streaming application is distributed monitoring, which has focused on continuous tracking of complex queries over collections of physically distributed data streams. Effective solutions in their setting also need to guarantee communication efficiency over the underlying network [37].

Another line of research has focused on multi-site distributed search engines [38, 39]. Such work has shown to reduce the resource consumption in query processing as well as user perceived latency when compared to single-site centralized search engines [40, 41]. It bears some resemblance to our work but in the context of information retrieval.

Other existing Big Data processing systems, such as Parameter Server, Graphlab, or Spark [4–6, 42], efficiently process data in the context of a single data center, which typically employs a high-bandwidth, relatively low-cost network. To the best of our knowledge, they have not been tested in multi-data center deployments (and were not designed for it), where scarce WAN bandwidth makes it impractical to naively communicate parameters between locations. Instead, our system was specifically optimized to perform well on this X-DC setting.

Since our initial work on GDML systems [43, 44], other studies have emerged in the area. Among the most prominent ones we find Gaia [45], which also focuses on leveraging intelligent communication mechanisms, with more emphasis on reducing training times rather than X-DC transfers. Further, the work by Konečný et al. [46–48] introduces the concept of Federated Optimization, where the idea is to train a global model with data residing in users' mobile devices, instead of DCs. Their setting is very similar to GDML in the sense that communication efficiency is of utmost importance, but the cardinality is quite different (millions of devices as opposed to tens of DCs). This poses other research questions, e.g., what sample of devices to choose at a given point in time, how to alleviate the fact that mobile devices are frequently offline, etc.

Besides the systems solutions, the design of efficient distributed Machine Learning algorithms has also been the topic of a broad research agenda [21–25, 49–52]. In general, all these algorithms perform more complex computations to decrease the overall number of communication rounds. Some recent work uses model quantization (i.e., reduce the number of bits of the model parameters at the expense of potentially losing some accuracy) to reduce the communication cost [53]. The Terascale method [21] might be the best representative method from the Statistical Query Model class and is considered a state-of-the-art solver. CoCoA [24] represents the class of distributed dual methods that, in each outer iteration, solve (in parallel) several local dual optimization problems. Alternating Direction Method of Multipliers (ADMM) [22, 23] is a dual method different from the primal method we use here, however, it also solves approximate problems in the nodes and iteratively reaches the full batch solution. Recent follow up work [25] shows that the algorithm described in §3.1 performs better than the aforementioned ones, both in terms of communication rounds and running time.

6 Discussion and Future Work

GDML is an interesting, challenging and open area of research. Although we have proposed an initial and novel geo-distributed approach that shows substantial gains over the centralized state-of-the-art in many practical settings, many open questions remain, both from a systems and a Machine Learning perspective.

Perhaps, the most crucial aspect is fault tolerance. With data centers distributed across continents, consistent network connectivity is harder to ensure than within a single data center, and network partitioning is more likely to occur. On the other hand, a DC-level failure might completely compromise the centralized approach (if the primary DC is down), while the geo-distributed solution might continue to operate on the remaining data partitions. There has been some initial work [54] to make ML algorithms tolerant to missing data (e.g., machine

failures). This work assumes randomly distributed data across partitions. Hence, a failure removes an unbiased fraction of the data. In production settings, this is the case when multi-DC deployments are created for load-balancing (e.g., within a region)—we are aware of multiple such scenarios within Microsoft's infrastructure. However, cross-region deployments are often dictated by latency-to-end-user considerations. In such settings, losing a DC means losing a heavily biased portion of the population (e.g., all users residing in Western US). Coping with faults and tolerating transient or persistent data unavailability, as well as understanding the impact of different data distributions in convergence speeds are still open problems that will likely require both systems and ML contributions.

In this work we have restricted ourselves to linear models with l_2 regularization, and shown results on logistic regression models. It would be interesting to validate similar observations in other regularizers (e.g., l_1). More broadly, studying geo-distributed solutions that can minimize X-DC transfers for other complex learning problems such as trees, deep neural networks, etc., is still an open area of research.

Further, a truly geo-distributed approach surely does no worse than a centralized method when analyzed from regulatory and data sovereignty angles. Questions in this area arise not only at the global scale, where different jurisdictions might not allow raw data sharing, but also at the very small scale, e.g., between data stored in a private cluster and data shared in the cloud. We believe that studying the setup presented here from a privacy-preserving and regulatory-compliance angle will yield important improvements, and potentially inform regulators.

One aspect we did not cover is related to the work-cycle of these global data repositories and its impact in the efficacy of geo-distributed learning. If the data gets crunched by X algorithms once it is gathered into a single DC, including, perhaps, by algorithms that depend on each others inputs and/or encompass interactive workflows, the centralized methodology might be more effective than the geo-distributed one. This latter approach would increase communication by X-fold, whereas the centralized method would not incur in any extra communication. We consider that a more in-depth study of which approach (centralized or geo-distributed) is more adequate for different problem settings is still missing. Even more, we have not yet addressed the issue of whether a hybrid method that combines both centralized and geo-distributed learning could be more suitable under certain circumstances.

Besides presenting early results in this area, this paper is intended as an open invitation to researchers and practitioners from both systems and ML communities. We foresee the need for substantial advances in theory, algorithms and system design, as well as the engineering of a whole new practice of Geo-Distributed Machine Learning (GDML). To that end, we contributed back all the changes done to Apache Hadoop YARN and Apache REEF as part of this work.

7 Conclusions

Large organizations have a planetary footprint with users scattered in all continents. Latency considerations and regulatory requirements motivate them to build data centers all around the world. From this, a new class of learning problems emerge, where global learning tasks need to be performed on data "born" in geographically disparate data centers, which we call Geo-Distributed Machine Learning (GDML).

To the best of our knowledge, this aspect of Machine Learning has not been studied in great detail before, despite being faced by practitioners on a daily basis.

In this work, we introduce and formalize this problem, and challenge common assumptions and practices. Our empirical results show that a geo-distributed system, combined with communication-parsimonious algorithms, can deliver a substantial reduction in costly and scarce cross data center bandwidth. Further, we speculate distributed solutions are structurally better positioned to cope with the quickly evolving regulatory frameworks.

To conclude, we acknowledge this work is just a first step of a long journey, which will require significant advancements in theory, algorithms and systems.

Acknowledgments

We are grateful to our reviewers and colleagues for their help and comments on earlier versions of this paper. One of the authors was supported by the Argentine Ministry of Science, Technology and Productive Innovation with the program BEC.AR. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors nor the authors' affiliations.

References

[1] Ashish Thusoo et al. Data Warehousing and Analytics Infrastructure at Facebook. SIGMOD, 2010.

[2] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. The Unified Logging Infrastructure for Data Analytics at Twitter. *PVLDB*, 2012.

[3] Aditya Auradkar et al. Data Infrastructure at LinkedIn. In *ICDE*, 2012.

[4] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2. USENIX Association, 2012.

[5] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012.

[6] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, October 2014. USENIX Association.

[7] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S. Pai, and Michael J. Freedman. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 275–288. USENIX Association, 2014.

[8] Ashish Vulimiri, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 323–336, Oakland, CA, 2015. USENIX Association.

[9] Nikolaos Laoutaris, Michael Sirivianos, Xiaoyuan Yang, and Pablo Rodriguez. Inter-datacenter Bulk Transfers with Netstitcher. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 74–85. ACM, 2011.

[10] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The Cost of a Cloud: Research Problems in Data Center Networks. *ACM SIGCOMM computer communication review*, 39(1):68–73, 2008.

[11] Martin Rost and Kirsten Bock. Privacy by Design and the New Protection Goals. *DuD, January*, 2011.

[12] European Commission press release. Commission to pursue role as honest broker in future global negotiations on Internet Governance. `http://europa.eu/rapid/press-release_IP-14-142_en.htm`.

[13] Ashish Vulimiri, Carlo Curino, Brighten Godfrey, Konstantinos Karanasos, and George Varghese. WANalytics: Analytics for a Geo-Distributed Data-Intensive World. *CIDR 2015*, January 2015.

[14] Court of Justice of the European Union Press Release No 117/15. The Court of Justice declares that the Commission's US Safe Harbour Decision is invalid, 2015. `http://g8fip1kplyr33r3krz5b97d1.wpengine.netdna-cdn.com/wp-content/uploads/2015/10/schrems-judgment.pdf`. Accessed 2015-10-17.

[15] Dhruv Mahajan, S. Sathiya Keerthi, S. Sundararajan, and Léon Bottou. A Functional Approximation Based Distributed Learning Algorithm. *CoRR*, abs/1310.8418, 2013.

[16] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16. ACM, 2013.

[17] Markus Weimer, Yingda Chen, Byung-Gon Chun, Tyson Condie, Carlo Curino, Chris Douglas, Yunseong Lee, Tony Majestro, Dahlia Malkhi, Sergiy Matusevych, Brandon Myers, Shravan Narayanamurthy, Raghu Ramakrishnan, Sriram Rao, Russel Sears, Beysim Sezgin, and Julia Wang. REEF: Retainable Evaluator Execution Framework. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1343–1355. ACM, 2015.

[18] Michael Kearns. Efficient Noise-tolerant Learning from Statistical Queries. *J. ACM*, 45(6):983–1006, nov 1998.

[19] Cheng tao Chu, Sang K. Kim, Yi an Lin, Yuanyuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y. Ng. Map-Reduce for Machine Learning on Multicore. In B. Schölkopf, J.C. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, 2007.

[20] Vitaly Feldman. A complete characterization of statistical query learning with applications to evolvability. *J. Comput. Syst. Sci.*, 78(5):1444–1459, 2012.

[21] Alekh Agarwal, Oliveier Chapelle, Miroslav Dudík, and John Langford. A Reliable Effective Terascale Linear Learning System. *Journal of Machine Learning Research*, 15:1111–1133, 2014.

[22] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Found. Trends Mach. Learn.*, 3(1):1–122, January 2011.

[23] Caoxie Zhang, Honglak Lee, and Kang G. Shin. Efficient Distributed Linear Classification Algorithms via the Alternating Direction Method of Multipliers. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2012, La Palma, Canary Islands, April 21-23, 2012*, pages 1398–1406, 2012.

[24] Martin Jaggi, Virginia Smith, Martin Takác, Jonathan Terhorst, Sanjay Krishnan, Thomas Hofmann, and Michael I. Jordan. Communication-Efficient Distributed Dual Coordinate Ascent. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3068–3076, 2014.

[25] Dhruv Mahajan, Nikunj Agrawal, S. Sathiya Keerthi, S. Sundararajan, and Léon Bottou. An efficient distributed learning algorithm based on effective local functional approximations. *Journal of Machine Learning Research*, 2015.

[26] Jonathan R Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical report, Pittsburgh, PA, USA, 1994.

[27] Chih-Jen Lin, Ruby C. Weng, and S. Sathiya Keerthi. Trust Region Newton Method for Logistic Regression. *J. Mach. Learn. Res.*, 9:627–650, 2008.

[28] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308. USENIX Association, 2011.

[29] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013.

[30] Apache Hadoop community. Scaling out YARN via Federation. `https://issues.apache.org/jira/browse/YARN-2915`, 2016.

[31] William D. Gropp, Steven Huss-Lederman, Andrew Lumsdaine, and Inc netLibrary. *MPI : The Complete Reference. Vol. 2. , The MPI-2 extensions*. Scientific and engineering computation series. Cambridge, Mass. MIT Press, 1998.

[32] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 242–253. ACM, 2011.

[33] Criteo Labs. Terabyte Click Logs, 2015. `http://labs.criteo.com/downloads/download-terabyte-click-logs/`. Accessed 2016-02-03.

[34] Kaggle Criteo Labs. Display Advertising Challenge, 2014. `https://www.kaggle.com/c/criteo-display-ad-challenge`. Accessed 2015-09-30.

[35] K.Q. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature Hashing for Large Scale Multitask Learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1113–1120. ACM, 2009.

[36] Qifan Pu, Ganesh Anantharayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low Latency, Geo-distributed Data Analytics. In *ACM SIGCOMM*, London, UK, 2015.

[37] Arnon Lazerson, Izchak Sharfman, Daniel Keren, Assaf Schuster, Minos Garofalakis, and Vasilis Samoladas. Monitoring Distributed Streams Using Convex Decompositions. *Proc. VLDB Endow.*, 8(5):545–556, January 2015.

[38] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vassilis Plachouras, and Luca Telloli. On the feasibility of multi-site web search engines. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 425–434. ACM, 2009.

[39] Guillem Francès, Xiao Bai, B. Barla Cambazoglu, and Ricardo Baeza-Yates. Improving the Efficiency of Multi-site Web Search Engines. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, pages 3–12. ACM, 2014.

[40] Berkant Barla Cambazoglu and Ricardo Baeza-Yates. Scalability Challenges in Web Search Engines. pages 27–50, 2011.

[41] E. Kayaaslan, B. B. Cambazoglu, and C. Aykanat. Document replication strategies for geographically distributed web search engines. 2013.

[42] Mu Li, David G. Andersen, Alexander Smola, and Kai Yu. Communication Efficient Distributed Machine Learning with the Parameter Server. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, pages 19–27, Cambridge, MA, USA, 2014. MIT Press.

[43] Ignacio Cano, Markus Weimer, Dhruv Mahajan, Carlo Curino, and Giovanni Matteo Fumarola. Towards Geo-Distributed Machine Learning. In *Learning Systems Workshop at NIPS 2015*, 2015.

[44] Ignacio Cano, Markus Weimer, Dhruv Mahajan, Carlo Curino, and Giovanni Matteo Fumarola. Towards geo-distributed machine learning. *CoRR*, abs/1603.09035, 2016.

[45] Hsieh Kevin, Harlap Aaron, Vijaykumar Nandita, Konomis Dimitris, Ganger Gregory R., Gibbons Phillip B., and Mutlu Onur. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds. In *NSDI*, 2017.

[46] Jakub Konecný, Brendan McMahan, and Daniel Ramage. Federated Optimization: Distributed Optimization Beyond the Datacenter. *CoRR*, abs/1511.03575, 2015.

[47] Jakub Konecný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated Learning: Strategies for Improving Communication Efficiency. *CoRR*, abs/1610.05492, 2016.

[48] Jakub Konecný, H. Brendan McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *CoRR*, abs/1610.02527, 2016.

[49] Yuchen Zhang, John C. Duchi, and Martin J. Wainwright. Communication-efficient algorithms for statistical optimization. *J. Mach. Learn. Res.*, 14(1):3321–3363, January 2013.

[50] McMahan H. Brendan, Moore Eider, Ramage Daniel, Hampson Seth, and Agüera y Arcas Blaise. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2017.

[51] Maria-Florina Balcan, Avrim Blum, Shai Fine, and Yishay Mansour. Distributed Learning, Communication Complexity and Privacy. *Journal of Machine Learning Research*, 2012.

[52] A. Bar-Or, D. Keren, A. Schuster, and R. Wolff. Hierarchical Decision Tree Induction in Distributed Genomic Databases. *IEEE Transactions on Knowledge and Data Engineering – Special Issue on Mining Biological Data*, 17(8), August 2005.

[53] Ananda Theertha Suresh, Felix X. Yu, H. Brendan McMahan, and Sanjiv Kumar. Distributed Mean Estimation with Limited Communication. *CoRR*, abs/1611.00429, 2016.

[54] Shravan Narayanamurthy, Markus Weimer, Dhruv Mahajan, Sundararajan Sellamanickam, Tyson Condie, and Keerthi Selvaraj. Towards Resource-Elastic Machine Learning. In *NIPS 2013, Workshop on Big Learning*, NIPS 2013, pages 1–6, 2013.

34th IEEE International Conference on Data Engineering 2018

April 16-20, 2018, Paris, France

http://icde2018.org/

http://twitter.com/icdeconf #icde18

Call for Participation

The annual IEEE International Conference on Data Engineering (ICDE) addresses research issues in designing, building, managing, and evaluating advanced data-intensive systems and applications. It is a leading forum for researchers, practitioners, developers, and users to explore cutting-edge ideas and to exchange techniques, tools, and experiences.

Conference Sessions:

- Keynotes
- Research Papers
- Industrial Papers
- Demos
- Panels
- Tutorials
- Lightning Talks (new track)
- Posters (ICDE & TKDE)

General Chairs:

Malu Castellanos (Teradata, USA)
Ioana Manolescu (Inria, France)

Affiliated Workshops:

- **Context in Analytics: Challenges, Opportunities and Trends**
- **Data Engineering meets the Semantic Web (DESWeb2018)**
- **The Joint Workshop of HardBD (International Workshop on Big Data Management on Emerging Hardware) and Active (Workshop on Data Management on Virtualized Active Systems)**
- **Data Engineering meets Intelligent Food and COoking Recipe 2018 (DECOR 2018)**
- **Emerging Data Engineering Methods and Approaches for Precision Medicine (DEPM 2018)**
- **Accelerating In-Memory Databases (AIMD 2018)**

Data Engineering

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name _____

IEEE Member # _____

Mailing Address _____

_____

Country _____

Email _____

Phone _____

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaoyong Du
Key Laboratory of Data Engineering and Knowledge Engineering
Renmin University of China
Beijing 100872, China
duyong@ruc.edu.cn

TCDE Chair

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398