Towards a Distributed OS for Data-Intensive Cloud Applications

Stephanie Wang

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-3
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-3.html

January 11, 2024

Towards a Distributed OS for Data-Intensive Cloud Applications

By

Stephanie Wang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Joseph Hellerstein
Professor Matei Zaharia
Professor Joseph Gonzalez

Fall 2023

Towards a Distributed OS for Data-Intensive Cloud Applications

Abstract

Towards a Distributed OS for Data-Intensive Cloud Applications

by

Stephanie Wang
Doctor of Philosophy in Computer Science

University of California, Berkeley
Professor Ion Stoica, Chair

Commodity hardware is reaching fundamental limits, while the demands of data-intensive applications continue to grow. Thus, we now rely on horizontal scale-out and hardware accelerators to improve application performance and scale, while developing a myriad of distributed execution frameworks that are specialized to specific application domains, from data analytics to machine learning. While this reduces burden for certain applications, it also creates three problems: (1) duplicated system implementation effort, (2) reduced framework evolvability, and (3) difficulty interoperating efficiently between applications, especially when large data is involved.

This thesis describes the first steps towards a distributed "operating system" that can provide essential services to such data-intensive applications. This would allow currently monolithic frameworks to be built as libraries instead, making them easier to build, evolve, and compose. Towards this vision, we propose an intermediate and interoperable execution layer that handles common problems in distributed execution and memory management.

We first propose distributed futures, a general-purpose programming interface that extends the RPC abstraction with pass-by-reference semantics and a shared address space. Distributed futures act as a virtual memory-like abstraction but for the distributed setting, enabling distributed memory management to be factored out into a common system. Next, we present a design for this system that provides flexible fault tolerance with low overheads. We first present a fault-tolerant architecture for distributed futures that provides automatic memory management. We show how this system factors out system complexity from data-intensive applications without sacrificing performance, using MapReduce workloads as an example. Finally, we show how stronger recovery guarantees can be layered on top of this core architecture to provide greater recovery flexibility to end applications. Thus, we show how an end-to-end approach to fault tolerance can expand system generality.

To my family.

Contents

Contents **ii**

List of Figures **vi**

List of Tables **xii**

1 Introduction **1**
1.1 The Landscape of Data-Intensive Applications 5
1.2 Alternative Solutions 10
1.3 Overview and Contributions 12

2 Distributed futures and RPC **15**
2.1 Introduction 15
2.2 API 17
2.3 Automatic memory management 19
2.4 Is the API enough for applications? 21
2.5 Related abstractions for distributed memory 24
2.6 System Implementation and Thesis Work 25
2.6.1 Memory management 25
2.6.2 Fault tolerance 27
2.7 Conclusion and Lessons Learned 28

3 Lineage stash **29**
3.1 Introduction 30
3.2 Background 33
3.2.1 Case Study: Stream Processing 33
3.2.2 System Model and Challenges 34
3.3 Lineage Stash Overview 36
3.4 Lineage Stash Implementation 41
3.4.1 Definitions 41
3.4.2 Protocol 43
3.4.3 Failure Model 48

3.5 Evaluation . . . 49
3.5.1 Microbenchmarks . . . 50
3.5.2 End-to-end Applications . . . 53
3.6 Related work . . . 56
3.7 Conclusion and Lessons Learned . . . 58

4 Ownership **59**
4.1 Introduction . . . 59
4.2 Distributed Futures . . . 64
4.2.1 API . . . 64
4.2.2 Applications . . . 64
4.3 Overview . . . 67
4.3.1 Requirements . . . 67
4.3.2 Existing solutions . . . 70
4.3.3 Our solution: Ownership . . . 71
4.4 Ownership Design . . . 72
4.4.1 Task scheduling . . . 74
4.4.2 Memory management . . . 76
4.4.3 Failure recovery . . . 79
4.5 Evaluation . . . 83
4.5.1 Microbenchmarks . . . 83
4.5.2 End-to-end applications . . . 87
4.6 Related Work . . . 88
4.7 Discussion . . . 89
4.7.1 Programming languages . . . 89
4.7.2 Impact on Ray . . . 90
4.8 Conclusion and Lessons learned . . . 91

5 Exoshuffle **93**
5.1 Introduction . . . 94
5.2 Motivation . . . 96
5.2.1 Shuffle Systems . . . 96
5.2.2 Random Shuffle in ML Training Pipelines . . . 98
5.3 Shuffle with Distributed Futures . . . 100
5.3.1 The Distributed Futures API . . . 100
5.3.2 Expressing Shuffle with Distributed Futures . . . 101
5.4 System Architecture . . . 102
5.4.1 Example: Push-based Shuffle . . . 102
5.4.2 Scheduling Primitives . . . 102
5.4.3 Transparent System Facilities . . . 105
5.5 Evaluation . . . 108
5.5.1 Shuffle Performance . . . 108

5.5.2 Implementation Complexity . . . 111
5.5.3 System Microbenchmarks . . . 111
5.6 Related Work . . . 114
5.7 Discussion . . . 115
5.8 Conclusion and Lessons Learned . . . 116

6 Exoflow **118**
6.1 Introduction . . . 119
6.2 Motivation . . . 122
6.2.1 Overview of recovery strategies . . . 122
6.2.2 Applications . . . 125
6.3 API . . . 127
6.3.1 Overview and requirements . . . 127
6.3.2 Model . . . 130
6.3.3 Guaranteeing exactly-once execution . . . 132
6.3.4 References . . . 133
6.4 Architecture . . . 136
6.5 Evaluation . . . 136
6.5.1 ML training pipelines . . . 137
6.5.2 Stateful serverless workflows . . . 139
6.6 Related Work . . . 140
6.7 Discussion . . . 142
6.8 Conclusion and Lessons Learned . . . 143

7 Conclusion **144**
7.1 Related Work . . . 144
7.1.1 Distributed dataflow . . . 144
7.1.2 RPC and actors . . . 146
7.1.3 High-performance computing (HPC) . . . 148
7.1.4 Distributed shared memory (DSM) . . . 149
7.1.5 Disaggregated memory . . . 150
7.1.6 Serverless . . . 151
7.1.7 Cluster managers . . . 152
7.2 Discussion . . . 153
7.2.1 Broader Impact: History of Ray as an open-source project . . . 153
7.2.2 Lessons Learned . . . 154
7.2.3 Limitations and future work . . . 158
7.3 Conclusion . . . 164

Appendix **186**

A Ownership system protocols **186**
A.1 Distributed Reference Counting 186
A.2 Formal Specification 187

B Exoshuffle libraries, applications, and evaluation **189**
B.1 Expressing Shuffle Strategies with Distributed Futures 189
B.1.1 Pre-Shuffle Merge 189
B.1.2 Push-based Shuffle 190
B.1.3 Straggler Mitigation 190
B.1.4 Data Skew 191
B.2 Expressing Shuffle Applications 191
B.2.1 Online Aggregation with Streaming Shuffle 192
B.2.2 Distributed ML Training with Pipelined Shuffle 192
B.3 Evaluation 193
B.3.1 Performance Comparison of Shuffle Libraries 193
B.3.2 Shuffle Scalability 194
B.3.3 Shuffle Applications 194
B.3.4 CloudSort 194
B.3.5 Online Aggregation with Streaming Shuffle 196
B.3.6 Distributed ML Training 196

C Exoflow system design and evaluation **198**
C.1 Architecture 198
C.1.1 Workflow execution 198
C.1.2 Workflow recovery 199
C.1.3 Execution backends 201
C.2 Implementation 201
C.3 Evaluation 202
C.3.1 Online-offline graph processing 202
C.3.2 Microbenchmarks 204

D Some random walks **208**

List of Figures

1.1 Application and execution models for data analytics workloads. . . . 7
1.2 Application and an example execution for distributed ML training with online data preprocessing. 9
1.3 Thesis overview. Light blue indicates a system discussed in this thesis, while dark blue indicates the abstraction exposed by that system. Light gray boxes indicate application libraries that may be built on top of the system abstractions. 13

2.1 A single "application" actually consists of many components and distinct frameworks. With no shared address space, data (squares) must be copied between different components. 16
2.2 Logical RPC architecture: **(a)** today, and **(b)** with a shared address space and automatic memory management. 16
2.3 Applications for a pass-by-reference API. Legend: gray circle is the client, other circles are RPCs, dashed arrows are RPC invocation, solid squares are data, solid arrows are dataflow. 23

3.1 A streaming mapreduce. **(a)** Logical representation. Mappers compute a stateless function over each record (rounded box) in the input and output the results to a Reducer. **(b)** Physical representation, as a dynamic dataflow. Solid arrows show data dependencies (record batches). White arrows show stateful dependencies [145], determined by the execution order on a given process. Mappers do not have application state, but they are stateful because they can buffer records and dynamically push them to Reducer by submitting tasks, which get executed in a nondeterministic order. Reducer fails during task 3 (red), and outlined tasks must be re-executed to preserve exactly-once semantics. Lineage reconstruction (green) exactly reconstructs Reducer by replaying its inputs since the last checkpoint in the same order. Global checkpointing (blue) re-executes *all* processes' tasks since the last checkpoint, possibly in a different order (e.g., Reducer may execute task 3 before 2). 30

3.2 **(a)** Lineage stash architecture, on top of a decentralized dataflow scheduler. A and B are processes that can submit tasks to each other (e.g., $A1$ submits $B1$). Dotted arrows show the protocols used to communicate between nodes. **(b)** Stream processing. D is a *nondeterministic* operator that reads dynamically sized batches (buffers) from multiple input sources (A, B, C) in any order and outputs results to downstream operator E. G is a *deterministic* operator that reads statically sized batches from a single source, F. 35
3.3 **(a)** A nondeterministic application and **(b)** a failure scenario showing why lineage must be forwarded. Because C executes tasks from A, B in a nondeterministic order, it must retrieve its lineage from D after a failure, shown by the red dashed arrows. 39
3.4 **(a)** A deterministic application and **(b)** a failure scenario showing what lineage must be remembered. To recover B after a failure, A simply resubmits (red dashed arrows) its previous tasks. 39
3.5 Lineage stash methods for getting and receiving a task's uncommitted lineage (Definition 3.5). A practical implementation can easily avoid forwarding duplicate lineage by recording which tasks have been sent to which nodes. 43
3.6 Node methods for task execution. `AssignTask` also records nondeterministic execution order by updating the task's `predecessorId`. Nondeterministic events during task execution are recorded by appending to the task's `applicationLog` (not shown). 44
3.7 Forwarding and flushing lineage. **(a)** Task $A2$ submits task $B2$, forwards the uncommitted lineage ($A2$) to B, and asynchronously flushes $B2$. **(b)** A and B receive commit acknowledgements for $A1$ and $B2$. $A1$ can be evicted because it has no dependencies, but $B2$ cannot. **(c)** A and B receive commit acknowledgements for the remaining tasks and it is safe to evict all tasks. 45
3.8 Lineage stash methods for flushing to the global store. `FlushTask` writes a task asynchronously to the global store with the callback `TryEvict`. Once a task (or a newer version) is committed and its dependencies have been evicted, it is evicted in `TryEvict` (`TryEvict` also tries to evict any dependent tasks, not shown). 46
3.9 Recovery procedure for the nondeterministic process from Fig 3.3 in detail. **(a)** (1) C contacts downstream process D, (2) D flushes its lineage, (3) D receives all acknowledgements, (4) D replies to C. **(b)** Processes A and B (not shown) resubmit their last submitted tasks ($A2$, $B1$) to C. This may happen concurrently with steps 1-4. **(c)** After steps 1–5, C recovers the lineage of $A2$ and $B1$, which includes the initial execution order, from the global store. 47

3.10 Task latency for deterministic and nondeterministic applications, with lineage stash vs `WriteFirst`. A ring of 64 processes is instantiated, one on each node. Each process submits no-op tasks with a unique token to its successor. Task latency is the time before the process receives its token again divided by the number of processes. For Fig 3.10c, we forward an uncommitted task up to f=8 times. 49
3.11 Median (and first and third quartiles) size of the forwarded uncommitted lineage, varying task duration for different values of f, the maximum number of concurrent failures tolerated. Above 10ms tasks, the uncommitted lineage size is stable. 50
3.12 **(a)** Allreduce duration on 64 workers (m5.2xlarge), averaged over 20 trials (with std. deviation). `WriteFirst` and the lineage stash use ring allreduce, with simulated global store latency as labeled. **(b)** Allreduce recovery time for lineage stash vs `WriteFirst` vs OpenMPI, with checkpoints to disk every 150 iterations. We kill and restart a worker at iteration 284. **(c)** Distributed SGD on the lineage stash vs Horovod v0.16.1, on 16 p3.8xlarge. Both use TensorFlow v1.12 on Resnet-101 with synthetic data and batch size 64. The lineage stash uses the same ring allreduce as in §3.5.2.1. Each worker checkpoints the model to disk every 640 iterations (˜7min). We kill and restart a worker at iteration 1200. 52
3.13 **(a)** Latency CDF for a streaming wordcount on 32 m5.xlarge workers at 400k records/s (4M words/s). Latency is sampled once every 1000 records. Both systems used a parallelism of 32 (per source, map, reduce, sink) and checkpoints to disk every 30s. **(b, c)** Failure and recovery for streaming wordcount on 32 m5.xlarge nodes at 300k records/s, checkpoints to disk every 30s. A worker is killed and restarted at t=˜45s (vertical red line), ˜15s after the first checkpoint. We report **(b)** median latencies seen by a single sink (with 1st and 3rd quartiles), x-axis is the record timestamp, and **(c)** total throughput, x-axis is physical time. The throughput drop at t=˜80s is due to checkpointing. 54

4.1 A distributed futures program. `compute` and `add` are stateless. `a_future`, `b_future`, and `c_future` are distributed futures. 60
4.2 Example executions of the program from Figure 4.1. **(a)** With RPC. **(b)** With RPC and distributed memory, allowing the system to reduce data copies. **(c)** With RPC and futures, allowing the system to manage parallel execution. **(d)** With distributed futures. 61
4.3 Distributed futures applications. 66
4.4 Failure detection. **(a)** `a`'s location is known by the time worker 2 receives the reference. **(b)** `a`'s location may not be known when worker 2 receives `add`, so worker 2 cannot detect the failure. 68

4.5 Failure recovery. **(a)** Data is passed by value, so the driver recovers by resubmitting `add`. **(b)** `b` is also lost. `f`'s description must be recorded during runtime so that `b` can be recomputed. 69
4.6 Distributed futures systems. **(a)** An application. **(b)** Master manages metadata and object failures. **(c)** Workers write metadata asynchronously, coordinate failure handling with leases. **(d)** Workers manage metadata. Worker 1 handles failures for workers 2 and 3. Worker 1 failure is handled by `A`'s owner elsewhere in the cluster. 70
4.7 Architecture and protocol overview. **(a)** Task execution. **(b)** Local task scheduling. **(c)** Remote task scheduling. **(d)** Object transfer. **(e)** Task output storage and input retrieval. Ownership layer manages distributed memory garbage collection and recovery. **(f)** Scheduler fetches objects in distributed memory to fulfill task dependencies. . . 74
4.8 Task scheduling and the method of recording a task's location for the program in Figure 4.6a. **(a)** Centralized master. **(b)** Distributed leases. **(c)** Scheduling with ownership. (1-2) Local scheduler redirects owner to node 2. (3) Update task location. (4-5) Remote scheduler grants worker lease. (6) Task dispatch. **(d)** Direct scheduling by the owner, using the worker and resources leased from node 2 in (c). **(e)** Length of critical path of local and remote task execution, in terms of local and remote RTTs. 75
4.9 **(a)** Distributed memory store API, and **(b-d)** Memory management for the program in Figure 4.6a. (1-2) `B` returns a large object `X` in distributed memory. The primary copy is pinned until all references have been deleted. (3) Worker 1 dispatches `C` once `X` is available. (4-5) `Get` the value from distributed memory (location lookup not shown). (6) `C` returns a small object `Y` directly to the owner. (7-8) Object reclamation. 77
4.10 Object recovery. 79
4.11 Owner recovery. 79
4.12 Throughput and scalability. **(a-d)** Task submission is divided across multiple intermediate drivers, either colocated on the m5.8xlarge head node or spread with one m5.8xlarge node per driver. 1 intermediate driver is added per 5 worker nodes. Each task returns either a small (short binary string) or large (1MB blob) object. **(e)** Scaling task submission using nested tasks and first-class distributed futures. . . 82
4.13 Task latency. Local means that the worker and driver are on the same node. Error bars for standard deviation (across 3k tasks). 85
4.14 Total run time (log-scale), relative to ownership without failures. The application is a chain of dependent tasks that execute on one node. Each task sleeps for the duration on the x-axis (total 10s) and returns either **(a)** a short binary string, or **(b)** a 10MB blob. 85

4.15 End-to-end benchmarks. **(a)** Image classification latency (right is p95-p100). **(b)** Online video stabilization latency. **(c)** Online video stabilization latency with failures (starting at p90). L=leases; O=ownership; CP=checkpointing; WF=worker failure; OF=owner failure. 86

5.1 Exoshuffle builds on an extensible architecture. Shuffle as a library is easier to develop and more flexible to integrate with applications. The data plane ensures performance and reliability. 95
5.2 Shuffle algorithms for various applications. Exoshuffle uses distributed futures to execute these DAGs. 97
5.3 Pipelining data preprocessing and shuffle with GPU tasks in an ML training application. 98
5.4 Comparing a monolithic vs. application-level shuffle architecture. (a) implements all coordination and block management through an external shuffle service on each node, in this case implementing the Magnet shuffle strategy (Appendix B.1.2). (b) shows the same shuffle strategy but implemented as an application on a generic distributed futures system. 104
5.5 Comparing job completion times on the Sort Benchmark. The dashed lines indicate the theoretical baseline (§5.5.1.1). Exoshuffle is abbreviated as ES. 109
5.6 Comparing shuffle time in Dask and Ray. Legends show number of processes × threads. 112
5.7 Effect of I/O optimizations in Ray. 112

6.1 **(a)** An example workflow with internal outputs (e.g., `a_out`) and external outputs (e.g., `put(key,val)`). **(b)** The most efficient recovery strategy depends on output visibility and nondeterminism. 123
6.2 **(a)** ETL workflow today, using external outputs for communication. **(b)** The same ETL workflow with internal outputs only. **(c)** ML training workflow today, with external outputs and manual orchestration within a task. **(d)** The same ML workflow with internal outputs only, and orchestration is handled by the workflow system. Third-party framework state (`TF workers`) can be passed between workflow tasks. 124
6.3 Serverless workflow systems [188, 216, 110] guarantee exactly-once semantics by interposing on all communication to external storage, e.g., through a transaction buffer, and explicitly managing visibility of these external effects. 126

6.4 **(a)** Task annotations. Edge cuts represent `checkpoint=True`. **(b)** Passing references (small boxes) in an ML workflow. Blue `Refs` are actors that wrap TensorFlow worker state. **(c)** Passing an `ActorRef` in an ETL workflow. `B` and `C` call read-only methods on the Spark context actor. 129
6.5 Workflow architecture. The controller and executors are RPC-like services built using Ray actors. Each invocation on these services returns a distributed future (system-internal `Refs`). 135
6.6 End-to-end duration for the ML workflow application shown in Figures 6.2d and 6.4b. **Left:** End-to-end duration without failure. **Right:** End-to-end duration with different failure types. The shadow represents the execution time without failure. 137
6.7 **(a)** Response latency percentile for a serverless travel reservation benchmark [87]. **(b)** Median latency of the trip reservation request from the travel reservation benchmark. Error bar represents 99-percentile latency. 139

B.1 Comparing job completion times on the Sort Benchmark. The dashed lines indicate the theoretical baseline (§5.5.1.1). Exoshuffle is abbreviated as ES. 193
B.2 Online aggregation. Dotted lines show map progress; solid, reduce progress. 195
B.3 Single-node ML training for 20 epochs. 195
B.4 4-node, distributed ML training for 20 epochs. 195

C.1 **(c)** Latency CDF of online-offline graph processing. 203
C.2 Microbenchmarks. **(a)** Triggering and data passing latency of Exoflow and other workflow systems, using AWS Lambda (λ) and Ray as execution backends. Missing bars indicate limitations in inter-task communication. **(b)** End-to-end run time for the ETL workflow shown in Figures 6.2b and 6.4c, compared with Airflow and native Spark. . 205
C.3 Microbenchmarks, cont. Maximum task throughput (a: 1 task/DAG; b: 100 tasks/DAG) of 10k tasks, compared against Ray as an optimal baseline, on 1 node and 4 nodes. 206

List of Tables

1.1 A comparison of abstractions used, from a traditional single-machine OS (Unix/POSIX) to a representative distributed dataflow system (Naiad) to general-purpose distributed execution platforms (RPC + container orchestrator vs. the work presented in this thesis). 3
1.2 DA=Data Analytics; ML=Machine Learning; G=General-purpose. A comparison of features across distributed execution systems, focusing on those relevant to data-intensive computing. The upper section includes examples specialized to some application domain, while the middle section includes general-purpose frameworks that may serve as lower-level building blocks and "glue" systems for those in the upper section. The last row describes the abstraction and system proposed by this thesis. 6

2.1 A language-agnostic pass-by-reference API. 18
2.2 RPC-like systems that expose a shared address space. Each system was designed for the listed application domain. Systems that implement both first-class references and futures are *distributed futures* systems. 22

3.1 Task specification (*version*, *predecessorId* and *applicationLog* may be updated after task creation to record nondeterminism) 42
3.2 Summary of mean latencies in milliseconds during normal operation and during recovery for Ray with the lineage stash (LS) compared to baseline systems on a variety of applications. For latency during a failure in streaming (§3.5.2.3), we take the mean of all reported latencies between the failure time to when the latency for new inputs converges to normal operation. For the other applications, we report the maximum latency. 53

4.1 Distributed futures API. The full API also includes an actor creation call. A task may also return a `DFut` to its caller (nested `DFuts` are automatically flattened). 65
4.2 Ownership table. The owner stores all fields. A borrower (Section 4.3.2) only stores fields indicated by the *. 73

5.1 Different shuffle systems are built to optimize shuffle for deployment in different storage environments. 97
5.2 Approximate lines of code for implementing shuffle algorithms in Exoshuffle versus in specialized shuffle systems. 111

6.1 Workflow API. Top: API calls exposed to the application. Middle: Task annotations specified by application or third-party library. Bottom: Exoflow-internal `Ref` API, pluggable by execution backend. . . 127

A.1 Full description of the `References` field in Table 4.2. Every process with an instance of the `DFut` (either the owner or a borrower) maintains these fields. 186

B.1 CloudSort costs over years. 196

Acknowledgments

One of my greatest qualms about starting a PhD was that I would end up working on problems in isolation. It's safe to say that this did not happen, and I have many people to thank for that.

First and foremost, I would like to thank my advisor Ion Stoica. Ion taught me nearly everything I know about how to do good research. At the risk of going against his primary advice to *focus*, I'd like to thank Ion for several of the qualities that he embodies and that I admire the most: his dedication to working on problems that matter, his ability to simplify down to the fundamentals, and his boundless energy in continuing to push the boundaries of what we believe is achievable.

I would like to thank the other members of my thesis and quals committees: Joe Hellerstein, Matei Zaharia, Alvin Cheung, and Joey Gonzalez. Their feedback and perspective throughout has been valuable and insightful. Be thankful that you are reading this version of the thesis and not the one I gave to my thesis committee!

I would like to thank my undergraduate and MEng advisors, Frans Kaashoek and Nickolai Zeldovich, who gave me my start in systems research an unknown number of years ago now. Now that I am more aware of just how many things professors are juggling, I feel even more grateful for the time and attention you gave to all of your students, both in the classroom and in research. Without your encouragement, mentorship, and support, I probably would have just done another internship.

I would like to thank my other wonderful coauthors and collaborators from graduate school. In roughly chronological order: Philipp Moritz and Robert Nishihara, for making my first few years of graduate school a delightful team experience; hope we can share some ginger tea again soon. Eric Liang, for contributions to nearly every chapter in this thesis, especially the ownership work, and for being an excellent judge of system design; I couldn't have asked for a better complement in building practical large-scale systems. Edward Oakes, for contributions to the ownership work and for being an all-around fun coding partner.

Although this thesis officially has only one author, two of the chapters in this thesis in particular should truly have multiple authors. I would like to highlight these two collaborators in particular: Frank Sifei Luan, for contributions to the Exoshuffle work, and Siyuan Zhuang, for contributions to the Exoflow work. I feel truly privileged to have gotten the chance to work with you both and look forward to seeing what you do next.

For the sake of keeping this short, I'd also like to acknowledge my other colleagues at Anyscale, many of whom I've worked closely with in developing Ray, and some of the many coauthors that I've worked with during this thesis: Romil Bhardwaj, SangBin Cho, Melih Elibol, Benjamin Hindman, Jaewan Hong, Zhuohan Li, John Liagouris, Richard Liaw, Jianan Lu, Ujval Misra, Rishabh Poddar, Johann Schleier-Smith, Alexey Tumanov, Zongheng Yang, Samyu Yagati, Siyuan Zhuang, and Danyang Zhuo. I sincerely hope we all get more opportunities to collaborate.

Those are some of the names on the official record, but of course graduate school would not have been the same without the wonderful community of students, postdocs, faculty, and staff at Berkeley, the RISELab, and NetSys. I learned so much from you all, and much more importantly, you made graduate school life fun!

Next a special shout-out to friends who kept me happy, healthy, and sane all during grad school. To my lifelong friends from Burton Third, here's to 333rd more years! And to the many wonderful friends I made during my years at Berkeley: my labmates, my neighbors in NetSys and on the 5th floor of Soda, my roommates, and my cohort from PhD visit days.

Finally, I want to thank my family. First to Tyler, my partner in life who has been with me through the good and the bad, the adventures and the everyday. I truly cannot imagine how grad school would have been without you, and I can't wait for our next chapter together. To Tyler's mother Ruth, who kept us well-supplied in cookies and love all through grad school. To my sister Jie, my brother Peter, and the beautiful family that you've brought to us; I'm so excited to see Forest and Orion grow up with us. And last but not least, to my parents, whose unconditional love, support, and sacrifice are the reasons I'm here.

Chapter 1

Introduction

Scaling applications with distributed execution has now become the norm. On one hand, commodity hardware is colliding with fundamental limits in compute and memory scalability. On the other hand, data-intensive applications are continuing to grow in scale, ubiquity, and diversity.

On the hardware side, the end of Moore's Law and Dennard scaling has meant that application performance must now be scaled horizontally, by leveraging multiple machines, and through specialized hardware, such as with GPUs for machine learning (ML). The difficulty of scaling memory capacity has meant that memory continues to be the limited resource in data-intensive and ML applications. Meanwhile, significant performance and/or durability differences persist between traditional DRAM versus emerging technologies in nonvolatile storage and remote memory access. Thus, reducing data copies and data movement continue to be important factors in ensuring performance for data-intensive applications.

On the applications side, an important and inevitable trend is *heterogeneity*. Application heterogeneity creates diversity and complexity in the underlying software systems. For example, diversity in end use cases for data analytics has led to a dichotomy between distributed systems that use batch vs. stream processing. The rise of ML has complicated this picture further, as these workloads often require both analytics-like data processing on CPUs and scientific computing-like processing on GPUs. Differences between GPUs and CPUs complicate the usual problems in distributed systems of resource management, communication, and fault tolerance.

Application heterogeneity has led to a myriad of specialized distributed execution frameworks designed to support applications within specific domains. While this reduces burden for standalone applications that fit into one domain, it also creates three problems.

1. **System implementation effort:** Frameworks are developed in isolation, leading to duplicated development effort across different frameworks.

2. **Evolvability:** Frameworks are often built as monolithic distributed systems

on top of low-level APIs such as remote procedure calls (RPCs) and OS system calls. This makes the frameworks difficult to evolve and extend as they mature. In combination with the above, the result is that features and optimizations introduced in one framework are not easily portable to another.

3. **Inter-framework applications:** Due to workload heterogeneity, an important end use case is applications that might span multiple domains. In these cases, application development requires stitching together different specialized frameworks. Doing so efficiently requires coordinated data movement and sharing of cluster resources across frameworks. These problems are left to the end developer.

The role of an operating system (OS) is to provide essential services among different applications that share the same physical resources. In addition, POSIX provides a common standard for operations such as filesystem access and inter-process communication. In the single-machine setting, this has of course been wildly successful. However, we lack such a system for the distributed setting. Application-specific distributed frameworks can play such a role, but only for a limited application domain. Lower-level interfaces such as remote procedure calls (RPC) are a general-purpose standard but limited in their ability to provide essential OS services such as memory management (Table 1.1).

An OS for distributed applications would allow specialized distributed frameworks to be developed as *libraries* that share a common underlying execution layer. Towards this vision, the focus of this thesis is on the design of an intermediate and interoperable execution layer that handles common problems in distributed execution and memory management. Such a layer would make domain-specific frameworks easier to build, evolve, and compose:

1. **System implementation effort:** Libraries can share key functionality such as garbage collection and fault tolerance for distributed memory. Development of future domain-specific libraries is also accelerated.

2. **Evolvability:** By factoring out a common intermediate layer, we can decouple system concerns that are currently entangled, internal to monolithic distributed frameworks. This allows greater evolvability at the library layer, which can now deal with higher-level and more domain-specific concerns.

3. **Inter-framework applications:** By sharing a common distributed scheduling and memory layer, composition of libraries can be accomplished with lower developer and performance overhead. For example, physical data copies between libraries can be shared.

Some essential distributed services have already become standard. For example, the modern analogy to a single-machine file system or database is a cloud blob store

	Unix/POSIX	Naiad	RPC + container orchestrator	Distributed OS (this thesis)
Computation	Processes, threads	Dataflow	Containers	Actors, tasks
Memory	Virtual memory	Event stream	-	Distributed futures
Communication	Sockets	Event stream	RPC	RPC
Storage	File system	Cloud storage	Cloud storage	Cloud storage

Table 1.1: A comparison of abstractions used, from a traditional single-machine OS (Unix/POSIX) to a representative distributed dataflow system (Naiad) to general-purpose distributed execution platforms (RPC + container orchestrator vs. the work presented in this thesis).

or database. However, we argue in this thesis that distributed *execution* does not yet have a comparable standard. We draw the requirements for a common execution layer from existing data-intensive frameworks, whose functionality we would like to factor out, and by analogy to Unix, whose functionality we would like to emulate but in the distributed setting.

1. **Computation:** Similar to POSIX processes and threads, the unit of computation must be *flexible*. In the distributed setting, this means that it must be possible to express different models of parallelism, from data parallelism in big data systems to model parallelism in distributed ML systems. Concretely, the abstraction should support both stateless and stateful computation, and allow for system implementations that can dynamically spawn computation in milliseconds or less, in order to support a wide range of computation granularities.

2. **Memory:** Similar to virtual memory, there must exist an abstraction for distributed memory that makes the location of the data *transparent*. This is to enable factoring out problems common in data-intensive applications, such as sharing physical memory, reducing data movement and swapping to disk. It also implies that the system must provide key automatic distributed memory management features, e.g., memory allocation, deallocation, and movement.

3. **Communication:** Similar to POSIX sockets, there must be a *standard primitive for communication* with other processes. This is to support orchestration of distributed execution, both intra- and inter-framework.

4. **Fault tolerance:** Handling process and memory faults becomes a much more complex problem in the distributed setting. Ideally, the system would provide *failure transparency*. However, a general-purpose solution is likely to add high run-time performance overheads. Thus, instead the goal in this thesis is to provide a *flexible spectrum* between recovery guarantees vs. run-time overheads.

This thesis describes a general-purpose distributed system called Ray that meets these requirements. We take inspiration from what is arguably the closest popular example of a distributed OS that exists today: a container orchestrator such as Kubernetes [53] combined with RPC for communication (Table 1.1). This combination is popularly used to support microservices applications but lacks three of the key features needed in data-intensive systems: a fine-grained and dynamic unit of parallelism, an abstraction for distributed memory, and (a choice of) failure transparency.

Thus, we first propose a general-purpose programming interface that extends RPC with these features. In particular, we use the *actor model* for computation, and a single function call as the smallest unit of parallelism, i.e. a *task*. We also propose a novel primitive which we call *distributed futures*, as an abstraction for distributed memory (Chapter 2). In traditional RPC, all data sent from the caller to the callee must be *copied by value*, which becomes expensive for large data. This means that data-intensive frameworks built directly on top of RPC must often implement additional systems to manage distributed memory efficiently. Distributed futures enable *pass-by-reference* semantics for RPC, allowing callers to send data without necessarily having it local. In this way, distributed futures allow distributed memory management to be factored out into a common underlying system.

The remainder of this thesis focuses on the design of this system. The presented architecture provides automatic management of distributed memory and transparent recovery for applications with deterministic and idempotent tasks (Chapter 4). We show how this system factors out system complexity from data-intensive applications without sacrificing performance, using MapReduce workloads as an example (Chapter 5). Finally, we show how stronger recovery guarantees can be layered on top of this core architecture to provide greater recovery flexibility to end applications (Chapter 6). Thus, we show how an end-to-end approach [173] to fault tolerance can expand system generality.

Thesis: *Distributed futures and actors, combined with an end-to-end approach to fault tolerance, can serve as a general-purpose and interoperable execution subtrate for data-intensive applications.*

Today, we already have substantial evidence towards this thesis. For example, in 2022, the Ray system was used to train the large language model (LLM) that powers ChatGPT and is one of the largest ML models to date. The main reasons for this choice, compared to other popular general-purpose choices such as MPI, were the dynamicity and fault tolerance of the system, as described in Chapter 4. We have

also used the Ray system as an intermediate execution substrate for a MapReduce-like framework, replacing RPC to enable greater evolvability and interoperability with other data-intensive applications. In 2023, this system was used to break the world record in CloudSort, which measures the cost of sorting 100TB of data in the public cloud [131].

1.1 The Landscape of Data-Intensive Applications

Here, we motivate the three outlined requirements: (1) a flexible unit of parallel computation, (2) an abstraction for distributed memory, and (3) developer choice over recovery strategy. In particular, we will argue that despite a myriad of distributed execution frameworks that aim to be general-purpose, application needs regarding these three features continue to evolve, making it challenging to provide a one-size-fits-all system.

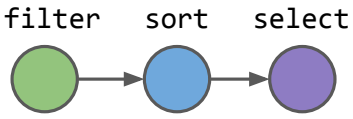
We overview these challenges by contrasting the requirements of distributed data analytics vs. more recent ML workloads in model training and inference, many of which are supported today by the first section of systems shown in Table 1.2. The key difference between these workloads is their *parallelism strategy*. While both employ data parallelism, the required systems are worlds apart. Data analytics transforms can often be applied independently to arbitrary partitions of data, making it more amenable to elastic, asynchronous, and heterogeneous compute. In contrast, ML computation requires data-parallel workers to communicate frequently and execute in lockstep; thus static, synchronous, and homogeneous clusters make efficiency much simpler to achieve. Furthermore, while data analytics can continue to use data parallelism to scale out and overcome the memory wall, larger ML models require additional model parallelism strategies. Together, these differences motivate the requirement for a *flexible unit of parallel computation*.

Because of these differences, data analytics and ML workloads have also inspired entirely separate implementations, even though both categories of systems often expose a dataflow-based API (Figures 1.1a and 1.2a). However, this state of the world is also not ideal, as many end-to-end workloads require both CPU and accelerator execution. For example, model training and inference on a GPU often require pipelining with CPU-based data loading and preprocessing (Figure 1.2b). Even predominantly GPU-based ML workloads offload compute and/or memory to the CPU [170]. These heterogeneous applications motivate the requirement for an *abstraction for distributed memory*, which would allow different frameworks or fine-grained units of computation to efficiently exchange data.

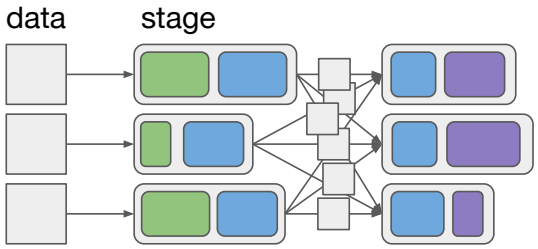
Finally, the differences in parallelism strategy also impact the appropriate choice of recovery strategy. In data analytics frameworks such as MapReduce, Spark, and Naiad, the challenges in fault tolerance often stem from the problem of producing consistent snapshots [60] with low overhead. In particular, each datum should appear

System	Applications	Message location transparency	Transparent fault tolerance	Automatic resource management	Stateful API	Abstraction
MapReduce [72, 205]	DA	✓	✓	✓	×	Map, reduce
Apache Spark [212]	DA	✓	✓	✓	✓/×	Dataflow (RDD)
Apache Flink [56]	DA	✓	✓	✓	✓	Dataflow (stream)
Naiad [146]	DA	✓	✓	✓	✓	Timely dataflow
Ciel [148]	DA	✓	✓	✓	×	Function call
Distributed TensorFlow [18]	ML	✓	✓	✓	✓	Dataflow (tensors)
PyTorch RPC [70]	ML	✓	×	×	✓	Function call (tensors)
RPC [44, 9]	G	×	×	×	✓	Function call
RPC + key-value store	G	✓	×	×	✓	Function call
Actors [195, 2, 42]	G	×	✓/×	✓	✓	Message/function call
Message-passing [190, 86, 78]	G	×	×	×	✓	Message
Datalog derivations [24, 65]	G	✓	✓	✓	✓	Declarative rules
Serverless functions [176, 52]	G	×	×	✓	×	Function call
Distributed shared memory [153]	G	✓	×	✓	✓	Threads+shared memory
Distributed futures + actors (Ray)	G	✓	✓/×	✓	✓	Function call

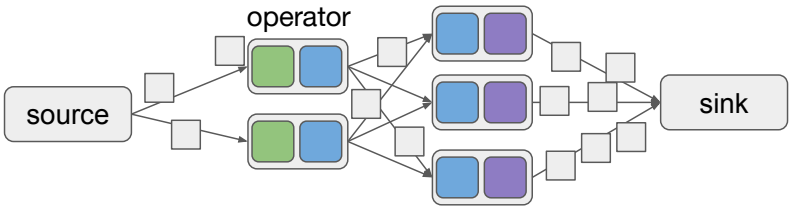
Table 1.2: DA=Data Analytics; ML=Machine Learning; G=General-purpose. A comparison of features across distributed execution systems, focusing on those relevant to data-intensive computing. The upper section includes examples specialized to some application domain, while the middle section includes general-purpose frameworks that may serve as lower-level building blocks and "glue" systems for those in the upper section. The last row describes the abstraction and system proposed by this thesis.



(a) A dataflow DAG of transforms specified by the application.



(b) Batch processing execution of the DAG.



(c) Stream processing execution of the DAG.

Figure 1.1: Application and execution models for data analytics workloads.

to processed exactly once in the snapshot. When the computation is inherently synchronous and nondeterministic, as in ML, producing a consistent snapshot is much simpler. Instead, the challenges in fault tolerance often stem from the tightly coupled and fate-sharing nature of ML clusters. These differences motivate the requirement for *developer choice over recovery strategy*.

We expand on these differences by first examining the generality and limitations of the primary execution models for data analytics: batch processing (Figure 1.1b) and stream processing (Figure 1.1c). Batch processing produces results over a fixed-size "batch" of data at a time, while stream processing may execute over an unbounded stream of data. Batch processing is typically used in offline scenarios, in which a large amount of input data must be processed at high throughput and there is no need for user interaction. Stream processing is more commonly used in online scenarios, as it can provide low-latency incremental results over a live stream of events.

Both batch and stream processing take advantage of data parallelism. Typically, the application "driver" first specifies a DAG of data transforms to the system (Fig-

ure 1.1a). Transforms are often implemented as map and/or reduce operations: a map operation can be applied to each record individually while reduce operations aggregate multiple records. For example, Figure 1.1a shows an application DAG where `filter` and `select` are examples of map while `sort` requires a map and reduce. Map and reduce operations can be transparently parallelized and scaled out by partitioning the input dataset.

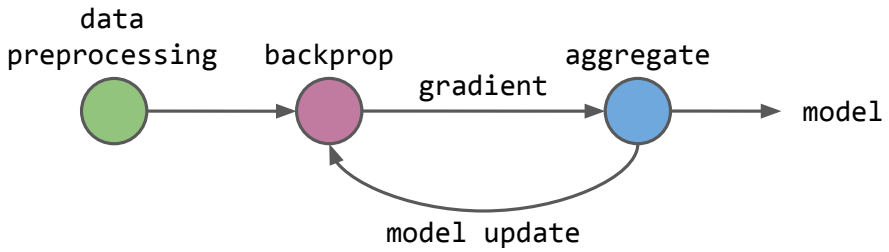
Batch processing systems such as MapReduce, Spark, and DryadLINQ require the developer to decompose their application into a series of deterministic transforms over an immutable dataset. Then, by partitioning the input dataset, each data transform can be executed as a "stage" of parallel tasks, one for each partition (Figure 1.1b). All-to-all transforms such as a sort require a (logically) synchronous barrier at the end of a stage to materialize results. Most batch dataflow systems use a centralized scheduler, and fault tolerance can be provided through transparent re-execution of lost tasks.

Stream processing uses an asynchronous and stateful model, in which each data transform is physically instantiated as one or more "operators" that share an input stream (Figure 1.1c). Operators are placed before execution, but data partitioning and task scheduling are done on the fly, as each operator independently reads from its input stream once it is ready to process another record. Operators may have local state, including an output stream buffer and user-defined state. Typically, fault tolerance is provided through global checkpointing and rollback.

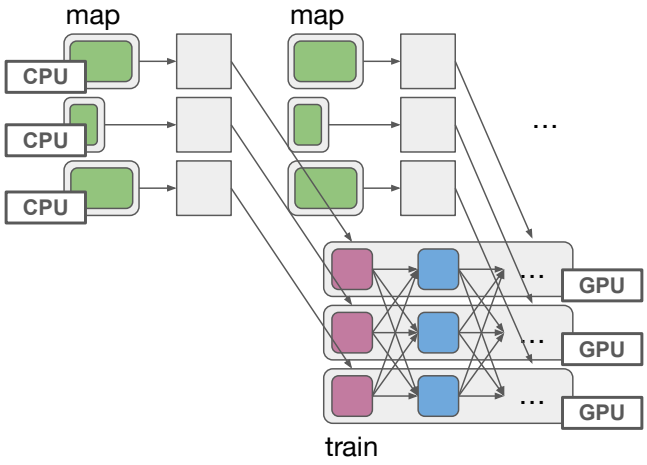
Batch and stream processing are both general-purpose models that can in principle capture distributed deep learning workloads in addition to data analytics. At a high level, ML workloads can be expressed as a logical DAG of map and reduce operations. For example, Figure 1.2a shows how an ML training job might be expressed as a DAG that takes the training dataset as input. However, in practice, extending existing data analytics frameworks to support deep learning requires high effort [208] and the result can incur significant performance overheads. The reasons stem from the fundamentally different core computation patterns of deep learning: stochastic gradient descent and accelerator-based computation.

Stochastic gradient descent (SGD) can be scaled out via data parallelism by creating multiple copies of a model, each executing over a disjoint subset of the training dataset. However, SGD is also inherently sequential, as weights must be updated before taking another step. Thus, scaling SGD with data parallelism is nontrivial and requires frequent communication and synchronization between workers.

Frequent communication is a poor fit for batch processing systems; training deep learning models on Spark for example can add an order of magnitude more overhead than a specialized system such as TensorFlow [144]. Meanwhile, stream processing systems often exploit asynchronous execution for better performance; frequent synchronization negates this edge. As an example, Naiad is a general-purpose and low-latency stream processing system that supports cyclical dataflow [146]. In principle, such a system could achieve high performance for ML dataflow graphs as well.



(a) A machine learning training job, expressed as a dataflow DAG over the training dataset.



(b) An example execution that involves both CPU-based preprocessing and GPU-based training.

Figure 1.2: Application and an example execution for distributed ML training with online data preprocessing.

However, Naiad's focus is on incremental processing and inherently unpredictable input events; ML dataflow systems assume synchronous computation and can thus use fixed-size "mini-batches" to improve performance [18].

In addition, these per-layer operations usually require accelerators (often GPUs) for sufficient performance. The use of accelerators creates significant challenges in fault tolerance and resource management. First, because GPU workers must synchronize frequently, the workers naturally fate-share. In contrast, batch processing systems structure the execution so that workers can fail and recover independently.

Second, resource management becomes more complex with accelerators in general and GPUs in particular. Frameworks such as Apache Spark [212] (batch processing) and Naiad [146] (stream processing) are designed for multicore machines. Their architectures rely on the OS to enable efficient sharing of physical memory and cores among worker threads on the same machine. Unfortunately, it is relatively expensive to share memory across CPU and GPU threads as it usually requires copy-

ing, which is limited by PCIe bandwidth [170]. Multiplexing GPU tasks ("kernels") is also challenging; since tasks are often memory-intensive, coordinating access to shared memory resources is critical but can also add significant overheads [210]. In addition, modern GPUs use specialized and proprietary communication links such as NVIDIA's NVLink that are physically separate from commodity networks.

Finally, the fully connected layers that are commonly found in deep learning models typically require all model weights to be in memory to achieve acceptable performance. When model size exceeds GPU memory capacity, this introduces an additional challenge of *model parallelism*, or scaleout of a single model copy across multiple GPUs [71, 183]. Model parallelism strategies can be further broken down into tensor vs. pipeline parallelism. Scalability of both strategies is fundamentally limited by increasing communication and/or memory cost [109, 218].

Given these issues, adding native and efficient GPU support to existing data-parallel execution frameworks has proven to require significant effort and complexity [208]. Meanwhile, it is also not sufficient to build completely distinct and GPU-based frameworks, as many workloads require both CPU and GPU execution, such as in Figure 1.2b.

1.2 Alternative Solutions

Monolithic frameworks. As a result of these differences, developers have built monolithic execution frameworks that are specialized for distributed ML. For example, Distributed TensorFlow, Apache MXNet, and PyTorch Distributed are all designed for parallel and distributed SGD. These frameworks eschew high-level execution models such as batch and stream processing and instead re-implement a dataflow interface using low-level message-passing interfaces. This allows for tight control over scheduling of parallel execution and distributed memory management. They similarly expose a DAG-based programming interface, but one that is highly specialized to neural networks. Fault tolerance is commonly supported through checkpointing.

Inevitably, however, universality turns out to be difficult to achieve. Hyperparameter search, for example, requires hierarchical and data-dependent algorithms that are difficult to express directly in DAG-based deep learning frameworks. Reinforcement learning requires both hierarchical algorithms and scale-out of black-box third-party simulators that may be CPU-based [127]. Some workloads require significant CPU-based data preprocessing and shuffling that must be overlapped with GPU computation while scaled and recovered independently [149]. End-to-end ML workflows that cover the entire model development lifecycle, from model design to training to (re-)deployment, may require these steps and more.

Building end-to-end ML workflows with monolithic execution frameworks thus requires one of two strategies. One option is to extend monolithic frameworks to support more and more types of workloads. This unfortunately has a limit in application

generality. As more varied workloads are supported, the framework becomes more complex and difficult to evolve. Even while ML training frameworks converge, *inference*, for example, turns out to require significantly different systems. This is due to the requirement of low latency for dynamic requests in inference, vs. high-throughput on static inputs for training.

Glue systems. The other option is to have application developers stitch together frameworks and custom code. This can be accomplished with general-purpose "glue" systems such as RPC, message-passing, and actor-based frameworks (the middle section of systems in Table 1.2). Such frameworks provide lower-level messaging primitives for general-purpose distributed execution and are explicitly designed to interoperate with third-party systems.

For example, to support an ML training workflow that involves data preprocessing, the data may first be preprocessed on CPUs with a Spark job, then sent to Distributed TensorFlow for GPU consumption (Figure 1.2b). This process would be repeated until training completes. The two job types could be coordinated and overlapped using RPC.

However, this strategy also puts the onus of achieving end-to-end properties such as performance and fault tolerance on the application developer, as glue frameworks typically do not take on the necessary responsibilities of managing resources, data movement, and recovery across frameworks.

Consider resource management in glue frameworks. RPC, message-passing and actor-based systems often support parallel execution through service sharding as well as multithreaded and/or asynchronous remote invocation. One caveat, however, is that these systems typically do not manage application *resources*. For example, suppose some application workflow involves parallel execution across two distributed application frameworks sharing a cluster, and both are built on top of RPC. The RPC layer has no visibility into the resources held by each application framework, and thus will not be able to ensure that resources are appropriately allocated between the two frameworks. This responsibility is typically left to a third party, e.g., a container orchestrator such as Kubernetes [53].

Efficient data movement is a greater challenge. While the location of a server (e.g., an RPC service or an actor) may be transparent, the message *data* that is sent to the server is not. In particular, *a client must have the message data local, and sending the data requires copying the data.* For request-response paradigms, the server must similarly copy the response data back to the client. This can be expensive when the data sent is large. It is also unnecessary when the response is meant to be processed by another server.

Monolithic frameworks often avoid this bottleneck by building custom subsystems for distributed memory management. Unfortunately, however, using a glue system to pass data *between* monolithic frameworks will still result in copying the data, often via disk or network.

Finally, glue systems do not provide the same level of failure transparency that is typically provided by an application-specific framework. This is due to their general-purpose nature: providing the same level of failure transparency and durability as a framework specialized to distributed ML, while also providing generality, would result in exorbitant performance overheads and/or system complexity. A single strategy for recovery, such as message logging, cannot provide optimal runtime overhead for all applications [78]. Offering multiple strategies increases complexity. As a result, most glue systems offer only weak recovery guarantees. Typically, this means providing at-most-once or at-least-once execution semantics, with optional utilities to aid in building fault-tolerant services [2, 3].

Thus, while glue systems offer useful utilities for scheduling and communication between monolithic frameworks, their functionality falls short of modern application demands. Managing resources, data movement, and recovery across frameworks becomes the developer's responsibility. Given the complexity of this problem, developers will sacrifice performance for a simple solution. For example, writing data to a highly available and scalable cloud storage system is a simple solution. It allows developers to move data in a fault-tolerant way between two different frameworks, and without having to manage any shared resources between the producer and consumer(s). However, this method can also result in unnecessarily high latency and wasted resources.

1.3 Overview and Contributions

In this thesis, we propose a general-purpose execution layer for data-intensive applications and a flexible approach to fault tolerance.

In Chapter 2, we first propose a general-purpose distributed programming interface based on *distributed futures*, actors, and tasks. Distributed futures extend RPC with a shared and immutable address space and thus provide a common abstraction for distributed memory. We explore how such an interface can and has been used to support different kinds of applications, as well as the implementation challenges that it imposes.

The remaining chapters describe an architecture for this interface. We use Chapter 3 to motivate the need for an architecture that provides a choice of recovery strategy, which is realized in Chapters 4 and 6. Chapter 4 also describes the design of the distributed memory subsystem, which is leveraged in Chapter 5 to support MapReduce-style applications.

Thus, in Chapter 3, we first present the *lineage stash*, a fault tolerance technique that targets the proposed interface and provides a strong guarantee of exactly-once semantics for distributed futures and actors. The lineage stash aims to resolve the tradeoff between checkpointing- vs lineage-based recovery approaches: results for applications in distributed training and stream processing show that the lineage stash provides task execution latencies similar to checkpointing alone, while incurring a

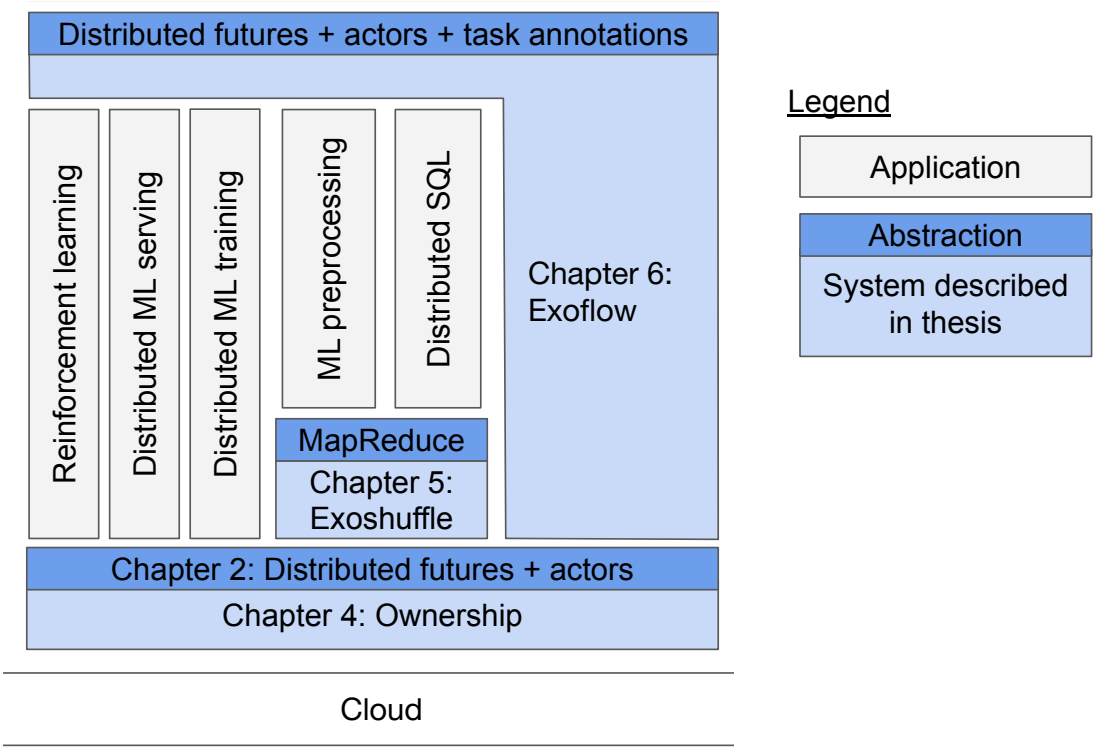


Figure 1.3: Thesis overview. Light blue indicates a system discussed in this thesis, while dark blue indicates the abstraction exposed by that system. Light gray boxes indicate application libraries that may be built on top of the system abstractions.

recovery overhead as low as other lineage-based approaches. However, the lineage stash is designed for a specific category of applications: low-latency and data-intensive applications that might contain nondeterministic control flow. Thus, this chapter also serves to illustrate the challenges in providing one-size-fits-all transparent recovery as part of a general-purpose execution layer.

In Chapter 4, we present *ownership*, which serves as the foundation for an alternative and extensible approach to fault tolerance. The ownership architecture for distributed futures and actors provides RPC-like latency and scalability, but extends RPC with a fault-tolerant distributed memory layer. The ownership architecture provides minimal but essential recovery guarantees. Exactly-once semantics is only guaranteed for deterministic and idempotent tasks. The system does not provide durability or high availability for values stored in distributed memory. However, it does provide essential fault tolerance features needed by most applications: a choice between at-least-once or at-most-once semantics, and fault-tolerant distributed memory management. The latter comprises two properties under failure: 1) prevention of distributed memory leaks (completeness), and 2) reachability and failure detection for values in distributed memory (soundness).

Key applications of the ownership system that fit these assumptions include those that can implement checkpointing at the application level with minimal effort and overhead. For example, distributed ML training jobs are often run offline and can straightforwardly checkpoint their state between epochs with minimal coordination. Furthermore, applications that can be implemented with deterministic and idempotent computations will benefit from transparent recovery. For example, in Chapter 5, we show how Exoshuffle extends the ownership layer with out-of-core processing to support the large-scale and fault-tolerant MapReduce applications commonly seen in data analytics. We show that Exoshuffle can achieve comparable performance and data scale as more specialized data processing frameworks, while also providing greater system extensibility and application interoperability with domains outside of data analytics, such as ML training.

In Chapter 6, we present *Exoflow*, a layer that builds upon the ownership system to expand the recovery guarantees and support a broader set of applications. In particular, Exoflow is an orchestrator for programmatic and distributed *workflows* that provides exactly-once semantics and durability to the application. Workflows are task pipelines have traditionally been used to coordinate team processes; programmatic workflows are ones that can be written as software while distributed workflows are a more recent concept that are used to coordinate across distinct services or frameworks [5, 8, 11]. Unlike other distributed workflow systems, Exoflow decouples the unit of recovery from the unit of execution. In particular, different units of execution may be recovered through different strategies, thus allowing more flexible tradeoffs between run-time and recovery overheads. To do this, Exoflow extends distributed futures with additional abstractions for expressing application semantics such as nondeterminism, which Exoflow then uses to decide how to recover each unit of execution.

We conclude in Chapter 7 with discussion of related work and future directions towards a distributed OS for data-intensive applications. Broadly, future directions are analogous to the problems found in traditional single-machine OSes. In particular, these include: providing performant but extensible system mechanisms, greater extensibility towards heterogeneous devices such as GPUs, and codesign of distributed execution systems with programming languages.

Chapter 2

Distributed futures and RPC

In this chapter, we present *distributed futures*, an extension to the popular RPC API and the key abstraction used in the following chapters. We argue that distributed futures provides a necessary feature for data-intensive RPC applications: a shared (but immutable) address space. Much like how RPC factors out communication from distributed applications, distributed futures factors out distributed memory management from distributed data-intensive applications. Here, we lay out the memory management functionalities that may be factored out and present some of the open challenges in building such a system. The architecture of such a system and the challenges relating to fault tolerance are addressed in later chapters.

2.1 Introduction

RPC has been remarkably successful. Most distributed applications built today use an RPC runtime such as gRPC [9] or Apache Thrift [6]. The key behind RPC's success is the simple but powerful semantics of its programming model. In particular, RPC has *no shared state*: arguments and return values are *passed by value* between processes, meaning that they must be copied into the request or reply. Thus, arguments and return values are inherently *immutable*. These simple semantics facilitate highly efficient and reliable implementations, as no distributed coordination is required, while remaining useful for a general set of distributed applications. The generality of RPC also enables *interoperability*: any application that speaks RPC can communicate with another application that understands RPC.

However, the lack of shared state, and in particular a shared address space, has its limitations. In particular, for data processing domains such as data analytics and machine learning, end applications do not typically use RPC directly. This is because pass-by-value works well when data values are small, but when data values are large, it may cause inefficient data transfers. For example, if a caller invokes $x = f()$ then

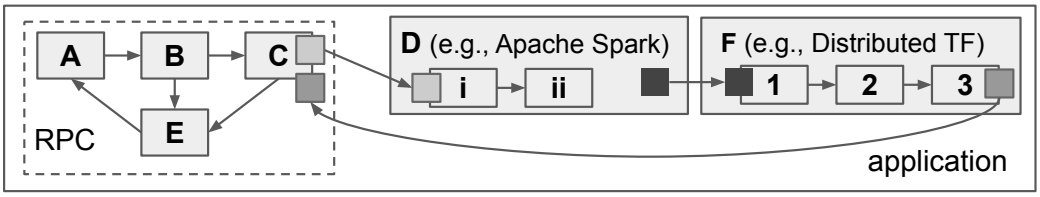


Figure 2.1: A single "application" actually consists of many components and distinct frameworks. With no shared address space, data (squares) must be copied between different components.

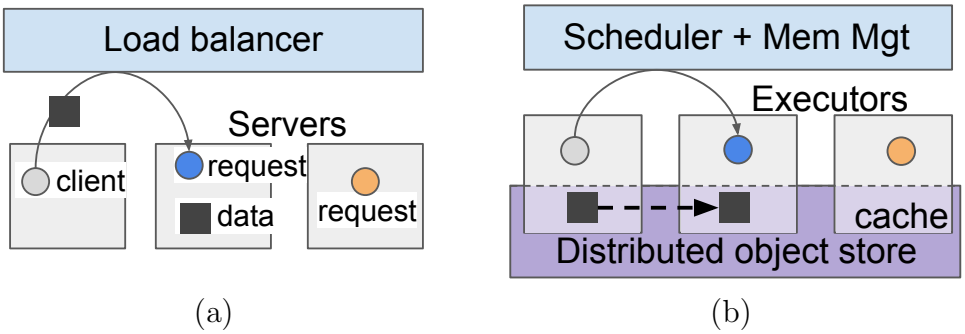


Figure 2.2: Logical RPC architecture: **(a)** today, and **(b)** with a shared address space and automatic memory management.

$y = g(x)$, it would have to receive x and copy x to g's executor, even if f executed on the same server as g.

There has been more than one proposal to address this problem by introducing a shared address space *at the application level* [62]. The common approach is to enable an RPC procedure to store its results in a shared data store and then return a *reference*, i.e., some metadata that acts as a pointer to the stored data. The RPC caller can then use the reference to retrieve the actual value when it needs it, or pass the reference on in a subsequent RPC request.

While a shared address space can eliminate inefficient or unnecessary data copies, doing it at the application level places a significant burden on the application programmer to manage the data. For example, the application must decide when some stored data is no longer used and can be safely deleted. This is a difficult problem, analogous to manual memory management in a non-distributed program.

Thus, instead of using RPC directly, distributed data-intensive applications tend to be built on top of specialized frameworks like Apache Spark [212] for big data processing or Distributed TensorFlow [18] for machine learning. These frameworks handle difficult systems problems such as memory management on behalf of the application. However, with no common foundation like RPC, *interoperability* between

these frameworks is a problem. Resulting applications resemble Figure 2.1, where some components communicate via RPC and others communicate via a framework-specific protocol. This often results in redundant copies of the same data siloed in different parts of an application.

We argue that RPC itself should be extended with a shared address space and *first-class* references. This has two application benefits: 1) by allowing data-intensive applications to be built directly on RPC, we promote interoperability, and 2) by shifting automatic memory management to a common system, we can reduce duplicated work between specialized frameworks. Indeed, we are already starting to see this realized by the latest generation of data systems, including Ciel [148], Distributed PyTorch [12], and Ray [145], the system described in this thesis. All implement an RPC-like interface with a shared address space. Our aim is to bring attention to the common threads and challenges of these and future systems.

At first glance, introducing a shared address space to RPC seems to directly contradict the original authors, who argued that doing so would make the semantics more complex and the implementation less efficient [44]. We believe, however, that these concerns stem not from a shared address space itself, but rather from supporting *mutability*. By adding an *immutable* shared address space, *we can preserve RPC's original semantics while enabling data-intensive applications.*

Immutability simplifies the system design, but what concretely would supporting a shared address space require? To answer this question, we first consider the design of RPC systems today. While RPC is often assumed to be a point-to-point communication between the caller and a specific callee, today's RPC systems look more like Figure 2.2a: a load balancer *schedules* an RPC to one of several servers (which can themselves act as an RPC client). Even the original authors of RPC provided such an option, known as *dynamic binding* [44]. Extending this architecture with an immutable shared address space would mean: (1) augmenting each "executor" with a local data store or cache, and (2) enhancing the load balancer to be *memory-aware* (Figure 2.2b).

In the rest of this chapter we explain what it means to add first-class support for immutable shared state and pass-by-reference semantics to RPC. We give examples of applications that already rely on this interface today. Then, drawing from recent data systems, we discuss the challenges and design options in implementing such an interface.

2.2 API

There are two goals for the API: (1) It should preserve the simple semantics of RPC, and (2) It should allow the system to manage memory on behalf of the application. We use immutability to achieve the former. It is the simplest approach because the system does not need to define and implement a *consistency model* as part of the API.

`Ref r`	A first-class type. Points to a value which may not exist yet and may be located on a remote node.
`shared(Ref r)` → `SharedRef`	Returns a copy of `r` that can be shared with another client by passing to an RPC.
`f.remote(Ref r)` → `Ref`	Invoke `f`. Pass the argument by *reference*: the executor receives the dereferenced argument. Returns a `Ref` pointing to the eventual reply.
`f.remote(SharedRef r)` → `Ref`	Invoke `f`. Pass the argument by *shared reference*: the executor receives the corresponding `Ref`. Returns a `Ref` pointing to the eventual reply.
`get(Ref r)` → `Val`	Dereference `r`. This blocks until the underlying value is computed and fetched to local memory.
`delete(Ref r)`	*Called implicitly when `r` goes out scope.* The client may not use `r` in subsequent API calls.

Table 2.1: A language-agnostic pass-by-reference API.

Meanwhile, the application remains free to implement mutability with local state.

We will use the term "client" to mean the process that invokes an RPC, while the "executor" refers to the server process that executes the RPC. Note that the executor may also be an RPC client, if it invokes RPCs on other executors.

First-class references. A *first-class* primitive is one that is part of the system API. For references, it means that the RPC system is involved in the creation and destruction of all clients' references. Compared to the original RPC proposal [44], there are three key differences in the API (Table 2.1):

First, all RPC invocations *return a reference* (of type `Ref`). The caller can dereference an RPC's reply when it needs it by calling `get`. We choose to have all RPC invocations return a reference so that the application never needs to decide whether an executor should return by value or by reference. Instead, this is decided transparently by the system. References are logical, so a system implementation can choose to pass back all replies by value, by reference, or both, e.g., depending on the size of the reply. A future extension to the API could allow applications to control this decision, if needed.

Second, the client can pass a reference as an RPC argument, in addition to normal values. There are two options for passing `Refs` as arguments. If a function with the signature `f(int x)` is passed a `Ref` argument, the system implicitly dereferences the `Ref` to its `int` value before dispatching `f` to its executor. Thus, the executor never sees the `Ref`.

In some cases, it may be useful for another executor receive the reference instead of the value. To support this, we also support remote functions with the signature

`f(Ref r)`, and the caller must pass a `SharedRef` instead of a noraml `Ref`. The executor in this case *shares* the `Ref` argument passed by its caller, meaning that the executor can further pass the `Ref` to another RPC or call `get`.

The intention behind the two different API options is to make the developer conscious of data movement and to allow the system to optimize memory management. With implicit dereferencing, the callee does not begin execution until *all* of its `Ref` arguments are local. This can require more memory usage and may not be suitable for incremental computation, but it also gives the system control over memory management, e.g., it can wait until all arguments are ready before scheduling the function. On the other hand, with `SharedRefs`, the system has less visibility into how the callee will use the received `Ref`, i.e., whether and when it will call `get`. `SharedRefs` do however allow for efficient delegation: the executor may forward the shared `Ref` to another executor for the actual computation, without needing the data local.

Third, a client uses the `delete` call to notify the system when it no longer needs a value. Note that this is *implicit*: it is not exposed to the application and should be called automatically by the language bindings when a `Ref` goes out of scope. This is important for memory safety, as we will see next.

Distributed futures. A third requirement for the API is that it should explicitly support *parallelism*. Parallelism is of course critical to the performance of many data-intensive applications. It has already been addressed in part by the addition of *asynchrony* to RPC, hence the focus in this chapter is primarily on first-class references and their implications for memory management.

A common asynchronous API has each RPC invocation immediately return a *future*, or a pointer to the eventual reply [35, 129]. In Section 2.3, we explain how this in conjunction with a *reference*, i.e. a pointer to a possibly remote value, gives the system insight into the application, by giving a view of the caller's future requests. Thus, we introduce the term *distributed future* to mean a future that is also a first-class reference to a possibly remote value. The impact on the API shown in Table 2.1 is that the function invocations are asynchronous.

2.3 Automatic memory management

First-class references allow the system to manage distributed memory on behalf of the application. For contrast, we will consider an application-level shared address space implementation that combines a key-value store with an existing RPC system. The application uses keys as references. We will call these *raw references* because their operations are not encapsulated by the RPC API. Thus, the system is fully or partially unaware of operations such as reference creation or deletion.

We consider four key operations in memory management, the concrete design of which we will explore in later chapters:

Allocation. The minimum requirement is the ability to allocate memory without specifying *where*. This is analogous to `malloc`, which handles problems such as fragmentation in a single-process program. This requirement is easily met by both raw and first-class references, as key-value stores do not require the client to specify where to put a value.

Reclamation. Reclaiming memory once there are no more references is a key requirement for applications with nontrivial memory usage. This is challenging in a distributed setting. It requires a fault-tolerant protocol for distributed garbage collection [160]. A key benefit of first-class references is that the API allows the system to implement this protocol on behalf of the application because *all reference creation and destruction operations are encapsulated in the API.*

In contrast, it is virtually impossible for the system to determine whether the application still holds a raw reference, analogous to determining whether a raw pointer is still in scope in a single-progress program. The problem is that raw references allow and even encourage the application to create a reference at any time, e.g., by hard-coding a string key. Thus, correctness requires manual memory management.

Movement. The primary motivation of pass-by-reference semantics is to eliminate unnecessary distributed data movement. The use of raw references shifts the responsibility of data movement to the system: the application has to specify when to move data (i.e., by calling `get`), but not how or where. The use of *first-class* references in combination with futures allows the system to also decide *when* to move data. By coordinating data movement with request scheduling, the system has greater control in optimizing data movement.

For example, a key system feature is *data locality*: when the data needed by a request is large, the system can choose an executor near the data rather than moving the data to the executor. This cannot be implemented with a `put`/`get` key-value store interface. It is straightforward with first-class references because each request specifies the `Refs` that it needs to the system, *before* placement.

Even if a key-value store were extended for data locality, we would still miss out on valuable optimizations, such as *pipelining of I/O and compute*. For example, if an executor had incoming requests $f()$ and $g(x)$, the system could schedule $f()$ while fetching x in parallel. The use of futures enables this optimization for requests from the *same* client, by allowing a client to make multiple requests in parallel.

Thus, the `Ref` API gives the system visibility into each request's data dependencies, affording unique opportunities in optimizing data movement with request scheduling. This also motivates our choice to expose two options for argument deref-

erencing, either implicit (by passing a `Ref`) or explicit (by passing a `SharedRef`). Much like raw references, there is ambiguity around if and when an executor will dereference a `SharedRef`, which affects the usefulness of system optimizations. For example, a request with `SharedRef` arguments may simply pass the references to another RPC instead of dereferencing them directly. In this case, scheduling the request according to data locality brings no benefit.

Memory pressure. To improve throughput, a single server machine generally executes multiple RPC requests concurrently. If the total memory footprint is higher than the machine's capacity, at least one process will be killed or swapped to disk by the OS, incurring high overheads [15, 172]. For example, with pure pass-by-value, the scheduler would not be able to queue new requests once the local memory capacity was reached. With raw references, each request would contain only references to its dependencies, so additional requests could be queued. However, this only defers the problem: the concurrent requests would still overwhelm the machine once they began execution and called `get` on their dependencies.

A *memory-aware scheduler* can ensure stability and performance by coordinating the memory usage of concurrent requests. This is true independent of a shared address space. However, a key challenge is determining each request's memory requirements. One could require developers to specify memory requirements, but in practice, this is very difficult.

First-class references give the system rich information about each request's memory requirements, *with no developer effort*. This is again because the scheduler has visibility into each request's dependencies. Thus, the scheduler could for example choose a subset of requests for which to fetch the dependencies, based on dependency size and the available memory.

Thus, only a system with first-class references can adequately handle reclamation, movement, and memory pressure for the application. In fact, we argue that *RPC systems with a shared address space should not expose raw references to the application*. If they do, it is at the developer's risk.

2.4 Is the API enough for applications?

The latest generation of distributed data systems shows us how valuable automatic memory management is to applications. We argue that these systems are in fact RPC systems with a *shared address space* (Table 2.2), even if they may not call themselves as such. All have a function invocation-like interface. Most use an immutable shared address space, and most use first-class references.

Despite the many API commonalities, these systems were originally proposed for a wide range of application domains, from data analytics to machine learning. We argue that the API proposed in Table 2.1 is sufficiently general because it can

System	Applications	Immutable	First-class refs	Futures	Shared refs	Stateless fns	Stateful fns
Ciel [148]	Data processing	✓	✓	✓	✓	✓	×
Ray [145]	RL, ML	✓	✓	✓	✓	✓	✓
Dask [171]	Data analytics	✓	✓	✓	×	✓	✓/×
Distributed PyTorch [12]	ML	×	✓	✓	✓	×	✓
Distributed TensorFlow [18]	ML	✓	✓	×	×	✓	✓
Cloudburst [186]	Stateful serverless	×	✓/×	✓	✓	✓	✓

Table 2.2: RPC-like systems that expose a shared address space. Each system was designed for the listed application domain. Systems that implement both first-class references and futures are *distributed futures* systems.

be used to express any application targeted by one of the systems in Table 2.2. To illustrate this, we will describe three concrete applications (Figure 2.3) that drove the development of some of these systems.

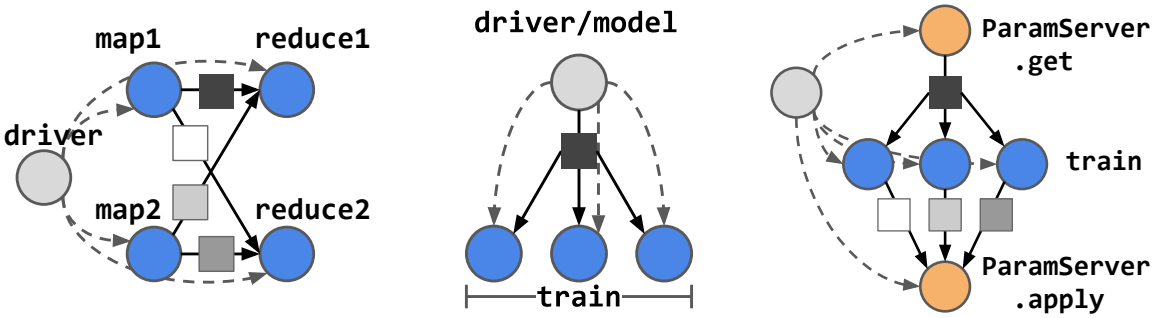
Data processing. Ciel is a universal execution engine for distributed dataflow processing that uses a distributed futures API virtually identical to that proposed [148]. Unlike data processing systems such as Apache Hadoop [205] and Spark [212], which implement a *data-parallel* interface, Ciel uses *task parallelism*: each task is a function that executes on a data partition. For example, Figure 2.3a shows a simple map-reduce application with two tasks per stage. Ciel programs can also have *nested* tasks, similar to an RPC executor that itself invokes RPCs.

Ciel shows equal or better performance than Hadoop for synchronous data processing [148]. Futures and first-class references are used to express the dataflow graph to the system. Ciel also shows that its interface is general, supporting both iterative and data-dependent control flow patterns [148].

Reinforcement learning. Ray [145], the system described in this thesis, uses a distributed futures API to support emerging AI applications such as reinforcement learning (RL). RL requires a combination of *asynchronous* and *stateful* computation [152]. A typical algorithm proceeds in asynchronous stages: a driver sends the current model to a number of `train` tasks (Figure 2.3b). The `train` tasks are stateful because the executors hold an environment simulator in local memory.

Futures allow the driver to process the `train` results asynchronously[1]. First-class

[1]Ray extends the API proposed in Table 2.1 with a `wait` call that returns the first ready reference,



(a) Data processing (b) RL (c) Parameter server

```
# (a) MapReduce.
map_out = [map.remote(i) for i in range(m)]
out = [reduce.remote(map_out[i][j] for i in range(m))
       for j in range(r)]
get(out)

# (b) RL, one round.
refs = [train.remote(i, weights_ref) for i in range(3)]
while ready_ref = wait(refs):
    result = get(ready_ref)
    # ... apply the result ...

# (c) Parameter server, one round.
weights_ref = ps.get.remote()
refs = [train.remote(i, weights_ref) for i in range(3)]
ps.apply.remote(refs)
```

(d) Pseudocode using a pass-by-reference API (Table 2.1).

Figure 2.3: Applications for a pass-by-reference API. Legend: gray circle is the client, other circles are RPCs, dashed arrows are RPC invocation, solid squares are data, solid arrows are dataflow.

references are used to reduce redundant data copies when sending the model weights to the `train` tasks (Figure 2.3b). Ray's distributed object store optimizes this with: (1) shared memory to eliminate copies between executors on the same machine, and (2) a protocol for large data transfer between machines [145].

Parameter server. A primary motivation for Distributed PyTorch [12] and Distributed TensorFlow [18] is model training. A standard algorithm is synchronous stochastic gradient descent (SGD), using a parameter server to store the model weights [63, 124] (Figure 2.3c). In each round, each worker gets the current weights from the parameter server, computes a gradient, and sends the gradient to the parameter server to be aggregated. Synchrony is important to ensure that gradients are not stale [63].

Similar to Figure 2.3b, first-class references allow the system to optimize the broadcast of the current weights. The driver can also use references to concisely *coordinate* each round without having the data local (Figure 2.3d).

Both Distributed PyTorch and Distributed TensorFlow use an API similar to Table 2.1, extended with higher-level primitives specific to machine learning, e.g., optimization strategies [18, 125]. Distributed PyTorch allows mutable memory, and Distributed TensorFlow requires the developer to specify a static graph instead of using futures[2].

Summary. Given the overlap in API, we believe that these systems have encountered many of the same challenges in automatic memory management. The result is duplicated effort and inconsistent feature support. For example, CIEL handles memory pressure by using a disk-based object store, but has no method of reclamation [150]. Distributed PyTorch implements distributed reference counting for reclamation but throws out-of-memory errors to the application [12]. Thus, we ask: *how can we create a common foundation for data-intensive applications that is simple, efficient, and general enough*?

2.5 Related abstractions for distributed memory

Distributed shared memory [153] (DSM) provides the illusion of a single globally shared address space across physically distributed threads. Our RPC proposal has two differences: (1) shared memory is *immutable*, and (2) the use of futures and first-class references. The former decision is informed by the historical difficulties of implementing DSM in practice [153, 41, 111, 123]. The latter is valuable for capturing richer application semantics that enable automatic memory management (Section 2.3).

similar to `get` with a timeout.

[2]Distributed TensorFlow v2 supports eager execution, which produces a dynamic graph, similar to futures.

Some systems have introduced novel and rich abstractions to manage consistency for mutable shared state [76, 206, 25]. For example, FaRM [76] uses distributed transactions while Anna [206] offers a range of consistency levels. We chose a minimal approach that preserves the pass-by-value semantics of RPC and avoids imposing a consistency model. This does not preclude developers from using or implementing other consistency models at the application level.

Distributed message queues [154, 114] provide a common communication system based on publish/subscribe event streams. They handle message dispatch to multiple consumers, and thus can be considered a higher-level form of RPC that provides a shared adddress space. For example, in Figure 2.1, a message queue could act as a replacement for RPC in the lefthand application. However, the focus of message queues is on supporting online applications, where messages are typically smaller and handlers are less compute-intensive than in data processing. In particular, message queue systems may not place as great an emphasis on problems such as CPU or memory contention between workers and reducing data movement.

Other systems introduce new abstractions for accessing remote memory, including distributed data structures [49] and primitives for a single "object" [172]. These are fundamentally different approaches to programming distributed memory. In particular, we call for tightly coupling the notion of *functions* with remote memory (Section 2.2) and *co-designing* function scheduling and memory management (Section 2.6).

Section 2.4 summarizes modern system manifestations of pass-by-reference RPC. Many have handled some but not all of the problems in memory management discussed in Sections 2.3 and 2.6. None have fully addressed the challenges of interoperability, which is unsurprising given that it was not their explicit goal. For example, the concept of ownership in the system Ray handles automatic memory reclamation and recovery, but requires all references to be coupled to their creator, which is a problem for interoperability [201].

2.6 System Implementation and Thesis Work

2.6.1 Memory management

The design of a shared address space requires some *distributed object store*. In particular, this object store should add some in-memory storage to each server that is co-located with the executor (Figure 2.2b). While the object store could be implemented directly with an external and distributed key-value store, we argue that *to realize the benefits of pass-by-reference, we must* ***co-design*** *the scheduler and memory management systems*. For example, we may decide what to cache locally based on what functions are queued locally.

The remainder of this thesis describes such a design, in which the RPC scheduler

and object manager are co-designed. We will use the term "task" to mean a single RPC invocation, to better match the terminology used in data processing systems, while a "distributed future" is the first-class reference and future returned by a task. As an introduction to the main thesis work, we examine each of the key memory management operations to show why co-design with the task scheduler is essential and to briefly describe how this thesis work handles the operation:

Reclamation. Reclamation must ensure *memory safety*, i.e. referenced values should not be reclaimed, and *liveness*, i.e. values that are not referenced should eventually be reclaimed. A common approach is reference counting, to avoid the need for global pauses [160]. Concretely, this means that for each `Ref`, the system tracks: (1) whether the caller still has the `Ref` in scope, and (2) whether any in-flight requests have the `Ref` as an argument. The latter requires cooperation between memory management and the scheduler. *Shared references* (Section 2.2) further require the system to track (3) which *other* clients have the `Ref` in scope.

We study the design of an efficient and fault-tolerant memory reclamation system in the ownership work (Chapter 4). At a high level, the idea is to design a system in which reference counts can be kept local to the caller, i.e. the "owner", for efficiency. To support fault-tolerant reclamation, we implement fate sharing, i.e. reference holders fail if the owner fails.

Movement. Co-design of scheduling and memory management gives the system greater control over data movement and sharing within the distributed object store. For example, a key-value store can improve data access time for skewed workloads by replicating a hot key. However, it must do this reactively, according to how often a key is used and where the keys are needed [33]. In contrast, a co-designed system could simultaneously schedule the consumers of some data with a broadcast protocol, e.g., using a dynamic multicast tree. There is also much previous work here that could be leveraged, including collective communication from HPC [86] and peer-to-peer networking systems such as BitTorrent [162].

The ownership work in Chapter 4 describes the mechanism for data movement, while the Exoshuffle work in Chapter 5 extends this design with more sophisticated policies on when to move data. In the latter chapter, we design system optimizations such as data locality and pipelining of task execution with data movement. An additional reference that may be of interest is the Hoplite work [219], which introduces a dynamic collective communication system for pass-by-reference RPC.

Memory pressure. First-class references allow the system to control how much memory is used by the arguments of concurrent requests (Section 2.3). However, this is not enough to ensure available memory, as the size of a request's *outputs* are not known until run time. Thus, barring user annotations, the system must have

a method for detecting and handling when additional memory is required by the application.

The system could: 1) throw an out-of-memory error, 2) kill and re-schedule a memory-hungry request, or 3) swap out memory (at an object granularity) to external storage. Options 1 and 2 are simple but cannot guarantee progress. Option 3, a standard feature in big data frameworks [205, 212, 163], guarantees progress but can impose high performance overheads. In some cases, simply limiting request parallelism can guarantee progress without having to resort to swapping. One challenge is in designing a scheduler that can efficiently identify, avoid, and/or handle out-of-memory scenarios.

The ownership work in Chapter 4 supports option 1, the most minimal of the options. The Exoshuffle work in Chapter 5 extends the system to support option 3. Supporting option 2 without adding undue overheads remains an open question for the future.

2.6.2 Fault tolerance

Failures are arguably the most complex part of introducing a shared address space, as it implies that a reference and its value can have separate failure domains. As such, the main body of this thesis focuses on the problem of providing fault tolerance in a distributed futures system. In particular, our goal is to provide a spectrum of recovery vs. run-time tradeoffs, analogous to the spectrum of RPC failure handling options ranging from at-most-once to exactly-once semantics. However, because we also want to support data-intensive applications, the problem contains an additional dimension: *the data passed may be large and the computation may be resource-intensive*. This necessitates additional solutions for failure handling, e.g., to record large intermediate values efficiently. To illustrate these differences, we examine each of the commonly used semantics for RPC and discuss how adding a shared address space affects the design space.

A minimal RPC implementation guarantees *at-most-once* semantics: the system detects failures for in-flight requests and returns an error to the application. The difference with pass-by-reference is that *failures can occur even after the original function has completed successfully*, as the value may be created then lost from the distributed object store.

RPC libraries often support automatic retries, or *at-least-once* semantics. This is enough to transparently recover idempotent functions. The key difference in a pass-by-reference API is that, in addition to the failed RPC, any arguments passed by reference may also require recovery. Many systems in Table 2.2 support this through lineage reconstruction [148, 145, 171], replication [206], and/or persistence. Compared to pass-by-value RPC, these methods require storing and maintaining additional system state, as an object may be referenced well past the RPC invocation that created it.

Both of these semantics are supported in the ownership work in Chapter 4. As described above, we use the ownership system to assign each reference an owner process that manages reclamation for the value. Similarly, the owner also coordinates failure detection, and if specified, recovery via recursive task re-execution.

However, for functions that return nondeterministic values and/or that have side effects, applications may benefit from systems that can provide *exactly-once* semantics. The challenges of doing so efficiently and at scale are much the same as with RPC without a shared address space [117]. One difference is that because it is assumed that data passed by reference may be large, the cost of recording values may be higher. In this thesis, this problem is first studied in the lineage stash work in Chapter 3. The lineage stash is an efficient, decentralized, and fault-tolerant technique for recording sources of nondeterminism as part of the data lineage.

While the use of pass-by-reference may complicate the problem of providing exactly-once semantics compared to pure RPC, the use of futures also provides unique opportunity. In particular, the dataflow graph that is expressed via futures provides some application information to the system before execution begins. We explore this idea in the Exoflow work in Chapter 6. Exoflow extends distributed futures with annotations for side effects and nondeterminism. These annotations allow Exoflow to decouple the unit of execution from the unit of recovery, as they give the system necessary information for deciding the optimal way to recover each task's results.

2.7 Conclusion and Lessons Learned

Memory management is a key part of all distributed systems and is especially important in data-intensive applications. This chapter addresses the first goal of this thesis: to extract a common API from recent data-intensive systems that can be used to factor out problems in distributed memory management. The result is pass-by-reference RPC and in particular the distributed futures primitive. We argue that distributed futures should be used as a unified abstraction for "virtual memory" in distributed applications, enabling interoperability and faster development of future applications.

In particular, the key lessons learned in this chapter are:

1. First-class references, as opposed to raw pointers, are necessary for factoring out distributed memory management into a common system.

2. Achieving good performance for specific application domains with a general-purpose pass-by-reference RPC system hinges on co-design of the task scheduler and memory management subsystems.

3. Achieving fault tolerance is complicated by the diverse space of possible application semantics and physical properties.

Chapter 3

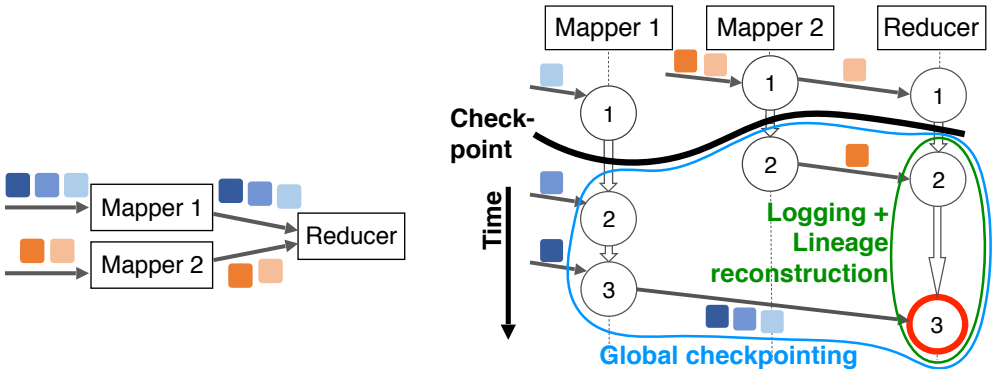
Lineage Stash: Transparent recovery for low-latency applications

Section 1.1 laid out the two major categories of system models used to support data analytics: batch vs. stream processing. Meanwhile, the previous chapter shows how we might build a common system core for these models based on the distributed futures interface. In this chapter, we begin to address some of the challenges in providing transparent recovery for such a system.

Batch vs. stream processing systems for data analytics generally employ one of two rollback recovery approaches for fault tolerance: lineage reconstruction vs. checkpointing. Lineage reconstruction exhibits low overhead during recovery but higher overhead during normal operation, while checkpointing makes the opposite tradeoff. Resolving this tradeoff will become a common theme in this thesis and in the study of fault tolerance techniques in general.

In this chapter, we present a first attempt at resolving this tradeoff. We propose the *lineage stash*, a decentralized causal logging [78] technique that significantly reduces the run-time overhead of lineage-based approaches without impacting recovery efficiency. With the lineage stash, instead of recording the task's information before the task is executed, we record it asynchronously and forward the lineage along with the task. This makes it possible to support large-scale, low-latency (millisecond-level) data processing applications with low run-time and recovery overheads. Experimental results for applications in distributed training and stream processing show that the lineage stash provides task execution latencies similar to checkpointing alone, while incurring a recovery overhead as low as traditional lineage-based approaches.

3.1 Introduction



(a) Logical. (b) Physical.

Figure 3.1: A streaming mapreduce. **(a)** Logical representation. Mappers compute a stateless function over each record (rounded box) in the input and output the results to a Reducer. **(b)** Physical representation, as a dynamic dataflow. Solid arrows show data dependencies (record batches). White arrows show stateful dependencies [145], determined by the execution order on a given process. Mappers do not have application state, but they are stateful because they can buffer records and dynamically push them to Reducer by submitting tasks, which get executed in a nondeterministic order. Reducer fails during task 3 (red), and outlined tasks must be re-executed to preserve exactly-once semantics. Lineage reconstruction (green) exactly reconstructs Reducer by replaying its inputs since the last checkpoint in the same order. Global checkpointing (blue) re-executes *all* processes' tasks since the last checkpoint, possibly in a different order (e.g., Reducer may execute task 3 before 2).

Recent data processing applications in domains ranging from stream processing [56, 146] to reinforcement learning [145] have become increasingly *online* and user-facing, making the need for low latency as critical as the need for high throughput. The *dynamic dataflow graph* [148, 145] is a flexible computation model that is ideal for developing large-scale online data processing applications because it can support both batch processing [148] and fine-grained stateful computation [145]. In this model, a program expresses task parallelism through asynchronous function invocations, called *tasks*, that return the distributed futures described in Chapter 2. Tasks may be stateless, i.e. free of side effects, or stateful, i.e. bound to a specific process. The directed graph of tasks forms a *dataflow* graph, and we call it *dynamic* because the graph may be created based on results returned by previous tasks [148]. Figure 3.1 shows an example stream processing application as a dynamic dataflow in which operators can dynamically push records to downstream operators.

Guaranteeing fault tolerance without sacrificing low latency during normal operation is an open challenge for dynamic dataflows when tasks are fine-grained, i.e. milliseconds long. This is a challenge because many applications require exactly-once semantics, i.e. all data inputs are reflected in the final output exactly once, for global consistency.

There are two general techniques for guaranteeing global consistency after a failure: global checkpointing and logging (Fig 3.1b). With global checkpointing, the system takes periodic application checkpoints, and in the event of failure, reruns the job from the latest checkpoint to a consistent but possibly different state (due to nondeterministic execution). With logging, the system durably logs events during execution, and in the event of failure, *exactly* replays the events to recover lost state, without needing to rollback any process that did not fail[1].

These differences have fundamental implications for the run-time and recovery overheads for data processing applications. In general, logging-based techniques incur a higher overhead during normal operation because they must record information during execution, but lower overhead during recovery, as they can use this information to reduce the amount of computation that must be replayed (Fig 3.1b). In particular, *all sources of nondeterminism* must be durably logged to ensure consistency after re-execution. In data processing, the computation is often logically deterministic, e.g., a program like WordCount should always return the same result given the same inputs. However, the *physical* computation may employ nondeterminism to improve performance, in particular when choosing which batch of data to process next. For instance, in the stream processing system shown in Fig 3.3a, processes A and B submit tasks concurrently to C, where they can be executed in any order to improve output latency. Logging would be a poor choice because it requires durably recording these events, which may occur at a millisecond time scale. Meanwhile, batch processing systems such as MapReduce and Apache Spark partition the dataset before execution and log only the *lineage*, or the computation graph, that produces each partition; this allows them to process partitions in any order and reduce the run-time overhead of logging, at the cost of having to reconstruct lost intermediate data in case of failure.

Still, the run-time overhead of lineage-based reconstruction has so far restricted its applicability to *coarse-grained* tasks that operate over large-enough partitions. This is a poor fit for latency-sensitive applications that often require processing in batches that are smaller and created on the fly. While this type of application can be easily expressed using dynamic dataflow, direct application of lineage-based reconstruction would add significant overhead. Global consistency requires that the lineage of each task be durably logged *before* execution, which requires replication to at least one remote node to tolerate non-transient failures.

Many systems for fine-grained data processing, including Naiad [146] and Flink [56],

[1]Note that in practice, logging techniques also use checkpointing, as an optimization to avoid replaying from the beginning.

rely on global checkpointing because it is simpler and adds low run-time overhead for nondeterministic choices in batching and ordering. Other than the overhead of the checkpoint itself [179], which can be reduced through asynchronous checkpointing [60, 57, 113], this approach adds minimal run-time overhead because there is no need to record nondeterministic events. On the other hand, recovery requires a coordinated global rollback of the entire system to the latest checkpoint [78], which is expensive at large scale [194]. This is because even previous work that is unaffected by the failure must be rolled back for consistency (Fig 3.1b), and new work cannot be accepted until recovery is complete. Also, because global checkpointing alone does not promise exact re-execution, guaranteeing exactly-once semantics for interactions with the outside world adds significant run-time overhead, since every such interaction requires a checkpoint to ensure it is never rolled back [78].

In this chapter, we introduce the *lineage stash*, a decentralized logging technique for dynamic dataflows that simultaneously achieves low recovery overhead and low run-time overhead. Like previous lineage-based systems, we rely on lineage reconstruction for fast recovery and low downtime. However, unlike these systems, the lineage stash doesn't require a task's lineage to be stored *before* the execution of the task. This removes the lineage overhead from the critical path during normal operation, allowing nondeterministic choices in batching and ordering to be made cheaply.

The main idea behind the lineage stash is that instead of storing the lineage in a reliable store on the critical path of execution, one can *forward the full lineage along with every task invocation*. Then, if the system needs to execute a task with a missing input (e.g., because of a failure), the worker running the task has full information about which upstream tasks need to be re-executed to reconstruct the missing input. Of course, this straw man solution is not practical as the lineage can grow very large, and the overhead of forwarding it can be prohibitive. To make this solution practical, we *asynchronously* store the lineage and forward only the most recent part which has not been durably stored yet. In particular, each worker keeps a lineage *stash* in local memory containing all tasks that it has seen recently. Each worker then runs a local protocol to flush its stash to a remote reliable store. Since flushing is asynchronous, it has negligible impact on application latency during normal operation.

The lineage stash is an example of causal logging [80, 27], a class of recovery techniques for message-passing systems in which processes asynchronously log nondeterministic events. The key challenge is to identify the minimum set of events that need to be logged such that we can guarantee *global consistency* after recovery while also guaranteeing *predictably low task latency* during normal operation. A naive logging approach could add prohibitive run-time overhead. For instance, one could log all messages, but in data processing, these messages can be arbitrarily large. The lineage stash minimizes the amount logged by exploiting the fact that the computation in data processing is usually deterministic, while the nondeterministic events can usually be encapsulated by the order of execution. For example, in Fig 3.1b,

the application's map and reduce functions are deterministic, but the order of task submission and execution is not.

In the lineage stash work, we extend ideas from both lineage reconstruction and causal logging to make them practical for large-scale, low-latency data processing. In particular, we identify the nondeterministic events that must be logged for application correctness and design an efficient protocol to store this information off the critical path of execution. We implement the lineage stash on Ray [145], a distributed framework for dynamic dataflows, and demonstrate the benefit on two representative applications in stream processing and distributed training. Whereas previous systems for these applications can achieve either low latency or low recovery time, we show that the lineage stash can achieve *both*. Thus, we present the following contributions:

1. An analysis of the nondeterministic events that must be logged in data processing applications.

2. A log storage architecture that enables simple, scalable protocols for flushing the stash and recovery.

3. The lineage stash: a causal logging technique that achieves low run-time *and* recovery overheads for fine-grained data processing applications.

3.2 Background

We present a case study of a stream processing application, which represents an important class of large-scale online data processing applications. We show how such applications can be expressed and executed as a dynamic dataflow, and present the open challenges in the proposed approach.

3.2.1 Case Study: Stream Processing

Stream processing provides the abstraction of *continuous operators* that compute a long-running computation over an infinite stream of data items, or records. Each operator consumes one or more input streams and produces an output stream. This imposes a set of requirements that is representative of large-scale fine-grained data processing applications.

First, stream processing applications have stringent performance requirements during normal operation, requiring both high-throughput data processing, because of the often large data ingest, and low latency, as the computation result will change over time and is generally desired as soon as possible.

Second, because stream processing applications often run online and computation results are needed as soon as possible, applications are sensitive to recovery

time. Especially at large scale, when the chance of a failure is greater, it is critical that applications experience little downtime after partial failures.

Finally, the types of computation performed vary widely even in a single application, which has implications on recovery correctness [107]. While much of the data processing computation may be deterministic (i.e. a function of the input stream), typically a stream processing application will also include local state, such as a sink operator that maintains computation results, as well as interactions with the external world, such as a sink operator that triggers an alert after a specified computation result. A deterministic computation can be safely re-executed many times, but computations with side effects on the outside world often require exactly-once semantics, since the outside world in general cannot be rolled back.

3.2.2 System Model and Challenges

Existing systems for stream processing fall under two categories. Systems like Flink [56] and Naiad [146] use global checkpointing for fault tolerance and instantiate physical instances of continuous operators, each of which consumes and produces buffers, or *batches*, of records. This allows for low-latency, record-at-a-time processing. In contrast, systems like Spark Streaming [213], execute *synchronous stages* over fixed-size partitions of the input stream, and record the *lineage* of each stage for fault tolerance.

Stream processing applications can be represented as a dynamic dataflow (Fig 3.1), with both *continuous operators*, as in Flink [56] or Naiad [146], and lineage-based recovery, as in Spark Streaming [213]. Each continuous operator is instantiated as a *process* with local state that can execute *tasks*, also known as methods or message handlers, submitted by upstream operators. Each task's argument is a record batch. Processes execute tasks as input batches become available (Reducer in Fig 3.1), and can flush batches to downstream processes by dynamically submitting tasks (Mappers in Fig 3.1), e.g., based on the maximum output buffer size.

The lineage in this model is recorded at the granularity of a batch. This is in contrast to Spark Streaming [213], which records lineage at the granularity of partitions, each of which may span many batches. In Fig 3.1, we show how the lineage of each batch is tracked, through data dependencies (solid arrows), and stateful dependencies (white arrows). Data dependencies are specified by the application through task arguments, while stateful dependencies are created between tasks that execute consecutively on the same process. The use of lineage can reduce downtime during recovery, as intact operators can continue processing records while lost operators can be replayed exactly from the lineage.

To execute this dataflow graph, we adopt the system model introduced by Ray [145], in which a distributed scheduler dispatches tasks to local worker processes based on their data and stateful dependencies (Fig 3.2a). Each Ray node can host multiple worker processes, which may be stateful (known as *actors* [145]). Worker processes on the same node also share an in-memory *object store*, which can be used

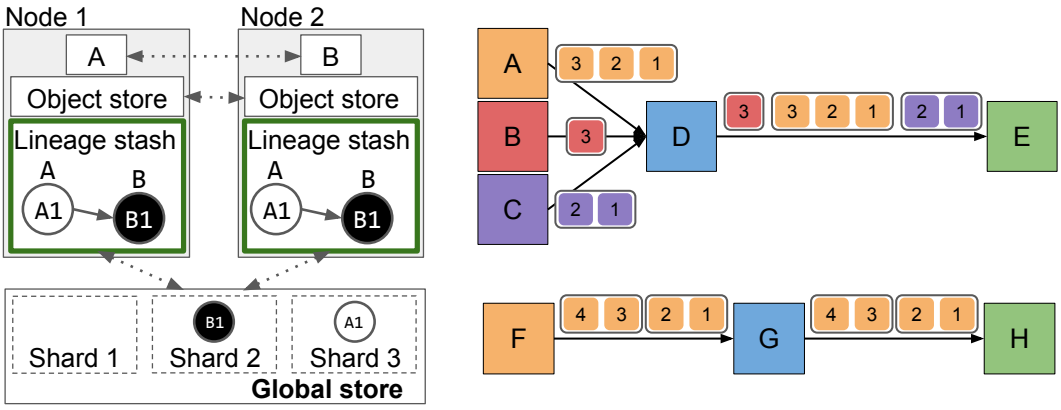


Figure 3.2: **(a)** Lineage stash architecture, on top of a decentralized dataflow scheduler. A and B are processes that can submit tasks to each other (e.g., $A1$ submits $B1$). Dotted arrows show the protocols used to communicate between nodes. **(b)** Stream processing. D is a *nondeterministic* operator that reads dynamically sized batches (buffers) from multiple input sources (A, B, C) in any order and outputs results to downstream operator E. G is a *deterministic* operator that reads statically sized batches from a single source, F.

to cache immutable copies of large task outputs. System metadata, such as task descriptions and object locations, is stored in a logically centralized *global store*, which can be sharded for scalability and replicated for durability.

There are a number of challenges in applying lineage reconstruction to this setting. First, the granularity at which lineage is recorded is much finer than in previous lineage-based systems, at the level of batches that can take milliseconds to process, compared to synchronous stages that can take seconds. Since the lineage is both significantly larger and updated more frequently than in existing lineage-based systems, the common approach of logging lineage to a centralized location [72, 205, 212] on the critical path of task execution would affect both task latency and throughput.

Second, the lineage is not only larger, it must also be updated at *runtime* to guarantee exactly-once record processing. This is because asynchronous record processing introduces *nondeterministic events* when an operator processes data from multiple sources. For example, in Fig 3.2b, operator D processes data from operators A, B, and C as they become available, then outputs results to E. If D fails but E remains active, then we must guarantee that when reconstructing D's outputs, we do so in an order consistent with what E has seen so far. This necessitates reliably recording the order in which D processed its inputs during execution, which adds latency if this must be done before E can process the results. Note that this is not an issue in lineage-based systems that execute in synchronous stages [213]; in such

systems, D would block results to E until it has processed a predetermined number of records from A, B, and C. Nor is it an issue when reading from a single input, as G does in Fig 3.2b.

These problems motivate an *asynchronous* logging approach, in which task specifications are logged to a centralized reliable storage system, but off the critical path of task execution. In particular, each node logs lineage directly to a local, in-memory *lineage stash* (Fig 3.2a), which is asynchronously flushed to the global store. However, this solution presents a third challenge: maintaining the decentralized state. The decentralized logging approach complicates both normal operation, as it creates local state that must be flushed, and recovery, as a failed operator's lineage is no longer guaranteed to be in a centralized location. The final challenge is thus in designing simple protocols for flushing local state and recovering after a failure.

In summary, the challenges are: (1) removing the cost of recording lineage from the critical path of task execution, (2) efficiently recording nondeterministic events, and (3) designing simple, scalable protocols for flushing and recovering lineage. In the remainder of this paper, we describe the lineage stash design and how it meets these challenges.

3.3 Lineage Stash Overview

The data processing applications that the lineage stash supports can be viewed logically as message-passing systems, a low-level abstraction in which a set of processes with local state communicate with each other by sending and receiving messages. For example, the continuous operators in a stream processing application can be viewed as a set of processes where each message contains a single record. In this section, we describe the relationship between a message and a *task*.

The lineage stash is a form of *rollback recovery* [78], in which information is recorded during execution to minimize the amount of work that must be redone after a failure. Informally, the lineage stash guarantees that if a process fails, then any messages that it received since its last checkpoint will be replayed in the same order. This implies that the system will recover to a *globally consistent state*–that is, for every message that has been delivered to a process, the corresponding event is reflected at the sender. This in turn implies exactly-once semantics for the application (e.g., every record is processed once in stream processing). The lineage stash can also support *end-to-end* exactly-once semantics for when the application outputs a result to the external world. As is standard in rollback recovery [78], the lineage stash targets the fail-stop model [177] and assumes that the application can identify and record any *nondeterministic* events, as well as inputs and outputs to the external world.

At a high level, we use a *causal logging* approach for recording and replaying computation. Causal logging [80, 27] is a technique that aims to lower both recovery and runtime overhead by logging asynchronously to a *stable storage* system, i.e. the

Ray system metadata store [145] or the Spark scheduler [212]. Each process buffers a log of all nondeterministic events (e.g., "received message m") that caused its current state and piggybacks any volatile records onto its messages to other processes. If a process fails, then it can retrieve logs from the remaining processes to guide its recovery. Since all nondeterministic events from the initial execution can be replayed from the logs, this guarantees global consistency.

However, practical use of causal logging in general message-passing applications remains challenging due to the sheer variety of nondeterministic events that could occur (e.g., writing to external memory, executing on a timer, etc.). Correctness requires that *all* such events are logged during execution, which can be cumbersome, expensive, or both. On the other hand, data processing applications by nature consist of mostly *pure* computation, i.e. side effect-free and the outputs are a deterministic function of the inputs. This makes causal logging a promising approach to providing rollback recovery for decentralized data processing applications. The key system challenge is then to identify and efficiently capture the sources of nondeterminism that do occur in data processing applications. There are three questions to answer: *what* information do we log, *how* do we log that information, and *how* do we recover the initial execution from these logs?

What information should we log? A general logging approach is to reliably record every message that every process receives, including the content and the execution order. Then, assuming deterministic message handlers, recovery is simply a matter of retrieving and replaying the logs.

However, this straw man approach is clearly expensive for data processing applications. The total message content in data processing applications can be much larger than the description of the computation. This is true in communication primitives like allreduce for large arrays, as the array is often much larger than the description of the reduce function. Other data processing applications consist instead of many small messages that all undergo the same computation on the receiver. An example of this is stream processing: logically, operators execute record-at-a-time, but physically, many records are batched together for efficiency. In this case, the execution order only needs to be logged at batch boundaries. In the allreduce case, the message ordering is deterministic, so it need not be recorded at all.

Fortunately, the message *content* in data processing applications is often the output of a deterministic computation performed by the sender. Thus, it can be perfectly recomputed, assuming the same inputs and sender state. This allows for a key optimization: recording the *lineage* instead of the raw data. In particular, we reliably store a pointer to the application data, called an *object*, and a concise description of the computation, called a *task*. Each task can take as input a process's local state and one or more objects, and can generate objects (return values) as well as other tasks (nested functions). The lineage of an object comprises the task that created it and the lineage of each of the task's arguments. Since object values are

deterministic, we can cache multiple immutable copies of the object across nodes. As in previous systems [212], this comes at the cost of having to recompute objects during recovery if all copies are lost. For small enough objects, the data can optionally be inlined in the task specification.

In some cases, nondeterminism is actually key to application functionality and performance. In particular, the ability to dynamically execute tasks based on data availability at runtime is essential for low latency in applications where a single process executes tasks from multiple other processes, such as in stream processing (Fig 3.2b). In this case, if another process sees the result, then the task execution order must be recorded in the lineage and made durable in case of failure. Otherwise, the nondeterministic process may recover to a state inconsistent with the witness, or *orphan process* [78]. For instance, in Fig 3.3a, processes A and B submit tasks concurrently to C, where they can be executed in any order, and the result is seen by D. If C were to fail, we must replay the tasks from A and B in the same order as before to guarantee consistency with D.

However, there are also applications where it is sufficient for a process to execute tasks in a deterministic order. For example, communication primitives like allreduce are *fully deterministic* because every process receives tasks from one other process. Determinism can also be enforced for a process with multiple callers if tasks are always executed in a specific order, e.g., round-robin. While this is more restrictive to applications, it does allow for more efficient logging, since it is not necessary to record task execution order. For instance, in Fig 3.4a, C is the only process to submit tasks to A, so the order of tasks that A executes is deterministic and need not be recorded. Note that it is possible to mix different logging levels in a single application, i.e. one process may execute tasks dynamically while others are fully deterministic.

In rare cases, a process may also execute nondeterministic events *during* a task. For instance, a stream processing application with strict latency requirements could choose to release outputs based on a timer. To support this case, we also allow the application to record such events as part of the task description so that they can be replayed exactly after a failure. While we provide system support, it is up to the application to identify and replay such events; in practice, we expect this to be done in application-level libraries.

How should we log information? While recording the lineage rather than the data greatly reduces the cost of logging, the rate at which tasks are generated can still be very high for fine-grained, decentralized applications. Thus, storing a task description reliably must be done off of the critical path. The main idea behind the lineage stash is to use a *causal logging* approach, where instead of storing the lineage reliably before the task is executed, we *forward the lineage of each of the task's inputs with the task invocation.* This way, the node executing the task has all the information to reconstruct the task's inputs, if necessary. Each node remembers the lineages of tasks that it generated or received for execution in a local store, called the *lineage*

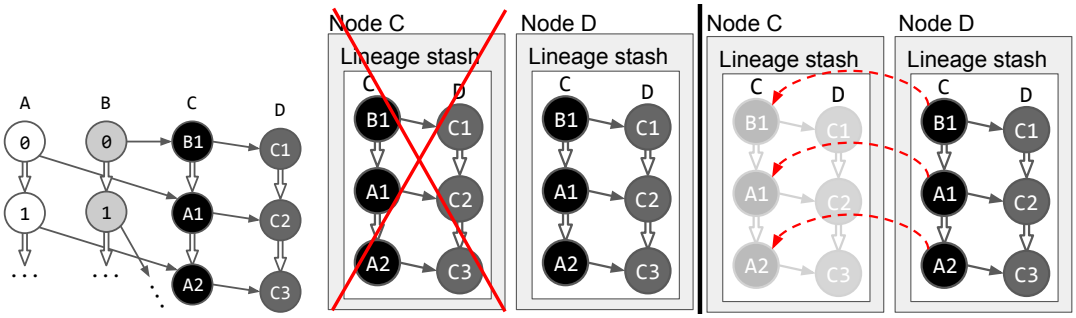


Figure 3.3: **(a)** A nondeterministic application and **(b)** a failure scenario showing why lineage must be forwarded. Because C executes tasks from A, B in a nondeterministic order, it must retrieve its lineage from D after a failure, shown by the red dashed arrows.

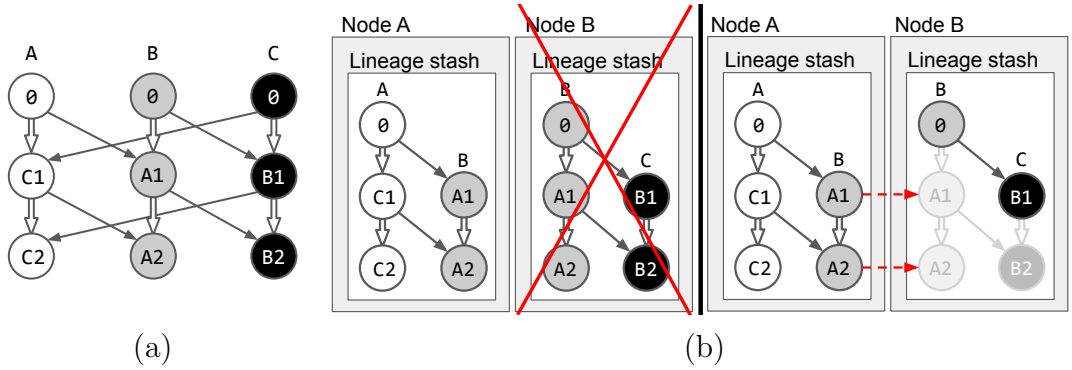


Figure 3.4: **(a)** A deterministic application and **(b)** a failure scenario showing what lineage must be remembered. To recover B after a failure, A simply resubmits (red dashed arrows) its previous tasks.

stash.

Only the nondeterministic events must be reliably recorded for recovery correctness. Thus, in deterministic applications, the lineage need not be forwarded during normal operation because it can be deterministically recreated after a failure. Each process only needs to *remember* the tasks that it has submitted, by storing them in its local lineage stash. If a process fails, then it recovers by simply asking the remaining processes to resubmit their stashed tasks. Figure 3.4b shows this for the deterministic application example in Fig 3.4a: during normal operation, A remembers the tasks that it submits to B in its local lineage stash. When B dies, A resubmits its stashed tasks ($A1$, $A2$) to recover B.

In nondeterministic applications, each process must also *forward* the lineage that it has seen so far. This is because task descriptions are updated during execution

based on nondeterministic events, such as the order of task execution. If a process fails, then it must recover the most recent copy of its tasks. For example, in Fig 3.3a, because C executes tasks in a nondeterministic order, it must also forward its own lineage to D during execution. This is so that if C dies, as in Fig 3.3b, D can resubmit these tasks to C in the correct order.

While this simple scheme of remembering and forwarding recent lineage removes the lineage store from the critical path of a task's execution, unfortunately it will not scale for realistic applications as the lineage can become very large and forwarding it can be prohibitive. Thus, timely flushing of the local stash is critical to maintaining predictably low task latency. Traditionally in rollback recovery systems, each process can asynchronously flush its volatile log to an individual stable storage system, which may be remote [80, 166, 78]. However, this requires each process to garbage-collect its stable storage and can lead to an unpredictably large storage footprint if a task is forwarded many times.

We simplify garbage collection by asynchronously flushing each stash to a global but physically decentralized (sharded) stable storage system, in which each task has a unique identifier and any process may read or append to any task (§3.4.2.2). This means that only a single copy of each task is reliably stored, and garbage collection of the stable storage can be handled by a single background process, which erases tasks previous to the last global checkpoint. Although it is not our focus in this work, this also facilitates logging for *stateless* tasks that are not bound to a specific process.

Because processes can die before they flush their lineage, task descriptions can be lost entirely before they are written to the global store. However, we guarantee that during normal execution, if a task is not yet in the global store, then all nodes that execute dependent tasks must have the task in their stash. Therefore, if we lose all nodes that have stashed a particular task, we can still guarantee consistency because no live node will have seen the result of the task.

How do we recover the logs? To recover a failed process to a globally consistent state, we must retrieve and re-execute its lineage. As noted above, for deterministic processes, it is enough for the other processes to remember the lineage of tasks that they have submitted so far. These can be resubmitted along with any stashed lineage during recovery (Fig 3.4b).

For nondeterministic processes, we must also recover the initial execution order, as well as any nondeterministic events that occured during a task's execution. For example, in Fig 3.3a, if C fails and its lineage has not yet been written to the global store, it must retrieve its lineage from D before it can accept further tasks from A and B. For an arbitrary application, the relevant lineage could reside at any process, so a failed process would have to retrieve and reconcile a subgraph of its lineage from every other process [80]. This can be expensive and complicated, especially when there are multiple simultaneous failures.

We can simplify the lineage recovery protocol with the global lineage store. Upon a failure, each remaining process simply flushes its local stash to the global store and replies to the recovering process once all writes have been acknowledged. The recovering process can then retrieve its lineage by walking the task dependencies in the global store, starting from the tasks resubmitted by the other processes. Because the global store is indexed by task rather than process, retrieving the lineage is likely slower than if the process's log were stored contiguously. While this may affect recovery performance, it allows for a simple recovery protocol; our implementation required only 125 lines of code (§3.5).

We can further optimize the recovery protocol by leveraging a property common to decentralized data processing applications: most processes send tasks to only a small set of other processes, which changes infrequently. Thus, we only need to contact this set, which is often much smaller than the total set of processes. For example, in Fig 3.3b, C only needs to retrieve lineage from D.

3.4 Lineage Stash Implementation

First, we expand on the architecture presented in Fig 3.2a and introduce the lineage stash protocols, which we present in §3.4.2 along with their guarantees. A process can send a task to a remote process using the lineage stash protocols for forwarding and remembering lineage (§3.4.2.1). Processes can also send or receive objects to or from remote processes through their in-memory object stores. We assume a fail-stop model: if a node fails, then its object store and lineage stash will be lost.

Each lineage stash flushes to a logically centralized *global lineage store*, a reliable key-value store that maps task ID to specification. All operations are over a single task, and we do not assume sequential consistency across tasks, i.e., operations on different tasks from the same node may be processed in any order. This allows us to shard the global store by task ID for horizontal scalability, as in Fig 3.2a. Each node communicates with the global store independently to flush its local stash and retrieve lineage during recovery (§3.4.2.2). As an optimization, each node can request an acknowledgement when a given task has been written to the global store.

3.4.1 Definitions

Next, we describe the lineage structure. All processes, tasks, and objects are assigned a unique identifier (ID), which can be deterministically recomputed during recovery. The task ID is a hash of the sender and receiver IDs and the number of tasks sent between the pair so far (Table 3.1), The object ID is a concatenation of the ID of the task that created it and the number of objects that the task has created so far.

The task specification (Table 3.1) can be monotonically updated to record non-deterministic events. The `predecessorId` is initially the ID of the previous task that

Field	Type	Description
id	TaskID	`hash`(*receiver*, *sender*, *taskCounter*)
version	int	# of updates to the task specification
receiver	ProcessID	ID of process that receives the task
sender	ProcessID	ID of process that sent the task
taskCounter	int	# of tasks sent from *sender* to *receiver*
applicationLog	string[]	nondeterministic events during task
parentId	TaskID	*id* of task that submitted this task
predecessorId	TaskID	`hash`(*receiver*, *sender*, *taskCounter* − 1)
argumentIds	ObjectID[]	object IDs of task arguments
dependencies	TaskID[]	[*parentId*, *predecessorId*] + *argumentIds*

Table 3.1: Task specification (*version*, *predecessorId* and *applicationLog* may be updated after task creation to record nondeterminism)

the sender submitted to the destination process. It may be overwritten once, before execution, to the task that the destination process executed immediately beforehand, to reflect the task order. During task execution, the application can also append nondeterministic events to the `applicationLog`. Each of these updates increments the task's `version`. To define global consistency, we first define a total order on tasks with the same ID. This also makes it safe to flush any version of a task; the global store simply rejects older versions.

Definition 3.1 (Task order)**.** *For tasks T and T' where $T.id = T'.id$, $T \leq T'$ if $T.version \leq T'.version$, $T.applicationLog$ is a prefix of $T'.applicationLog$, and either $T.predecessorId = T'.predecessorId$ or $T.predecessorId$ and $T.version$ are equal to their initial values (Table 3.1).*

Definition 3.2 (Lineage)**.** *The lineage of a task T consists of T itself and the lineage of all of its* `dependencies` *(Table 3.1). For convenience, we will also say that the lineage of an object is the lineage of the task that created it, and the lineage of a process is the lineage of the last task that it executed.*

$$Lineage(T) = \{T\} \bigcup_{T' \in T.dependencies} Lineage(T')$$

Recovery correctness is defined via global consistency [60], i.e., every message received by a process is also reflected in the sender's history. In terms of lineage, this means that for every task that a process has executed, if the same task appears in some other process's lineage, then the process that executed the task must have the most recent version.

Definition 3.3 (Lineage consistency)**.** *For any processes p, q, if $T_p.id = T_q.id$, p executed T_p, and $T_q \in Lineage(q)$, then $T_q \leq T_p$.*

```
def GetUncommittedLineage(stash, T):
  lineage = {}
  for D in T.dependencies:
    if D in stash:
      lineage.add(D)
      lineage.update(GetUncommittedLineage(stash, D))
  return lineage
```

(a) Getting uncommitted lineage from the local stash.

```
def AddUncommittedLineage(stash, T, lineage):
  for D in T.dependencies:
    if D in lineage and D not in stash:
      stash.add(D); AddUncommittedLineage(stash, D, lineage)
```

(b) Receiving uncommitted lineage in the local stash.

Figure 3.5: Lineage stash methods for getting and receiving a task's uncommitted lineage (Definition 3.5). A practical implementation can easily avoid forwarding duplicate lineage by recording which tasks have been sent to which nodes.

We also define task durability.

Definition 3.4 (Durability)**.** *A task T is durable if it can be found in a live process's local stash or in the global store.*

Lineage consistency after failure of a process p is guaranteed as follows: if T is the last task executed by p that is in the lineage of any live process, then all tasks in $Lineage(T)$ are durable. We ensure this property by forwarding *uncommitted lineage* with each submitted task.

Definition 3.5 (Uncommitted lineage of T)**.** *The tasks in $Lineage(T)$ that are not yet committed in the global store.*

3.4.2 Protocol

3.4.2.1 Forwarding Lineage

We describe the protocol for submitting a task from one process to another, first without flushing. For processes hosted by separate nodes, this requires a minimum of one message to send the task itself. We design the lineage stash protocol so that

```
def SubmitTask(T):
  stash.add(T); FlushTask(T)
  AssignTask(T, T.receiver, GetUncommittedLineage(stash, T)
    if P.NONDETERMINISTIC else {})
```

(a) Submit a task and forward uncommitted lineage.

```
def AssignTask(T, P, uncommitted_lineage):
  if P.NONDETERMINISTIC:
    T.predecessorId = P.lastTaskId; P.lastTaskId = T.id
    T.version += 1; FlushTask(T)
  stash.add(T)
  AddUncommittedLineage(stash, T, uncommitted_lineage)
```

(b) Assign a task and add the forwarded lineage.

Figure 3.6: Node methods for task execution. `AssignTask` also records nondeterministic execution order by updating the task's `predecessorId`. Nondeterministic events during task execution are recorded by appending to the task's `applicationLog` (not shown).

all additional information needed for recovery correctness, i.e. the task's lineage, can be computed locally by the sender and piggy-backed on this message.

As described in §3.3, only the nondeterministic events need to be forwarded to receiving nodes. If the application is deterministic, i.e. the task specifications are immutable, then it is only necessary to *remember* tasks that have been submitted so far, by adding these tasks to the local stash (Fig 3.6a).

For nondeterministic processes, the sender process retrieves the task's lineage from its local stash (Fig 3.5a) and forwards the result along with the task (Fig 3.6a). As in previous work [80], it is only necessary to forward *new* uncommitted lineage that has not yet been forwarded to the receiving process (not shown). Next, the receiver adds the forwarded lineage to its own stash (Fig 3.5b) before assigning the task (Fig 3.6b). If an added task is already present in the receiver's lineage stash, the more recent version is used (not shown). The task submission protocol is illustrated in Fig 3.7a.

The `SubmitTask` and `AssignTask` procedures maintain lineage consistency during normal execution, assuming no flushing yet.

Invariant 3.1 (Lineage durability without flushing)**.** *For each process p and task T that p has executed or submitted, T's lineage is durable.*

We include a brief informal proof by induction on the global state (all process and lineage stash state). The base case is the first task T submitted, from $p1$ to $p2$.

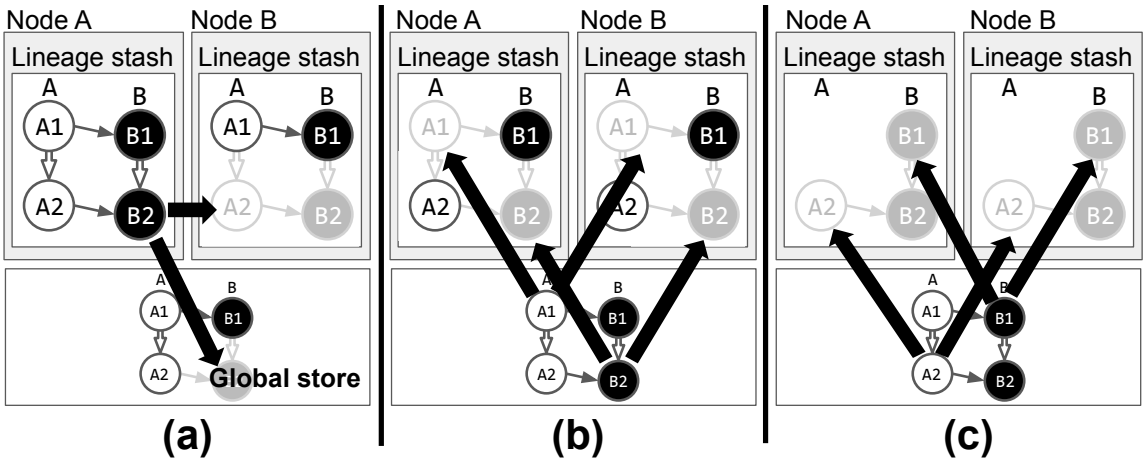


Figure 3.7: Forwarding and flushing lineage. **(a)** Task $A2$ submits task $B2$, forwards the uncommitted lineage ($A2$) to B, and asynchronously flushes $B2$. **(b)** A and B receive commit acknowledgements for $A1$ and $B2$. $A1$ can be evicted because it has no dependencies, but $B2$ cannot. **(c)** A and B receive commit acknowledgements for the remaining tasks and it is safe to evict all tasks.

T has no dependencies and T is added to both $p1$ and $p2$'s stash. If either process fails, the other process will still have T in its local stash. If both processes fail, then we need not guarantee durability; this is equivalent to restarting from the beginning. The induction step assumes that the invariant is true, and $p1$ submits T to $p2$. In this case, `GetUncommittedLineage` returns T's full lineage, which is added to $p2$'s stash. Similarly, in any failure case, T's lineage will remain in at least one live process's stash, or it is unnecessary to guarantee durability because all affected processes will also have failed.

3.4.2.2 Flushing the Stash

To prevent lineage stashes from growing indefinitely, each process flushes its stash to the shared global store. Because task versions are ordered, each lineage stash could safely flush any task that it sees. However, to avoid overloading the global store with many writes of the same task, we choose to instead flush a task every time it is updated, i.e., its `version` (Table 3.1) is incremented. When a task is submitted (Fig 3.6a), the sender asynchronously flushes the initial version of the task, as A does for $B2$ in Fig 3.7a. When the execution order is nondeterministic, the node updates the assigned task's specification to reflect its predecessor task and flushes again (`AssignTask` in Fig 3.6b). If a task executes a nondeterministic event, the node adds the application-provided entry to the task `applicationLog` and flushes again (not shown).

Each node receives commit acknowledgements for particular task versions from

```
def FlushTask(T):
  global_store.Write(T, TryEvict)
def TryEvict(stash, T):
  if T.version >= stash[T.id].version:
    for D in T.dependencies:
      if D in stash: return
    stash.erase(T)
```

Figure 3.8: Lineage stash methods for flushing to the global store. `FlushTask` writes a task asynchronously to the global store with the callback `TryEvict`. Once a task (or a newer version) is committed and its dependencies have been evicted, it is evicted in `TryEvict` (`TryEvict` also tries to evict any dependent tasks, not shown).

the global store and evicts tasks from its stash accordingly (`TryEvict` in Fig 3.8) . `TryEvict` only evicts a task if it has been committed and if its dependencies have also been evicted from the local stash. This is to guarantee that for every task still in the local stash, there is a connected subgraph in the stash that contains the task's uncommitted lineage, to ensure `GetUncommittedLineage` correctness when flushing is enabled.

As an example, in Fig 3.7b, both processes receive acknowledgements for tasks $A1$ and $B2$. Note that these acknowledgements can arrive in any order since we do not assume sequential consistency from the global store. Both stashes can evict $A1$ because $A1$ does not have any dependencies. However, $B2$ cannot be evicted yet because it has uncommitted dependencies $A1$ and $B1$. This ensures that `GetUncommittedLineage` will correctly return the uncommitted lineage for $B2$ or any future tasks dependent on $B2$.

The `FlushTask` and `TryEvict` procedures maintain the same invariant as above, but for *uncommitted* lineage. The remaining lineage is in the global store and therefore durable.

Invariant 3.2 (Lineage durability with flushing)**.** *For each process p and task T that p has executed or submitted, T's* ***uncommitted*** *lineage is in p's local stash.*

3.4.2.3 Recovery Protocol

During the recovery protocol, the failed process retrieves and re-executes the lineage of the last task that it executed before failure that exists in another live process's lineage. Note that this may differ from the last task that the failed process actually executed but is enough to guarantee global consistency.

First, each process that submitted a task to the failed process resubmits its last submitted task. Step 5 in Figure 3.9b shows this for operator C from the application example in Fig 3.3a. For a deterministic process, the lineage does not change after it

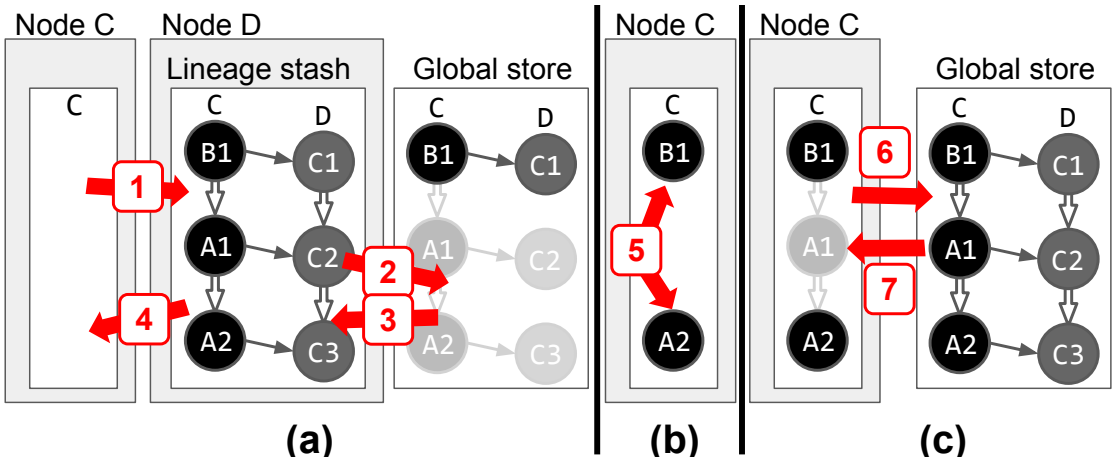


Figure 3.9: Recovery procedure for the nondeterministic process from Fig 3.3 in detail. **(a)** (1) C contacts downstream process D, (2) D flushes its lineage, (3) D receives all acknowledgements, (4) D replies to C. **(b)** Processes A and B (not shown) resubmit their last submitted tasks ($A2$, $B1$) to C. This may happen concurrently with steps 1-4. **(c)** After steps 1–5, C recovers the lineage of $A2$ and $B1$, which includes the initial execution order, from the global store.

is generated, so the resubmission step with Invariant 3.2 is enough to guarantee global lineage consistency (after re-execution). Invariant 3.2 implies that any uncommitted lineage will be forwarded with the resubmitted task. All other lineage can be retrieved from the global store.

When the execution is nondeterministic, the failed process must also retrieve the *latest* version of each task that it executed. We adopt a standard causal logging procedure [80], in which the failed process contacts other processes to retrieve their stashed lineage. However, rather than have the processes reply directly with the uncommitted lineage, each process instead flushes its entire local stash to the global store (via `FlushTask`, Fig 3.8), waits for all tasks to commit (via `TryEvict`, Fig 3.8), then acknowledges to the recovering process, which can then retrieve the flushed lineage from the global store (Fig 3.9). Using Invariant 3.2 and the fact that the global store only accepts writes for higher task versions, this guarantees lineage consistency (after re-execution).

The recovering process can then re-execute its tasks based on the lineage retrieved from the global store, for both deterministic and nondeterministic applications. The process may resubmit tasks that have already executed, which get added to its stash as during normal execution to guarantee that Invariant 3.2 will hold after recovery completes. Receiving processes can easily deduplicate these tasks with a counter.

3.4.3 Failure Model

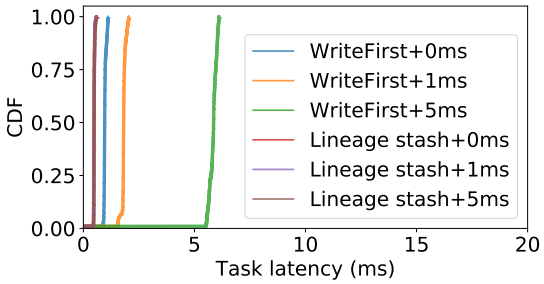
The protocols in §3.4.2 guarantee exactly-once semantics within an application. They can also support *end-to-end* exactly once semantics, e.g., if a sink operator in a stream processing application outputs to an external system. The lineage stash can support a task that outputs to the external world by first flushing the task and its local uncommitted lineage (§3.4.2.2), then waiting for the commit acknowledgements from the global store. This guarantees that if the operator fails later on, it will replay its execution in the same order and restore to a state consistent with the external world. In contrast, a global checkpointing approach alone must take a checkpoint during every such interaction to guarantee that the execution will not be rolled back [79, 4].

Like other rollback recovery systems, we target a fail-stop model [78]. Each node's local state (processes, in-memory object store, and lineage stash) can be rebuilt after a failure, possibly on a different physical node. As in previous causal logging work [26], we also allow the user to configure the maximum number of times that an uncommitted task is forwarded to lower-bound f, the number of simultaneous failures tolerated. f may be greater than the task forwarding limit depending on application properties: communication structure (e.g., in acyclic graphs [26]) and the mix of deterministic versus nondeterministic processes. We discuss further application-specific failure handling considerations here.

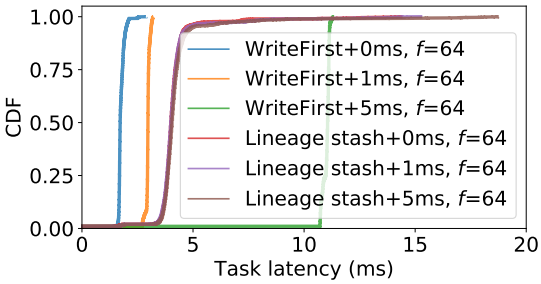
Checkpointing. Long-running applications must still take checkpoints to bound re-execution time after a failure. In theory, the application can take inconsistent checkpoints with the lineage stash. However, as prior work has shown [79], it is much simpler to take globally consistent checkpoints, to avoid coordination between processes for garbage collection of the global store. Since many applications today, e.g., distributed training [18] and stream processing [57], already provide support for efficient global checkpointing, we recommend adopting these methods to simplify and roughly bound recovery. The lineage stash can be used in conjunction to further guarantee exact replay, to reduce recovery time and runtime overheads for end-to-end exactly-once semantics.

Intermediate State. In general, logging approaches collect state during execution in order to reduce recovery overheads and garbage-collect the state after a checkpoint. Therefore, in every logging approach, it is possible for this intermediate state to exceed storage capacity.

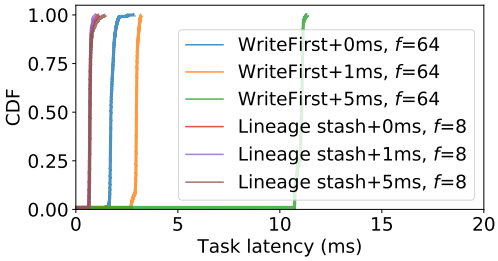
For the lineage stash, there are three types of intermediate state. First, for the lineage in the local stash, the node can apply backpressure on the local processes until enough tasks have been flushed, via the protocol in §3.4.2.2. Second, for the lineage in the global store, the options are to scale up the capacity (e.g., by adding shards), force an application checkpoint, or fall back to a global rollback in case of failure. Third, there are the objects in the local in-memory store. This is unique to



(a) Deterministic.



(b) Nondeterministic, unlimited forwarding.



(c) Nondeterministic, forward up to 8 nodes.

Figure 3.10: Task latency for deterministic and nondeterministic applications, with lineage stash vs `WriteFirst`. A ring of 64 processes is instantiated, one on each node. Each process submits no-op tasks with a unique token to its successor. Task latency is the time before the process receives its token again divided by the number of processes. For Fig 3.10c, we forward an uncommitted task up to f=8 times.

the lineage stash because we decouple the object metadata (i.e., the lineage) from the object data. The options are similar: spill to external storage, force a checkpoint, or evict some objects and fall back to a global rollback.

3.5 Evaluation

We study the performance of the lineage stash compared to a `WriteFirst` method, which persists tasks to a global store before execution. We also evaluate the performance of the lineage stash on two end-to-end applications, distributed model training with ring allreduce and stream processing, and show that the lineage stash can provide faster recovery than a checkpoint-only solution with little to no additional runtime overhead. In summary, we study:

1. What is the latency overhead of the `WriteFirst` method compared to the lin-

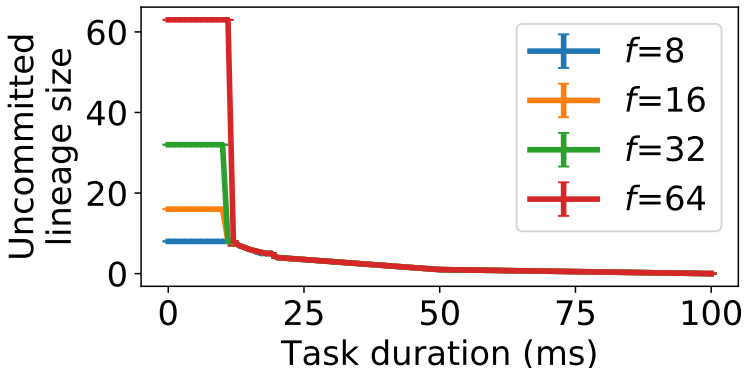


Figure 3.11: Median (and first and third quartiles) size of the forwarded uncommitted lineage, varying task duration for different values of f, the maximum number of concurrent failures tolerated. Above 10ms tasks, the uncommitted lineage size is stable.

eage stash?

2. How can an application maintain a stable amount of uncommitted lineage?

3. How does the lineage stash benefit data processing applications vs a global checkpoint-only approach?

We ran all experiments on Amazon EC2 (instance types inline). We implemented the lineage stash in 1k LoC (C++) on Ray [145], a low-latency system for distributed dynamic dataflows that normally uses `WriteFirst`. The recovery protocol for nondeterministic applications (§3.4.2.3) was implemented in an additional 125 LoC. For each Ray cluster, we used one non-replicated Redis instance per global store shard and one m5.8xlarge node separate from the workers to host the shards. In benchmarks that simulate global store write latency, we modified Ray to submit writes on a timer.

3.5.1 Microbenchmarks

Task latency distribution. We measure the latency of the lineage stash relative to `WriteFirst`. We also simulate global store latencies of +1 and +5ms. In Fig 3.10, each process in a ring of 64 processes simultaneously submits a no-op task to its successor in the ring with a unique token, and we measure task latency based on the round-trip time of each token. Because of the ring structure, every task's lineage includes nearly every other task executed so far.

Figure 3.10a shows the latency distribution for applications with deterministic lineage. While `WriteFirst` can achieve p50=0.96ms and p99=1.12ms latency, it suffers greatly with simulated delays of +1 and +5ms (maximum of ¿6ms). Meanwhile,

the lineage stash achieves p50=0.48ms and p99=0.58ms latency, even with +5ms of simulated delay. This is because each node only needs to remember uncommitted tasks that it submitted in its local stash (§3.3).

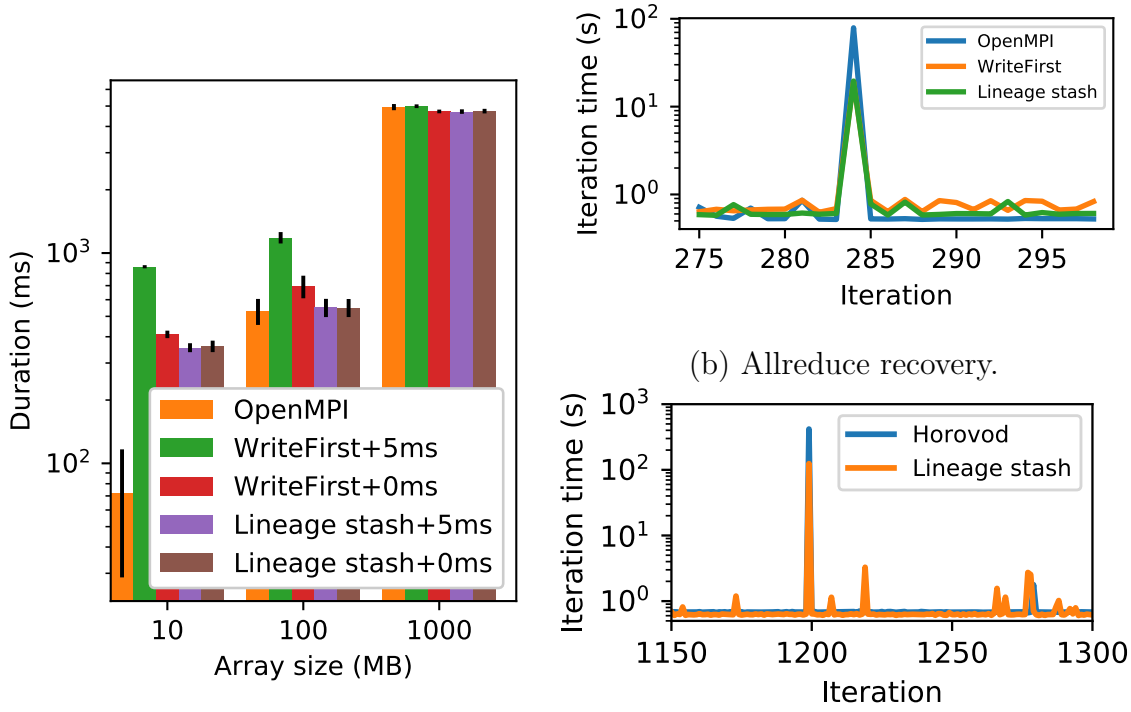
For nondeterministic applications, both systems flush each task a second time, before dispatch to the process, to record the execution order (§3.4.2). For the lineage stash, in Figures 3.10b and 3.10c, we forward uncommitted tasks infinitely many and up to 8 times, to tolerate up to f=64 (the number of nodes) or f=8 simultaneous failures, respectively.

As expected, logging execution order doubles `WriteFirst` latency compared to deterministic applications (p50=1.72ms at +0ms; p50=11.07ms at +5ms). For the lineage stash with f=64 (Fig 3.10b), the latency is much higher than for deterministic applications (p50=4ms and p99=11ms at +5ms). This is because every process has a path to every other process, which causes the uncommitted lineage to grow too large when the task duration is too short relative to the global store latency. Once we limit the number of times a task can be forwarded (Fig 3.10c), latency is stable. The lineage stash's p50 latency at +5ms delay is 0.70ms, 15× lower than `WriteFirst`'s at +5ms and lower even than `WriteFirst`'s at +0ms.

Uncommitted lineage. The amount of forwarded uncommitted lineage depends on: (1) the global store latency, (2) the task arrival rate, (3) f, the number of simultaneous failures tolerated, and (4) the application structure (§3.4.3). For instance, if a process submits one task every T seconds to another process and the global store latency is $10T$, then we expect each task to forward an average of 10 tasks.

In Fig 3.11, we vary task duration as a proxy for task arrival rate and report the forwarded lineage size, per submitted task. We also vary the maximum number of times an uncommitted task can be forwarded, to demonstrate how to cap the forwarded lineage at the cost of only tolerating f failures (§3.4.3). The workload is a ring of 64 nondeterministic processes as in Fig 3.10, with a simulated global store latency of 100ms. This communication structure is challenging for the lineage stash because each process has a path to every other process, so each task must be forwarded to f other nodes to tolerate f failures. Also, the global store should have much lower latency in practice, but we configure this to accurately show the effect of millisecond task durations.

Below task duration 11ms, the forwarded lineage in all cases grows unbounded and is capped only by f. This has consequences on the task latency: 61 forwarded tasks translates to 3.4ms latency, versus 1.1ms latency for 8.8 forwarded tasks. Interestingly, no matter the value of F, all configurations converge on 8-9 forwarded tasks at task duration 11ms. This suggests that for a given application structure and global store latency, there is a maximum task arrival rate under which the uncommitted lineage will remain stable.



(a) Allreduce latency.

(b) Allreduce recovery.

(c) SGD recovery.

Figure 3.12: **(a)** Allreduce duration on 64 workers (m5.2xlarge), averaged over 20 trials (with std. deviation). `WriteFirst` and the lineage stash use ring allreduce, with simulated global store latency as labeled. **(b)** Allreduce recovery time for lineage stash vs `WriteFirst` vs OpenMPI, with checkpoints to disk every 150 iterations. We kill and restart a worker at iteration 284. **(c)** Distributed SGD on the lineage stash vs Horovod v0.16.1, on 16 p3.8xlarge. Both use TensorFlow v1.12 on Resnet-101 with synthetic data and batch size 64. The lineage stash uses the same ring allreduce as in §3.5.2.1. Each worker checkpoints the model to disk every 640 iterations (˜7min). We kill and restart a worker at iteration 1200.

	Allreduce		Distributed SGD		Streaming WC	
	OpenMPI	LS	Horovod	LS	Flink	LS
Mean latency w/o failure	530	550	684	674	79	92
Mean latency during failure	79,012	19,557	417,655	124,296	8,869	435

Table 3.2: Summary of mean latencies in milliseconds during normal operation and during recovery for Ray with the lineage stash (LS) compared to baseline systems on a variety of applications. For latency during a failure in streaming (§3.5.2.3), we take the mean of all reported latencies between the failure time to when the latency for new inputs converges to normal operation. For the other applications, we report the maximum latency.

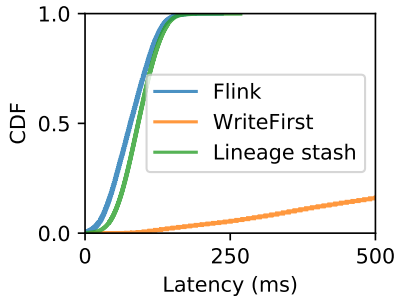
3.5.2 End-to-end Applications

3.5.2.1 Ring allreduce

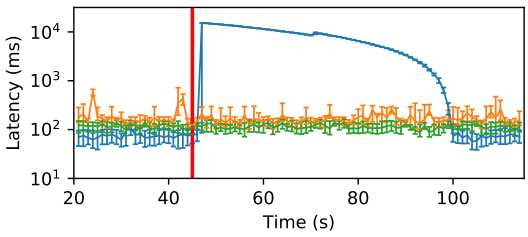
Allreduce is an important collective communication routine commonly used in high-performance computing in which all processes start with an input element and end with the reduced sum of the inputs. Ring allreduce is an implementation optimized for large arrays, in which a ring of P processes exchange inputs over $2(P-1)$ rounds of communication with P messages (tasks) each. The runtime of this algorithm is especially important for machine learning, where it is used in data-parallel synchronous distributed training to exchange gradients between copies of the model. Ring allreduce can be written as a deterministic application on the lineage stash. Also, because the application data is large, we cache all object data in Ray's per-node shared-memory store.

In Fig 3.12a, we compare the runtime of ring allreduce on the lineage stash against the same implementation but with `WriteFirst` and against OpenMPI v1.10 [86]. We show that the latency with the lineage stash is comparable to that of OpenMPI and consistently lower than `WriteFirst`. On 100MB arrays, the mean duration on the lineage stash is 550ms versus 530ms on OpenMPI. The lineage stash outperforms OpenMPI on 1GB arrays but is 5× worse on 10MB, in both cases possibly because of OpenMPI's use of a different allreduce algorithm. Meanwhile, the lineage stash is 1.26× faster than the `WriteFirst` method on 100MB. With a global store delay of 5ms, the lineage stash iteration time stays constant, since it is insensitive to global store latency, while the `WriteFirst` iteration time increases to 1184ms.

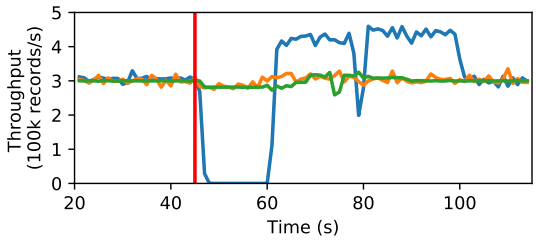
We also compare recovery in Fig 3.12b on an application that iteratively calls allreduce on a 100MB array on 64 workers. We checkpoint the allreduce data to



(a) Latency without failure.



(b) Latency during failure.



(c) Throughput during failure.

Figure 3.13: **(a)** Latency CDF for a streaming wordcount on 32 m5.xlarge workers at 400k records/s (4M words/s). Latency is sampled once every 1000 records. Both systems used a parallelism of 32 (per source, map, reduce, sink) and checkpoints to disk every 30s. **(b, c)** Failure and recovery for streaming wordcount on 32 m5.xlarge nodes at 300k records/s, checkpoints to disk every 30s. A worker is killed and restarted at t=˜45s (vertical red line), ˜15s after the first checkpoint. We report **(b)** median latencies seen by a single sink (with 1st and 3rd quartiles), x-axis is the record timestamp, and **(c)** total throughput, x-axis is physical time. The throughput drop at t=˜80s is due to checkpointing.

disk every 150 iterations (˜1min), kill and restart a node near iteration 280, and measure the time to recover all of the allreduce outputs since the last checkpoint. For OpenMPI, we restart the benchmark from the latest checkpoint on failure. For the lineage-based systems, the failed process retrieves all lost allreduce outputs since the last checkpoint from the remaining nodes' in-memory stores and replays the last allreduce iteration from the lineage. Figure 3.12b shows that the lineage stash (and `WriteFirst`) achieves 4× better recovery time than OpenMPI with only a small runtime overhead during normal operation (Table 3.2).

3.5.2.2 Distributed Training

Data-parallel distributed training is an increasingly important workload in which many copies of a model train on different batches of a dataset. In synchronous training, all workers iteratively compute a local gradient (in 100s of ms on GPUs),

sum gradients with allreduce, and apply the summed gradient to their model copy. Thus, fast allreduce is critical for distributed training throughput. While distributed training is often long-running, meaning that fast recovery may be less important than in an online application, we show that the lineage stash can provide faster recovery than state-of-the-art systems with no perceivable runtime overhead.

In Fig 3.12c, we compare distributed stochastic gradient descent (SGD) on Ray with the lineage stash vs Horovod v0.16 [178] (both with Tensorflow [18] v0.12) and show that we can achieve a similar mean iteration time of 674ms, compared to 684ms on Horovod, during normal operation (Table 3.2). Also, we show that we can recover from the failure at iteration 1200 in 124s, more than 3× faster than Horovod (417s). Approximately half of the lineage stash's recovery time is due to TensorFlow initialization, which could be reduced with a standby worker, while the rest is spent recovering and reapplying the lost gradients to the restored model.

3.5.2.3 Stream Processing

In this section, we measure the benefits of the lineage stash for an online stream processing workload. Stream processing at scale requires low-latency scheduling across many nodes. Since applications are also long-running, the chance of a failure is high, so reducing downtime during recovery is critical. Finally, these applications often interact with the external world, which in general cannot be rolled back, so exact replay is important for end-to-end exactly-once semantics.

We implement a streaming wordcount application on top of Ray with and without the lineage stash, with one long-running actor per mapper and reducer instance. Each actor batches records in the stream and submits one task per batch to a downstream actor. Mapper tasks compute over an input batch and contain only the lineage (no application data), while reducer tasks contain inlined task arguments. Reducers execute tasks *nondeterministically*, i.e. they process tasks from the mappers in order of arrival. This order is recorded in the lineage, as described in §3.4.1. To test the overhead of the lineage stash for nondeterministic processes, we record the latency for each reducer at a different node, so that the reducer must forward any uncommitted lineage to a remote node. We also implement asynchronous, globally consistent checkpointing, using the same algorithm as Flink [57].

Latency without failures. In Fig 3.13a, we show that the lineage stash on Ray can achieve similar latencies as Flink (v1.8.1) at a throughput of 400k records/s (4M words/s) on 32 nodes. The p50 and p90 latency for Flink is 79ms and 125ms, respectively, vs. 92ms and 132ms for the lineage stash. Meanwhile, `WriteFirst` cannot keep up with the target throughput because the global store is a bottleneck.

Recovery time. In Figures 3.13b and 3.13c, we run the same workload at 300k records/s and kill a worker ˜15s after the first checkpoint. For both systems, we

immediately restart the worker so that Flink has enough resources to continue.

For Flink, because the entire job must roll back and play forward again, new records are blocked by recovery and throughput drops to 0 (t=48-60s in Fig 3.13c). Once all lost work has been replayed, at t=61s in Fig 3.13c, the system can process new records that entered the stream during recovery. Because the system is overprovisioned for the target load, the system is able to use the extra capacity to eventually catch up to the input stream, returning to normal throughput at t=101s in Fig 3.13c. Note that the higher the expected load during normal operation, the more the system must be overprovisioned for failure, or else the system will never catch up with the input stream after recovery. The records that are processed during this period (t=48-100s in Fig 3.13b) all experience higher latency than normal (¿15s) since their processing was blocked by the global rollback (Table 3.2).

For Ray with and without the lineage stash, the failed node has one source, mapper, reducer, and sink, each of which is replayed after the failure. The mapper can skip most tasks during replay since it is stateless, but the reducer must recompute its state from its last checkpoint. While new records scheduled to the recovering operators are delayed by task replay, those scheduled to intact operators can be safely processed. Thus, the total throughput drops only slightly after the failure, to ˜280k records/s (t=48-65s in Fig 3.13c). Once the failed operators have finished re-execution, they process the new records (t=66-80s in Fig 3.13c). During this period, the total throughput increases (to ˜320k records/s), as in Flink, but much less additional capacity is needed. Also, although the maximum per-record latency is about the same as for Flink, since the maximum work replayed by any single process is the same, most of the record latencies during recovery (t=48-80s in Fig 3.13b) are actually the same as during normal operation, since they were not blocked by recovery (Table 3.2).

3.6 Related work

Message-passing systems. Because almost any distributed application could be logically viewed as a message-passing system [78], there are many framework examples, including parallel computing frameworks [86], distributed training frameworks [178, 18], low-latency data processing frameworks [146, 56], and actor frameworks [195, 54]. Out of these systems, the ones that provide explicit fault tolerance support [146, 56, 178, 18] use global checkpointing alone, most likely because this is the simplest to implement and understand and adds low and predictable runtime overhead. Previous work has studied techniques for asynchronous global checkpointing [60] that are optimize runtime overheads for particular applications, such as stream processing [56, 115]. However, in general, a global checkpoint-only approach introduces higher recovery overheads, as well as high runtime overhead when end-to-end exactly once semantics are needed [78], i.e., when outputting to the external world.

Causal logging [80, 27] is a general class of techniques in which processes log nondeterministic events asynchronously and piggyback volatile records onto messages to other processes. Potentially because of protocol complexity and difficulty in guaranteeing low runtime overhead in practice, causal logging is not used in any practical application that we are aware of. A primary difficulty in any logging approach, causal or otherwise, is that all possible sources of nondeterminism must be logged, which is complicated for a general application that can make system calls, share memory, etc. Our primary contribution is in identifying distributed data processing as a promising application for causal logging and describing how to efficiently capture the necessary nondeterministic events. We also present a system architecture for the stable log storage system that reflects the design of modern cloud storage systems, which are often highly available and horizontally scalable but guarantee only eventual consistency and do not promise low latency [59, 73, 61].

Lineage-based systems. MapReduce [72], Apache Hadoop [205], and Apache Spark [212] implement a *bulk synchronous parallel* model in which the user specifies data parallelism through a lineage graph of coarse-grained transformations that apply the same operation to each item in an arbitrarily sized dataset. A centralized scheduler then schedules tasks in each stage to execute over a data partition. For fault tolerance, the lineage is stored reliably at a centralized location, usually the scheduler, on the critical path of task execution. Drizzle [194] amortizes the scheduler overhead for applications where the lineage is known a priori, as in stream processing [213]. However, this does not solve the problems inherent to BSP systems, namely that the job must proceed in synchronous stages and each stage must be statically sized (e.g., the static microbatch size in Spark Streaming [213]).

CIEL [148], Ray [145], and Noria [89] are examples of lineage-based systems that support dynamic dataflows, but again with synchronous logging to a centralized location. More importantly, none of these systems support exact replay of nondeterministic execution. They target only computations that can be rolled back and replayed without side effects on the external world. Noria guarantees exactly-once semantics for client reads, but at the cost of rolling back and replaying all computation downstream of the failed node.

Transactional systems. Our work is closely related to logging and recovery in database systems, with a long history of work on "write-ahead logging" methods in which changes are durably logged before the transaction commits [141]. This technique is widely applicable and has also been used to reduce recovery time and guarantee exactly-once semantics for large-scale stream processing in MillWheel [21], at the cost of higher latency during execution. MillWheel writes all operator state and intermediate records to a persistent storage system [61] on the critical path of execution, while the lineage stash logs only the lineage to persistent storage and does

so off of the critical path.

There is a complementary line of work on data *provenance* or lineage in the database community. While this work has not yet impacted the design of recovery protocols, it has been used towards incremental view maintenance, such as in differential dataflow [138]. In the future, there may be opportunity to apply the lineage stash towards recovery in such systems as well.

3.7 Conclusion and Lessons Learned

We introduce the lineage stash, a causal logging technique for simultaneously achieving predictably low latency during normal execution and rapid recovery after a failure. While others [78, 194, 56] have shown that there is a fundamental tradeoff between these axes, we show here that the tradeoff need not affect the application. We achieve this by recording lineage off the critical path of task execution and replaying the lineage to reconstruct lost data after a failure. We evaluate the concept empirically on end-to-end applications in machine learning and stream processing, and show how the lineage stash enables large-scale, online data processing with fine-grained dynamic dataflows.

However, this chapter also serves to demonstrate the significant obstacles in providing efficient recovery for a generic, dynamic, and fine-grained interface such as distributed futures. While the lineage stash can provide efficient recovery, its fully decentralized design adds significant complexity and it does not fully address related problems in fault tolerance including: garbage collection and availability of recovery metadata, dealing with tasks with side effects in the external world, and conflict prevention during concurrent task failures and re-executions.

Thus, the key lessons learned are:

1. For distributed memory systems, fault tolerance must be designed holistically with other memory management operations such as garbage collection.

2. The recovery techniques needed for deterministic vs. nondeterministic applications vary greatly, to the point that they often necessitate disparate systems. Thus, providing recovery *flexibility* is in itself a substantial research challenge.

These lessons are incorporated in Chapters 4 and 6, respectively.

Chapter 4

Ownership: A Distributed Futures System for Fine-Grained Tasks

In the remainder of this thesis, we will turn our focus to the works overviewed in Figure 1.3. In the previous chapter, we attempted to provide transparent recovery, low run-time overhead, and low recovery overhead in a single general-purpose system. Here, we instead take a layered approach to address the fault tolerance needs of different classes of applications. These classes are based roughly on the execution semantics required, i.e. at-most-once, at-least-once, or exactly-once, as described in Section 2.6.2.

In this chapter, we will present *ownership*, an architecture for distributed futures that will serve as the system foundation for the following chapters. Following from this role, ownership will provide the strongest performance and the weakest recovery guarantees out of the systems presented in this thesis, as this promotes application generality. In particular, ownership provides an alternative method of decentralization compared to the lineage stash, and provides 1ms latency for tasks, faster and more accurate failure recovery, and greater scalability. To achieve these properties, ownership focuses on providing exactly-once semantics for applications with deterministic and idempotent tasks, and at-most-once or at-least-once semantics for all others.

4.1 Introduction

RPC is a standard for building distributed applications because of its generality and because its simple semantics yield high-performance implementations. The original proposal uses *synchronous* calls that *copy* return values back to the caller (Figure 4.2a). As discussed in Chapter 2, several recent systems [145, 148, 171, 12] have extended RPC so that, in addition to distributed communication, the system may

```
a_future = compute()
b_future = compute()
c_future = add(a_future, b_future)
c = system.get(c_future)
```

Figure 4.1: A distributed futures program. `compute` and `add` are stateless. `a_future`, `b_future`, and `c_future` are distributed futures.

also manage *data movement* and *parallelism* on behalf of the application.

Data movement. Pass-by-value semantics require all RPC arguments to be sent to the executor by copying them directly into the request body. Thus, performance degrades with *large* data. Data copying is both expensive and unnecessary in cases like Figure 4.2a, where a process executes an RPC over data that it previously returned to the same caller.

To reduce data copies, some RPC systems use *distributed memory* [153, 155, 76, 116, 148]. This allows large arguments to be passed by *reference* (Figure 4.2b), while small arguments can still be passed by value. In the best case, arguments passed by reference to an RPC do not need to be copied if they are already on the same node as the executor (Figure 4.2b). Note that, like traditional RPC, we make all values *immutable* to simplify the consistency model and implementation.

Parallelism. RPCs are traditionally *blocking*, so control is only returned to the caller once the reply is received (Figure 4.2a). *Futures* are a popular method for extending RPC with asynchrony [36, 129], allowing the system to execute functions in parallel with each other and the caller. With *composition* [129, 148], i.e., passing a future as an argument to another RPC, the application can also express the parallelism and dependencies of future RPCs. For example, in Figure 4.2c, `add` is invoked at the beginning of the program but only executed by the system once `a` and `b` are computed.

Distributed futures are an extension of RPC that combines futures with distributed memory: a *distributed future* is a reference whose eventual value may be stored on a remote node (Figure 4.2d). An application can then express distributed computation without having to specify when or where execution should occur and data should be moved. As discussed in Section 2.4, this is an increasingly popular interface for developing distributed applications that manipulate large amounts of data [148, 145, 12, 171].

As with traditional RPC, a key goal is generality. To achieve this, the system must minimize the overhead of each function call [44]. For example, the widely used

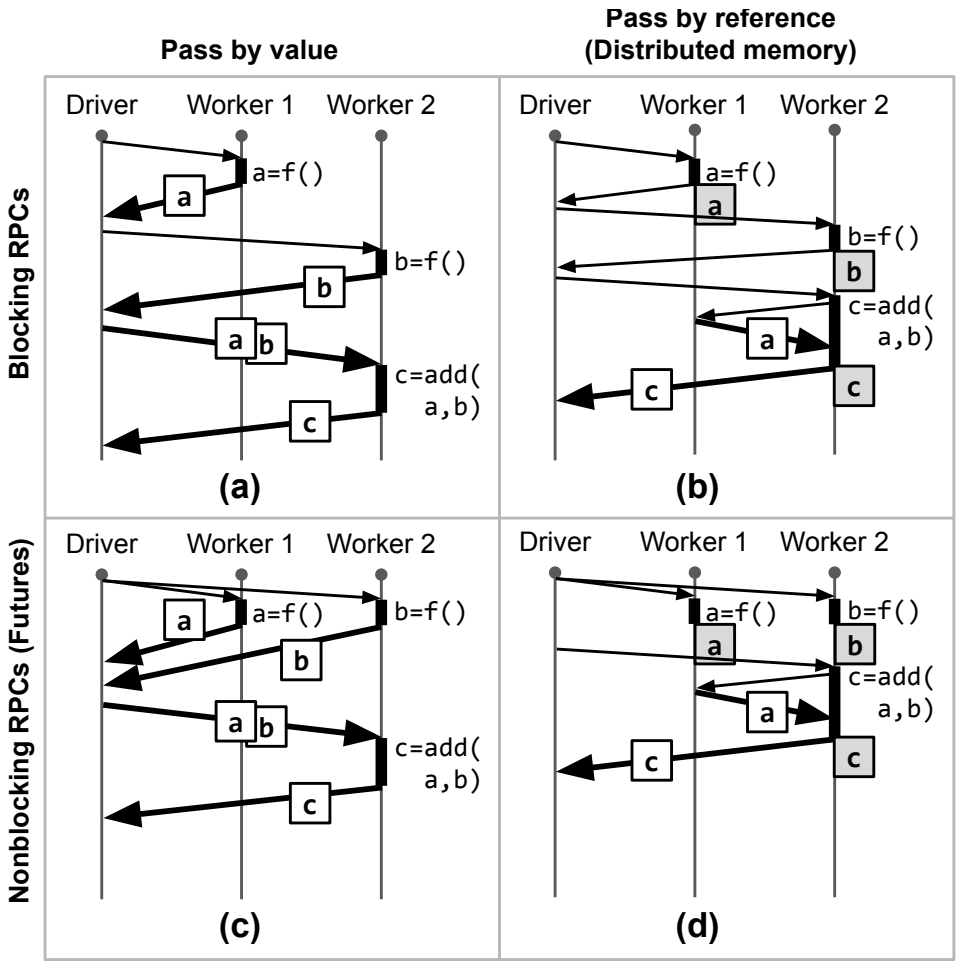


Figure 4.2: Example executions of the program from Figure 4.1. **(a)** With RPC. **(b)** With RPC and distributed memory, allowing the system to reduce data copies. **(c)** With RPC and futures, allowing the system to manage parallel execution. **(d)** With distributed futures.

gRPC provides horizontal scalability and sub-millisecond RPC latency, making it practical to execute millions of *fine-grained* functions, i.e. millisecond-level "tasks", per second [9].

Similarly, there are emerging examples of large-scale, fine-grained applications of distributed futures, including reinforcement learning [145], video processing [99, 161], and model serving [187]. These applications must optimize parallelism and data movement for performance [152, 161, 187], making distributed futures apt. Unfortunately, existing systems for distributed futures are limited to *coarse-grained* tasks [148].

In this chapter, we present a distributed futures system for fine-grained tasks. While others [145, 148, 171] have implemented distributed futures before, our contribution is in identifying and addressing the challenges of providing *fault tolerance for fine-grained tasks without sacrificing performance*.

In particular, the problem of interest is: *if a task holds a reference to a distributed future, can we guarantee that it will eventually be able to dereference the value?* This requires tracking the lifetime of each distributed future, i.e. determining when it is still in scope and how physical failures may have affected any reference holders and/or locations of the physical value. The challenge comes in doing this *efficiently*, in terms of both run-time and recovery overheads, and *in the presence of failures*.

Distributed futures make this problem more challenging than in traditional RPC because they introduce *shared state* between processes. In particular, an object and its metadata are shared by its reference holder(s), the RPC executor that creates the object, and its physical location(s). To ensure that each reference holder can dereference the value, the processes must coordinate, a difficult problem in the presence of failures. In contrast, traditional RPC has no shared state, since data is passed by value, and naturally avoids coordination. This property is critical to scalability and low latency in RPC systems.

For example, in the traditional RPC model shown in Figure 4.2a, once worker 1 copies `a` to the driver, it does not need to be involved in the execution of the downstream `add` task. In contrast, worker 1 stores `a` in Figure 4.2d, so the two workers must coordinate to ensure that `a` is available long enough for worker 2 to read. Also, worker 1 must garbage-collect `a` once worker 2 executes `add` and there are no other references. Finally, the processes must coordinate to detect and recover from the failure of another process.

The common solution in previous systems is to use a centralized master to store system state and coordinate these operations [148, 145]. A simple way to ensure fault tolerance is to record and replicate metadata at the master *synchronously* with the associated operation. For example, in Figure 4.2d, the master would record that `add` is scheduled to worker 2 *before* dispatching the task. Then, if worker 2 fails, it is simple to detect `c`'s failure; because the metadata write happened synchronously and at a centralized location, it is quick and trivial to decide that `c` was indeed executing on worker 2 when the failure occurred. However, this design adds significant overhead for applications with a high volume of fine-grained tasks.

Thus, decentralizing the system state is necessary for scalability. The question is how to do so without requiring coordination and undue complexity. The key insight in our work is to exploit the application structure: a distributed future may be shared by passing by reference, but *most distributed futures are shared within the scope of the caller*. For example, in Figure 4.1, `a_future` is created then passed to `add` in the same scope.

We thus propose *ownership*, a method of decentralizing system state across the RPC *executors*. In particular, the caller of a task is the *owner* of the returned future and all related metadata. In Figure 4.2d, the driver owns `a`, `b`, and `c`.

This solution has three advantages. First, for horizontal scalability, the application can use nested tasks to "shard" system state across the workers. Second, since a future's owner is the task's caller, task latency is low because the required metadata writes, though synchronous, are local. This is in contrast to an application-agnostic method of sharding, such as consistent hashing. Third, each worker becomes in effect a centralized master for the distributed futures that it owns. The owner coordinates all system operations such as reference counting, for memory safety, and lineage reconstruction, for recovery, for the futures that it owns. Thus, failure handling remains simple.

Of course, it is also possible that the owner itself fails. In this case, we must ensure that when recovering the owner, any remaining reference holders can eventually dereference the value. One option is to durably record the reference holders during execution and reconcile them with the recreated owner during recovery. However, this can add both run-time overhead and recovery complexity.

Instead, we rely on lineage reconstruction and a second key insight into the application structure: in many cases, the references to a distributed future are held by tasks that are descendants of the failed owner. The failed task can be recreated through lineage reconstruction by *its* owner, and the descendant tasks will also be recreated in the process. Therefore, if reference holders *fate-share* with the future's owner[1], we can achieve two desirable properties: a) while owner failures may cause other tasks to rollback, the tree structure ensures that the blast radius will be minimized, and b) during recovery, the new owner and descendant tasks will naturally be consistent with each other via the normal execution path, without any need for additional protocols. As we expect failures to be relatively rare, we argue that this reduction in system overheads and complexity outweighs the cost of additional re-execution upon a failure.

In summary, our contributions are:

- A decentralized system for distributed futures with transparent recovery and automatic memory management.
- A lightweight technique for transparent recovery based on lineage reconstruction

[1]Note that reference holders fate-share with their owner, but not vice versa.

and fate sharing.

- An implementation in the Ray system [145] that provides high throughput, low latency, and fast recovery.

4.2 Distributed Futures

4.2.1 API

The key benefit of distributed futures is that the system can transparently manage parallelism and data movement on behalf of the application. Here, we describe the API (Table 4.1).

To spawn a *task*, the caller invokes a *remote function* that immediately returns a `DFut` (Table 4.1). The spawned task comprises the function and its arguments, resource requirements, etc. The returned `DFut` refers to the *object* whose value will be returned by the function. The caller can *dereference* the `DFut` through `get`, a blocking call that returns a copy of the object. The caller can *delete* the `DFut`, removing it from scope and allowing the system to reclaim the value. Like other systems [148, 145, 171], all objects are *immutable*.

After the creation of a `DFut` through task invocation, the caller can create other references in two ways. First, the caller can pass the `DFut` as an argument to another task. `DFut` task arguments are implicitly dereferenced by the system. Thus, the task will only begin once all upstream tasks have finished, and the executor sees only the `DFut` *values*.

Second, the `DFut` can be passed or returned as a *first-class value* [98], i.e. passed to another task without dereferencing. Table 4.1 shows how to cast a `DFut` to a `SharedDFut`, so the system can differentiate when to dereference arguments. We call the process that receives the `DFut` a *borrower*, to differentiate it from the original caller. Like the original caller, a borrower may create other references by passing the `DFut` or casting again to a `SharedDFut` (creating further borrowers).

Like recent systems [145, 12, 171], we support *stateful* computation with actors. The caller creates an actor by invoking a *remote constructor function*. This immediately returns a reference to the actor (an `ARef`) and asynchronously executes the constructor on a remote process. The `ARef` can be used to spawn tasks bound to the same process. Similar to `DFuts`, `ARefs` are first-class, i.e. the caller may return or pass the `ARef` to another task, and the system automatically collects the actor process once all `ARefs` have gone out of scope.

4.2.2 Applications

Typical applications of distributed futures are those for whom performance requires the flexibility of RPC, as well as optimization of data movement and parallelism. We

Operation	Semantics
`f(DFut x) → DFut`	Invoke the remote procedure `f`, and pass `x` by reference. The system implicitly dereferences `x` to its `Value` before execution. Creates and returns a distributed future, whose value is returned by `f`.
`get(DFut x) → Value`	Dereference a distributed future. Blocks until the value is computed and local.
`del(DFut x)`	Delete a reference to a distributed future from the caller's scope. Must be called by the program.
`Actor.f(DFut x) → DFut`	Invoke a stateful remote procedure. `f` must execute on the actor referred to by `Actor`.
`shared(DFut x) → SharedDFut`	Returns a `SharedDFut` that can be used to pass `x` to another worker, without dereferencing the value.
`f(SharedDFut x) → DFut`	Passes `x` as a first-class `DFut`: The system dereferences `x` to the corresponding `DFut` instead of the `Value`.

Table 4.1: Distributed futures API. The full API also includes an actor creation call. A task may also return a `DFut` to its caller (nested `DFuts` are automatically flattened).

describe some examples here and evaluate them in Section 4.5.2.

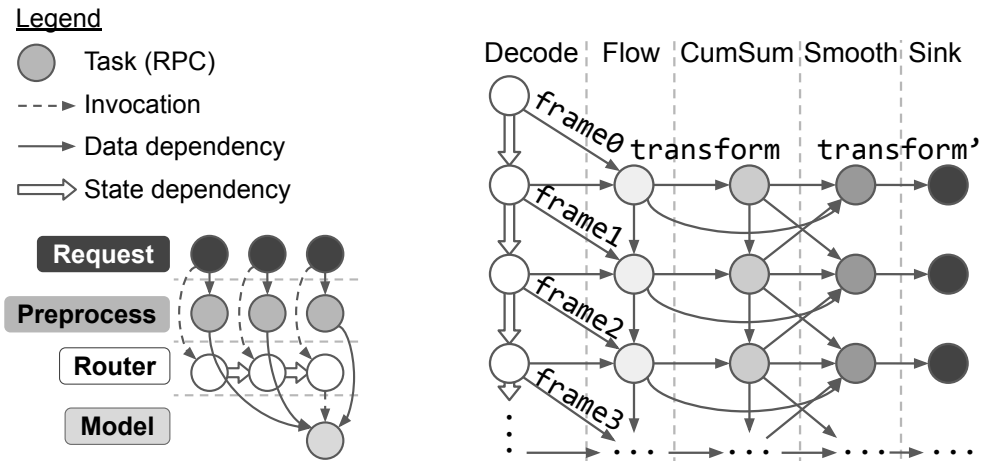
Distributed futures have previously been explored for data-intensive applications [148, 145]. Ciel identified the key ability to *dynamically* specify tasks during execution, e.g., based on previous results, rather than specify the entire graph upfront [148]. This makes it simple to express task-parallel algorithms, making distributed futures an especially good fit for iterative and recursive algorithms that contain data-dependent control flow.

Our goal is to expand the application scope to include those with *fine-grained* tasks that run in the milliseconds. We also explore the use of actors and first-class distributed futures.

Model serving. The goal is to reduce request latency while maximizing throughput, often by using model *replicas*. Depending on the model, a latency target might be 10-100ms [95]. Typically, an application-level scheduling policy is required, e.g., for staged rollout of new models [175].

Figure 4.3a shows an example of a GPU-based image classification pipeline. Each client passes its input image to a `Preprocess` task, e.g., for resizing, then shares the returned `DFut` with a `Router` actor. `Router` implements the scheduling policy and passes the `DFut` by reference to the chosen `Model` actor. `Router` then returns the results to the clients.

Actors improve performance in two ways: (1) each `Model` keeps weights warm



(a) Model serving (b) Video processing

Figure 4.3: Distributed futures applications.

in its local GPU memory, and (2) `Router` buffers the preprocessed `DFuts` until it has a batch of requests to pass to a `Model`, to leverage GPU parallelism for throughput. With *dynamic* tasks, the `Router` can also choose to flush its buffer on a timeout, to reduce latency from batching.

First-class distributed futures are important to reduce routing overhead. They allows the `Router` to pass the references of the preprocessed images to the `Model` actors, instead of copying these images. This avoids creating a bottleneck at the `Router`, which we evaluate in Figure 4.15a. While the application could use an intermediate storage system for preprocessed images, it would then have to manage additional concerns such as garbage collection and failures.

Online video processing. Video processing algorithms often have complex data dependencies. For example, video stabilization (Figure 4.3b) works by tracking objects between frames (Flow), taking a cumulative sum of these trajectories (CumSum), then applying a moving average (Smooth). Frame-to-frame dependencies are common, such as the video decoding state stored in an actor in Figure 4.3b. Each stage runs in 1-10s of milliseconds per frame.

Safe and timely garbage collection in this setting can be challenging because a single object (e.g., a video frame) may be referenced by multiple downstream tasks. Live video processing is also latency-sensitive: output must be produced at the same frame rate as the input. Low latency relies on pipeline parallelism between frames, as the application cannot afford to wait for multiple input frames to appear before beginning execution, and stateful computation, i.e. the video decoder state. Finally, for online applications, low downtime during a failure is an important requirement.

Data-parallel batch processing systems such as Apache Spark [212] can provide

fast recovery by only recovering the intermediate data that was lost during a failure, but they are not suited to efficient execution of video processing algorithms. Batch processing systems generally require stateless transforms, and transforms that cannot be expressed as one-to-one operations may require the intermediate dataset to be materialized. Thus, direct application of a batch processing system towards video processing would add significant overheads [99, 161]

In principle, these requirements can be met by data-parallel stream processing systems such as Apache Flink [56] of Naiad [146], which are well-suited to expressing the stateful and complex inter-frame dependencies found in video processing algorithms. However, simultaneously achieving low downtime is difficult. Most stream processing systems use global checkpointing for fault tolerance, which can lead to long pauses during recovery due to rollback.

With distributed futures, the application can specify the logical task graph dynamically, as input frames appear, making it simple to express inter-frame dependencies. Concurrent video streams can easily be supported using nested tasks, one "driver" per stream. Meanwhile, the system manages all aspects of the physical execution, including pipelining tasks, garbage collection of objects, and recovery. Previous work has shown that it is possible to achieve low latency with such a system [145]; what is missing is simultaneously achieving *fast recovery*.

4.3 Overview

4.3.1 Requirements

The system guarantees that each `DFut` can be *dereferenced* to its value. This involves three problems related to object lifetime tracking: automatic memory management, failure detection, and failure recovery.

The key question is where and when to record the system metadata needed for these operations such that the metadata is *consistent*[2] and *fault-tolerant*. By consistent, we mean that the system metadata matches the current physical state of the cluster. By fault-tolerant, we mean that the metadata should survive individual node failures.

We briefly describe the implications of distributed futures and failures on the design of these operations.

Automatic memory management is a system for dynamic memory allocation and reclamation of objects. The system must decide at run time whether an object is currently referenced by a live process, e.g., through reference counting [160].

[2]Unrelated to the popular definition of replica consistency [190].

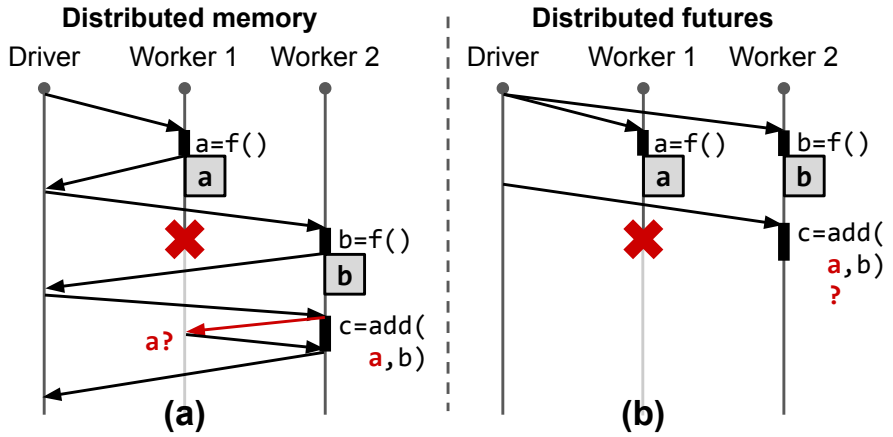


Figure 4.4: Failure detection. **(a)** `a`'s location is known by the time worker 2 receives the reference. **(b)** `a`'s location may not be known when worker 2 receives `add`, so worker 2 cannot detect the failure.

Failure detection is the minimum functionality needed to ensure progress in the presence of failures. The system detects when a `DFut` cannot be dereferenced due to worker failure.

With distributed memory but no futures, this is straightforward because *the location of the value is known by the time the reference is created.* In Figure 4.4a, for example, the driver learns that `a` is stored on worker 1 and could then attach the location when passing `a` to worker 2. Then, when worker 2 receives `add`, it can detect `a`'s failure.

The addition of futures complicates failure detection because references can be created *before* the value. Even the future *location* of the value may not be known at reference creation time. Of course, the system could wait until a task has been scheduled before returning the reference to the caller. However, this would defeat the purpose of futures as an asynchronous construct. It is also impractical because a realistic scheduler must be able to update its decision at run time, e.g., according to changes in the environment such as resource availability and worker failures.

Thus, it is possible that there are no locations for `a` when worker 2 receives the `add` RPC in Figure 4.4b. Then, worker 2 must decide whether `f` is still executing, or if it has failed. If it is the former, then worker 2 should wait. But if there is a failure, then the system must recover `a`. To solve this problem, the system must record the locations of all *tasks*, i.e. pending objects, in addition to created objects.

Failure recovery. The system must also provide a method of recovering from a failed `DFut`. The minimum requirement is to throw an error to the application if it tries to dereference a failed `DFut`. We further provide an option for *transparent* recovery, i.e. the system will recover a failed `DFut`'s value.

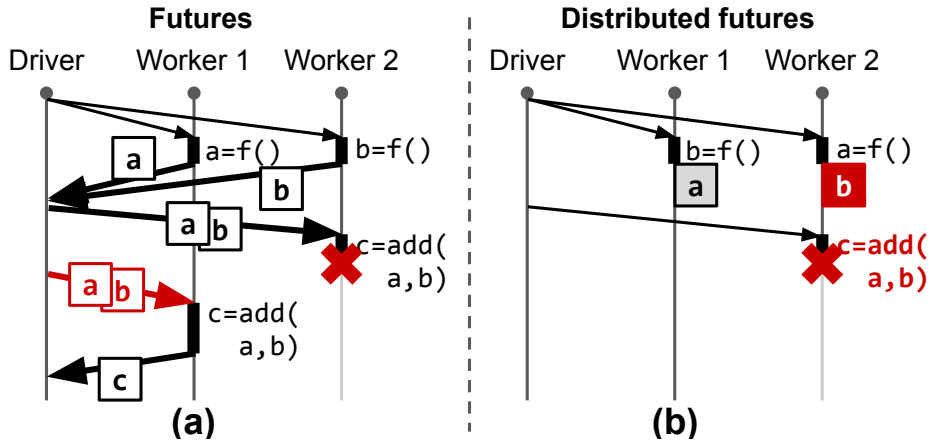


Figure 4.5: Failure recovery. **(a)** Data is passed by value, so the driver recovers by resubmitting `add`. **(b)** `b` is also lost. `f`'s description must be recorded during runtime so that `b` can be recomputed.

With futures but no distributed memory, if a process fails, then we will lose the reply of any pending task on that process. Assuming idempotence, this can be recovered through retries, a common approach for pass-by-value RPC. For example, in Figure 4.5a, the driver recovers by resubmitting `add(a,b)`. Failure recovery is simple because *all data is passed by value*.

With distributed memory, however, tasks can also contain arguments passed by *reference*. Therefore, a node failure can cause the loss of an object value that is still referenced, as `b` is in Figure 4.4b. A common approach to this problem is to record each object's *lineage*, or the subgraph that produced the object, during runtime [132, 212, 78]. The system then walks a lost object's lineage and recursively reconstructs the object and its dependencies through task re-execution. This approach reduces the runtime overhead of logging, since the data itself is not recorded, and the work that must be redone after a partial failure, since objects cached in distributed memory do not need to be recomputed. Still, achieving low run-time overhead is difficult because the lineage itself must be recorded and collected at run time and it must survive failures.

Note that we focus specifically on *object recovery* and, like previous systems [148, 145, 212], assume *idempotence* for correctness. Thus, our techniques are directly applicable to idempotent functions and actors with read-only, checkpointable, or transient state, as we evaluate in Figure 4.15c. Although it is not our focus, these techniques may also be used in conjunction with known recovery techniques for actor state [78, 145] such as recovery for nondeterministic execution [200].

Metadata requirements. In summary, during normal operation, the system must at minimum record (1) the location(s) of each object's value, so that reference holders

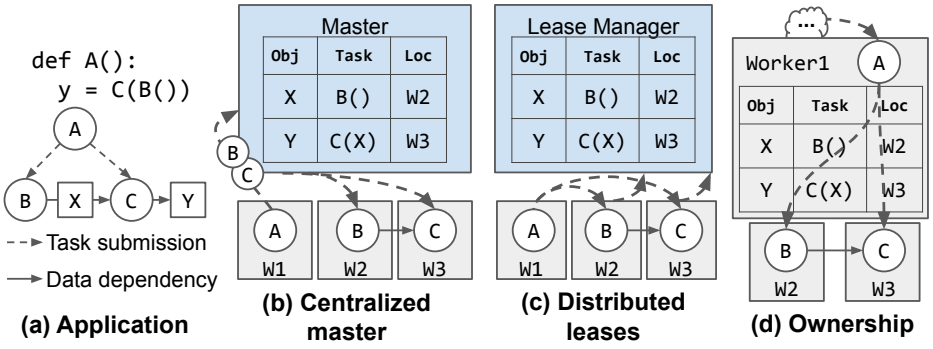


Figure 4.6: Distributed futures systems. **(a)** An application. **(b)** Master manages metadata and object failures. **(c)** Workers write metadata asynchronously, coordinate failure handling with leases. **(d)** Workers manage metadata. Worker 1 handles failures for workers 2 and 3. Worker 1 failure is handled by `A`'s owner elsewhere in the cluster.

can retrieve it, and (2) whether the object is still referenced, for safe garbage collection. For failure detection and recovery, the system must further record, respectively, (3) the location of each *pending* object, i.e. the task location, and (4) the object lineage.

We must decide both where, i.e. which processes, and when, i.e. before or asynchronous with, to record these operations. In some cases, it is safe for metadata to be *asynchronously updated*, i.e. there is a transient mismatch between the system metadata and the system state. For example, the system may transiently believe that an object `x` is still on node A even though it has been removed. This is safe because a reference holder can resolve the inconsistency by asking A if it has `x`.

On the other hand, metadata needed for failure handling should ideally be *synchronously updated*. For example, the metadata should never say that a task T is on node A when it is really on node B. In particular, if node A then fails, the system would incorrectly conclude that T has failed. As we will see next, synchrony simplifies fault tolerance but can add significant runtime overhead if done naively.

4.3.2 Existing solutions

Centralized master. Failure handling is simple with a synchronously updated centralized master, but this design can also add significant runtime overhead. For example, failure detection requires that the master record a task's scheduled location *before* dispatch (Figure 4.6b). Similarly, the master must record every new reference before it can be used. This makes the master a bottleneck for scalability and latency.

The master can be sharded for scalability, but this can complicate operations that coordinate multiple objects, such as garbage collection and lineage reconstruc-

tion. Also, the latency overhead is fundamental. Each task invocation must first contact the master, adding at minimum one round-trip to the critical path of execution, even without replicating the metadata for fault tolerance. This overhead can be detrimental when the task itself is milliseconds long, and especially so if the return value is small enough to be passed by value. Small values may be stored in the master directly as an optimization, but still require 1 RTT for retrieval [150].

Distributed leases. Decentralization can remove such bottlenecks, but often leads to complex coordination schemes. One approach is to use *distributed leases* [92]. This is similar to a centralized master that is updated *asynchronously*.

As an example, consider asynchronous task location updates (Figure 4.6c). To account for a possibly stale master, the worker nodes must coordinate to detect task failures, in this case using leases. Each worker node acquires a lease for each locally queued task and repeatedly renews the lease until the task has finished. For example, in Figure 4.6c, worker 3 can detect a failure of `B` by waiting for worker 2's lease to expire.

This design is horizontally scalable through sharding and reduces task latency, since metadata is written asynchronously. However, the reliance on timing to reconcile system state can slow recovery (Figure 4.14). Furthermore, this method of decentralization introduces a new problem: the workers must also coordinate on *who* should recover an object, i.e. re-execute the creating task. This is trivial in the centralized scheme, since the master coordinates all recovery operations.

4.3.3 Our solution: Ownership

The key insight in our work is to "shard" the centralized master, for scalability, but to do so based on the application structure, for low run-time overhead and simple failure handling. In ownership, the worker that calls a task stores the metadata related to the returned `DFut`. Like a centralized master, it coordinates operations such as task scheduling, to ensure it knows the task location, and garbage collection. For example, in Figure 4.6d, worker 1 owns `X` and `Y`.

The reason for choosing the task's caller as the owner is that in general, it is the worker that accesses the metadata most frequently. The caller is involved in the initial creation of the `DFut`, via task invocation, as well as the creation of other references, by passing the `DFut` to other RPCs. Thus, task invocation latency is minimal because the scheduled location is written locally. Similarly, if the `DFut` stays in the owner's scope, the overhead of garbage collection is low because the `DFut`'s reference count can be updated locally when the owner passes the `DFut` to another RPC. These overheads can be further reduced for small objects, which can be passed by value as if without distributed memory (see Section 4.4.2).

Of course, if all tasks are submitted by a single driver, as in BSP programs, ownership will not scale beyond the driver's throughput. Nor indeed will any sys-

tem for dynamic tasks. However, with ownership, the *application* can scale horizontally by distributing its control logic across multiple nested tasks, as opposed to an application-agnostic method such as consistent hashing (Figure 4.12e). Furthermore, the worker processes hold much of the system metadata. This is in contrast to previous solutions that push all metadata into the system's centralized or per-node processes, limiting the *vertical* scalability of a single node with many worker processes (Figure 4.12).

However, there are problems that are simpler to solve with a fully centralized design, assuming sufficient performance:

First-class futures. First-class futures (Section 4.2) allow non-owning processes to reference a `DFut`. While many applications can be written without first-class futures (Figure 4.3b), they are sometimes essential for performance. For example, the model serving application in Figure 4.3a uses first-class futures to delegate task invocation to a nested task, without having to dereference and copy the arguments.

A first-class `DFut` may leave the owner's scope, so we must account for this during garbage collection. We avoid centralizing the reference count at the owner, as this would defeat the purpose of delegation. Instead, we use a distributed hierarchical reference counting protocol (Section 4.4.2). Each borrower stores a local reference count for the `DFut` on behalf of the owner (Table 4.2) and notifies the owner when the local reference count reaches zero. The owner decides when the object is safe to reclaim. We use a reference counting approach as opposed to tracing [160] to avoid global pauses.

Owner recovery. If a worker fails, then we will also lose its owned metadata. For transparent recovery, the system must recover the worker's state on a new process and reassociate state related to the previously owned `DFuts`, including any copies of the value, reference holders, and pending tasks.

We choose a minimal approach that guarantees progress, at the potential cost of additional re-execution on a failure: we *fate share* the object and any reference holders with the owner, then use *lineage reconstruction* to recover the object and any of the owner's fate-shared children tasks (Section 4.4.3). This method adds minimal run-time overhead and is correct, i.e. the application will recover to a previous state and the system guarantees against resource leakage. A future extension is to persist the owner's state to minimize recovery time at the cost of additional recovery complexity and run-time overhead.

4.4 Ownership Design

Each node in the cluster hosts one to many workers (usually one per core), one scheduler, and one object store (Figure 4.7). These processes implement future resolution,

Field	Value
`*ID`	The `ObjectID`. Also used as a distributed memory key.
`*Owner`	Address of the owner (IP address, port, `WorkerID`).
`*Value`	(1) Empty if not yet computed, (2) Pointer if in distributed memory, or (3) Inlined value, for small objects (Section 4.4.2).
`*References`	A list of reference holders: Number of dependent tasks and a list of borrower addresses (Section 4.4.2 and Appendix A.1).
`Task`	Specification for the creating task. Includes the `ObjectIDs` and `Owners` of any `DFuts` passed as arguments.
`Locations`	If `Value` is empty, the location of the task. If `Value` is a pointer to distributed memory, then the locations of the object.

Table 4.2: Ownership table. The owner stores all fields. A borrower (Section 4.3.2) only stores fields indicated by the *.

resource management, and distributed memory, respectively. Each node and worker process is assigned a unique ID.

Workers are responsible for the resolution, reference counting, and failure handling of distributed futures. Each worker executes one task at a time and can invoke other tasks. The root task is executed by the "driver".

Each task has a unique `TaskID` that is a hash of the parent task's ID and the number of tasks invoked by the parent task so far. The root `TaskID` is assigned randomly. Each task may return multiple objects, each of which is assigned an `ObjectID` that concatenates the `TaskID` and the object's index. A `DFut` is a tuple of the `ObjectID` and the owner's address (`Owner`).

The worker stores one record per future that it has in scope in its local *ownership table* (Table 4.2). A `DFut` borrower records a subset of these fields (* in Table 4.2). When a `DFut` is passed as an argument to a task, the system implicitly resolves the future's value, and the executing worker stores only the `ID`, `Owner`, and `Value` for the task duration. The worker also caches the owner's stored `Locations`.

An actor is a stateful task that can be invoked multiple times. Like objects, an actor is created through task invocation and *owned* by the caller. The ownership table is also used to locate and manage actors: the `Location` is the actor's address. Like a `DFut`, an `ARef` (an actor reference) is a tuple of the `ID` and `Owner` and can be passed as a first-class value to other tasks.

A worker requests resources from the scheduling layer to determine task placement (Section 4.4.1). We assume a decentralized scheduler for scalability: each scheduler manages local resources, can serve requests from remote workers, and can redirect a worker to a remote scheduler.

The distributed memory layer (Section 4.4.2) consists of an immutable distributed object store (Figure 4.7d) with `Locations` stored at the owner. The `Locations` are updated asynchronously. The object store uses shared memory to reduce

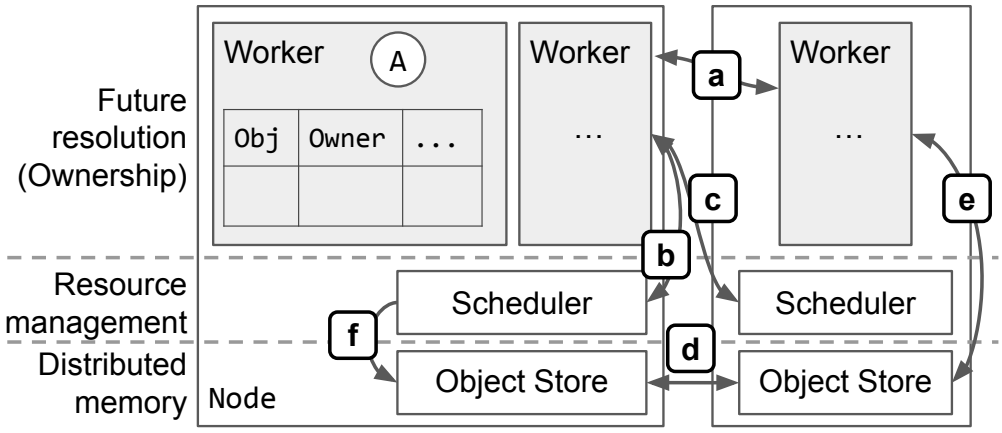


Figure 4.7: Architecture and protocol overview. **(a)** Task execution. **(b)** Local task scheduling. **(c)** Remote task scheduling. **(d)** Object transfer. **(e)** Task output storage and input retrieval. Ownership layer manages distributed memory garbage collection and recovery. **(f)** Scheduler fetches objects in distributed memory to fulfill task dependencies.

copies between reference holders on the same node.

Workers store, retrieve, reclaim, and recover large objects in distributed memory (Figure 4.7f). The scheduling layer sends requests to distributed memory to fetch objects between nodes according to worker requests (Figure 4.7g).

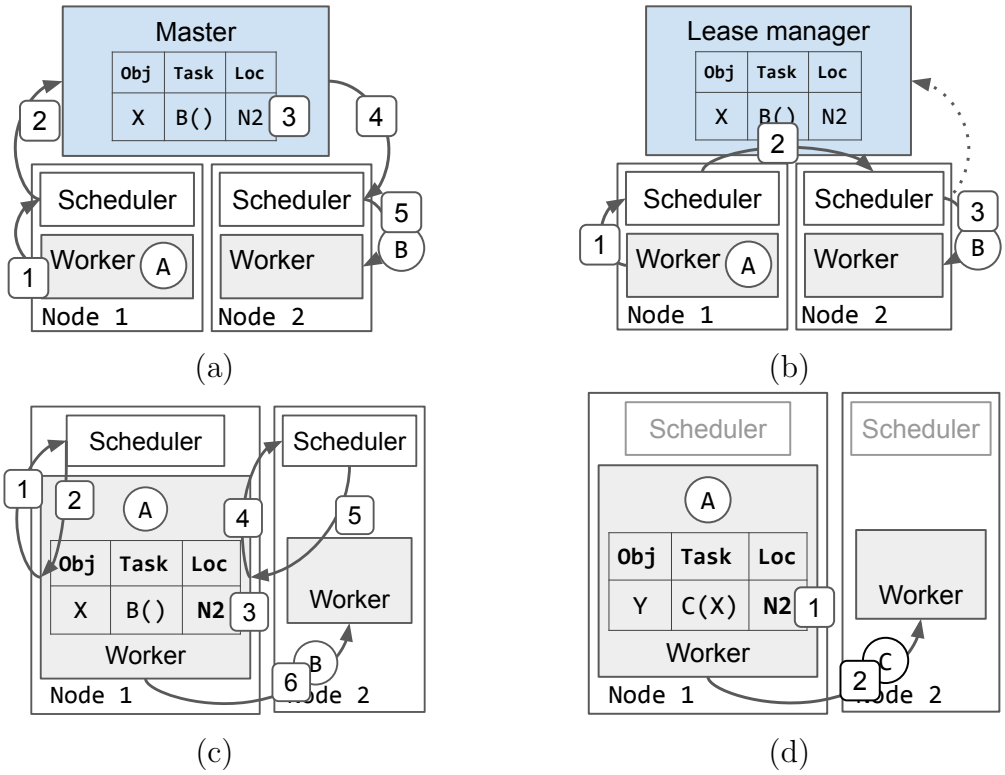
4.4.1 Task scheduling

We describe how the owner coordinates task scheduling. At a high level, the owner dispatches each task to a location chosen by the distributed scheduler. This ensures that the task location in the ownership table is updated synchronously with dispatch. We assume an abstract scheduling policy that takes in resource requests and returns the ID of a node where the resources should be allocated. The policy may also update its decision, e.g., due to changes in resource availability.

Figure 4.8c shows the protocol to dispatch a task. Upon task invocation, the caller, i.e. the owner of the returned `DFut`, first requests resources from its local scheduler[3]. The request is a tuple of the task's required resources (e.g., `{"CPU": 1}`) and arguments in distributed memory. If the policy chooses the local node, the scheduler accepts the request: it fetches the arguments, allocates the resources, then *leases* a local worker to the owner. Else, the scheduler rejects the request and redirects the owner to the node chosen by the policy.

In both cases, the scheduler responds to the owner with the new location: either the ID of the leased worker or the ID of another node. The owner stores this new

[3]The owner can also choose a remote scheduler, e.g., for data locality.



	Node	Local RTTs	Remote RTTs
Master	Local	1	1
	Remote	1	1
Leases	Local	1	0
	Remote	1	0.5
Owner-ship	Local	0.5 or 1.5	0
	Remote	0 or 1	0.5 or 1.5

(e)

Figure 4.8: Task scheduling and the method of recording a task's location for the program in Figure 4.6a. **(a)** Centralized master. **(b)** Distributed leases. **(c)** Scheduling with ownership. (1-2) Local scheduler redirects owner to node 2. (3) Update task location. (4-5) Remote scheduler grants worker lease. (6) Task dispatch. **(d)** Direct scheduling by the owner, using the worker and resources leased from node 2 in (c). **(e)** Length of critical path of local and remote task execution, in terms of local and remote RTTs.

location in its local ownership table before dispatching the task to that location. If the request was granted, the owner sends the task directly to the leased worker for execution; otherwise, it repeats the protocol at the next scheduler.

Thus, the owner always dispatches the task to its next location, ensuring that the task's pending `Location` (Table 4.2) is synchronously updated. This also allows the owner to *bypass* the scheduler by dispatching a task directly to an already leased worker, if the task's resource requirements are met. For example, in Figure 4.8d, worker 1 reuses the resources leased from node 2 in Figure 4.8c to execute `C`. The owner returns the lease after a configurable expiration time, or when it has no more tasks to dispatch. We currently do not reuse resources for tasks with different distributed memory dependencies, since these are fetched by the scheduler. We leave other policies for lease revocation and worker reuse for future work.

The *worst-case* number of RTTs before a task executes is higher than in previous solutions because each policy decision is returned to the owner (Figure 4.8e). However, the throughput of previous solutions is limited (Figure 4.12) because they cannot support direct worker-to-worker scheduling (Figure 4.8d). This is because workers do not store system state, and thus all tasks must be routed through the master or per-node scheduler to update the task location (Figures 4.8a and 4.8b).

Actor scheduling. The system schedules actor constructor tasks much like normal tasks. After completion, however, the owner holds the worker's lease until the actor is no longer referenced (Section 4.4.2) and the worker can only execute actor tasks submitted through a corresponding `ARef`.

A caller requests the actor's location from the owner using the `ARef`'s `Owner` field. The location can be cached and requested again if the actor restarts (Section 4.4.3). The caller can then dispatch tasks directly to the actor, as in Figure 4.8d, since the resources are leased for the actor's lifetime. For a given caller, the actor executes tasks in the order submitted.

4.4.2 Memory management

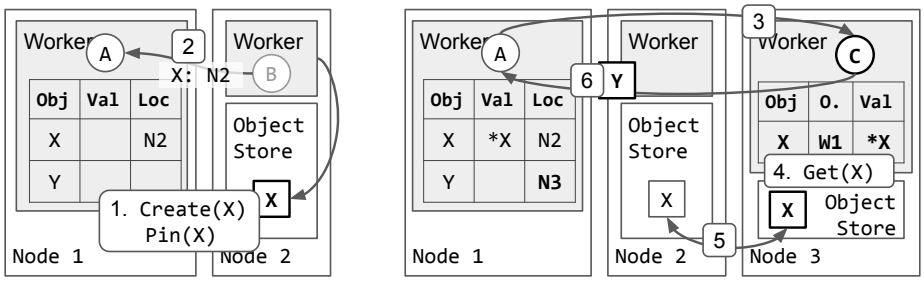
Allocation. The distributed memory layer consists of a set of object store nodes, with locations stored at the owner (Figures 4.9b to 4.9d). It exposes a key-value interface (Figure 4.9a). The object store may replicate objects for efficiency but is not required to handle recovery: if there are no copies of an object, a `Get` call will block until a client (i.e. a worker) `Creates` the object.

Small objects may be faster to copy than to pass through distributed memory, which requires updating the object directory, fetching the object from a remote node, etc. Thus, at object creation time, the system transparently chooses based on size whether to pass by value or by reference.

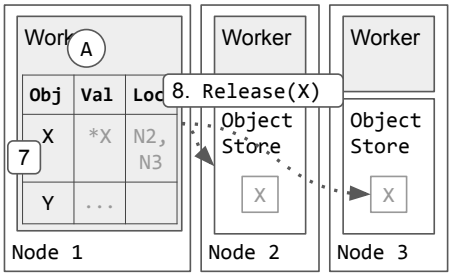
Objects over a configurable threshold are stored in the distributed object store (step 1, Figure 4.9b) and returned by reference to the owner (step 2). This reduces the

Operation	Semantics
`Create(ObjID o, Value v)`	Store an object.
`Pin(ObjID o, NodeID loc)→ bool`	Pin `o` on `loc` until released. Returns false if `loc` failed.
`Release(ObjID o)`	Object `o` is safe to evict.
`Get(ObjID o)→ Value`	Get the object value. May fetch copy from remote node.

(a)



(b) (c)



(d)

Figure 4.9: **(a)** Distributed memory store API, and **(b-d)** Memory management for the program in Figure 4.6a. (1-2) `B` returns a large object `X` in distributed memory. The primary copy is pinned until all references have been deleted. (3) Worker 1 dispatches `C` once `X` is available. (4-5) `Get` the value from distributed memory (location lookup not shown). (6) `C` returns a small object `Y` directly to the owner. (7-8) Object reclamation.

total number of copies, at the cost of requiring at least one IPC to the distributed object store for `Get` (steps 4-5, Figure 4.9c). Small objects are returned by value to the owner (step 6, Figure 4.9c), and each reference holder is given its own copy. This produces more copies in return for faster dereferencing.

The initial copy of a large object is known as the *primary*. This copy is pinned (step 1, Figure 4.9b) until the owner releases the object (step 8, Figure 4.9d) or fails. This allows the object store to treat additional capacity as an LRU cache without having to consult the owners about which objects are safe to evict. For example, the *secondary* copy of `X` created on node 3 in Figure 4.9c is cached to reduce `Get` and recovery time (Section 4.4.3) but can be evicted under memory pressure.

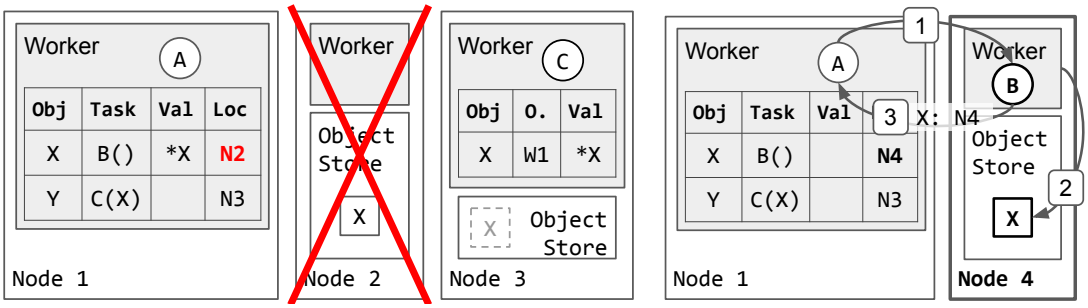
Dereferencing. The system dereferences a task's `DFut` arguments before execution. The task's caller first waits for the `Value` field in its local ownership table to be populated (Figure 4.9b), then copies the `Value` into the dispatched task description. The executing worker then copies the received `Value` into its local table (Figure 4.9c). For large objects, the sent value is a pointer to distributed memory, so the worker must also call `Get` to retrieve the actual value (step 4, Figure 4.9c).

If the task's caller is also the owner of its `DFut` arguments, the above protocol is sufficient. If the task's caller is *borrowing* an argument, then it must populate the `Value` field through a protocol with the owner. Upon receiving a `DFut`, the borrower sends the associated `Owner` a request for the `Value`. The owner replies with the `Value` (either the inlined value or a pointer) once populated. The borrower populates its local `Value` field by copying the reply.

Reclamation. The owner reclaims the object memory once there are no more reference holders (Figure 4.9d) by deleting its local `Value` field (step 7) and, if necessary, calling `Release` on the distributed object store (step 8). An object's reference holders are tracked with a distributed reference count maintained by the owner and borrowers.

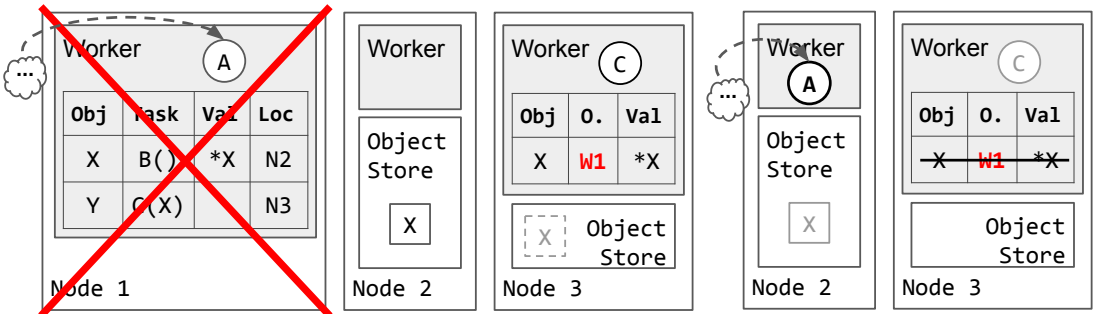
Each process with a `DFut` instance keeps a local count of submitted tasks (`References`, Table 4.2). The task count is incremented each time the process invokes a dependent task and decremented when the task completes. Each process also keeps a local set of the worker IDs of any borrowers that it created, by passing the `DFut` as a first-class value. This forms a tree of borrowers with the owner at the root (see Appendix A.1). The owner releases the object once there are no more submitted tasks or borrowers anywhere in the cluster.

Actors. Actors are reference-counted with the same protocol used to track borrowers of a `DFut`. Once the set of reference holders is empty, the owner of the actor reclaims the actor resources by returning the worker lease (Section 4.4.1).



(a) Failure detection. (b) Lineage reconstruction.

Figure 4.10: Object recovery.



(a) Failure detection. (b) Fate sharing.

Figure 4.11: Owner recovery.

4.4.3 Failure recovery

The system guarantees that any reference holder will eventually be able to resolve the value in the presence of failures.

Failure detection. Failure notifications containing a worker or node ID are published to all workers. Workers do not exchange heartbeats; a worker failure is published by its local scheduler. Node failure is detected by exchanging heartbeats between nodes, and all workers fate-share with their node.

Upon receiving a node or worker failure notification, each worker scans its local ownership table to detect a `DFut` failure. A `DFut` is considered failed in two cases: 1) loss of an owned object (Figure 4.10a), by comparing the `Location` field, or 2) loss of an owner (Figure 4.11a), by comparing the `Owner` field. We discuss the handling for these two cases next, using lineage reconstruction and fate sharing, respectively.

Note that a non-owner does not need to detect the loss of an object. For example, in Figure 4.10a, node 2 fails just as worker 3 receives `C`. When worker 3 looks up `X` at the owner, it may not find any locations. From worker 3's perspective, this means that either node 2's write to the directory was delayed, or node 2 failed. Worker 3

does not need to decide which it is; it simply waits for `X`'s owner to handle the failure.

Object recovery. The owner recovers a lost value through lineage reconstruction. During execution, the owner records the object's lineage by storing each invoked `Task` in its ownership table (Table 4.2). Then, upon detecting a `DFut` failure, the owner resubmits the corresponding task (Figure 4.10b). The task's arguments are recursively reconstructed, if needed.

Like previous systems [212, 148, 145], we can avoid lineage reconstruction if other copies of a required object still exist. Thus, when reconstructing an object, the owner will first try to locate and designate a secondary copy as the new primary. To increase the odds of finding a secondary copy, object reclamation (Section 4.4.2) is done lazily: the owner releases the primary copy once there are no more reference holders, but the copy is not evicted until there is memory pressure.

Often, the owner of an object will also own the objects in its lineage (Section 4.5.2). Thus, upon failure, the owner can locally determine the set of tasks to resubmit, with a recursive lookup of the `Task` fields. In some cases, an object's lineage may also contain *borrowed* references. Then, the borrower requests reconstruction from the owner.

The owner can delete the `Task` field once the task has finished and all objects returned by reference will never be reconstructed again. When a worker returns an object by value, the owner can immediately delete the corresponding `Task` field. This is safe because objects passed by value do not require reconstruction (Section 4.3.1).

For an object passed by reference, the owner keeps a *lineage reference count* to determine when to collect the `Task`. The count is incremented each time the `DFut` is passed to another task and decremented when that `Task` is itself collected. The owner collects a record after collecting both the `Task` and `Value` (Section 4.4.2) fields. We also plan to support object checkpointing to allow the lineage to be collected early.

Owner recovery. An owner failure can result in a "dangling pointer": a `DFut` that cannot be dereferenced. This can happen if the object is simultaneously lost from distributed memory. For example, `C` in Figure 4.11a will hang if node 2 also fails.

We use *fate sharing* to ensure that the system can make progress upon an owner's failure. First, all resources held by the owner and any reference holders are reclaimed. Specifically, upon notification of the owner's failure, either the distributed object store frees the object (if it exists) or the scheduling layer reclaims the worker lease (if the object is pending), shown in Figure 4.11b. All reference holders, i.e. borrowers and dependent tasks, also fate-share with the owner.

Then, to recover the fate-shared state, we rely on *lineage reconstruction*. In particular, the task or actor that was executing on the failed owner must itself have been owned by another process. That process will eventually resubmit the failed task. As the new owner re-executes, it will recreate its previous state, with no system

intervention needed. For example, the owner of `A` in Figure 4.11a will eventually resubmit `A` (Figure 4.11b), which will again submit `B` and `C`.

For correctness, we show that all previous reference holders are recreated, with the address of the new owner. Consider task T that computes the value of a `DFut` x. T initially executes on worker W and re-executes on W' during recovery. The API (Section 4.2) gives three ways to create another reference to x: (1) pass x as a task argument, (2) cast x to a `SharedDFut` then pass as a task argument, and (3) return x from T.

In the two former cases, the new reference holder must be a child task of T. In case (2), when x is passed as a first-class value, the child task can create additional reference holders by passing x again. All such reference holders are therefore descendants of T. Then, when T re-executes on W', W' will recreate T's descendants.

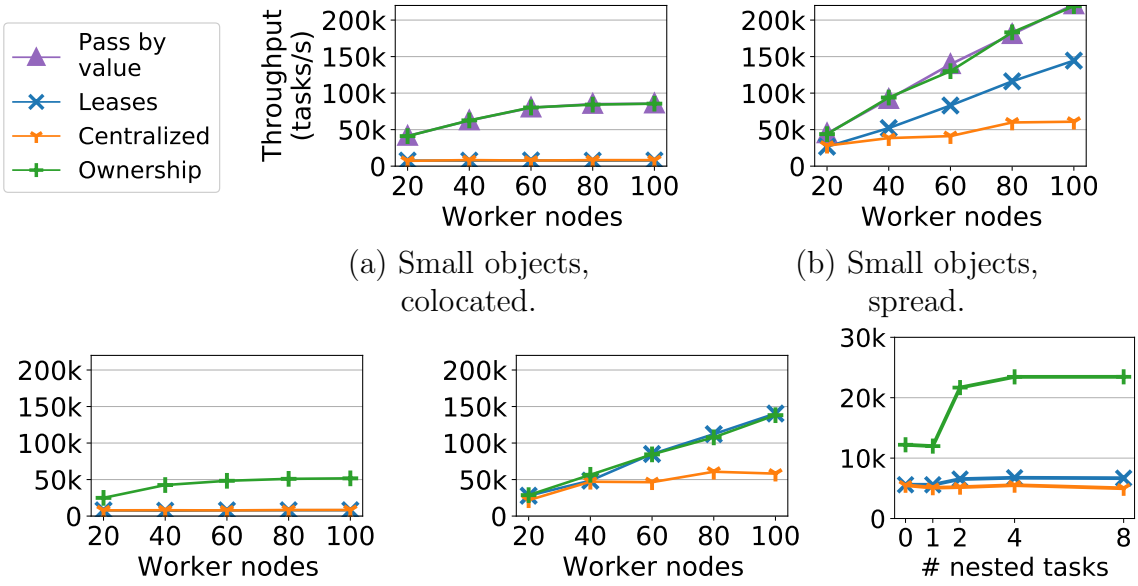
T can also return x, which can be useful for returning a child task's result without dereferencing with `get`. Suppose T returns x to its parent task P. Then, P's worker becomes a borrower and will fate-share with W. In this case, P is recovered by *its* owner, and again submits T and receives x.

Thus, because any borrower of `x` must be a child or ancestor of T, fate-sharing and re-execution guarantees that the borrower will be recreated with W' as the new owner. Note that for actors, this requires that an actor not store borrowed `DFuts` in its local state. Of course, this is only required for transparent recovery; the application may also choose to handle failures manually and rely on the system for failure detection only.

While fate-sharing and lineage reconstruction add minimal run-time overhead, the combination is not suitable for all applications. In particular, the application will fate-share with the driver. In fact, this is the same failure model offered by some BSP systems [10], which can be written as a distributed futures program in which the driver submits all tasks. As shown by these systems, this approach can be extended to reduce the re-execution needed during recovery. We leave such extensions, including application-level checkpointing (Section 4.5.2), and persistence of the ownership table, for future work.

Actor recovery. Actor recovery is handled through the same protocols. If an actor fails, its owner restarts the actor through lineage reconstruction, i.e. resubmitting the constructor task. If the owner fails, the actor and any `ARef` holders fate-share.

Unlike functions, actors have local state that may require recovery. This is out of scope for this work, but is an interesting future direction. Ownership provides the infrastructure to manage and restart actors, while other methods can be layered on top for transparent recovery of local state [78, 145, 200].



(a) Small objects, colocated.

(b) Small objects, spread.

(c) Large objects, colocated.

(d) Large objects, spread.

(e) Single node, nested tasks.

Figure 4.12: Throughput and scalability. **(a-d)** Task submission is divided across multiple intermediate drivers, either colocated on the m5.8xlarge head node or spread with one m5.8xlarge node per driver. 1 intermediate driver is added per 5 worker nodes. Each task returns either a small (short binary string) or large (1MB blob) object. **(e)** Scaling task submission using nested tasks and first-class distributed futures.

4.5 Evaluation

We study the following questions:

1. Under what scenarios is distributed futures beneficial compared to pass-by-value RPC?
2. How does the ownership architecture compare against existing solutions for distributed futures, in terms of throughput, latency, and recovery time?
3. What benefits does ownership provide for applications with dynamic, fine-grained parallelism?

We compare against three baselines: (1) a pass-by-value model with futures but no distributed memory, similar to Figure 4.2c, (2) a decentralized lease-based system for distributed futures (Ray v0.7), and (3) a centralized master for distributed futures (Ray v0.7 modified to write to a centralized master before task execution). All distributed futures systems use sharded, unreplicated Redis for the global metadata store, with asynchronous requests. All systems use the Ray distributed scheduler and (where applicable) distributed object store. Ownership and pass-by-value use gRPC [9] for worker-to-worker communication. All benchmarks schedule tasks to predetermined nodes to reduce scheduling variation.

All experiments are run on AWS EC2. Global system metadata, such as an object directory, is hosted on the same node as the driver, where applicable. Unless stated otherwise, this "head node" is an m5.16xlarge instance. Other node configuration is listed inline. All benchmark code is available at [202].

4.5.1 Microbenchmarks

Throughput and scalability. The driver submits one nested task for every 5 worker nodes (m5.8xlarge). Each intermediate "driver" submits no-op tasks to its 5 worker nodes. We report the total throughput of the leaf tasks, which return either a short string (Figures 4.12a and 4.12b) or a 1MB blob (Figures 4.12c and 4.12d). The drivers are either colocated (Figures 4.12a and 4.12c) on the same m5.8xlarge node as the root driver, or spread (Figures 4.12b and 4.12d), each on its own m5.8xlarge node. We could not produce stable results for pass-by-value with large objects due to the lack of backpressure in our implementation.

At <60 nodes, the centralized and lease-based architectures achieve about the same throughput because the centralized master is not yet a bottleneck. In general, ownership achieves better throughput than either because it distributes some system operations to the workers. In contrast, the baselines handle all system operations in the global or per-node processes.

The gap between ownership and the baselines is more significant with small return values (Figures 4.12a and 4.12b). For these, ownership matches pass-by-value because small objects are returned directly to their owner. The baseline systems

could implement a similar optimization, e.g., by inlining small objects in the object directory (Section 4.4.2), but this would still require at minimum one RPC per read.

When the drivers are spread (Figures 4.12b and 4.12d), ownership and leases both scale linearly. Ownership scales better than leases in Figure 4.12b because more work is offloaded onto the worker processes. Ownership and leases achieve similar throughput in Figure 4.12d, but the ownership system also includes memory safety (Section 4.4.2). The centralized design (2 shards) scales linearly to ∼60 nodes. Adding more shards would raise this threshold, but only by a constant amount.

When the drivers are colocated (Figures 4.12a and 4.12c), both baselines flatline because of a centralized bottleneck: the scheduler on the drivers' node. Ownership also shows this, but there is less scheduler load overall because the drivers reuse resources for multiple tasks (Section 4.4.1). A comparable optimization for the baselines would require each driver to batch task submission, at the cost of latency. Throughput for ownership is lower in Figure 4.12c than in Figure 4.12a due to the overhead of garbage collection.

Thus, because ownership decentralizes system state among the workers, it can achieve vertical (Figures 4.12a and 4.12c) and horizontal (Figures 4.12b and 4.12d) scalability. Also, it matches the performance of pass-by-value RPC while enabling new workloads through distributed memory (Section 4.2.2).

Scaling through borrowing. We show how first-class futures enable delegation. Figure 4.12e shows the task throughput for an application that submits 100K no-op tasks that each depend on the same 1MB object created by the driver. The tasks are submitted either by the driver (x=0) or by a number of nested tasks that each *borrow* a reference to the driver's object. All workers are colocated on an m5.16xlarge node.

For all systems, the throughput with a single borrower (x=1) is about the same as when the driver submits all tasks directly (x=0). Distributing task submission across multiple borrowers results in a 2× improvement for ownership and negligible improvement for the baselines. Thus, with ownership, an application can scale past the task dispatch throughput of a single worker by *delegating* to nested tasks. This is due to (1) support for first-class distributed futures, and (2) the hierarchical distributed reference counting protocol, which distributes an object's reference count among its borrowers instead of centralizing it at the owner (Section 4.4.2). In contrast, the baselines would require additional nodes to scale.

Latency. Figure 4.13 measures task latency with a single worker, hosted either on the same node as the driver ("local"), or on a separate m5.16xlarge node ("remote"). The driver submits 3k tasks that each take the same 1MB object as an argument and that immediately returns a short string. We report the average duration before each task starts execution.

First, distributed memory achieves better latency than pass-by-value in all cases

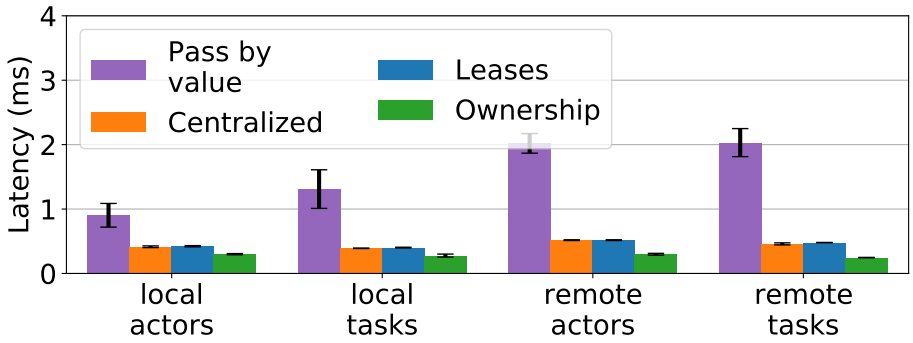
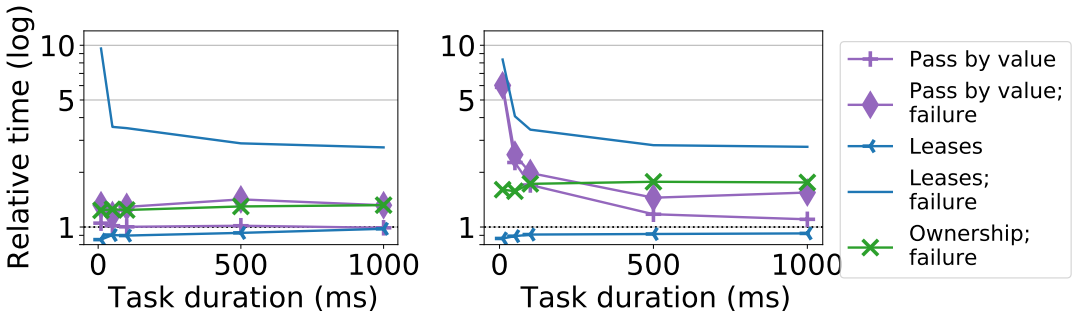


Figure 4.13: Task latency. Local means that the worker and driver are on the same node. Error bars for standard deviation (across 3k tasks).



(a) Small objects. (b) Large objects.

Figure 4.14: Total run time (log-scale), relative to ownership without failures. The application is a chain of dependent tasks that execute on one node. Each task sleeps for the duration on the x-axis (total 10s) and returns either **(a)** a short binary string, or **(b)** a 10MB blob.

because these systems avoid unnecessary copies of the task argument from the driver to the worker.

Second, compared to centralized and leases, ownership achieves on average 1.6× lower latency. This is due to (1) the ability to write metadata locally at the owner instead of a remote process, and (2) the ability to reuse leased resources, in many cases bypassing the scheduling layer (Section 4.4.1).

Recovery. This benchmark submits a chain of tasks that execute on a remote m5.xlarge node. Each task depends on the previous, sleeps for the time on the x-axis (total duration 10s), and returns either a short binary string (Figure 4.14a) or a 10MB blob (Figure 4.14b). We report the run time relative to ownership without failures. To test recovery, the worker node is killed and restarted 5s into the job (1s heartbeat timeout). We do not include centralized due to implementation effort.

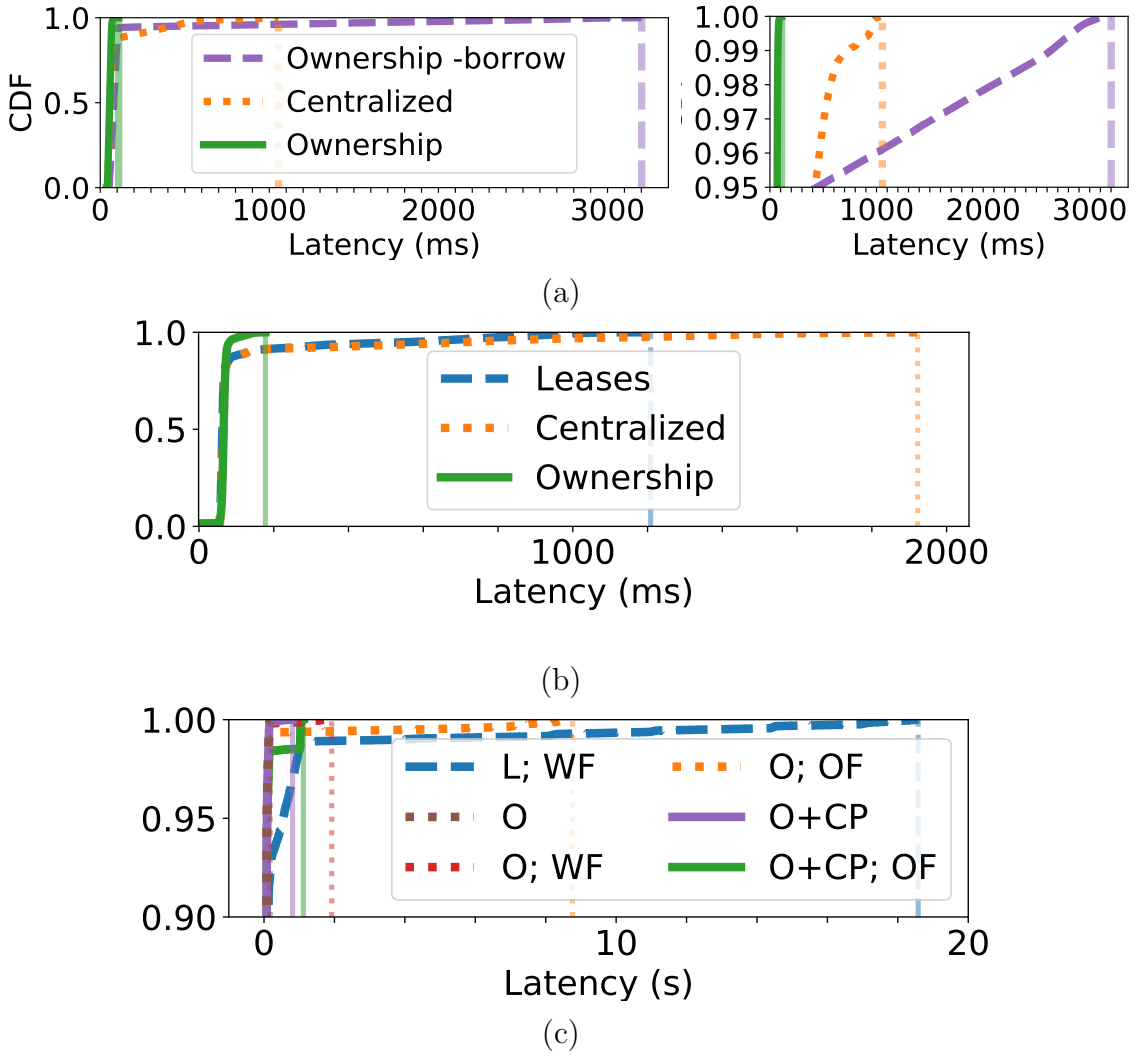


Figure 4.15: End-to-end benchmarks. **(a)** Image classification latency (right is p95-p100). **(b)** Online video stabilization latency. **(c)** Online video stabilization latency with failures (starting at p90). L=leases; O=ownership; CP=checkpointing; WF=worker failure; OF=owner failure.

Normal run time for leases is up to 1.18× faster than ownership, but recovery time is more than double, worse than restarting the application. This is because a task's lease must expire before it can be re-executed, adding delay for short tasks. The recovery delay for longer tasks is also high because the implementation (Ray v0.7) repeatedly doubles a lease's expiration time to reduce renewal overhead. A shorter lease interval would reduce recovery delay but can be unstable.

Ownership recovers within 2× the normal run time. Recovery time is the same as pass-by-value for small objects because only in-flight tasks are re-executed (Figure 4.14a). For large objects (Figure 4.14b), ownership achieves better normal run time than pass-by-value because arguments are passed by reference; the gap decreases as task execution dominates.

Thus, ownership can achieve the same or better normal run-time performance as leases *and* pass-by-value, while also guaranteeing timely recovery through lineage reconstruction.

4.5.2 End-to-end applications

Model serving. We implement Figure 4.3a. Figure 4.15a shows the latency on 4 p3.16xlarge nodes, each with 1 `Router` and 8 ResNet-50 [100] `Models`. We use a GPU batch size of 16 and generate 2300 requests/s. Ownership and centralized achieve the same median latency (54ms), but the tail latency for centralized is 9× higher (1s vs. 108ms). We also show the utility of first-class distributed futures: in "-borrow", the `Router` receives the image *values* and must copy these to the `Model`. As expected, the `Router` is a bottleneck (p50=80ms, p100=3.2s).

Online video processing. We implement Figure 4.3b with 60 concurrent videos. The tasks for each stream are executed on an m5.xlarge "worker" node (1 per stream) and submitted by a driver task on a separate m5.xlarge "owner" node. Each owner node hosts 4 drivers. Each video source uses an actor to hold frame-to-frame decoder state. However, tasks are idempotent: a previous frame may be reread with some latency penalty. We use a YouTube video with a frame rate of 29 frames/s and a radius of 1s for the moving average.

Figure 4.15b shows latency without failures. All systems achieve similar median latency (∼65ms), but leases and centralized have a long tail (1208ms and 1923ms, respectively). Figure 4.15c shows latency during an injected failure, 5s after the start, of the `Decoder` actor (Figure 4.3b). Lease-based recovery is slow because the decoder actor must replay all tasks, and each task accumulates overhead from lease expiration. Checkpointing the actor was infeasible because the leases implementation does not safely garbage-collect lineage.

Figure 4.15c also shows different failure scenarios for ownership, with a failure after 10s. The owner uses lineage reconstruction to recover quickly from a worker failure (1.9s in O;WF). Owner recovery is slower because the failed owner must re-execute

from the beginning (8.8s in O;OF). To bound re-execution, we use application-level checkpoints (O+CP, checkpoints to a remote Redis instance once per second). Each checkpoint includes all intermediate state needed to transform the given frame, such as the cumulative sum so far (Figure 4.3b). When the sink receives the transformed frame, it "commits" the checkpoint by writing the frame's index to Redis. This results in negligible overhead (O vs. O+CP) and faster recovery (1.1s in O+CP;OF).

4.6 Related Work

Distributed futures. Several systems [148, 171, 145, 12, 186, 200] have implemented a distributed futures model. Most [148, 171] use a centralized master (Section 4.3.2). In contrast, ownership is a decentralized design that stores system state directly in the workers that invoke the tasks. Ray [145] shards the centralized state, but must still write to the centralized store *before* task execution and does not support automatic memory management. Lineage stash [200] is a complementary technique for recovering nondeterministic execution; ownership provides infrastructure for failure detection and memory management.

Other dataflow systems. Distributed data-parallel systems provide high-throughput batch computation and transparent data recovery [72, 205, 108, 212]. Many of our techniques build on these systems, in particular the use of distributed memory [212, 108] and lineage re-execution [72, 205, 108, 212]. Indeed, a data-parallel program is equivalent to a distributed futures program with no nested functions.

Most distributed data-parallel systems [72, 205, 108, 212] employ some form of centralized master, a bottleneck for applications with *fine-grained* tasks [194, 164, 135]. Naiad [146, 147] and Canary [164] support fine-grained tasks but, like other data-parallel systems, implement a *static* task graph, i.e. all tasks must be specified upfront. In contrast, distributed futures are an extension of RPC, which allows tasks to be dynamically invoked. Nimbus [135] supports both fine-grained *and* dynamic tasks with a centralized controller by leveraging *execution templates* for iterative computations. In contrast, ownership distributes the control plane and schedules tasks one at a time. These approaches are complementary; an interesting future direction is to apply execution templates to distributed futures.

Actor systems. Distributed futures are compatible with the actor model [104, 31]. Other actor frameworks [42, 2] already use futures for asynchrony, but with pass-by-value semantics, making it expensive to process large data. Actors can be extended with distributed memory to enable pass-by-reference semantics. Since distributed memory is immutable, it does not violate the condition of no shared state.

Our fault tolerance model is inspired by *supervision* in actor systems [31]. In this model, a supervisor actor delegates work to its children actors and is responsible

for handling any failures among its children. By default, an actor also fate-shares with its supervisor. Our contribution is in extending the supervision model, which focuses on actor-level recovery, to *task- and object-level* recovery.

Parallel programming systems. MPI [86] exposes a low-level pass-by-value interface. In contrast, distributed futures supports pass-by-reference and heterogenerous processes.

Distributed futures are more similar in interface to other parallel programming runtimes [45, 40, 184, 133, 98]: the user annotates a sequential program to designate procedures that can be executed in parallel. Out of these systems, ownership is perhaps most similar to Legion [40], in that the developer specifies a task hierarchy that dictates system behavior. Our contribution is in identifying and addressing the challenges of failure detection and recovery for distributed futures.

Distributed memory. Distributed shared memory [153] provides the illusion of a single globally shared and *mutable* address space across a physically distributed system. Transparency has historically been difficult to achieve without adding exorbitant runtime overhead. Mutability makes consistency a major problem [153, 41, 111, 123], and fault tolerance has never been satisfactorily addressed [153].

More recent distributed memory systems [155, 76, 37, 116, 28] implement a higher-level key-value store interface. Most target a combination of performance, consistency, and durability. Similar to our use of distributed memory (Section 4.4.2), in-memory data replicas are used to improve durability and recovery time. Indeed, many of these systems could likely be used in place of our distributed memory subsystem.

However, the requirements of our distributed memory subsystem are minimal compared to previous work, e.g., durability is only an optimization. This is because we target an even higher-level interface that integrates directly with the programming language: unlike a key, a `DFut` is a first-class reference that can be used to express rich application semantics to the system, such as an RPC's data dependencies. Also, like previous data processing systems [72, 212, 148], data is immutable. Thus, fine-grained mutations are expensive, but consistency is not a problem.

4.7 Discussion

4.7.1 Programming languages

The ownership architecture for distributed futures is directly inspired by the concept of *ownership types* [66] from programming languages, popularized by languages such as Rust [17]. A key application of ownership types is in automatic memory safety: by only allowing a single owner for an object at a time, it becomes feasible at compile

time to guarantee that, at run time, objects will only be reclaimed when they go out of scope. Similarly, we apply the concept of having a single owner towards automatic *distributed* memory safety: by only allowing a single owner for an object at a time, we can guarantee with minimal coordination that objects will only be reclaimed when they are out of scope in the cluster. Thus, our work can also be considered an extension of ownership types to the distributed setting, studying how ownership relates to distributed systems design and how failures affect the problem of memory safety.

There are two key differences in our work: 1) ownership cannot be transferred, and 2) we use distributed reference counting to reclaim objects instead of inserting reclamation calls at compile time. These were practical decisions, and neither is fundamental.

Supporting ownership transfer would help mitigate unnecessary fate-sharing, but so far proved to be unnecessary for the evaluated applications. It would be important, however, for passing data across application boundaries while keeping failure domains isolated from each other. An interesting possibility in the distributed setting that does not appear in the language runtime setting is that of a "global" owner that is highly available and lives forever. This can be useful, for example, in cases where the user wants to manually decide when to delete the objects, e.g.., when storing persistent datasets.

The primary reason for choosing distributed reference counting is that it works for both interpreted and compiled languages. Because the primary frontend language in Ray is Python, distributed reference counting is a good fit. Supporting compile-time ownership checks in the frontend language is an interesting avenue for future work: it could be used, for example, to avoid distributed reference counting, to support ownership transfer while minimizing protocol complexity, and potentially even to enforce stronger fault tolerance guarantees. As an example of possible fault tolerance guarantees, ownership type checking could inform the system of the visibility of a particular result; this can be an important bit of information when deciding how to safely rollback computation after a failure. We have begun to explore these types of guarantees in Chapter 6, but a true compiler for a distributed futures language could unlock many more possibilities.

4.7.2 Impact on Ray

Ownership is the basis of the Ray architecture in v1.0+ [14], implemented in ∼14k C++ LoC. Previously, Ray used a sharded global metadata store [145]. There were two problems with this approach: (1) latency, and (2) worker nodes still had to coordinate for operations such as failure detection. Ray v0.7 introduced leases (Section 4.3.2) and an experimental version of the lineage stash that is described in Chapter 3, which solved the latency problem but not coordination. It became impractical to introduce additional distributed protocols needed for operations such as garbage collection. We designed ownership as a simpler and more efficient system for both

garbage collection and failure detection, and eventually it replaced Ray's previous control plane entirely.

4.8 Conclusion and Lessons learned

At first glance, the fault tolerance properties that ownership provides seem weaker than those described earlier in [145] and Chapter 3: it does not handle automatic state recovery for actors and when owners fail, it can cause other tasks to exit as well. Critically, however, ownership provides efficient and reliable failure detection and garbage collection, both necessary features. It also provides transparent object recovery as an option for pure functions, meaning that only applications that need such an option pay the cost.

This shows the importance of designing holistically for necessary fault tolerance features first. Meanwhile, failure masking and faster recovery should be considered optimizations, albeit important ones. Because of this, the ownership-based architecture turned out to be more general than previous versions of Ray, more suited to production use, and longer-lasting. Thus, one of the key lessons learned from Ray's history is:

1. The end-to-end principle as applied to fault tolerance: Carefully reducing the provided recovery guarantees can paradoxically produce a more powerful system.

The concept of fate-sharing in the ownership system is an excellent example of the end-to-end principle as applied to fault tolerance. When deciding how to handle failed owners, there is an alternative option of recording the children during execution, and reconciling the recreated owner with the remaining children after a failure. However, this brings up both protocol complexity (e.g. guaranteeing that there are no inconsistencies when multiple processes fail simultaneously) as well as limitations on the application. In particular, a recreated owner would have to deterministically recreate its children tasks, whereas in the ownership architecture, it is safe for the recreated owner to create a completely different task subtree, as long as its *final* task output is deterministic. Thus, by "giving up" a feature, in this case improved failure resilience, we gain much more in return.

The key idea behind fate-sharing and the overall ownership architecture is that *the design is based on the structure of distributed futures applications*, i.e. the tree formed by nested function calls. In contrast, previous versions of Ray were in a sense designed for a totally flat application structure: any process could talk to any other process, and all processes were equal. For example, the design described in [145] is completely decentralized, with all worker processes performing the same role. This design is general and fits the traditional message-passing model [78], and yet it is unnecessary here, as we know that the application is not flat and that most processes

will only communicate with a handful of other processes. Meanwhile, the flat design also leads to coordination overhead for garbage collection and failure detection. The ownership architecture shows that we can avoid such problems by using a different decentralization scheme:

2. Decentralized systems do not imply decentralized decision making. Structuring the system according to the structure of the application can simplify the system.

Finally, one concept that is visited both here and in the previous chapter is the idea that after failures, *only outputs that are visible to others need special recovery handling*. In this work, we leverage this to handle owner failures, by requiring the children to fate-share, effectively rolling back the lost outputs' visibility. In the lineage stash work, each process stashes the lineage that is visible to it, i.e. upstream in the lineage DAG. Thus, when considering how to provide transparent recovery for an application, it is crucial to consider what outputs the application has and who may view them when. This is a lesson that we formalize and leverage further in Chapter 6:

3. During failures, outputs that are not visible to others are safe to roll back. Rolling back is often simpler and improves run-time overhead, but it generally increases recovery time.

In the following chapters, we build upon the ownership work. Chapter 5 studies one category of applications that benefits directly from the transparent recovery provided by ownership. Meanwhile, for more complex end applications, we note that achieving efficient run-time and recovery may require significant application information and higher-level functionality. We study this idea in Chapter 6, where we build upon ownership to also support applications with impure functions.

Chapter 5

Exoshuffle: An Extensible Shuffle Architecture

One of the primary goals of the distributed futures interface is to factor out distributed execution and memory management from data-intensive cluster computing frameworks, thus reducing system development effort and converting previously siloed frameworks into libraries. In this chapter, we extend and build upon the ownership architecture to show how this can be done for one class of distributed applications: MapReduce [72] applications.

The key primitive in MapReduce applications is *shuffle*, the all-to-all data transfer from mappers to reducers. We choose to focus on this primitive because it is one of the most expensive communication primitives in distributed data processing and is difficult to scale. This is evidenced by the many prior works on addressing shuffle scalability challenges, all of which are built into a monolithic shuffle system [167, 217, 182, 34].

In this chapter, we present Exoshuffle, a library for distributed shuffle that offers competitive performance and scalability as well as greater flexibility than monolithic shuffle systems. Exoshuffle does so by decoupling the shuffle control plane from the shuffle data plane. Exoshuffle libraries implement the shuffle control plane, defining a logical shuffle schedule for tasks and data movement, and depend on the ownership architecture as their data plane, which handles data movement, recovery, and pipelining with execution. Here, we also extend the ownership architecture presented in the previous chapter to include support for out-of-core data processing via object spilling. With these components, we are able to (1) rewrite previous shuffle optimizations as application-level libraries with an order of magnitude less code, (2) achieve shuffle performance and scalability competitive with monolithic shuffle systems, and break the CloudSort record as the world's most cost-efficient sorting system, and (3) enable new applications such as ML training to easily leverage scalable shuffle.

This chapter is based on the published work [130] and includes significant con-

tributions of the coauthors Frank Sifei Luan et al.

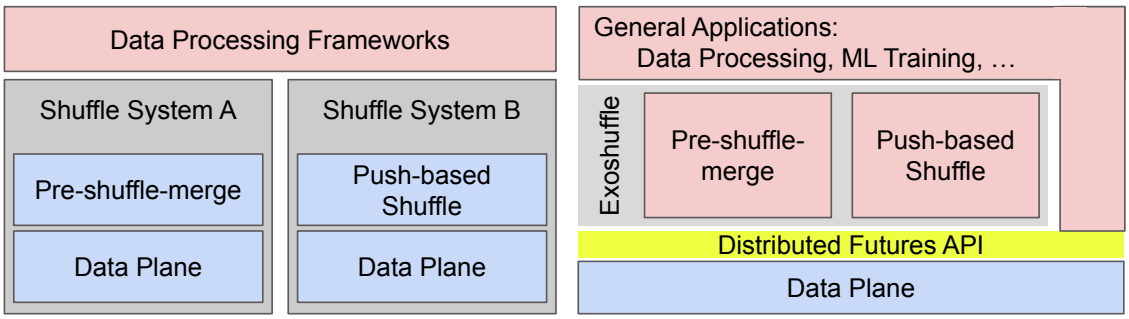
5.1 Introduction

Shuffle is a fundamental operation in distributed data processing systems. It refers to the all-to-all data transfer from mappers to reducers in a MapReduce-like system [72]. Shuffle is one of the most expensive communication primitives in these systems and is difficult to scale. Scaling shuffle requires efficiently and reliably moving a large number of small blocks from each mapper to each reducer across memory, disk, and network. It requires both high I/O efficiency, and robustness to failures and data skew. Furthermore, as the data size increases, the number of shuffle blocks grows quadratically, making shuffle the most costly operation in some workloads.

The difficulty of scaling shuffle has inspired many solutions from both the industry and research community. These shuffle implementations improve the performance and reliability of large-scale shuffle by optimizing I/O in different storage environments, such as HDD, SSD and disaggregated storage [167, 217, 182, 34]. Since performance at scale is a priority, these prior solutions are built as monolithic shuffle systems from scratch using low-level system APIs. However, these systems are costly to develop and integrate. For example, cloud providers each have to build proprietary services to support shuffle on their own storage services [34, 23, 185]. Magnet, a push-based shuffle system for Spark [182], took 19 months between publication and open-source release in the Spark project [75] because it required significant changes to system internals [181].

Furthermore, the above shuffle systems generally expose a batch execution API, often used to support end applications written with SQL or dataframes. Most of these systems are synchronous in nature: the results are available only after the entire shuffle operation completes. This poses challenges for applications that require *fine-grained* integration with the shuffle operation to improve their performance by processing data as it is being shuffled, i.e., pipeline data processing with the shuffle operation. For example, ML training often requires repeatedly shuffling the training dataset between epochs to improve learning quality [140, 139]. Doing this efficiently requires fine-grained pipelining between shuffle and training: ML trainers should consume partial shuffle outputs as soon as they become ready. Today's ML developers are faced with two undesirable choices: (1) they either rebuild shuffle from scratch, once again dealing with the performance challenges of large-scale shuffle, or (2) interface with existing shuffle systems through the synchronous APIs: the shuffle results can only be consumed after all partitions are materialized, leaving pipelining opportunities on the table.

To simplify the development of new shuffle optimizations targeting different environments, and to provide fine-grained pipelining for new applications, we propose an *extensible* architecture for distributed shuffle that enables flexible, efficient,



(a) Monolithic shuffle systems. (b) Exoshuffle.

Figure 5.1: Exoshuffle builds on an extensible architecture. Shuffle as a library is easier to develop and more flexible to integrate with applications. The data plane ensures performance and reliability.

and scalable implementations. Unlike previous solutions built as monolithic systems (Fig 5.1a), we propose building distributed shuffle as a library (Fig 5.1b). Such an architecture allows: (1) shuffle builders to easily develop and integrate new shuffle designs for new environments, and (2) a broader set of applications to leverage scalable shuffle in a more flexible manner.

How can we implement shuffle at the application level (as a library) while providing high performance? To answer this question, we first identify the optimizations in past shuffle systems that are key to performance and reliability. (1) Coordination: managing the timing and placement of mapper and reducer tasks, and implementing optimizations such as merging intermediate shuffle blocks. (2) Efficient data transfer: pipelining I/O with computation to maximize throughput, and spilling data to disk to accommodate larger-than-memory datasets. (3) Fault tolerance: guaranteeing data is reliably transferred to reducers via retries or replication.

Our key observation is that we can split these optimizations between a control and a data plane. Optimizations for coordinating shuffle are implemented by the *control plane* at the application layer, while the *data plane* provides efficient data transfer and fault tolerance at the system layer. This enables developers to easily implement a variety of shuffle solutions at the application layer, while having the underlying system handle efficient data transfer and fault tolerance.

The next question is what interface should the data plane provide to the application. Our answer is of course *distributed futures*. As described in Chapter 2, distributed futures extend RPC with an immutable shared address space, by allowing callers to pass objects *by reference* regardless of their physical location in the cluster. This abstraction allows the application to specify remote task invocations, while a common dataplane implements the physical data transfers. Distributed futures can also be passed before the data object is created, allowing the system to parallelize

remote calls and pipeline data transfer with task execution. We show that this abstraction can express a variety of shuffle algorithms, including dynamic strategies to handle data skew and stragglers (§5.3).

Although many distributed futures implementations exist, none of the these systems have been able to match the scale and performance of a monolithic shuffle system. CIEL [148] is the first to show MapReduce programs can be implemented using distributed futures, but it lacks an in-memory object store which is crucial for efficient pipelining and data transfers. Dask [171], another distributed futures-based dataframe system, supports in-memory objects but cannot scale beyond hundreds of GBs (§5.5.3.1). Previous versions of Ray [145] support shuffle within the capacity of its distributed shared memory object store, but lack disk spilling mechanisms and therefore do not support out-of-core processing.

In this work, we extend Ray with the necessary features to support large-scale shuffle (§5.4). These include: (1) locality scheduling primitives to enable colocating tasks to better exploit shuffle data locality; (2) a full distributed memory hierarchy with disk spilling and recovery; (3) asynchronous object fetching to pipeline task execution with disk and network I/O. We present Exoshuffle, a flexible and scalable library for distributed shuffle built on top of Ray. We demonstrate the advantages of this extensible shuffle architecture by showing that (§5.5):

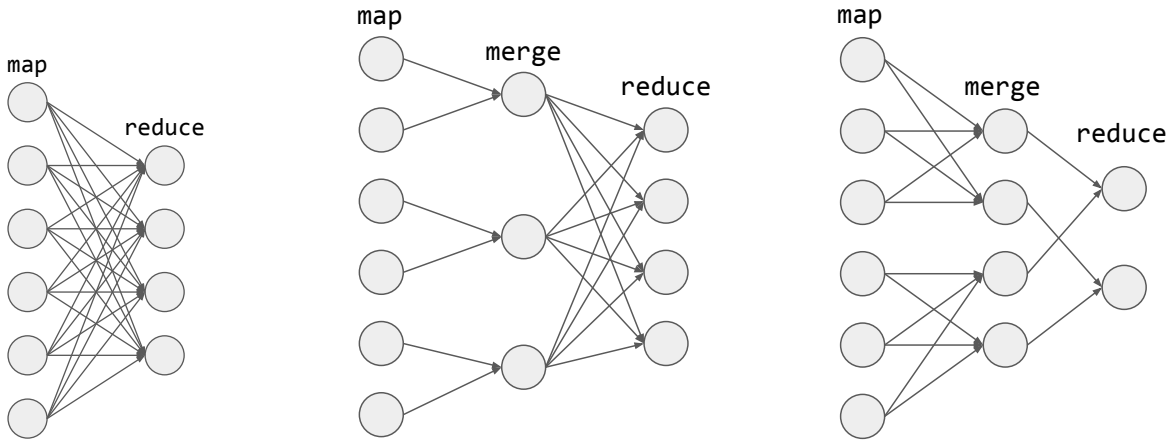
- A variety of previous shuffle optimizations can be written as distributed futures programs in Exoshuffle, with an order of magnitude less code.
- The Exoshuffle implementations of these shuffle optimizations match or exceed the performance of their monolithic counterparts.
- Exoshuffle can scale to 100 TB, outperforming Spark and Magnet by 1.8×, and breaking the CloudSort record as the world's most cost-efficient sorting system.
- Exoshuffle can easily integrate with a diverse set of applications such as distributed ML training, improving end-to-end training throughput by 2.4×.

5.2 Motivation

In this section, we overview two lines of previous work in building shuffle systems to illustrate the challenges in simultaneously achieving shuffle scalability and flexibility.

5.2.1 Shuffle Systems

In a MapReduce operation with M map tasks and R reduce tasks, shuffle creates $M \times R$ intermediate blocks. Each of these blocks must be moved across memory, disk, and network. As the number of tasks grow, the number of blocks increases and the



(a) "Simple" shuffle [72]. (b) Pre-shuffle merge [217]. (c) Push-based shuffle [182, 47, 197].

Figure 5.2: Shuffle algorithms for various applications. Exoshuffle uses distributed futures to execute these DAGs.

Storage target	Shuffle systems
Hard disk	Sailfish [167], Riffle [217], Magnet [182]
SSD	Zeus [38]
Cloud storage	Alibaba E-MapReduce Shuffle [23], AWS Glue Shuffle [34], Google Cloud Dataflow Shuffle [185]

Table 5.1: Different shuffle systems are built to optimize shuffle for deployment in different storage environments.

block size decreases both quadratically. At terabyte scale, this can result in hundreds of millions of very small blocks. This creates great challenges for I/O efficiency, especially for hard drives with low IOPS limits. Many shuffle systems have been built to optimize I/O efficiency in different storage environments. Table 5.1 shows an incomplete list of these systems, grouped by their target storage environments.

Previous I/O optimizations fall under two general categories: (1) reducing the number of small and random I/O accesses by *merging* intermediate blocks into larger ones at various stages [217, 47, 182] (Figures 5.2b and 5.2c), and (2) using *pipelining* to overlap I/O with execution [182, 97]. For example, *push-based shuffle* [197, 96] involves pushing intermediate outputs directly from the mappers to the reducers, allowing network and disk I/O to be overlapped with map execution, and optionally merging results on the reducer (Figure 5.2c) to improve disk write efficiency [182, 47].

While these solutions can improve throughput, they also come with high de-

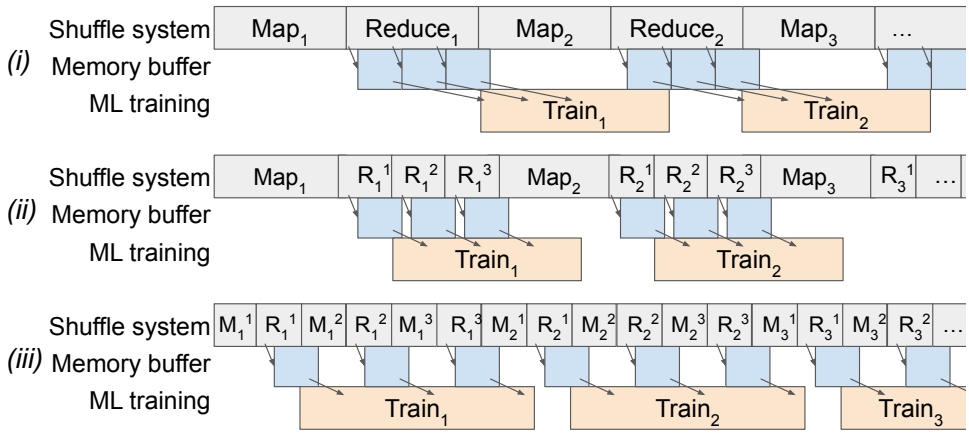


Figure 5.3: Pipelining data preprocessing and shuffle with GPU tasks in an ML training application.

velopment cost. Each new operation, such as reduce-side merge, requires building additional protocols for managing block transfers. However, although the ideas may be system-agnostic, the physical artifacts are often tightly integrated with proprietary storage systems, making them difficult to port to open-source frameworks. For example, many cloud providers build proprietary shuffle services to work with their own disaggregate storage offerings [34, 23, 185]; meanwhile, Magnet [182] is open-sourced as part of Spark but has yet to support disaggregated storage.

Furthermore, large-scale shuffle systems often come with more complicated deployment models. They are often deployed as auxiliary services to existing data processing systems. Shuffle services decouple block lifetimes from task executors to minimize interruptions upon executor failures [207], which are more frequent in large clusters. Shuffle services are also used to coordinate more sophisticated shuffle protocols, such as push-based shuffle and reduce-side merge [182]. However, because these shuffle services are only necessary at very large scale, they are not enabled by default in systems like Spark and require a separate deployment process.

Thus, while there has been significant innovation in new shuffle designs, few of these are widely deployed. Furthermore, it is difficult for an application to choose on the fly whether to use a particular shuffle algorithm; it requires both a priori knowledge of the application scenario and potentially an entirely different system deployment.

5.2.2 Random Shuffle in ML Training Pipelines

While much of the existing shuffle literature has focused on large-scale batch processing, there is also a need for performant shuffle in other application scenarios, such as online aggregation [67] and pipelining with downstream applications. An example of the latter is the *random shuffle* operation commonly used in machine learning train-

ing jobs. Note that by random shuffle, we mean the application-level transform that randomly permutes the rows of a dataset, rather than the generic system-level shuffle that is used to execute MapReduce applications.

To improve model convergence in deep learning, it is common practice to randomly shuffle the training dataset before feeding into GPU trainers to avoid bias on the order of the data [140, 139]. In some cases, the data must be shuffled on *every epoch*. To minimize GPU pauses, the shuffle should be pipelined with the training execution (Section 5.1). Furthermore, it is desirable for developers to be able to trade off between performance and accuracy: they might wish to run shuffle in a smaller window to reduce training latency, at the cost of overall end model accuracy.

These differences make it difficult for ML pipelines to directly leverage existing monolithic shuffle systems. Systems like Hadoop and Spark are highly optimized for global shuffle operations, but are not designed to pipeline the shuffle with downstream executions: shuffle results cannot be read until the full shuffle is complete [67]. The results must be written out to an external store before they can be read by the training workers (Section 5.1*i*). However, this leads to either high memory footprint, as it requires holding an additional copy of the dataset, or higher I/O overhead, if the shuffled data is written to disk before transfer to the GPU.

Fine-grained pipelining can improve efficiency. Section 5.1*ii* shows an example in which the reduce tasks for a particular epoch are pipelined with the training computation. This allows results to be used as they become available while limiting memory footprint to a single partition. Alternatively, the application can also choose to shuffle the dataset in windows (Section 5.1*iii*), improving pipelining at the cost of accuracy. Unfortunately, existing shuffle systems are not built for such fine-grained pipelining, and most big data systems that offer high-performance shuffle use an execution model that is incompatible with deep learning systems [69].

Instead, ML training frameworks often end up re-implementing shuffle within specialized data loaders and thus run into known problems that have been solved by traditional shuffle systems. Typically, data loaders are implemented with a pool of CPU-based workers colocated with the GPU trainers [149, 192, 93]. Each worker loads a partition of the dataset from storage (e.g., Amazon S3), preprocesseses it, and feeds the resulting data into the colocated trainers. To support random shuffle, the workers may read a random partition of the dataset on each epoch. However, to improve I/O efficiency, data must still be read in batches. Thus, to de-correlate data within the same batch, workers further shuffle the data by mixing records within a fixed-size local memory buffer. This effectively ties the shuffle window size to the size of the memory buffer. Setting the buffer size too large results in out-of-memory errors and poor pipelining, but if the buffer is too small, data de-correlation may be insufficient. In Appendix B.3.6, we demonstrate how Exoshuffle can bring distributed shuffle optimizations to ML training applications, achieving both high performance and flexibility.

5.3 Shuffle with Distributed Futures

For distributed futures to serve as an intermediate abstraction layer for shuffle, they should: (1) abstract out the common implementation details of different shuffle implementations, (2) be general enough to allow heterogeneous end applications to interface with the shuffle library, and (3) provide the same performance and reliability as monolithic shuffle systems. This narrow waist for distributed shuffle would enable both faster development for new shuffle implementations and extensibility to new application use cases.

Monolithic shuffle systems use messaging primitives, like RPC, as an intermediate abstraction layer. RPC is both general-purpose and high-performance, but it is too low-level to be a useful intermediate layer for shuffle. Integrating push-based shuffle into Spark, for example, required 1k+ LoC for the RPC layer changes alone [181]. Much of this development effort lies in implementing new inter-task protocols for data transfer and integrating them alongside existing ones.

In contrast, distributed futures decouple the shuffle control plane from the data plane. This abstraction enables different shuffle libraries to share a common data plane. Optimizations like push-based shuffle can be implemented in an order of magnitude less code as a result (§5.5.2).

In this section, we briefly overview the expression of known shuffle optimizations and application-specific shuffle variants as application-level libraries with the distributed futures API. These simplified examples capture the logical execution DAG of the shuffle. A full description and evaluation of the libraries can be found in Appendix B.

5.3.1 The Distributed Futures API

A distributed futures program invokes remote functions, known as *tasks*, that execute and return data on a remote node. When calling a remote function, the caller immediately gets a distributed future that represents the *eventual* return value. The future is "distributed" because the return value may be stored anywhere in the cluster, e.g., at the node where the task executes. This avoids copying return values back to the caller, which can become expensive for large data.

The caller can make use of a distributed future in two ways. First, it can create a DAG by passing a distributed future as an argument to another task. The system ensures that the dependent task runs only after all of its arguments are computed. Note that the caller can specify such dependencies before the value is computed and that the caller need not see the physical values. This gives the system control over parallelism and data movement, e.g., pipelining task execution with dependency fetching for other tasks, and allows the caller to manipulate data larger than local memory. Second, the caller can get the value of a distributed future using a `get` call, which fetches the value to the caller's local memory. This is useful when consuming

the output of a shuffle, as it allows the caller to pipeline its own execution with the shuffle. The caller can additionally use a `wait` call, which blocks until a set of tasks complete (without fetching the return values), for synchronization and for avoiding scheduling too many concurrent tasks.

5.3.2 Expressing Shuffle with Distributed Futures

Listing 1 demonstrates how these APIs can be used to express various shuffle optimizations (Fig 5.2) as application-level programs. We use Ray's distributed futures API for Python [145] for illustration. The `@ray.remote` annotation designates remote functions, and the `.remote()` operator invokes tasks.

```
def simple_shuffle(M, R, map, reduce):
  map_out = [map.remote(m) for m in range(M)]
  return ray.get([
    reduce.remote(map_out[:,r]) for r in range(R)]

def shuffle_riffle(M, R, F, map, reduce, merge):
  map_out = [map.remote(m) for m in range(M)]
  merge_out = [
    merge.remote(map_out[i*F:(i+1)*F, :])
    for i in range(M/F)]
  return ray.get([
    reduce.remote(merge_out[:,r]) for r in range(R)]

def shuffle_magnet(M, R, F, map, reduce, merge):
  map_out = [map.remote(m) for m in range(M)]
  merge_out = [
      [merge.remote(map_out[i*F:(i+1)*F, r])
      for i in range(M/F)] for r in range(R)]
  return ray.get(
    [reduce.remote(merge_out[:,r]) for r in range(R)]
```

Listing 1: Shuffle algorithms as distributed futures programs.

As an illustrative example, we discuss how "simple shuffle" is implemented with distributed futures. In Listing 1, `simple_shuffle` shows a straightforward implementation of the MapReduce paradigm illustrated in Figure 5.2a. The shuffle routine takes a `map` function that returns a list of map outputs, and a `reduce` function that takes a list of map outputs and returns a reduced value. `M` and `R` are the numbers of map and reduce tasks respectively. The two statements produce the task graph shown in Figure 5.2a. Note that the `.remote()` calls are non-blocking, so the entire task graph can be submitted to the system without waiting for any one task to complete.

This is effectively *pull-based shuffle*, in which shuffle blocks are *pulled* from the map workers as reduce tasks progress. Assuming a fixed partition size, the total

number of shuffle blocks grows quadratically with the total data size. Section 5.5.1 shows empirical evidence of this problem: as the number of shuffle blocks increases, the performance of the naive shuffle implementation drops due to decreased I/O efficiency. Prior work [217, 182, 47] have proposed solutions to this problem; a full study of adapting these strategies with distributed futures can be found in Appendix B.

5.4 System Architecture

Section 5.3 shows how shuffle DAGs can be expressed as distributed futures programs. However, achieving high performance shuffle also requires a set of critical system facilities. In this section, we describe the architecture of Exoshuffle via a realistic implementation of the push-based shuffle described in Appendix B.1.2. We describe the additional system APIs used by Exoshuffle (§5.4.2), and the transparent features provided by the underlying distributed futures implementation (§5.4.3) that are key to performance.

5.4.1 Example: Push-based Shuffle

Listing 2 implements push-based shuffle (Appendix B.1.2) for a cluster of `NUM_WORKERS` nodes. The library takes a `map` and a `reduce` function as input. The remaining constants are chosen by the library according to the user-specified number of input and output partitions.

Lines 11–25 comprise the map and merge stage, in which map results are shuffled, pushed to the reducer nodes, and merged. This stage pipelines between CPU (`map` and `merge` tasks), network (to move data between `map` and `merge`), and disk (to write out `merge` results). The map and merge tasks are scheduled in rounds for pipelining: Lines 18–19 ensures that there is at most one round of merge tasks executing, and that they can overlap with the following round's map tasks. Each round submits one merge task per worker node. Each merge task takes in one intermediate result from each map task from the same round and returns as many merged results as there are reduce partitions on that worker.

Once all map and merge tasks are complete, we schedule all reduce tasks (lines 28–31) and return the distributed future results. Each reduce task performs a final reduce on all merge results for its given partition. To minimize unnecessary data transfer, the reduce tasks are co-located with the merge tasks whose results they read.

5.4.2 Scheduling Primitives

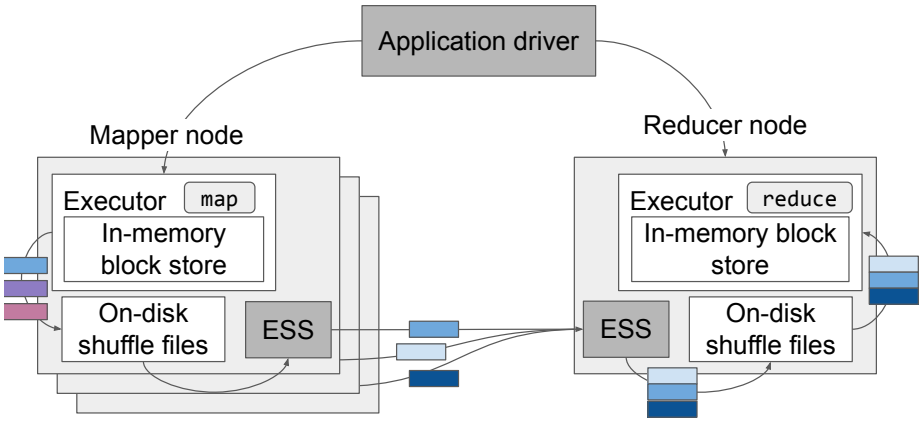
For complex applications like distributed shuffle, it is difficult for a general-purpose system to make optimal decisions in every context. For instance, optimally scheduling

```
def push_based_shuffle(map, reduce):
  @ray.remote
  def merge(*map_results):
    for results in zip(*map_results):
      yield reduce(*results)

  merge_results = numpy.empty((NUM_WORKERS,
    NUM_ROUNDS, NUM_REDUCERS_PER_WORKER))

  # Map and shuffle.
  for rnd in range(NUM_ROUNDS):
    for i in range(NUM_TASKS_PER_ROUND):
      map_results = [
        map.options(num_returns=NUM_WORKERS).remote(
          parts[rnd * NUM_TASKS_PER_ROUND + i])
        for i in range(NUM_TASKS_PER_ROUND)]

    if rnd > 0:
      ray.wait(merge_results[:, rnd - 1, :])

    for w in range(NUM_WORKERS):
      merge_results[w, rnd, :] = merge.options(
        worker=w, num_returns=NUM_REDUCERS_PER_WORKER
      ).remote(*map_results[:, w])
    del map_results

  # Reduce.
  return flatten(
    [[reduce.remote(*merge_results[w, :, rnd])
      for rnd in range(NUM_REDUCERS_PER_WORKER)]
      for w in range(NUM_WORKERS)])
```

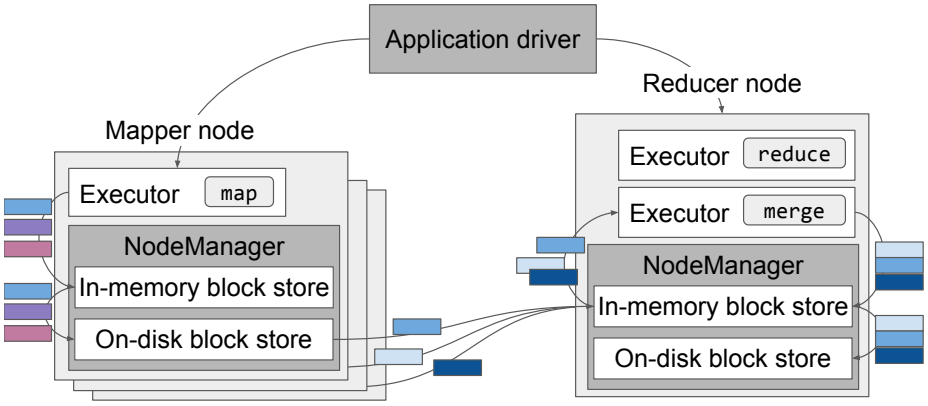
Listing 2: Implementation of two-stage shuffle.

a computation DAG on a set of nodes is NP-hard [55]. It is therefore more robust to allow the application or library developer to apply domain-specific knowledge to achieve better performance.

By default, Ray provides a two-level distributed scheduler that balances between bin-packing vs. load-balancing [145]. This is sufficient for map and reduce tasks in simple shuffle, as these can be executed anywhere in the cluster. However, more advanced shuffle strategies (Appendices B.1.1 and B.1.2) require more careful placement and scheduling of tasks to improve performance. In this section, we describe the additional APIs designed to give the shuffle library more control over the physical execution of the shuffle DAG.



(a) Example of a monolithic shuffle architecture.



(b) Exoshuffle.

Figure 5.4: Comparing a monolithic vs. application-level shuffle architecture. (a) implements all coordination and block management through an external shuffle service on each node, in this case implementing the Magnet shuffle strategy (Appendix B.1.2). (b) shows the same shuffle strategy but implemented as an application on a generic distributed futures system.

5.4.2.1 Scheduling for Data Locality

Ray provides automatic locality-based scheduling when possible. For example in Listing 2, lines 28–31, Ray automatically schedules the reduce tasks on the workers on which the upstream merge results reside. In some other cases, hints must be provided to the system to achieve better data locality. For example, a group of merge tasks must be colocated with the downstream reduce task, but this is impossible for the system to determine because the reduce task's dependency is not known to the system yet. To handle this problem, we introduce *node-affinity scheduling* in Ray, which allows the application to pin tasks to a particular node. For example, Listing 2 uses this in line 23 to colocate merge tasks for the same reducer. Node affinity is soft, meaning that Ray will choose another suitable node if the specified node fails.

5.4.2.2 Scheduling for Task Pipelining

The map and merge tasks should be pipelined to allow map results to be shuffled concurrently with map execution. This task-level pipelining is challenging for a distributed futures system to determine automatically: Too many concurrent map tasks will reduce resources available to downstream merge tasks, and scheduling the wrong set of map and merge tasks concurrently prevents map outputs from being consumed directly by merge tasks, resulting in unnecessary disk writes. The shuffle library is better placed to determine that it should apply backpressure by limiting the number of concurrent map and merge tasks. The library can also determine that a round of merge tasks should be executed concurrently with the following round of map tasks. Exoshuffle achieves this with the `wait` API (Listing 2, line 19), which blocks until a task completes.

5.4.2.3 Controlling Redundancy with Reference Counting

Distributed futures are reference-counted in Ray. While an object reference is in scope, Ray attempts to ensure its value exists in the cluster. By selecting which references to keep or drop, the shuffle library can make tradeoffs between reducing write amplification and improving data redundancy. For example, line 25 of Listing 2 deletes the intermediate map results from the current round. This reduces write amplification, as the map results can be immediately dropped from memory without spilling to disk, but requires additional re-execution upon failure. Alternatively, the shuffle library can instead keep the intermediate references, resulting in additional disk writes but improved data redundancy.

5.4.3 Transparent System Facilities

The actual data transfer, or *shuffle*, is managed by the distributed futures system according to the application specifications. For example in Listing 2, lines 23–25

specifies that one column of the distributed futures in `map_results` should be sent to one merge task. This prompts the data plane to transfer the corresponding physical data to the `merge` task's location. In this section, we describe the transparent storage and I/O mechanisms provided by the distributed futures system to facilitate this data movement.

5.4.3.1 Shared Memory Object Store

Previous monolithic shuffle systems implement distributed coordination via an *external shuffle service*, a specialized process deployed to each node that orchestrates block transfers (Fig 5.4a). This process is external to the executors, decoupling block transfers from map and reduce task execution. In Exoshuffle, we replace this service with a generic node manager that is responsible for both in-memory and spilled objects (Fig 5.4b).

We build on Ray's shared memory object store [145] for immutable objects. Each node manager hosts a shared memory object store shared by all executors on that node (Fig 5.4b). This decouples executors from blocks: once a task's outputs are stored in its local object store, the node manager manages the block. This keeps executors stateless and allows them to execute other tasks or exit safely while the node manager coordinates block movement. Shared memory enables *zero-copy* reads of object data on the same node, which avoids CPU and memory overhead. By making objects immutable, we also avoid consistency concerns between object copies.

Next, we describe extensions to the original Ray architecture [145] made in this work that improves pipelining disk and network I/O with task execution. These improvements are made at the system level without knowledge of the application-level shuffle semantics, and thus can benefit a wide range of data-intensive applications.

5.4.3.2 Pipelined Object I/O

Object Allocation and Fetching. There are two categories of object memory allocations: new objects created for task returns (e.g., `map` task outputs), and copies of objects fetched remotely as task arguments (e.g., `merge` task inputs). The memory subsystem queues and prioritizes object allocations to ensure forward progress while keeping memory usage bounded to a limit. This is critical for reducing thrashing within the object store, caused by requesting objects for too many concurrent requests, while leaving sufficient heap memory for task executors.

All memory allocations on a Ray worker node go into an allocation queue for fulfillment. If there is spare memory, the allocation is fulfilled immediately. Otherwise, requests are queued until the spilling process or garbage collection frees up enough memory. If memory is still insufficient, Ray falls back to allocating task output objects on the filesystem to ensure liveness. Spare memory besides the memory allocated to executing task arguments and returns is used to fetch the arguments

of queued tasks. This enables pipelining between execution and I/O, i.e. restoring objects from disk or fetching objects over the network. For example, at line 28 in Listing 2, all merge results are already spilled to disk and all reduce tasks are submitted at once. While earlier reduce tasks execute, the system uses any spare memory to restore merge results for the next round of reduce tasks from disk.

Object Spilling. Object spilling is transparent, so the application need not specify if or when it should occur. When the memory allocation subsystem has backlogged requests, the spilling subsystem migrates referenced objects to disk to free up memory. When a spilled object's data is required locally for a task, e.g., because it is the argument of a queued task, the node manager copies it back to memory as described above. When requested by a remote node, the spilled object is streamed directly from disk across the network to the remote node manager. To improve I/O efficiency, Ray coalesces small objects into larger files before writing to the filesystem.

5.4.3.3 Fault Tolerance

Exoshuffle relies on lineage reconstruction for distributed futures to recover objects lost to node failures [201], a similar mechanism to previous shuffle systems [72, 212]. In Ray, the application driver stores the object lineage and resubmits tasks as needed upon failure. This process is transparent to Exoshuffle, which runs at the application level. Still, Exoshuffle can use object references (§5.4.2.3) to specify reconstruction or eviction for specific objects.

Executor process failures are much more common than node failures; executors can fail due to transient application errors and/or out-of-memory errors, whereas nodes should only fail if there is a critical system bug or machine failure. If reconstruction is required each time an executor fails, it can impede progress [207, 182]. Many previous shuffle systems use an external shuffle service to ensure map output availability in the case of executor failures or garbage collection pauses. Similarly, in Exoshuffle, executor process failures do not result in the loss of objects, because the object store is run inside the node manager as a separate process.

More sophisticated shuffle systems require additional protocols such as deduplication to ensure fault tolerance [47]. Distributed futures prevent such inconsistencies because they require objects to be immutable, task dependencies to be fixed, and tasks to be idempotent.

To reduce the chance of data loss, some shuffle system uses on-disk [182] or in-memory [47] replication of intermediate blocks to guard against single node failures. In Ray, objects are spilled to disk and transferred to remote nodes where they are needed, which also results in multiple copies as long as the object is in scope. The application can also disable this optimization by deleting its references to the object (Listing 2, L25). In the future, we could allow the application to more finely tune the number of replicas kept, e.g., by passing this as a parameter during task invocation.

5.5 Evaluation

We study the following essential questions in the evaluation:

- Can Exoshuffle libraries achieve performance and scalability competitive with monolithic shuffle systems? (§5.5.1)
- Is it easier to implement shuffle optimizations in Exoshuffle? (§5.5.2)
- How do the features in the distributed futures backend contribute to Exoshuffle performance? (§5.5.3)

A more complete evaluation can be found in Appendix B, which further evaluates the performance and scalability of Exoshuffle libraries as well as the benefits for end applications, including CloudSort, online aggregation and ML training (Appendix B.3.3).

5.5.1 Shuffle Performance

5.5.1.1 Setup

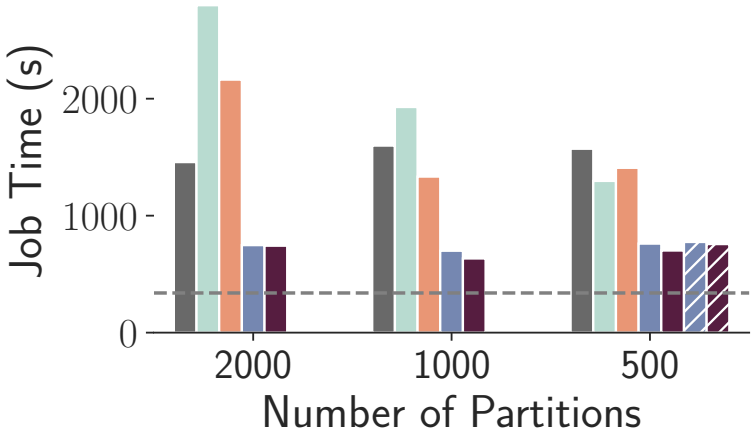
We create test environments on Amazon EC2 using VMs targeted at data warehouse use cases. We test on a HDD cluster of d3.2xlarge instances (8 CPU, 64 GiB RAM, 6× HDD, 1.1 GB/s aggregate sequential throughput, 18K aggregate IOPS, 15 Gbps network), and a SSD cluster of i3.2xlarge instances (8 CPU, 61 GiB RAM, NVMe SSD, 720 MB/s throughput, 180K write IOPS, 10 Gbps network).

Workload. We run the Sort Benchmark (a.k.a. TeraSort or CloudSort) [180], as it is a common benchmark for testing raw shuffle system performance. This benchmark requires sorting a synthetic dataset of configurable size, consisting of 100-byte records with 10-byte keys.

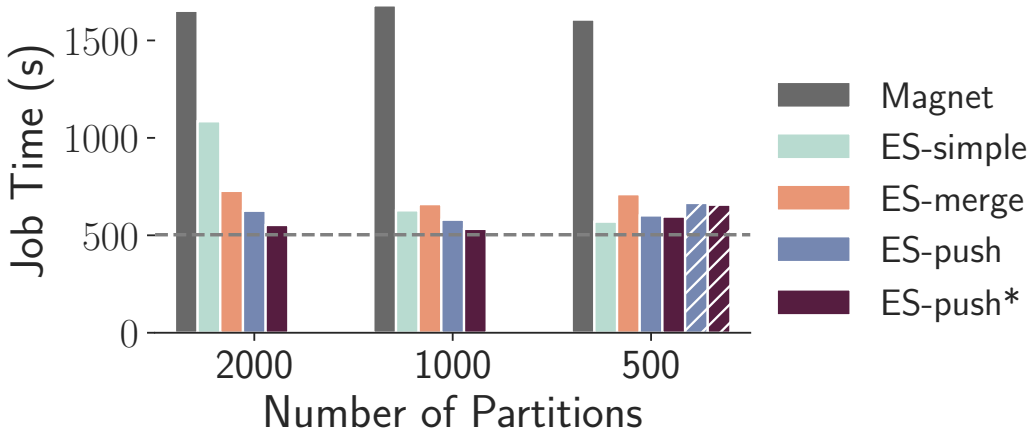
Baselines. We compare to the push-based shuffle service in Spark, a.k.a. Magnet, and a theoretical baseline.

Magnet is integrated into Spark in its 3.2.0 release as an external push-based shuffle service. We run Spark 3.2.0 on Hadoop 3.3.1 with Magnet shuffle service enabled. We disable compression of shuffle files according to the rules of TeraSort. This allows for a fair comparison in terms of total bytes of disk I/O.

For the theoretical baseline, we assume disk I/O is the bottleneck since empirically we find that disk I/O takes longer than networking and CPU processing in this benchmark. The baseline is calculated by $T = 4D/B$, where D is the total data size and B is the aggregate disk bandwidth. D is multiplied by 4 because each datum needs to be read twice and written twice, a theoretical minimum for external sort [168].



(a) 1 TB sort on 10 HDD nodes.



(b) 1 TB sort on 10 SSD nodes. Semi-shaded bars are runs with failures (§5.5.3.3).

Figure 5.5: Comparing job completion times on the Sort Benchmark. The dashed lines indicate the theoretical baseline (§5.5.1.1). Exoshuffle is abbreviated as ES.

Exoshuffle variants. We run Exoshuffle on Ray 1.11.0. We compare implementations of the following shuffle libraries:

- ES-simple, the simple shuffle variant (§5.3.2).
- ES-merge, pull-based shuffle with pre-shuffle merge, similar to that in Riffle (Appendix B.1.1).
- ES-push, push-based shuffle similar to Magnet (Appendix B.1.2).
- ES-push*, push-based shuffle further optimized to reduce write amplification (§5.4.1).

5.5.1.2 Performance Comparison of Shuffle Libraries

Performance on HDD. Figure 5.5a shows the job completion times of Exoshuffle variants running 1 TB sort on 10 HDD nodes. ES-simple shows the well-known scaling problem: performance degrades as the number of partitions increases, because the intermediate shuffle blocks become more in number and smaller in size both quadratically, quickly reaching disk IOPS limit. The push-based shuffle variants (ES-push, -push*) achieve better performance regardless of the number of partitions, thanks to the merging of shuffle blocks to increase disk I/O efficiency and the pipelining of disk and network I/O. ES-merge runs slower than -simple because merging the map output blocks incurs additional disk writes, which outweighs the I/O efficiency savings when the number of partitions is small, and only shows benefits when the number of partitions increases. The Magnet baseline shows comparable performance. In summary, Exoshuffle libraries demonstrate performance benefits that match the characteristics of their monolithic counterparts.

Performance on SSD. Figure 5.5b shows the same benchmark and variants running on the SSD cluster. All variants of Exoshuffle outperform the PBS baseline, and display similar trends as on the HDD cluster. The run times of the optimized versions of Exoshuffle are also close to the theoretical baseline. Since the NVMe SSD supports much higher random IOPS, the I/O efficiency gains are less pronounced.

For additional performance and scalability benchmarks, see Appendix B.

[1] Total lines of code in `org.apache.spark.shuffle`.
[2] As reported by Zhang et al. [217]
[3] Total added lines in `https://github.com/apache/spark/pull/29808/files`.

Shuffle Algorithm	System LoC	Exoshuffle LoC
Simple (§5.3.2)	2600 (Spark[1])	**215**
Pre-shuffle merge (Appendix B.1.1)	4000 (Riffle[2])	**265**
Push-based shuffle (Appendix B.1.2)	6700 (Magnet[3])	**256**
with pipelining (§5.4.1)	–	**256**

Table 5.2: Approximate lines of code for implementing shuffle algorithms in Exoshuffle versus in specialized shuffle systems.

5.5.2 Implementation Complexity

In Exoshuffle, shuffle is expressed as application-level programs. Table 5.2 compares the amount of code of several monolithic shuffle systems with the lines of code needed to implement the corresponding shuffle algorithms in Exoshuffle. Exoshuffle libraries may not provide all the production features of the monolithic counterparts, but many shuffle optimizations can be implemented in Exoshuffle with an order of magnitude less code, while keeping the same performance benefits. By offering shuffle as a library, Exoshuffle also allows applications to choose the best shuffle implementation at run time without deploying multiple systems.

5.5.3 System Microbenchmarks

The Exoshuffle architecture requires high-performance components from the distributed futures system to deliver good performance. In this section, we study the the impact of these system components on shuffle performance.

5.5.3.1 Shared-Memory Object Store

We study the effect of a shared-memory object store that decouples objects from executors by comparing Dask (v2021.4.0) and Ray (v1.11.0). Dask and Ray are both distributed futures systems, but they differ in architecture. Ray uses a shared-memory object store that is shared by multiple executor processes on the same node (§5.4.3.1). Dask stores objects in executor memory and requires the user to choose between multiprocessing and multithreading. With multithreading, multiple Dask executor threads share data in a heap-memory object store, but the Python Global Interpreter Lock can severely limit parallelism. Dask in multiprocessing mode avoids this issue but uses one object store per worker process, so objects must be copied between workers on the same node. Thus, the lack of a shared-memory object store results in either reduced parallelism (multithreading) or high overhead for sharing objects (multiprocessing). It is also less robust as objects are vulnerable to executor failures.

We study these differences by running the same Dask task graph on Dask and Ray backends [198]. Figure 5.6 shows dataframe sorting performance on a single node

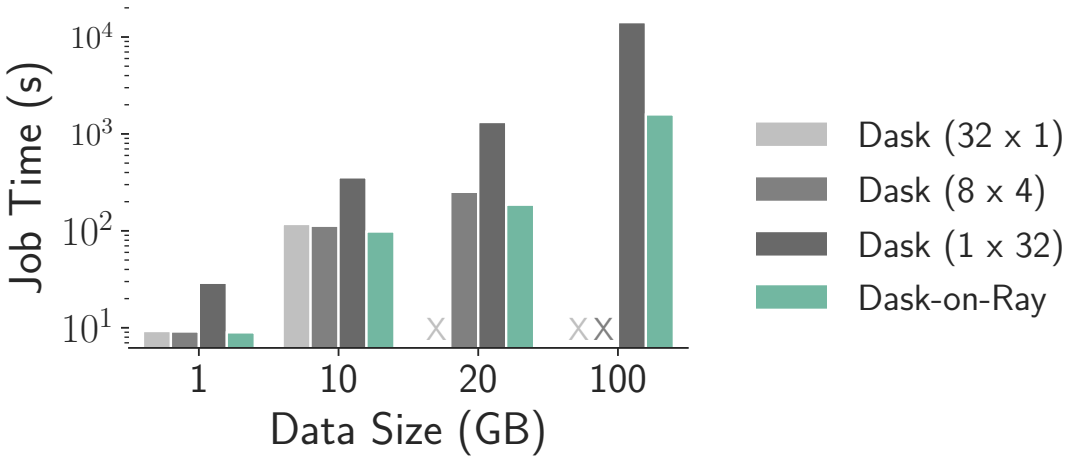


Figure 5.6: Comparing shuffle time in Dask and Ray. Legends show number of processes × threads.

Figure 5.7: Effect of I/O optimizations in Ray.

(32 CPU, 244 GB RAM, 100 partitions). For Dask, we vary the number of executor processes and threads to show the tradeoff between memory usage and parallelism. Ray requires no configuration and uses 32 executor processes, 1 per CPU.

On small data sizes, Dask with multiprocessing achieves about the same performance as Ray, but it is 3× slower with multithreading due to reduced parallelism. On larger data sizes, Dask with multiprocessing fails due to high memory pressure from extra object copies. Meanwhile, Ray's shared-memory object store enables better stability and lower run time on all data sizes.

5.5.3.2 Small I/O Mitigations

Ray implements two system-level optimizations for mitigating the small I/O problem: fusing writes of spilled objects to avoid small disk I/O, and prefetching task arguments to hide network and disk latency (§5.4.3.2). To show the impact of these optimizations, we run a single-node microbenchmark that creates 16 GB total objects in a 1 GB object store, forces them to spill to disk, then restores the objects from disk. We use object sizes ranging from 100 KB to 1 MB, as these are comparable to the shuffle block sizes. We use a sc1 HDD disk since the disk I/O bottleneck is more pronounced on slower storage.

Fusing Writes. Ray fuses objects into at least 100 MB files then writes them to disk. Figure 5.7 shows the total run time stays constant across object sizes with default fusing. When fusing is off, the run time is 25% slower for 1 MB objects, and up to 12× slower when spilling 100 KB objects.

Prefetching Task Arguments. Ray prefetches task arguments in a pipelined manner so that arguments are ready on a worker by the time the task is scheduled. Figure 5.7 shows that pipelined fetching of task arguments reduces the run time by 60–80%, comparing with a baseline implementation that only starts fetching objects after the task is scheduled.

5.5.3.3 Fault Tolerance

To test fault recovery, we fail and restart a random worker node 30 seconds after the start of the run. This results in both executor failure and data loss, as the worker's local object store is also lost. In all cases, we rely on the distributed futures system to re-execute any lost tasks and to reconstruct any lost objects. Lineage reconstruction (§5.4.3.3) minimizes interruption time during worker failures. Figures 5.5a and 5.5b show run times with failures indicated with semi-shaded bars. For ES-simple and -merge, a known bug in Ray currently prevents fault recovery from completing. For ES-push and -push*, recovering from a worker failure adds 20–50 seconds to the job

completion time. The system uses this time to detect node failures and re-execute tasks to reconstruct lost objects.

5.6 Related Work

Shuffle in Data Processing Systems. Many solutions to shuffle have been proposed [167, 96, 203, 121, 169, 168] since MapReduce [72] and Hadoop [205], with a focus on optimizing disk I/O and pipelining. Sailfish [167] is a notable example deployed at Yahoo which depends on a modified filesystem to batch disk I/O. Many recent shuffle systems have been built in industry for large-scale use cases [217, 47, 182, 38], but few have been open-sourced. Today's cloud providers often offer managed shuffle services [23, 185, 34]. However these are tightly integrated with proprietary cloud data services and are not accessible by other shuffle applications.

Hardware Environments. Hardware typically poses a range of constraints on shuffle design. For example, compute and memory may be either disaggregated [217, 47, 163] or colocated [167, 207, 182]. Disk constraints also affect system design, e.g., SSDs provide better random IOPS than HDDs but wear out more quickly. Many existing shuffle systems have been motivated by such hardware differences. In Exoshuffle, because the distributed futures API abstracts block management, a shuffle developer can plug in different storage backends and optimize shuffle at the application level.

Other Shuffle Applications. Machine learning research [140, 139] shows that SGD-based model training benefits from random shuffling of the training dataset. Both TensorFlow [149] and PyTorch [192] have built specialized systems designed specifically to pipeline data loading with ML training. These data loaders, in addition to Petastorm [93], support distributed data loading and random shuffling but shuffling is limited to a local buffer capped by worker memory (Appendix B.3.6).

Dataframes [204, 171, 159] are another class of applications in data science that depend on shuffle for operations such as group-by. While systems like Dask [171] and Spark [214] provide distributed dataframes, developers continue to build new engines that optimize for specific application scenarios, such as multi-core [74], out-of-core performance [48], or supporting SQL [85]. These new dataframe libraries, along with new embedded query engines such as DuckDB [165] and Velox [158], can directly use Exoshuffle to support distributed query processing.

Distributed Programming Abstractions for Shuffle. CIEL [148] is the first to propose using distributed futures to express iterative distributed dataflow programs, including MapReduce. Its implementation does not include features critical to large-scale shuffle performance, including intra-node parallelism, in-memory object storage,

and automatic garbage collection [150]. Dask [171] is another distributed futures-based system that has trouble scaling shuffle due to the lack of shared-memory objects (§5.5.3.1). While we build on Ray's design, such as a shared-memory object store [145] and lineage reconstruction [201], previous versions are not sufficient to support large-scale shuffle as they do not include spilling to disk or pipelining between execution and I/O. Thus, while others have implemented shuffle on distributed futures before, ours is the first that we know of to reach the scale, performance, and reliability of monolithic shuffle systems.

Serverless functions, as used in Locus [163], are one alternative to distributed futures. While Locus leverages an existing serverless cache and persistent storage, it still must manage block movement manually. In contrast, distributed futures abstract block management in full and manage execution, memory, and disk collectively on each node.

Hoplite [219] shows that it is possible to provide a high-performance and fault-tolerant collective communication layer on top of distributed futures, supporting operations such as scatter, gather, and reduce. Shuffle in MapReduce-like systems is a more challenging problem because it involves scheduling arbitrary compute tasks along with all-to-all communication. In this work we show that a distributed futures system can support shuffle at TB+ scale and provide competitive performance and reliability.

5.7 Discussion

Extensible Architectures. The decoupling of control and data planes in software-defined networking [137] has led to great innovations in the past two decades [82]. Operating systems research also advocates for extensible architectures to build OS kernels, such as microkernels [209] and exokernels [81]. We hope our work can drive more innovations in shuffle designs and applications through an extensible architecture for distributed shuffle.

Distributed Futures. Distributed futures are rising in popularity due to their ease of use and flexibility [148, 145, 199]. However, the question of flexibility versus performance remains. Large-scale shuffle is one of the most challenging problems in big data processing, inspiring years of work. By showing that large-scale shuffle is possible on a generic and flexible distributed futures system, we hope to show that other complex applications can be built on this framework, too.

Limitations. The ability to specify arbitrary tasks and objects with distributed futures is the key to its flexibility, but it is also the primary obstacle to performance. The system assumes that each task is independent for generality and stores metadata separately for each task and object. In contrast, monolithic shuffle systems have

semantic information and can share metadata for tasks and objects in the same stage. Currently metadata overhead is the main limitation to executing Exoshuffle at larger scales. We plan to address this in the future by "collapsing" shared metadata, i.e., keeping one metadata entry for multiple outputs of a task.

Architecturally, the primary limitation in Exoshuffle is the fact that an object must be loaded in its entirety into the local object store before it can be read (§5.4.3.1). Generators allow tasks to "stream" large outputs by breaking them into many smaller physical objects; future improvements include the described metadata optimizations and/or introducing APIs to stream objects larger than the object store, similar to Ciel [148]. Another limitation is in scheduling. Currently the distributed futures system may require hints from the shuffle library to determine which tasks should be executed concurrently and where to place tasks (§5.4.2). A more sophisticated scheduler may be able to determine these automatically.

Finally, Exoshuffle does not yet address the problem of providing a single shuffle solution that can meet the requirements of all applications. Doing so would require automatically picking the best shuffle algorithm and parameters based on application, environment, and run-time information. Instead, we focus on the problem of shuffle *evolvability*, a necessary step towards this overarching goal.

5.8 Conclusion and Lessons Learned

The history of Exoshuffle begins from the early days of Ray, when we showed how one could write MapReduce programs using distributed futures, with the same "simple shuffle" algorithm shown in Listing 1. While this version of shuffle did "run", it predated the ownership architecture and thus there were significant limitations, including the lack of memory safety, reliable failure detection and recovery, out-of-core processing, and pipelined execution. The former two were addressed in the ownership architecture in Chapter 4, while Exoshuffle deals with the latter two. These new features open the door to many applications beyond shuffle systems. In particular, the combination of low (millisecond-level) overheads and generic, dynamically defined tasks opens up two important possibilities: (1) low-latency streaming execution with fine-grained rollback recovery, and (2) the ability to pipeline between heterogeneous resources. Thus, one of the key lessons learned in Exoshuffle is:

1. Implementing generic performance optimizations such as millisecond-level task pipelining can produce a step change in the class of applications that are practical to execute.

In building Exoshuffle, the sheer range of possibilities in shuffle system design became clear. While at first the goal was simply to leverage the ownership system to show that distributed futures could be used to execute MapReduce applications efficiently, later the goals expanded to include shuffle system *extensibility*. Of course,

extensibility turns out to be an important goal in all sufficiently complicated systems. Thus, a second key lesson learned in this chapter was:

2. In addition to deduplicating effort and promoting interoperability across application domains, having a common interface such as distributed futures is also important for improving system extensibility within a *single* application domain.

Chapter 6

Exoflow: A Universal Workflow System for Exactly-Once DAGs

The ownership architecture provides low performance overheads and fast recovery for distributed futures applications composed of millisecond-level dynamic tasks. However, it also has two key limitations: (1) to guarantee exactly-once semantics, tasks must be deterministic and idempotent, and (2) it does not provide automatic durability, i.e. recovered tasks are re-executed from the beginning. Taken together, these properties mean that nontrivial applications may still require developers to implement significant parts of recovery. This can become especially challenging if applications are composed together, even if some sub-applications can indeed meet ownership's assumptions.

We address these limitations in Exoflow. Exoflow provides end-to-end exactly once semantics for distributed *workflow* applications, i.e. DAGs that compose arbitrary application tasks, each of which may itself be a distinct distributed application. This can be easily accomplished by a strawman recovery strategy that materializes and saves all task outputs before making them visible to others, thus imposing high performance overheads on each workflow task. Instead, Exoflow's goal is to enable a *flexible* choice of recovery vs. performance tradeoffs.

Exoflow accomplishes this by *decoupling the unit of execution from the unit of recovery*. To do so, Exoflow uses and extends the distributed futures interface to gain application semantics that inform the choice of recovery strategy for each task. In particular, similar to the works presented in previous chapters, we first use the distributed futures interface to capture the application's dataflow graph. This gives the system freedom to choose how the moved data should be saved and recovered. Second, we extend the distributed futures interface to include *task annotations* that allow the system to identify and recover tasks that are nondeterministic and/or that have external effects.

Exoflow thus generalizes recovery for existing workflow applications ranging from

ETL pipelines to stateful serverless workflows. This greater flexibility in recovery strategy enables Exoflow to match or exceed the run-time and recovery performance of other workflow systems, even ones that are application-specific. For example, Exoflow can achieve a 3× reduction in job run time for ETL workflows compared to Apache Airflow, as well as a 2× reduction in task latency for serverless workflow systems.

This chapter is based on the published work [220] and includes significant contributions of the coauthors Siyuan Zhuang et al.

6.1 Introduction

A key requirement for distributed applications is *fault tolerance*, i.e. the appearance of execution without failures even when failures occur. In general, there is a tradeoff between recovery and run-time overhead. For example, logging generally adds higher execution overhead but reduces recovery time by allowing the system to only re-execute computations that failed [78]. Meanwhile, checkpointing reduces execution overhead but can impose higher recovery overhead as the system must roll back additional computation after a failure.

Current distributed systems often choose different tradeoff points between recovery and performance based on the application. For example, Apache Spark uses lineage-based logging for batch processing [212], and Apache Flink uses checkpointing for stream processing [57].

However, it is becoming increasingly common for different applications to be composed into heterogeneous pipelines. For example, a machine learning pipeline might use batch ingest to build a training dataset, then stream the data to a batch distributed training job to reduce latency and memory overhead. If we use a single recovery strategy for the entire pipeline, performance and recovery may be suboptimal because different recovery strategies are suited to different applications. Thus, to optimize end-to-end performance and recovery, we need to *compose different recovery strategies*.

Implementing multiple, interoperable recovery techniques within the same system, let alone a single one, is challenging. For example, Spark introduced "continuous processing" to reduce performance overheads for stream processing applications, but this mode does not yet provide exactly-once semantics during failures [29]. On the other hand, Flink has added a batch processing mode, but this required building an entirely separate recovery system from the streaming path [58].

Overall, these challenges have led to patchy support for applications that have diverse requirements in the recovery-performance tradeoff space. Users must choose between: (1) building on a single system, and face a fixed choice of performance vs. recovery overheads, or (2) stitching together multiple systems that offer different application-specific tradeoffs. The latter, however, is challenging and requires

coordinating the flow of data, control, and recovery across disparate systems. This is true even in a single system, if using disparate execution modes such as batch vs. streaming.

In this chapter, we propose a *universal workflow* system that enables a flexible choice of recovery vs. performance tradeoffs, even within the same application. A *workflow* is a directed acyclic graph (DAG) of *tasks*, where each task encapsulates a function call and edges between tasks represent data dependencies. Workflows are used to orchestrate execution across systems and thus prioritize generality. The DAG API is popular because it allows arbitrary application code in each task, from submitting a Spark job to invoking a microservice.

In contrast to other workflow systems, however, we *decouple the unit of execution from the unit of recovery*. In particular, Exoflow guarantees fault tolerance by durably logging the workflow DAG and coordinating task checkpoint and recovery, while execution of the DAG is handled by a generic "backend". This has three key benefits. First, it enables heterogeneous application pipelines that need multiple recovery strategies for performance. Second, it augments existing distributed execution frameworks that provide only at-most-once or at-least-once semantics with strong exactly-once semantics. Third, it disaggregates the execution backend from recovery, allowing independent deployment and scaling.

Previous workflow systems provide exactly-once semantics but with significant limitations. For generality, workflow systems such as Apache Airflow [5] assume that each task is nondeterministic and may have side effects on external systems that in general cannot be rolled back. Thus, each task must synchronously checkpoint its outputs *before* they can be made visible to any downstream tasks. Otherwise, the system may have to re-execute the task in case of a failure. If the re-execution produces a different result, this can cause an inconsistent view among downstream tasks and external systems.

Thus, by assuming the worst, the workflow system has only one option of ensuring fault tolerance: no task can start before its upstream tasks have finished checkpointing all of their outputs. This limits the workflow system's ability to incorporate key optimizations often employed by application-specific frameworks that exploit the application's semantics. For example, large datasets passed between tasks can often be deterministically regenerated, making checkpointing unnecessary. In addition, while some tasks may indeed have external effects, e.g., starting a transaction on an external database, some effects can also be rolled back, e.g., by aborting the transaction.

Our goal is to hand control over recovery to Exoflow and ultimately the end user. Thus, we use two key interfaces to enable awareness of application semantics. First, we extend the typical workflow DAG API with pluggable *first-class references* to enable more flexible workflow-internal communication. A workflow task can return references to its outputs, which the workflow system then passes to downstream tasks. In contrast, current workflow systems require the application to pass data by

explicitly copying and checkpointing, which can be expensive for large data, or implicitly through external storage, which makes it difficult to guarantee exactly-once semantics. By using references to capture arbitrary data movement between workflow tasks, Exoflow leverages third-party systems' existing communication and recovery mechanisms while retaining control over workflow-level recovery.

Second, we introduce user annotations that specify relevant task semantics, i.e. whether to checkpoint a task, whether the outputs are deterministic, and whether the task has externally visible outputs. Before execution, Exoflow checks the safety of the user's specification. During execution, Exoflow synchronizes task execution and checkpointing. During recovery, Exoflow coordinates *rollback*, e.g., deletion of outputs from a previous execution, and task replay. For example, before executing a task with an externally visible output, Exoflow will first synchronize upstream checkpoints to *commit* any nondeterministic outputs, i.e. ensure they will never be rolled back. This allows the user to flexibly and safely optimize the recovery technique.

Exoflow is built on Ray [145] and consists of a per-workflow centralized controller, a pluggable checkpoint storage, and a pluggable execution backend. Centralizing controller logic makes it simple to guarantee recovery correctness. Meanwhile, checkpointing and execution are fully disaggregated, allowing these to be scaled independently of the controller.

We demonstrate the benefits of Exoflow with two execution backends, the Ray framework and AWS Lambdas, both distributed frameworks that provide at-most-once or at-least-once tasks. We show that references can enable $\sim 5\times$ speedup for Spark data processing workflows compared to Apache Airflow, while task annotations enable 51% lower latency for transactional serverless workflows compared to Beldi [216]. These optimizations are possible because correctness is ultimately guaranteed by Exoflow. These results also demonstrate Exoflow's *universality*, as the system is not specific to data processing or serverless environments. In summary, our contributions are:

1. Decoupling execution from recovery to enable a flexible tradeoff between performance and fault tolerance.
2. Designing a universal workflow system that guarantees exactly-once DAG execution.
3. Demonstrating benefits for a diverse set of applications, including an ML pipeline, serverless transactions, and graph processing that mixes stream and batch execution.

6.2 Motivation

6.2.1 Overview of recovery strategies

We use *exactly-once semantics* as our correctness condition. This condition often implies application-specific correctness properties, such as global consistency in message-passing systems [78] or linearizability in storage systems [103].

More precisely, exactly-once semantics require all outputs to appear consistent with a physical execution where all inputs were processed without failures. In a workflow setting, the inputs are the DAG and the root task arguments. Outputs are values produced by a task that are viewed by others.

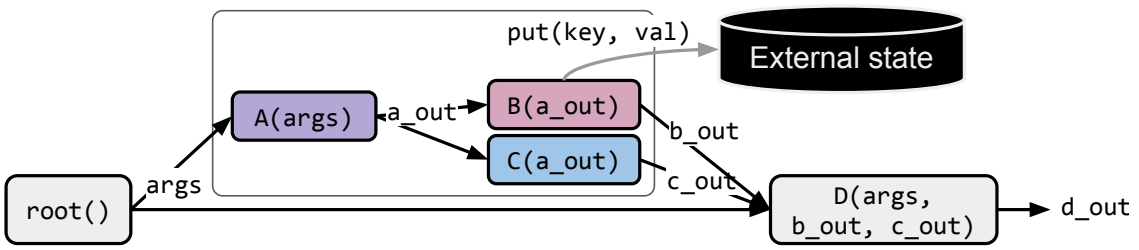
Output visibility can be *internal* or *external*. For example, values passed between tasks in Figure 6.1a are internal because they are viewed only by other tasks. Meanwhile, `(key,val)` is external because it is sent to a key-value store. Once outputs are made external, the workflow system no longer has control over how they will be used, e.g., via reads from external key-value store clients. Outputs can also be either *deterministically* or *nondeterministically* generated.

Output visibility and determinism are important because together they determine the recovery procedures that will guarantee exactly-once semantics (Figure 6.1b). For example, consider the cases if `A` is nondeterministic and we do not checkpoint `a_out` in Figure 6.1a. Suppose `C` views an initial value $\texttt{a\_out}_1$ and produces $\texttt{c\_out}_1$, but we lose $\texttt{a\_out}_1$ due to a failure. If we re-execute `A` to produce $\texttt{a\_out}_2$ and pass this to `B`, the outputs of `B` and `C` will not be consistent with a failure-free execution. To handle this case, we also need to "rollback" $\texttt{c\_out}_1$ and re-execute `C` on $\texttt{a\_out}_2$.

We encounter additional problems in the opposite case where `B` finishes and we then lose $\texttt{a\_out}_1$. `B` has already made `(key,val)` external and these values may depend on $\texttt{a\_out}_1$. If we execute `C` on $\texttt{a\_out}_2$, `c_out` will be inconsistent with `(key,val)`. Thus, the only way to guarantee correctness in this case is to either: (1) "commit" $\texttt{a\_out}_1$ before executing `B`, e.g., by checkpointing it, or (2) gain application semantics about how to roll back visibility of `(key,val)`.

Meanwhile, deterministic outputs are safe to view as long as the task can be replayed on its original inputs and recomputed outputs can be deduplicated. The external output in Figure 6.1a can for example be deduplicated by attaching a deterministic `req_id`.

Solution space. Handling nondeterministic outputs is generally done in two ways: (1) global checkpointing and rollback on failure, or (2) logging and deterministic replay on failure [78]. Both "commit" a prefix of a failure-free execution by saving the outputs of a task frontier, allowing recovery to resume execution from a consistent set of intermediate outputs. Global checkpointing advances this frontier several tasks at a time and upon failure, rolls back to the last frontier to undo partially visible



(a) Workflow DAG

	Internal	**External**
Nondeterministic	Commit output OR on failure, rollback visibility	Commit output *before* visibility OR if possible, rollback visibility on failure
Deterministic	Replay failed task(s) on previous inputs, dedupe outputs	Also dedupe external outputs

(b) Recovery strategies for workflow DAGs

Figure 6.1: **(a)** An example workflow with internal outputs (e.g., `a_out`) and external outputs (e.g., `put(key,val)`). **(b)** The most efficient recovery strategy depends on output visibility and nondeterminism.

nondeterministic outputs. For outputs that cannot be rolled back, however, upstream nondeterministic outputs must first be committed by taking a global checkpoint. Logging-based methods advance the frontier one task at a time by committing each nondeterministic output before making it visible, thus avoiding additional rollback on failure.

Note that rollback and durability options vary based on output visibility. External outputs may be impossible to roll back, e.g., a transaction commit cannot be undone, or make durable, as third-party system context is not always serializable.

Current workflow systems guarantee exactly-once semantics by: (1) durably checkpointing each internal output before making it visible, and (2) requiring the developer to make external outputs idempotent and durable. This one-size-fits-all approach does not leverage application-specific recovery methods (Figure 6.1b). Furthermore, existing workflow systems have fundamental limits on internal outputs, usually because they must be sent between tasks through the workflow controller. Apache Airflow uses a database to coordinate tasks, which imposes a maximum output size on the order of MBs [5], and direct task communication in FaaS is limited [84]. Together, these force developers to use external outputs for much of their task communication [84, 163].

Our goal is to support different recovery methods in a single workflow system and even within a single application. The key insight behind Exoflow is that knowing the DAG structure makes it simple to identify a consistent execution frontier, allowing the recovery methods before and after the frontier to be decoupled. For example, `a_out`

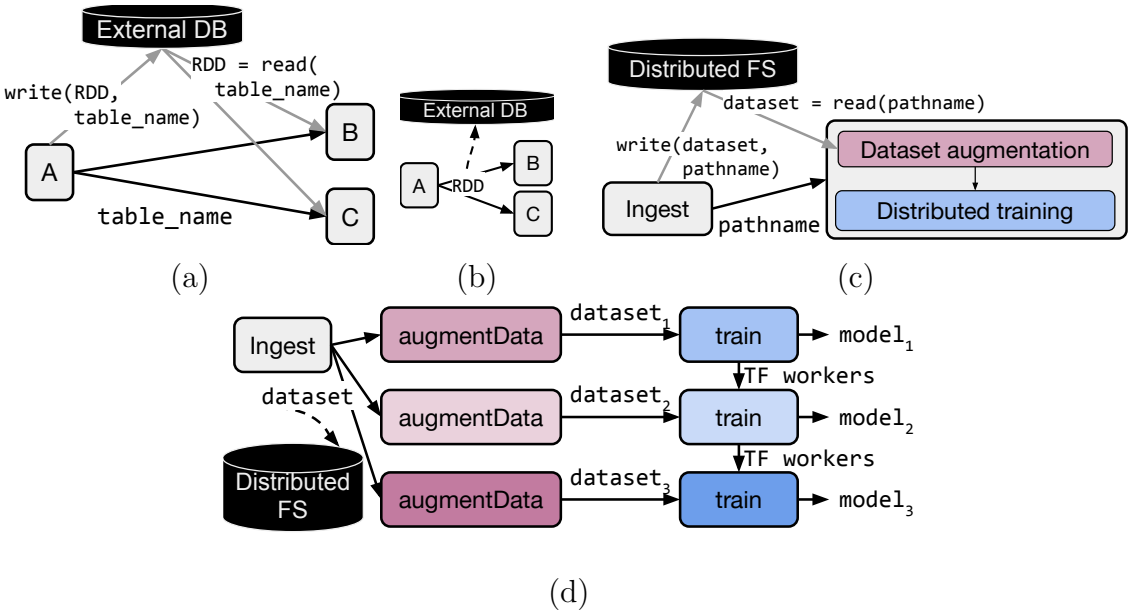


Figure 6.2: **(a)** ETL workflow today, using external outputs for communication. **(b)** The same ETL workflow with internal outputs only. **(c)** ML training workflow today, with external outputs and manual orchestration within a task. **(d)** The same ML workflow with internal outputs only, and orchestration is handled by the workflow system. Third-party framework state (`TF workers`) can be passed between workflow tasks.

is internal to the outlined sub-DAG in Figure 6.1a and thus its recovery method can be chosen flexibly as long as the inputs (`args`) and outputs (`b_out,c_out,key,val`) are consistent.

Thus, our solution consists of two parts. First, *references* enable Exoflow to capture a broader range of inter-task communication as internal outputs, without being involved in the physical communication. This encourages recovery flexibility within a sub-DAG and recovery independence across sub-DAGs. References enable efficient passing of task outputs of any size and location as well as outputs that may not be serializable.

Second, we support *annotations* to specify task semantics (checkpointing, nondeterminism, output visibility). These allow the system to determine recovery correctness before execution. The system "commits" the application to this specification by durably logging the DAG before execution, then coordinates and synchronizes task checkpoints during execution. The annotations are set to a safe default, i.e. each task's output(s) must be checkpointed, is assumed to be nondeterministic, and any external outputs must be made idempotent. This produces write-ahead logging behavior equivalent to that of a workflow system such as Apache Airflow.

6.2.2 Applications

We use three representative applications to show the value of: (1) making workflow-internal outputs more flexible, and (2) exposing application semantics to the workflow controller:

1. Extract-transform-load (ETL) pipelines: Using references to pass large data as internal outputs.
2. Machine learning (ML) pipelines: Using references to pass large data and leveraging application semantics.
3. Serverless workflows: Leveraging application semantics to reduce recovery overheads, in a way that is agnostic to external systems.

ETL pipelines. Workflow systems such as Apache Airflow are commonly used to orchestrate extract-transform-load (ETL) pipelines composed of data processing jobs. Figure 6.2a shows an example in which a Spark job `A` performs batch data cleaning and writes the data to an external database, e.g., Delta Lake [30]. Jobs `B` and `C` then load the data for querying.

Current practice for exactly-once workflow execution requires all of `A`'s outputs to be made durable *before* executing `B` and `C`. Synchronous checkpointing adds high overhead for large and distributed data. In addition, `B` and `C` must each reload the data, imposing an unnecessary memory copy. This is of course unnecessary if `A` is deterministic. Execution systems such as Spark leverage this property to natively support distributed in-memory caching. Ideally, `A` should pass its output as a cached RDD [212] to `B` and `C` (Figure 6.2b), avoiding the round trip to external storage, allowing `B` and `C` to share physical memory, and enabling asynchronous checkpointing.

Building such optimizations into a workflow system would enable orchestration of arbitrary DAGs and third-party frameworks. However, even with awareness of task determinism, current workflow systems cannot execute Figure 6.2b due to limitations in workflow-internal data passing.

ML pipelines. Machine learning (ML) pipelines are similar to ETL pipelines, but with an ML application as the end consumer. This requires composition of traditional ETL systems with distributed ML frameworks for training and inference. Figure 6.2c shows a typical ML training workflow, in which training data is extracted and transformed in the `Ingest` task, then consumed by a distributed training job. Loading data into the training job may itself require complex and possibly distributed data processing, with computations such as random transforms to augment datasets [149]. Furthermore, datasets are often large enough that preprocessing must be pipelined with training to maximize GPU utilization.

Current workflow systems cannot effectively orchestrate within the training task, as training data and worker state must be passed through distributed memory. Expanding workflow-internal outputs would enable workflows such as Figure 6.2d. To

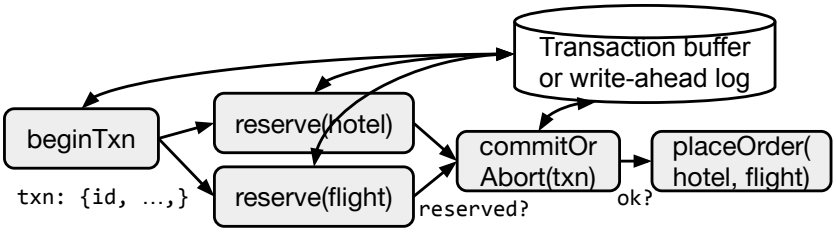


Figure 6.3: Serverless workflow systems [188, 216, 110] guarantee exactly-once semantics by interposing on all communication to external storage, e.g., through a transaction buffer, and explicitly managing visibility of these external effects.

reduce the overhead of recovery, however, the workflow system also requires application semantics, such as whether dataset augmentation is deterministic. Also, the model output can be consumed in a variety of ways, from local one-off testing during development to deployment on an ML serving system during production. All of these factors affect the optimal correct recovery strategy.

Serverless workflows. In the functions-as-a-service (FaaS) model, the user breaks their application into small functions that can be transparently executed and scaled without explicit resource provisioning. Serverless functions have a limited lifetime, all local state is transient, and failure handling is usually limited to function retries. This makes it challenging to build fault-tolerant nontrivial applications directly on FaaS [101].

Recently, *serverless workflow* systems [52, 188, 216] have gained popularity as a solution, especially for stateful applications. A common strategy for guaranteeing exactly-once execution is to provide fault-tolerant APIs to capture external outputs. For example, Figure 6.3 shows an example of a trip reservation workflow [87] that places the order if and only if both the hotel and flight were successfully reserved. Systems such as Aft [188], Beldi [216], and Boki [110] guarantee exactly-once semantics by providing a transactional key-value store to manage external output visibility.

However, each system offers different isolation levels that require different recovery strategies. Aft buffers uncommitted writes, which are safe to rollback, while Beldi and Boki use write-ahead logging. Thus, each system implements their own recovery procedures, e.g., durability and task re-execution.

Exoflow factors out workflow recovery to enable flexibility and optimizations. Instead of providing opinionated APIs for external outputs, we treat external systems such as the transaction buffer in Figure 6.3 as a black box. Exoflow does not interpose on the communication to this external system and instead requires that the application can specify task semantics such as whether the external effect can be rolled back. These semantics can be specified by a particular transaction system, i.e. Aft or Beldi.

Workflow API	Semantics
`f.options(Opts).bind(Value | WorkflowDAG) → WorkflowDAG`	Create a workflow task `f`. Creates and returns a `WorkflowDAG`, whose value is lazily evaluated. The caller may pass the `WorkflowDAG` to another task. The return value of `f` can be a `WorkflowDAG`, i.e. a nested workflow.
`run(WorkflowDAG w, str name) → Value`	Run the workflow `w` and return the result. Optionally take a string identifier for this workflow.
`run_async(WorkflowDAG w, str name) → Fut`	Run the workflow `w` asynchronously and return a future that can be used to retrieve the result.
`Ref.get() → Value`	Used by the application to dereference to a value. `Ref` construction is backend-specific.
`bool Opts.checkpoint=True`	True if the task's output should be saved.
`bool Opts.deterministic=False`	True if outputs are deterministically generated.
`bool Opts.can_rollback=False`	True if task has no external outputs, or if they can be rolled back. If False, the task must be idempotent.
`Fn Opts.rollback=null`	If external outputs can be rolled back, a function to do so. The function must be idempotent, and any `WorkflowDAG` arguments must be a subset of the original workflow task `f`'s arguments.
`Ref._id() → ID`	Used by the workflow system to compare equality.
`Ref._checkpoint() → Fut[Value]`	Used by the workflow system to coordinate checkpointing. The `Value` is the checkpoint data or metadata.
`Ref._restore(Value)`	Used by the workflow system to reload from a saved checkpoint.

Table 6.1: Workflow API. Top: API calls exposed to the application. Middle: Task annotations specified by application or third-party library. Bottom: Exoflow-internal `Ref` API, pluggable by execution backend.

6.3 API

6.3.1 Overview and requirements

Exoflow is a general workflow layer that abstracts a workflow *backend*, i.e. a distributed framework providing at-least-once and/or at-most-once remote function invocation. We overview the application-facing API (Table 6.1) and requirements. The application must be able to: (1) differentiate deterministic tasks, and (2) for tasks with external outputs, ensure that the task is idempotent or specify an idempotent rollback function.

DAG interface. The application invokes workflow tasks and specifies arguments using `f.bind` (Table 6.1). The caller receives a `WorkflowDAG` that represents the task's output and that can be passed to other tasks as dependencies. Workflow execution is *lazy*: to evaluate a `WorkflowDAG`, the developer must `run` it. This is to simplify

recovery, as the workflow system can check DAG-level properties before executing it.

The workflow backend should implement an RPC-like interface. Within a task, the application can invoke arbitrary local or distributed execution. For greater generality, we also adopt the *dynamic* task model [148]: tasks can dynamically invoke exactly-once nested workflows by returning a `WorkflowDAG`.

Task annotations. The application specifies semantics relevant to recovery at task invocation time (Table 6.1). The workflow system uses these to ensure correctness of: (1) coordination of distributed workflow checkpoints during execution, and (2) output rollback and task re-execution upon failure.

First, the application specifies whether to skip checkpointing a task's output. Note that the workflow system guarantees correctness, so this can be considered an optimization hint, e.g., to avoid recomputation for long tasks,

Next, the application can specify whether a task's outputs (both internal and external) are deterministic. This allows the workflow system to minimize rollback during recovery.

Finally, the application specifies whether a task can be rolled back and if yes, how to do so. Tasks with no external outputs, such as the data processing tasks in Figure 6.2, should set `can_rollback=True`. Tasks that have external outputs that cannot be rolled back should set `can_rollback=False` and ensure idempotence, as recovery may require re-execution.

Non-idempotent tasks with external outputs that can be rolled back should set `can_rollback=True` and the `rollback` callback. On failure, Exoflow executes these rollback "tasks" in reverse dependency order before resuming execution. The rollback task can take any arguments available to the original workflow task, but the application must additionally guarantee that the rollback task is idempotent. For example, to implement the transaction in Figure 6.3, rollback for the `beginTxn` and `reserve` tasks could simply abort.

On `run`, Exoflow checks the `WorkflowDAG` for specification errors and throws an exception if any are found. In particular, correctness requires the application to set checkpoints between each nondeterministic task and each downstream task with external output. Section 6.3.3 makes this precise.

Internal outputs. Direct task outputs are subject to limits of the execution backend. For greater flexibility, Exoflow allows outputs to include `Refs` created by the task. `Refs` are (optionally) pluggable by the execution backend. They are intended to capture volatile outputs that would be expensive or complex to natively support in Exoflow, e.g., large distributed data or third-party framework context. For an AWS Lambdas backend, for example, values can be stored in an external (volatile) key-value store and the key can be passed in a `Ref`. Other tasks can dynamically `get` the value, which can throw an error if the value is irretrievable due to failure.

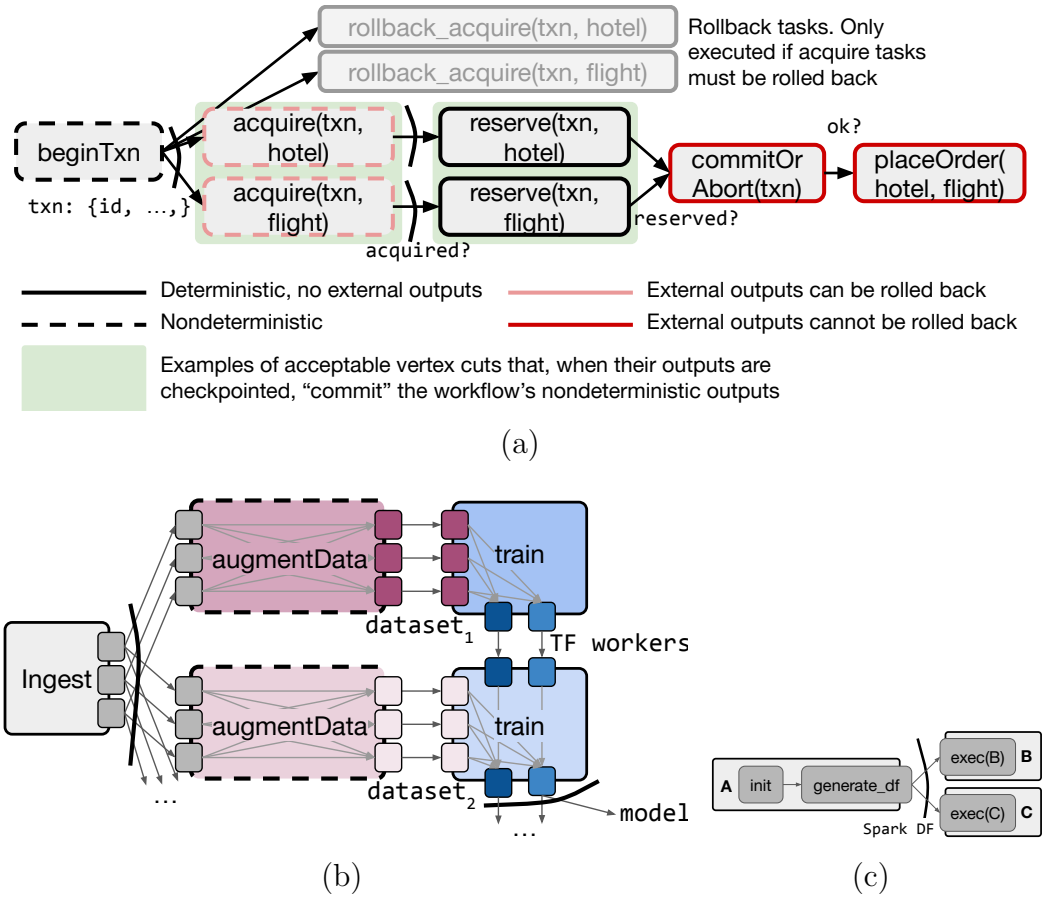


Figure 6.4: **(a)** Task annotations. Edge cuts represent `checkpoint=True`. **(b)** Passing references (small boxes) in an ML workflow. Blue `Refs` are actors that wrap TensorFlow worker state. **(c)** Passing an `ActorRef` in an ETL workflow. `B` and `C` call read-only methods on the Spark context actor.

`Refs` are uniquely identifiable objects typically containing backend-specific metadata. A task can only return `Refs` that it created or that were passed to it by an upstream task. Then, upon failure, Exoflow can either restore the `Ref` from a checkpoint, or trace the DAG back to the creating task. On re-execution, the task need not return the same `Refs` as its original execution. For example, with the annotation `deterministic=True`, it is only necessary that the *value* of a returned `Ref` is deterministic; the `Ref` itself may have a nondeterministic ID. This is safe because Exoflow simply cancels tasks using the previous `Refs` and re-executes with the new `Refs`.

By default, `Ref` values are *immutable*. This improves recovery efficiency, as it simplifies checkpointing and minimizes task rollback. To capture task outputs that are expensive or impossible to materialize, we also support *stateful* references, i.e. *actors* [104]. An `ActorRef` extends `Refs` with application-defined methods that

execute on the actor's state (Listing 3). However, mutable state is more complex to recover efficiently and correctly. Thus, compared to `Refs`, we limit how `ActorRefs` can be passed between workflow tasks (Section 6.3.4).

6.3.2 Model

We present a formal model of workflows to more precisely capture the API and assumptions. A *workflow* $G = (V, E)$ is a directed acyclic graph with vertices V and edges E. Each vertex v_i has:

- F_i: An associated function
- $\mathcal{N}_i$: A function representing a (potential) source of nondeterminism
- $\mathcal{R}_i$: An optional rollback function
- The set of annotations described in Table 6.1.

A workflow execution produces one internal and one external output per vertex, both optional. For brevity, the presented model only considers tasks with single outputs, although the system in reality supports multiple outputs.

We denote an execution's outputs by O_{Int} and O_{Ext}. O is a mapping from vertex to a single output value o, and the subscripts Int and Ext denote internal and external outputs, respectively. F_i outputs o_{Ext} by adding it to a global set $\mathcal{W}$, which can be read by other tasks and by external processes.

Each F_i takes as inputs:

- $args_i$: Direct arguments, one for each vertex with an edge to v_i.
- w_i: A set of external outputs.
- n_i: A nondeterministic value. If F_i's output does not depend on n_i, then F_i is deterministic.

In other words, an edge (v_i, v_j) indicates that v_i's internal output is passed to task v_j. Internal outputs passed between vertices are analogous to messages passed between processes in a message-passing model [78], except that the application must declare the "messages" (dependencies) before execution.

$\mathcal{N}_i$ captures nondeterministic inputs. For example, if F_i depends on the current time, then $\mathcal{N}_i$ returns the current time. We assume that if $\mathcal{N}_i$ reads some external state, the external state will not be rolled back (unless F_i is also rolled back via $\mathcal{R}_i$).

We define a *failure-free* execution of G as one where the individual output of each task v_i corresponds to an execution of F_i over inputs such that:

- The direct arguments are the internal outputs produced by vertices with an edge to v_i. Formally, this can be written as $args_i = \{O_{Int}[j] \mid (v_j, v_i) \in E\}$.
- The set of external outputs is equal to the external outputs of all tasks that precede v_i in G. Formally, this can be written as $w_i = \{O_{Ext}[j] \mid v_j <_G v_i\}$.

- The nondeterministic value is one returned by $\mathcal{N}_i$, i.e. $n_i = \mathcal{N}_i()$.

The correctness condition says that to an external process, it must appear as if the DAG has executed failure-free. Thus, we also define W: a sequence of snapshots of the external outputs produced so far by the DAG execution. W represents a series of reads of $\mathcal{W}$ made by an external process during execution. Then, we just need to make sure that once an external output is visible, i.e. it appears in a snapshot w in W, it should be visible in all following snapshots in W. In other words, W must be monotonic.

This definition is analogous to *global consistency* in message-passing [78], i.e. that every visible output has a corresponding task that created it. The goal is to provide a consistent execution under a crash failure model. Formally, we can define this as:

Definition 6.1 (Consistency)**.** *O_{Int}, O_{Ext} are consistent with a workflow $G = (V, E)$ if for all possible W, W is monotonic and the outputs O_{Int} and O_{Ext} correspond to a failure-free execution of G.*

The application assumptions are as follows. For each v_i:

1. We assume that if v_i and v_j cannot be ordered in the graph, then they cannot read each other's external outputs. Formally, if $v_i \not<_G v_j$ and $v_j \not<_G v_i$, then $F_i(I_{Int}, w_i, n_i) = F_i(I_{Int}, w_i \setminus \{O_{Ext}[j]\}, n_i)$. If the application requires v_i to depend on a task v_j's external output, then the ordering should be specified as part of the task graph. If this is not possible, then to ensure consistency, v_j's external output should be considered part of v_i's nondeterministic input, and the application must set `can_rollback=False` for v_j.

2. Tasks that set `deterministic=true` must produce outputs that are a deterministic function of their internal and external outputs, i.e. F_i is not dependent on the value returned by $\mathcal{N}_i$.

3. If the o_{Ext} returned by F_i is not null, then either `can_rollback=False` or $\mathcal{R}_i$ is not null.

   (a) If `can_rollback=False`, then F_i is idempotent. That is, if an invocation of F_i produces an external output o_{Ext}, and F_i is run again on the same internal outputs and a later snapshot of $\mathcal{W}$, then F_i should still produce the same external output.

   (b) If $\mathcal{R}_i$ is provided, then it is a deterministic and idempotent function of the task's *internal inputs* only. Intuitively, $\mathcal{R}$ removes the previous external output from the external world. Formally, this means that if the first invocation of $F_i(I_{Int}, w, n_i)$ produces (o_{Int}, o_{Ext}), then $\mathcal{R}_i(I_{Int})$ removes o_{Ext} from all past reads of $\mathcal{W}$.

Regarding (3b), note that the meaning of removing o_{Ext} from past reads is application-dependent. For example, suppose F_i executes a transaction and $\mathcal{R}_i$ aborts the transaction; if uncommitted reads are allowed, then $\mathcal{R}_i$ does not need to roll back the reader.

Nested tasks and references. While not explicitly captured in the above model, nested tasks can be thought of as tasks that expand into a sub-workflow. `Refs` and `ActorRefs` are native data types that can be returned in a function's internal output. Because actors are mutable, `ActorRefs` are versioned: if a caller writes to an actor by calling a method on its `ActorRef`, the caller's resulting `ActorRef` is of a different version. This becomes relevant in Section 6.3.4, which discusses the rules that the application must follow to ensure exactly-once semantics when `ActorRefs` are passed between workflow tasks.

6.3.3 Guaranteeing exactly-once execution

Task annotations simplify the decision of when to commit task outputs. To illustrate this, we use Figure 6.4a, a modified version of the workflow described in Figure 6.3. We show the annotations for a workflow using an external two-phase locking (2PL) transaction system. `beginTxn` generates a transaction context with a random `txn_id`. The `acquire` tasks each attempt to acquire a lock on an external table row. If this is successful, we attempt to `reserve` the flight and hotel if available, then finally commit the transaction and place the order if both succeed. The cuts in Figure 6.4a indicate `checkpoint=True`.

As an example, we first consider the `acquire` and `commitOrAbort` tasks. `acquire` tasks are nondeterministic because they depend on the run-time state of the external table. `commitOrAbort` has `can_rollback=False` because it is impossible to abort a committed transaction and vice versa. Although `acquire` can be rolled back (e.g., by aborting the transaction and releasing the lock), once we have started the `commitOrAbort` task, it is no longer safe to do so because the transaction may already be committed. Thus, we must ensure that both `acquire` outputs are saved before `commitOrAbort` starts. We can generalize this rule for the application as follows:

Invariant 6.1 (External output commit)**.** *For each workflow task v_i with `deterministic=False`, let G be the minimal subgraph that contains v_i and all downstream tasks (tasks for which there is a path from v_i). Then, for each workflow task v_j with `can_rollback=False` in G, there must exist a vertex cut that partitions v_i from v_j such that all tasks in the cut have `checkpoint=True`.*

Intuitively the vertex cut (green shaded box in Figure 6.4a) of the sub-DAG defines a commit point for the nondeterministic output of v_i. There may exist multiple such cuts. For example, another acceptable specification in Figure 6.4a is the righthand vertex cut, which instead checkpoints the `reserve` outputs.

Exoflow guarantees that at least one task frontier is fully checkpointed by the time `commitOrAbort` (v_j) starts. Interestingly, this also tells us that we do not need to commit the `acquire` outputs synchronously. In particular, the `reserve` tasks in this case are deterministic, as their outputs depend only on whether the lock was acquired and the value stored in the external table, which cannot be modified while locked. Furthermore, their external outputs are not visible while the lock is held. Thus, in this case, it is safe to annotate the `reserve` tasks with `deterministic=True` and `can_rollback=True`. Together, these annotations allow Exoflow to *overlap* the checkpoint of `acquire`'s outputs with execution of the `reserve` tasks, as long as the checkpoints are synchronized before `commitOrAbort`.

There is a similar requirement for rollback tasks. The rollback tasks in Figure 6.4a are conditionally invoked by the workflow system to undo external outputs of the `acquire` tasks. We must ensure that all inputs to the original `acquire` task are recoverable *before* execution. Otherwise, if the rollback task and its inputs fail simultaneously, it will be impossible to finish rollback. Thus, in Figure 6.4a, the application must set `checkpoint=True` for `beginTxn`, and Exoflow synchronizes this checkpoint before executing the `acquire` tasks.

Invariant 6.2 (Rollback durability)**.** *For each path beginning at a task v_i with `deterministic=False` and ending at a task v_j that has a rollback function R_j, there must exist at least one vertex along the path with `checkpoint=True`.*

Unlike Invariant 6.1, here we only require checkpointing a single task to handle nondeterminism, as the availability of a rollback function R_j means that we do not need to commit to the original output. The checkpointed task can also be a task other than v_i or v_j. For example, if there were additional deterministic tasks between `beginTxn` (T) and `rollback_acquire` (R), then checkpointing any is sufficient.

Both invariants can be easily checked by walking the DAG passed to `run`. If an invariant is not met, the system throws an exception to the user. Annotations do therefore require user cooperation, but note that a user with minimal performance needs can use the defaults in Table 6.1. This specification trivially satisfies the invariants and indeed corresponds to current workflow systems that commit all task outputs. Section 6.4 describes how Exoflow leverages the invariants to improve run-time performance for more sophisticated specifications.

Note that the system will not durably record a nested workflow returned by a task with `checkpoint=False`. To simplify recovery, we disallow sub-tasks with `checkpoint=True`, as we may lose all references to these checkpoints upon failure. We also disallow `can_rollback=False` and `rollback`, as these are challenging to recover without workflow durability.

6.3.4 References

```
@ray.remote
class SparkActor:
  def __init__(self):
    self.spark_context = connect(); self.df = None
  def generate_df(self):
    self.df = generate_df(self.spark_context).cache()
  @const
  def exec(self, seed: int) -> int:
    return exec(self.df, seed=seed).count()
  def _checkpoint(self):
    return self.spark_context.save(self.df)
  def _restore(self, path):
    self.df = self.spark_context.load_df(path)
```

Listing 3: Psuedocode for passing a Spark DataFrame by actor. The execution backend implements the actor. Public methods are user-defined. Methods prepended by _ are called internally by Exoflow.

Immutable `Refs` enable efficient passing of large and distributed data between workflow tasks. For example, Figure 6.4b shows how the `Ingest` task from Figure 6.2d can use `Refs` to return distributed in-memory data. Exoflow tracks inter-task `Ref` dependencies for recovery purposes, while the execution backend handles intra-task execution (e.g., `get`).

Some cases require stateful actors for performance. For example, the blue boxes passed between train tasks in Figure 6.4b are `ActorRefs` representing a training worker's state, e.g., a Distributed TensorFlow session. This helps avoid expensive materialization, such as the worker's local model copy.

Guaranteeing exactly-once semantics for state is challenging. If one task writes the `ActorRef`'s state, the output is visible to any other task holding a reference to the same actor. This can cause cascading rollbacks on failure depending on how `ActorRefs` are passed. Furthermore, checkpointing is more challenging if multiple tasks write concurrently to the actor, as the system must ensure that the actor checkpoint is consistent.

To simplify recovery, we limit `ActorRef` passing to two patterns, analogous to a read-write lock. By default, the `ActorRef` is in "write" mode. In this mode, only one workflow task may have a reference to the actor at a time. That task can call any actor methods as long as they finish before the task returns. For example, in Figure 6.4b, only one train task refers to each actor at a time. Exoflow can then checkpoint the actors' state between tasks, and on failure, roll back the actors with the workflow. This pattern is useful for abstracting and checkpointing distributed workers in third-party frameworks such as Distributed TensorFlow [18] and Flink [58].

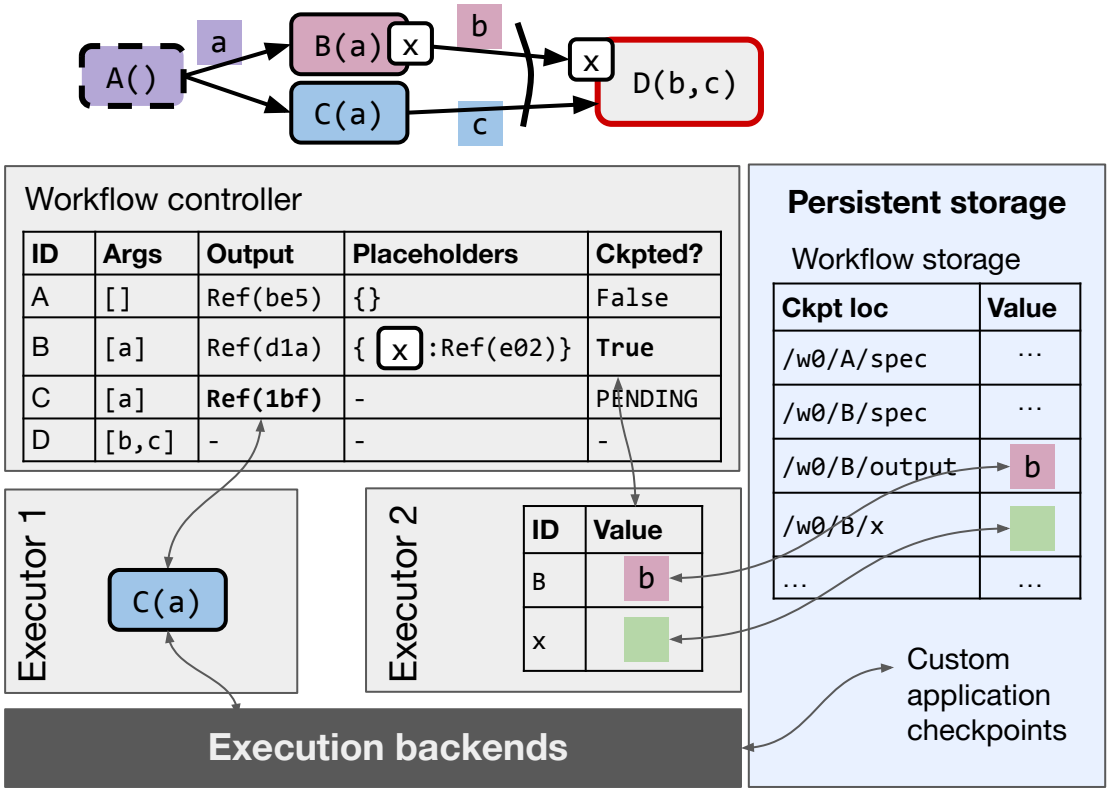


Figure 6.5: Workflow architecture. The controller and executors are RPC-like services built using Ray actors. Each invocation on these services returns a distributed future (system-internal `Refs`).

If there are multiple concurrent workflow tasks with a reference to the same actor, however, the tasks are restricted to read-only methods annotated by the user, as shown in Listing 3. Figure 6.4c shows an expanded Figure 6.2b in which we use an `ActorRef` to capture a Spark DataFrame. Initially, `A` has the only `ActorRef`, so it can write to the actor's state (`generate_df`). `B` and `C` share the actor concurrently, however, and so they are limited to read-only methods (`exec`). Invoking a write method such as `generate_df` would throw a run-time error.

Similar to a read-write lock, Exoflow can only provide correctness if the application respects certain conditions. In particular, the workflow tasks must explicitly pass `ActorRefs` through their outputs and arguments. Any other `ActorRefs` cannot be tracked by Exoflow and exactly-once semantics is not guaranteed, similar to reading a variable without holding the lock. Also, while methods may be called asynchronously on an `ActorRef`, a workflow task must synchronize any outstanding calls to an actor before returning.

6.4 Architecture

The Exoflow architecture (Figure 6.5) comprises a logically centralized workflow controller, a pluggable execution backend, and a pluggable persistent storage system.

The Exoflow controller is a long-running service that can be sharded by workflow (Figure 6.5). Persistent storage can be implemented by any durable blob storage supporting puts and gets with read-after-write consistency, such as Amazon S3. The execution backend should implement a *remote function invocation* interface, used by the controller to scale checkpointing and task execution. The backend should provide: (1) ability to detect and report task and `Ref` failures, and (2) guarantee no resource leaks for failed task execution and `Refs`.

The controller runs as an event loop with the following events: task or checkpoint completes, and task or checkpoint failed. All critical workflow state, such as the workflow DAG, is cached by the workflow controller and written-through to persistent storage, making it simple to also recover the workflow controller. Checkpointing is carried out asynchronously by background threads on the executors, enabling parallel and distributed checkpoints that are not bottlenecked by the centralized controller. The Exoflow controller coordinates checkpoint synchronization during execution as needed, according to the user-defined annotations. Then, on restart, the controller simply scans the storage for any unfinished workflows, coordinates rollback as needed, and re-`runs` to completion.

See Appendix C for a full description of the execution and recovery procedures, including correctness arguments and implementation details.

6.5 Evaluation

Our evaluation covers the following questions:

1. How can applications leverage first-class references and task annotations to have greater flexibility in recovery?

2. How does this flexibility in recovery strategy affect performance during execution and recovery?

Appendix C includes additional end application evaluation, as well as microbenchmarks evaluating:

1. What overheads does Exoflow add to at-least-once or at-most-once execution backends?

We compare primarily against these baselines: (1) exactly-once workflow systems: Airflow [5], "standard mode" AWS Step Functions [50], and the serverless

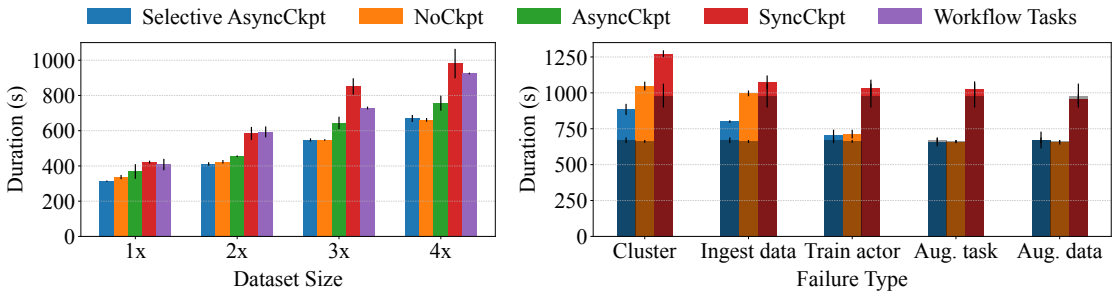


Figure 6.6: End-to-end duration for the ML workflow application shown in Figures 6.2d and 6.4b. **Left:** End-to-end duration without failure. **Right:** End-to-end duration with different failure types. The shadow represents the execution time without failure.

workflow system Beldi [216]; and (2) at-least-once distributed DAG systems: "express mode" AWS Step Functions [50] and Ray [145].

Given the high execution overheads of exactly-once workflow systems such as Airflow (Appendix C.3.2), to fairly address questions (1) and (3), we also compare against the following Exoflow modes:

1. `SyncCkpt`: Task outputs are synchronized before executing downstream tasks. This is used to simulate the recovery strategy of exactly-once workflow systems such as Airflow.

2. `NoCkpt`: All task outputs except the final are skipped. This is used to simulate the recovery strategy of an at-least-once or at-most-once system. The application must guarantee that all tasks are deterministic and idempotent to achieve exactly-once semantics.

3. `AsyncCkpt`: The default mode of Exoflow. Task outputs are only synchronized where necessary, to provide exactly-once semantics.

We conduct all of the experiments using the AWS cloud, specifically in the us-east-1 region. Exoflow and execution backends are hosted on EC2 and use Amazon S3 (or EFS in Section 6.5.2) for persistent storage.

6.5.1 ML training pipelines

We show how Exoflow enables a flexible recovery-performance tradeoffs for the workflow in Figure 6.2d. We use an image classification example adapted from Azure MLOps [7]. An ETL `Ingest` task (1 r3.2xlarge node) downloads the compressed data from S3. "1×" in Figure 6.6 indicates one data copy with 569 raw image files and

total size 225MB. The task loads the images into memory, and performs data cleaning and normalization with at-least-once parallel Ray tasks. The dataset (1.4GB of memory per data copy) is partitioned and passed using `Refs` to the dataset augmentation tasks, via Ray's shared-memory object store. Dataset augmentation again uses Ray at-least-once tasks to apply random cropping, flipping, and color adjustments to the base dataset, once per epoch. Dataset augmentation requires repeatedly processing the same dataset in a tight loop with training. Therefore, the dataset augmentation stage accumulates a total intermediate and checkpoint size of 67GB and 18GB respectively, per data copy. Training tasks are colocated and pipelined with dataset augmentation (1 g4dn.12xlarge node, 4 NVIDIA T4 GPUs). We use PyTorch data-parallel distributed training and the ConvNeXt Tiny (28.6M parameters) model. PyTorch workers are passed using `ActorRefs`.

Figure 6.6L shows end-to-end duration of 25 epochs without failures of different Exoflow recovery modes, as a function of dataset size. Here, we also include `Selective AsyncCkpt` (skip checkpointing dataset augmentation outputs) and `Workflow Tasks` (include at-least-once Ray tasks for data processing in the workflow DAG instead of passing volatile `Refs`).

Duration predictably grows approximately linearly with the dataset size for all strategies. The overhead of `Workflow Tasks` is high because each data processing task is durably (and unnecessarily) logged as part of the workflow. For the same workflow graph, the overhead for larger data varies depending on the recovery strategy. `NoCkpt` represents the best possible performance, where only the final model is checkpointed. `SyncCkpt` represents existing workflow systems (Figure 6.2c) and its overhead grows the most because checkpointing overhead grows faster than computation overhead. `AsyncCkpt`'s overhead grows less because checkpointing of augmented datasets is overlapped with training tasks. `Selective AsyncCkpt` has nearly identical duration as `NoCkpt` because the `Ingest` checkpoint is perfectly overlapped with training tasks.

Meanwhile, Figure 6.6R shows end-to-end duration in different failure scenarios compared to normal run-time execution (dark): whole cluster failure (including the Exoflow controller); in-memory ingest data lost; PyTorch worker actor lost; augmentation task lost; and in-memory augmented data lost. Here, we see the tradeoff between recovery and performance. `SyncCkpt` has similar or better recovery time overhead than `NoCkpt` for cluster and ingest data failures because it avoids re-executing the `Ingest` task, but overall it does worse because of high normal run-time overhead. `Selective AsyncCkpt` checkpoints the `Ingest` data asynchronously, so recovering from cluster and ingest data failures is fast because it simply restores the `Refs` from the checkpoint. Together, Figure 6.6L and R demonstrate how *the developer can flexibly choose the best recovery strategy.*

Figure 6.6R also demonstrates Exoflow's *broad failure coverage and ability to integrate with Ray's built-in recovery*: Ray automatically reconstructs deterministic data processing results but does not handle persistence or actor recovery [201]. Thus,

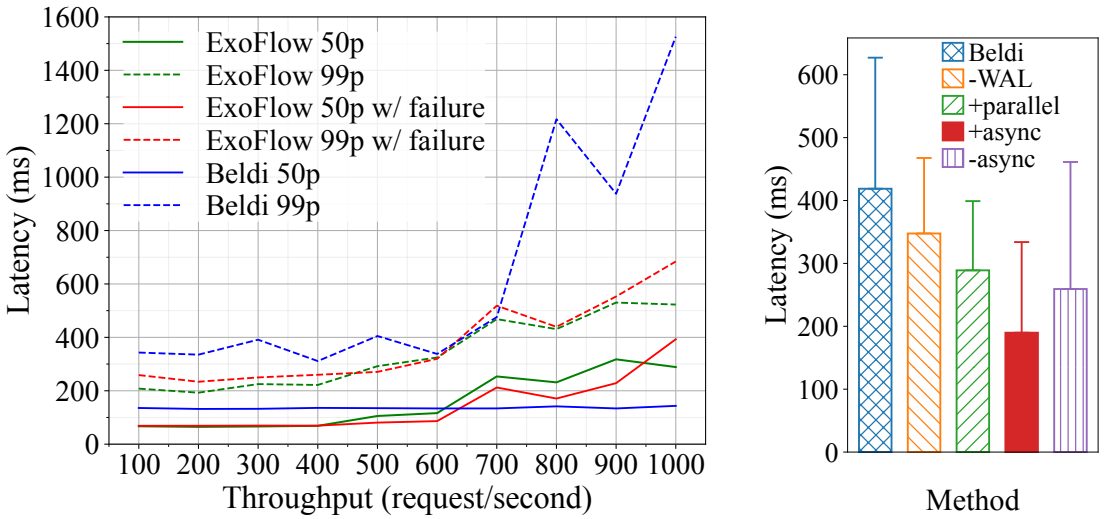


Figure 6.7: **(a)** Response latency percentile for a serverless travel reservation benchmark [87]. **(b)** Median latency of the trip reservation request from the travel reservation benchmark. Error bar represents 99-percentile latency.

Exoflow handles the first four failures, while Ray handles the last. Recovery for the last two failures is fast because rollback and checkpoint restore are unnecessary.

6.5.2 Stateful serverless workflows

We compare Exoflow on a travel reservation benchmark [87] to Beldi [216], a recent system for fault-tolerant and transactional stateful serverless workflows that uses *intent logging* to ensure exactly-once semantics. Our implementation uses Beldi's APIs for reading and writing state but the Exoflow controller with an AWS Lambdas backend for workflow execution and recovery. We use a single m5.16xlarge instance to host Exoflow and EFS for persistent storage, which provides lower latency than S3. The benchmark procedure follows [216], and we report response latency in Figure 6.7a.

Exoflow achieves about 51% lower p50 latency than Beldi for request rates up to 400, despite using the same execution system (AWS Lambdas) and state APIs (Beldi). This is because most of the workflows have deterministic computation and no external effects (i.e. read-only), so the additional logging used by Beldi is unnecessary for correctness. Furthermore, Beldi schedules an additional Lambda function to orchestrate others, while Exoflow directly schedules Lambdas[1]. When requests/s

[1]Note that unlike Beldi, Exoflow requires a server. However, because Exoflow's controller is fault-tolerant and horizontally scalable, it would be straightforward to deploy Exoflow as a serverless system using any autoscaling container orchestrator.

is higher than 700, Exoflow's median latency is greater than Beldi's. This is due to the Lambdas invocation bottleneck at the Exoflow controller node and can be easily removed through sharding across workflows. The Lambdas gateway used in Beldi is likely sharded internally.

The use of Exoflow as a Lambdas gateway has benefits in recovery time. Figure 6.7a also shows latency with a 10% failure rate for all Lambdas. Exoflow directly invokes Lambdas, so it can detect failures and recover virtually instantaneously, resulting in 0-31% extra overhead in p99 latency. In contrast, Beldi is fully decentralized and relies on timeouts for recovery correctness. Thus, although Beldi-style logging may reduce re-execution on recovery, the actual recovery time would be lower-bounded by a timeout ([216] evaluates 1min as a possible lower bound).

Figure 6.7b further demonstrates the performance benefit of exposing application semantics to the workflow system. We report latency of the most complex workflow in the benchmark, the trip reservation request described in Figure 6.3. Beldi implements the transaction using two-phase locking (2PL). We demonstrate progressive improvement over the original solution by varying the execution and recovery strategy. First, we eliminate Beldi logs for dynamic task invocation, as the DAG can be easily specified upfront, reducing p50 and p99 latency by 17% and 25% respectively (`-WAL`). Next, we parallelize the hotel and flight reservation tasks, further reducing p50 and p99 latency by 17% and 15% respectively (`+parallel`). Beldi executes these tasks sequentially because asynchronous invocation does not allow retrieval of the reply. Finally, we split each reservation task into two steps: lock acquisition and reservation, as seen in Figure 6.4a. `-async` shows that with synchronous checkpoints, this actually increases latency due to the added task. However, `+async` shows that by overlapping checkpointing with execution, we can further reduce p50 and p99 latency by 34% and 16% respectively, without compromising correctness.

6.6 Related Work

Workflow systems. Industry workflow systems [11, 5, 8, 50] orchestrate execution and recovery for distributed applications by durably logging the workflow, checkpointing task outputs and replaying failed tasks. However, they require external outputs to be idempotent and significantly limit how tasks can pass data to each other (Section 6.2).

Many workflow systems for FaaS focus on stateful serverless workflows. Several provide a fault-tolerant transactional key-value store interface [216, 188, 186]. Exoflow is agnostic to external state APIs and implementation and factors out execution and recovery orchestration from such systems.

Some stateful workflow systems offer a fault-tolerant actor programming model [52, 16, 51]. A common recovery technique is *event sourcing*, i.e. durably logging nondeterministic events. However, this requires the developer to use special APIs for non-

deterministic code and can add higher overheads than necessary when deterministic replay is not required for application correctness [78, 146]. Exoflow also supports pluggable actors but only with coarse-grained logging (i.e. recording the workflow DAG) and checkpoint-based recovery (Section 6.3.4). This is intentionally minimal, as it enables composition of both log- and checkpoint-based actor implementations.

Exoflow is similar to DARQ [126]: both use composable atomic steps (tasks) and asynchronous checkpointing. Unlike DARQ, Exoflow exposes references and annotations to avoid materializing and/or persisting outputs where possible.

Dataflow systems. Many dataflow systems use the DAG model [72, 108, 212]. Several use *lineage reconstruction* for recovery, a form of logging that records the DAG but not the data, to reduce run-time overhead. CIEL [148, 150] also introduces *dynamic tasks*, which we adopt. However, these systems target data processing applications in which all tasks are stateless and deterministic. Ray proposes a unified API for DAGs and actors [145], which we also adopt, but cannot support exactly-once semantics or persistence [201]. Tachyon [119] proposes a method of optimizing checkpoints for lineage-based systems; this could be applied to a future version of Exoflow.

Other systems such as Naiad [146], Apache Flink [56] and Canary [164] implement both batch and streaming dataflow with message passing and global checkpoints at run time for recovery. This produces lower latency but requires more rollback on failure; it can also add more overhead for applications with frequent external outputs [78]. Exoflow augments log- and checkpoint-based systems by orchestrating recovery across systems with different internal strategies (Appendix C.3.1).

Falkirk Wheel [90] targets efficient and flexible recovery for batch *and* streaming. It uses logical message timestamps to transparently determine the minimum to roll back on failure. Exoflow provides practical recovery for black-box functions (tasks) by asking semantics from the developer through references and task annotations.

Actor systems. The actor model is a distributed programming model where processes communicate through asynchronous method calls [104]. Most systems do not guarantee exactly-once semantics [31, 2, 54, 201]. Exoflow provides a limited exactly-once actor model to support workflows that pass actors between tasks. Meanwhile, the application has full flexibility of existing actor systems within a task.

Message-passing systems. Message-passing systems are a generalization of actors in which processes communicate through message sends and receives. There is a large body of work on recovery for message passing, primarily focusing on logging vs. checkpointing [78]. Our work adapts these techniques to the distributed workflow setting and aims to compose log- and checkpoint-based applications.

6.7 Discussion

References for framework interoperability. Like other dataflow systems, Exoflow captures the *logical* data movement in an application. Exoflow also aims to enable *interoperability* across distributed execution frameworks, unlike data processing abstractions such as RDDs [212] or timely dataflow [146] that are tightly coupled to a specific execution framework. This motivates some of the differences between `Refs` and `ActorRefs` vs. other dataflow abstractions: they can be used to capture third-party data and context, they are serializable, and they do not impose a particular model of parallelism.

These decisions are intentional. Pluggability for data movement is important for allowing applications to decide the best way to move data from one place to another. Actors are important because many execution frameworks have some type of context that should be passed between logical steps of an end-to-end workflow, e.g., the driver state in Spark. Neither of these is necessary in an execution framework that natively handles all worker communication and process state.

Serializability is of course important for moving any type of data across process boundaries. Supporting serializable references further allows moving large and potentially distributed data by reference instead of needing to first copy the values into one central location. In contrast, serializing an RDD or timely dataflow graph makes little sense; the deserialized copy may be useless if the receiver is not in the same cluster.

Finally, using a generic task parallelism model allows references to be flexibly passed between applications. In contrast, consuming data within a typical dataflow system often requires the consumer to be expressed as part of the dataflow graph, or else for the system to provide special data connectors to third-party systems.

Limitations. Using Exoflow effectively requires developer effort. Exoflow offers recovery flexibility but the developer must choose the right tradeoff for their application. For example, the developer must decide how large a workflow task should be, and whether checkpointing the output is desirable. Currently task annotations are also very coarse-grained, which makes the system general-purpose but also makes it more challenging for an application to achieve optimal performance and recovery overheads.

There are a number of future directions towards improving Exoflow's interfacing with external systems. First, while `Refs` allow the application to efficiently pass data between workflow tasks, reading and writing a `Ref`'s data may still require data movement to or from an external framework. Second, currently Exoflow does not support transactions, i.e. there is no way to specify that a task should be rolled back if another task fails. In this case, the developer must manually roll back the effects of both tasks, e.g., in a final `commitOrAbort` task. Finally, for cases where tasks

read and write external state, capturing more fine-grained semantics could reduce developer burden and improve performance. For example, native support for popular types of external state (e.g., a database) could be added.

6.8 Conclusion and Lessons Learned

Many existing distributed systems provide specialized, efficient, and transparent recovery for specific application domains. Exoflow has an orthogonal and complementary goal. To unify heterogeneous applications, we must provide *general* and *interoperable* recovery methods. The greatest challenge is to gain sufficient application semantics without sacrificing flexibility. Exoflow presents one approach that strikes a balance between usability (minimal annotations, compile-time safety checks) and functionality (flexible `Refs`, automatic recovery). In doing so, we hope to provide universal recovery that matches a universal API: the workflow DAG.

Initially, this work focused mainly on data-intensive workflows and did not include support for serverless microservices workflows nor a solution involving user annotations. However, eventually it became clear that the same techniques that were being used in data workflow systems such as Apache Airflow were also being used in serverless workflow systems; despite significant differences in programming interfaces, all were based on a dataflow graph combined with some form of write-ahead logging. Thus we began to investigate what was needed to additionally support serverless workflows. This led to the introduction of user annotations, as serverless workflows are likely to interact with the external world more frequently than ETL or ML pipelines.

Thus, the key lessons learned from the Exoflow work are:

1. Although previous work, including the lineage stash work in Chapter 3, has shown us that implementing different recovery strategies in the same system is hard, the right abstraction can yield a core system for recovery that is common to many different applications.

2. Broadening application support does not necessarily require adding features; sometimes providing lower system overheads and increasing interoperability with other systems is enough.

Chapter 7

Conclusion

7.1 Related Work

There have been many attempts to design flexible systems for distributed computing. While some have been successful, the emergence of data-intensive applications has shown how current solutions are lacking. Here, we discuss some of the models that have been used by previous systems, many of which are listed in Table 1.2.

7.1.1 Distributed dataflow

Data analytics. In data analytics, popular examples include MapReduce [72], Dryad/ DryadLINQ [108, 83], Apache Hadoop [205], Apache Spark [212], Apache Flink [58], and Naiad [146]. The common paradigm among these systems is that the application specifies a static dataflow graph, i.e. a DAG of data transforms in which each transform is applied in a data-parallel fashion over a dataset. Some [72, 108, 205, 212] are designed for batch processing, while the latter three [213, 58, 146] are designed to support both batch and stream processing.

These systems provide a powerful common set of features needed by data processing applications. In particular, they scale transparently to larger datasets and clusters. The developer does not need to manage allocation or placement of data or compute. Fault tolerance is also provided, either through lineage-based reconstruction [72, 212] or global checkpointing and rollback [60, 56, 57, 146].

Machine learning. However, due to the differences described in Section 1.1, these systems have not been suitable for ML applications, which require fundamentally different parallelism patterns. ML-specific systems focus on training and inference for deep learning models and thus offer native support for GPU-based execution. Similar to data processing systems, they offer an advanced feature set and manage all

resources on behalf of the developer. Fault tolerance is supported through checkpointing and manual reloads [18].

Combining data analytics and machine learning. Where both categories of systems fall short is at the interface between the two. For example, in distributed training, it is often necessary to preprocess and randomly shuffle input data during training. Efficiently shuffling a distributed dataset by row is a challenging problem that has been solved by data analytics systems but not machine learning systems, as described in the Exoshuffle work in Chapter 5. Meanwhile, simply deploying both systems is also unsatisfactory, as it would require overlapping execution, moving data, and sharing resources between two different frameworks.

The fundamental cause of this difficulty is that dataflow systems expose a coarse-grained unit of execution: a dataflow graph applied to a dataset. This is what makes it possible to provide a high level of performance and fault tolerance, as it affords the systems full command over execution and recovery within the graph. However, only exposing higher-level APIs also introduces impedance mismatch and makes it difficult to interoperate between frameworks at a finer granularity of execution. In the online data preprocessing for training case, for example, passing the preprocessed data between a batch processing system and an ML training system would require large amounts of memory for data buffering.

Fine-grained dataflow. A notable example of a dataflow system that exposes a finer-grained unit of execution is Ciel [148]. Ciel offers task parallelism instead of data parallelism. As discussed in Chapter 2, Ciel implements a distributed futures interface that allows computations to be expressed at the unit of a single function call over a fixed value(s), rather than a single function call over many records of a dataset. However, the lack of a stateful API in Ciel also makes the API less suitable to interoperating with other frameworks compared to APIs such as RPC or streaming dataflow (which often supports stateful operators). For example, to support third-party libraries that have context that cannot be materialized, Ciel would need to re-initialize the context on each function call, which can be expensive, e.g., in reinforcement learning [152].

Another use of a distributed futures-like API is PyTorch RPC [12], a lower-level primitive that is used within PyTorch Distributed to implement model parallelism strategies. Similar to distributed futures, PyTorch RPC "remote references" are used to abstract data placement and movement and are reference-counted. Compared to the implementation described in this thesis, PyTorch RPC is designed with ML applications specifically in mind, offering features such as automatic differentiation and multiple transport options that can leverage specialized GPU links. However, references cannot tolerate failures, and physical data movement operations must be explicitly specified by the application, which allows greater control but also imposes

greater burden on the developer.

Distributed programming and consistency. A major difficulty in building distributed systems is the problem of data consistency, i.e. when there are multiple views of the same logical piece of data, what guarantees can be made about the differences between the views compared to the history of data operations? In distributed dataflow systems such as Noria [89], Bloom [25] and Hydroflow [65], data consistency is a first-class concept. The use of dataflow eliminates complex consistency and/or performance bugs that can occur in distributed applications that are written in an imperative style. Unlike offline dataflow systems such as MapReduce, these systems are also well-suited to online workloads where timely access to incremental results is critical.

Thus far, data consistency has not been a focus in this work. In the distributed futures and actors model, all memory is either mutable but private (i.e. actors) or shared but immutable (i.e. distributed future values). There is no built-in replication for actors. Thus, we see the presented model as an intermediate building block upon which others can build replicated systems that may implement varying levels of consistency. An interesting line of future work lies in investigating the performance opportunities and gaps for such an approach, compared to using a lower-level primitive such as RPC directly. This is analogous to the Exoshuffle work in Chapter 5, which provides an alternative building block to RPC for building MapReduce systems.

7.1.2 RPC and actors

Actors are heavily used in many applications of the ownership architecture and Exoflow. Chapter 2 covers the main difference between distributed futures vs. RPC and actors; here we discuss the additional differences between RPC and actors to motivate the decision to use actors for expressing stateful computation.

RPC and actors were both intended as constructs for building distributed systems. RPC was initially designed as a paradigm for communicating between two disparate programs over the network [151, 44]. Concurrency is controlled by creating and destroying the communicating processes. Meanwhile, the actor model was conceived as a computation model for concurrent applications, in which units of computation (actors) can asynchronously send and receive messages [104, 19]. Thus, in the context of developing an end-to-end distributed application that may involve multiple communicating entities, RPC can be considered a lower-level building block, while actor systems often provide a more fully-featured runtime.

Since their inception, however, some interface features of RPC and actors have converged. Notably, RPC was originally proposed as a synchronous interface but has since been extended to support asynchrony, often via futures [129]. Modern RPC implementations also no longer try to hide partial failures, which are inevitable in the distributed setting and had been incorporated early on in actor systems dating

back to Erlang. Conversely, while actor systems such as Erlang are based strictly on message-passing, modern implementations such as Akka and Orleans also support request-response interaction much like RPC.

Today, RPC and actor systems see significant overlap in their use cases. Both are a popular choice for building web service backends. For example, the actor system Orleans is used as the backend for Microsoft's Halo cloud services [42], while gRPC is often used as an alternative to REST in microservices. In fact, the Ray system described in this thesis uses gRPC to implement an actor API.

In our view, the remaining fundamental differences between RPC and actors are in *lifecycle management* of services and *application interoperability*. By lifecycle management, we refer to the responsibility of creating, deleting, and recovering services. This is an important consideration in developing online applications that must remain highly available, respond to changing load, and be upgraded on the fly. In keeping with their origin as a model for developing end-to-end concurrent applications, actor systems typically offer explicit APIs for lifecycle management. Some actor systems such as Orleans also handle actor lifecycles entirely transparently, through a *virtual actor* interface [42]. In contrast, RPC is a paradigm for communication only and thus RPC implementations have no control over the lifetime of services. This responsibility is typically left to some cluster manager such as Kubernetes that deploys the RPC application.

As low-level APIs that expose execution at the granularity of a function call (or message) and that support remotable state, both RPC and actors are highly flexible in their use with third-party systems. As a communication paradigm, however, RPC affords applications innate interoperability and decoupling. Any program can communicate with an RPC service as long as it uses the same protocol. The client and server programs may be decoupled from one another, as long as they agree on the protocol. Meanwhile, actor deployments are fully encapsulating. Features that are supported within one actor deployment may not be supported across two different deployments, even when the same underlying actor system is used. A pair of sender and receiver actors is coupled together, as they share the same code and physical deployment. Note that this is not necessarily a negative; coupling may reduce autonomy between the sender and receiver but it also prevents developer errors such as mismatched versions during upgrades.

As an actor system, Ray shares the same properties in lifecycle management and interoperability. For example, the ownership architecture follows the supervision model proposed by Erlang [195] (Chapter 4) to manage failures for both actors and (stateless) tasks. Also, as of this thesis, a distributed future created within one Ray cluster would have to be copied by value for use in another Ray cluster, even if the two clusters share the same physical resources.

There has also been a long line of work on fault tolerance in RPC and actor systems, in particular studying how to provide at-most-once, at-least-once, and/or exactly-once message delivery. RPC has long had a debate on whether and how to

expose failures [196, 191]. Recently, given the popularity of RPC for microservices, there have been several works on providing exactly-once semantics for RPC [117, 22]. In practice, however, deployments continue to prefer at-least-once or at-most-once semantics, likely due to the performance overheads and application restrictions imposed by exactly-once semantics. Actor systems such as Akka [2] typically rely on supervision, as proposed in Erlang [195], and provide at-least-once or at-most-once semantics. Virtual actor systems [42] can be considered a special case of supervision, in which all actors are supervised and automatically restarted by the system. Out of these, Orleans provides optional persistence and at-most-once or at-least-once message delivery, while Ambrosia [91] provides exactly-once delivery.

Compared to these works, the main contribution of this thesis is in showing how *distributed futures* can provide unique opportunities in efficient and flexible fault tolerance. First, the ownership work in Chapter 4 draws from the actor supervision model but further extends it to provide efficient task- and object-level recovery. Second, the Exoflow work in Chapter 6 uses the DAG produced by distributed futures programs to show how one can compose applications that use a mix of different execution semantics. In contrast, past work on actor and RPC systems typically focus on providing a *single* type of execution semantics for a particular deployment.

7.1.3 High-performance computing (HPC)

High-performance computing (HPC) refers to a class of systems supporting computationally intensive problems often found in the scientific domain, such as physics simulation. Traditionally, HPC focuses on bare-metal execution on supercomputers that are specially designed for massively parallel compute, with many cores and high-speed interconnects. This is in contrast to cloud computing offerings, which generally consist of much cheaper commodity hardware and can offer customers a dynamically sized resource allocation through VMs.

Recently, HPC systems have gained widespread popularity through ML use cases, which are also computationally intensive and require collective communication primitives such as allreduce. MPI [86] for example has long been a standard for HPC application development and has been used directly in machine learning systems such as Horovod [178]. The single program, multiple-data (SPMD) programming model often used in MPI has also been adopted in other popular machine learning frameworks [18, 70].

Two of the main limitations found in the MPI model are a lack of dynamicity, as the participating programs must be specified before execution, and fault tolerance, as a single process failure will terminate all other processes. These are acceptable in the supercomputer scenario, where resources are fixed and machine failure is uncommon, but become significant obstacles in cloud computing scenarios.

Some parallel computing systems offer native support for dynamicity. These include Cilk [45] and Legion [40], both of which offer the ability to dynamically

spawn nested tasks, i.e. function calls, similar to Ray. Cilk-NOW [46] provides a distributed runtime for the Cilk API and fault tolerance through checkpointing. As shown by later dataflow-based systems, checkpointing alone is often undesirable for data-intensive applications that may produce large amounts of intermediate data.

Compared to Cilk, Legion additionally offers explicit support for data-intensive applications; developers specify their data partitioning and placement with logical regions while Legion handles the physical data movement. This has a similar goal as distributed futures of shifting data movement responsibilities from the application to the system. Unlike distributed futures, the data partitioning must be specified before execution and each Legion task must specify its data privileges and coherence requirements, in addition to the input regions that each task will access. Data locality is also explicitly controlled by the developer through Legion's mapping interface. Thus, Legion's data partitioning API offers developers greater control but also requires greater effort. Providing failure transparency also remains an open challenge for Legion. This is due to the extra bookkeeping, coordination, and data persistence required for recovery, which can impose significant complexity and performance overheads when tasks are both dynamic and fine-grained.

Legion shines in building high-performance applications on heterogeneous parallel architectures such as multicore NUMA and/or multi-GPU machines. Ray does not offer such a high level of architecture portability but it reduces the developer effort required to scale to multiple machines and it provides interoperability with other distributed applications. Taken together, Legion and Ray could produce a powerful combination for building high-performance distributed applications. An interesting line of future work is in the development of a deeper integration between the two systems, e.g., to allow ownership handoff of memory regions from one system to another.

7.1.4 Distributed shared memory (DSM)

Distributed shared memory (DSM) is a paradigm that aims to abstract the physically distributed memory of a cluster to provide the illusion of a shared address space [122, 153]. This allows the application to be written as a multithreaded program, as if for a single machine, but executed on a cluster of machines. Thus, DSM's goal is closely aligned with the goals of this thesis. However, DSM systems failed to abstract away distributed memory with suitable overheads and system complexity and is generally believed to have failed for two reasons [32].

First, while the goal of complete transparency would have simplified application development, the choice of programming model turned out to be inappropriate. Performance transparency was unattainable for local vs. remote memory, as remote memory access latency continues to be an order of magnitude or more higher. Thus, much of the DSM efforts were aimed towards the coherence mechanism [153], and good performance was only possible if data and threads were carefully placed to minimize sharing and communication [41, 111]. The same lessons were learned and incorpo-

rated into successful paradigms such as RPC and actors, both of which require the developer to explicitly specify remote execution and neither of which provide a shared address space. Meanwhile, distributed futures support memory sharing between distributed threads, but with two key differences compared to DSM: (1) shared memory is immutable, and (2) sharing is not global and must be made explicit through the use of references. The former avoids the overheads of coherence experienced by DSM systems, and the latter enables more efficient bookkeeping and placement for distributed memory.

Second, fault tolerance proved to be an essential property for scalable cloud computing applications, which are susceptible to failures and variable network latency. Transparent recovery of DSM applications would have required huge performance overheads, as the unit of execution is extremely fine-grained, at the level of individual memory accesses. In contrast, the programming model targeted in this thesis is coarser-grained, at the level of function calls, and more structured, as the program specifies a dataflow DAG ahead of execution. The latter property is critical towards applying the layered approach to fault tolerance proposed in this thesis; it is unclear how such an approach could be applied to a multithreaded programming model, where each thread's data access pattern is arbitrary and opaque to the system until execution.

7.1.5 Disaggregated memory

Disaggregated memory architectures decouple memory from processors [20]. This is in contrast to traditional server architectures where the ratio and amount of CPUs vs. RAM on each node is fixed. The goal is to allow both processors and memory to be more efficiently utilized, by increasing memory sharing between processors and right-sizing each application's physical allocation. Systems for disaggregated memory typically disallow memory sharing by distributed threads. Thus, compared to DSM, disaggregated memory systems similarly aim to share distributed memory but do not attempt to provide the illusion of a shared address space.

Many systems for disaggregated memory aim to leverage remote memory to expand a single processor's available memory beyond that of a typical server. This can be done entirely transparently to applications, by implementing remote memory support at the hardware or OS level [94, 120], or semi-transparently, by providing APIs to mark regions of memory as remotable [172].

Distributed futures could be used as a disaggregated memory API in the latter approach. A key difference in the distributed futures interface is that it also gives the system control over task placement. This advantage is typically lacking in disaggregated memory systems that focus mainly on memory placement. Compared to previous solutions, distributed futures would thus offer the system greater opportunity to leverage optimizations such as moving execution to the data or overlapping execution with data movement between servers, as implemented in Exoshuffle (Chap-

ter 5).

Furthermore, the distributed futures architectures proposed in Chapters 4 and 6 address fault tolerance holistically, recovering both execution *and* data. While this makes the problem more complex, it also offers greater flexibility in the recovery approach. For example, all of the works in this thesis rely on the ability to choose between logging data vs. lineage, i.e. the execution that produced the data.

Finally, thus far the benefits of memory pooling via physically disaggregated memory have been shown for small pools (8-16 CPU sockets) only [120], and in general a scalability limitation is likely to persist due to physical constraints. This makes integration of disaggregated memory inside a more scalable distributed futures system a promising approach towards supporting large-scale data-intensive applications.

7.1.6 Serverless

Serverless execution is a recent paradigm in which developers execute their applications in the cloud without needing to provision VMs. Instead, they specify their applications as the code to run and a description of dependencies. The code is then transparently deployed and autoscaled by the cloud provider.

Typically, the code to run must be a short (minutes-long) and stateless function, and the functions-as-a-service (FaaS) provider guarantees at-most-once or at-least-once semantics. These requirements significantly simplify deployment and autoscaling, as tasks are only allowed to hold resources while running, the duration of the run is limited, and the execution semantics can be guaranteed in the presence of failures through retries. However, unsurprisingly these restrictions have also turned out to be limiting for applications [101]. We see the distributed futures and actors model as a potential higher-level programming layer for FaaS applications. In particular, the Exoflow chapter (Chapter 6) explores what extensions are needed for the model to adequately wrap FaaS backends to help overcome two of their key limitations.

First, the cost of requiring all function state to be materialized and copied between functions can be significant. This has inspired a number of works that aim to make state and/or storage a first-class citizen in FaaS APIs, such as Cloudburst [186], Pocket [112], and Azure's Durable Functions [52]. The same trends can be seen in this thesis: distributed futures provide a FaaS-like API but promote physical memory sharing between functions, while actors provide a stateful API. In the Exoflow work (Chapter 6), we show how such an API can be backed by a serverless execution backend.

Second, the weak execution semantics offered by popular FaaS solutions can be difficult to build upon in the presence of failures. This has led to a number of systems that provide exactly-once semantics transparently to serverless applications, often by providing exactly-once APIs for reads and writes of external state [188, 216, 110]. Such systems can be used in conjunction with Exoflow (Chapter 6), which also aims to provide exactly-once semantics but with an additional goal of making

recovery more flexible. In particular, in Exoflow we observe that there is a spectrum of how individual functions in a serverless application may interact with external state, ranging from no interaction to requiring exactly-once operations. Thus, an application using a fault-tolerant serverless system such as Beldi or Aft could use Exoflow's annotations to describe which tasks will use the custom APIs and with what semantics, and Exoflow would ensure that end-to-end exactly-once semantics are met while minimizing re-execution upon failure.

7.1.7 Cluster managers

Cluster managers such as Apache Mesos [105], Borg and Kubernetes [53] can be considered a type of OS for the cloud. They are responsible for application deployment and resource allocation in multinode environments. Typically, the interface is a YAML-style static configuration that specifies the application's entrypoint, software dependencies, and system-level configuration such as resource requirements, placement constraints, and networking configuration. Often, the unit of execution is a container.

Cluster managers are powerful tools that are widely used for managing multi-tenant clusters hosted in the cloud. Because the unit of execution is a container, they are extremely general-purpose. The tenants of these clusters can range from deployments of multi-node systems such as Ray to individual microservices.

However, the choice of a container as an API also means that cluster managers are intentionally limited in the performance and fault tolerance features that they can provide to the end application code. In fact, cluster managers have no visibility into the application code. Thus, features such as dynamic creation and destruction of containers based on application-defined control flow are complex to support natively. It is also the application's responsibility to manage any cross-container interactions, including managing dependencies between containers and how data should be moved across container boundaries.

These limitations are logical in the context of cluster managers, which are targeted towards DevOps and cluster administrators. However, as large-scale applications are becoming more prevalent and complex, we are also seeing that developers need more direct control over cluster resources and placement. This in part motivates this thesis, which targets end developers and offers cluster management features that are embedded in the application code.

In particular, the system proposed here offers a subset of container-like features but at the granularity of tasks and actors (i.e. functions and classes), including runtime dependency management, autoscaling, and resource requests and limits. Some of these features require significant implementation work compared to containers, which enjoy native support from the OS and modern cluster managers. However, we believe that the investment is worthwhile, as application preference for finer-grained execution units is also seen in other trends such as serverless execution. The finer-

grained interface also affords the system greater visibility into and control over the application. Thus, we believe that natively providing such system-level features will be an important feature of future cloud systems.

At the time of this thesis, there is a well-defined boundary between cluster managers and distributed execution frameworks. However, we hypothesize that execution frameworks in the future will take on more of the current role of cluster managers, as finer-grained resource management and application interoperability become more important. We see this trend in other domains as well, such as in microservices frameworks [88]. The precise way in which this will play out, and affect both cluster managers and distributed execution frameworks, remains remains an open question.

7.2 Discussion

7.2.1 Broader Impact: History of Ray as an open-source project

The open-source project Ray [13] is the realization of the goals and ideas described in this thesis. Ray started in 2016 as a system for distributed Python. The first versions of the RLlib [127] and Tune [128] libraries, for distributed reinforcement learning and hyperparameter search, respectively, were released soon after. Since then, Ray has evolved into an ecosystem of distributed Python libraries that can be composed to build end-to-end data processing and machine learning applications. Many of the works in this thesis have been instrumental towards this journey.

While Ray saw some success with early users of RLlib and Tune, especially among academic researchers, the "core" backend system was not yet stable. In 2017, we rewrote the Ray core based on lessons learned from the initial versions. Some of the major changes included coupling the scheduler and the object store processes, an initial implementation of the lineage stash system described in Chapter 3, and a switch from C to C++. Importantly, Ray core continued to use the same decentralized control plane architecture described in [145].

As the rest of the system matured, this fully decentralized design become an increasingly significant obstacle when it came to further improving system performance and robustness. Two major robustness features were sorely lacking at this time: (1) garbage collection or reference counting of application objects, and (2) failure detection and recovery of application objects. Both procedures needed to be fault-tolerant, correct, and timely. As Ray was ultimately meant to be a production-ready system, this also called for simple solutions. Meanwhile, Ray's then-decentralized design also proved to add significant complexity and performance overheads, requiring convoluted paths through various nodes and system processes just to execute a single task. This prevented the development of high-performance libraries such as model serving on top of the Ray core.

Initially, the hope was to solve the performance problem for small actor-to-actor messages only, the prevalent pattern in early Ray libraries compared to non-actor tasks. This would be done by introducing a fast path for actor task execution, allowing actor tasks to be scheduled and executed in a single round trip. It also would have added significant complexity, as it essentially would have required two separate systems.

Thus, one motivating goal of the ownership architecture was to *improve performance for all application paths*, not just in the special case. Fortunately, it turned out that this design requirement simplified the rest of the system as well.

In particular, ownership was a comprehensive solution for the three most pressing problems at the time: object garbage collection, object failure recovery, and actor task performance. By centralizing system state at the workers, we were able to simplify and harden object management protocols while also reducing task overheads. Both actor *and* non-actor tasks were executed with essentially the same codepath, and still in a single round trip. Thus in 2019, we rewrote the Ray backend a second time, this time to use the ownership design, spurring the first major release of the Ray project.

As Ray's object management and recovery protocols stabilized, it became possible to build on top of these to achieve more sophisticated memory management features and applications. One of these features was to extend Ray's object store with disk spilling, to support larger-than-memory workloads. Exoshuffle (Chapter 5) became a driving application for this feature. Another of these features was to extend Ray's fault tolerance capabilities with application durability, to optimize recovery time after a failure and to be able to survive total cluster failure in addition to partial failures. This ultimately led to the Exoflow work described in Chapter 6.

7.2.2 Lessons Learned

Here I summarize some of the personal lessons learned over the course of this thesis, both in the broader design of systems and on the road to practical adoption.

1. Fault tolerance is not one thing. Fault tolerance is often listed as one of the key requirements for a distributed system. While many (including me) speak of it as if it is a single concept, the actual meaning has a depth of nuance, to the point that saying that a system is "fault-tolerant" alone says very little. The difficulty is that there are many scenarios and assumptions (application model, failure model), desired properties (execution semantics, consistency, durability, availability), and metrics (run-time overheads, recovery overheads) to consider. In stating a fault tolerance goal, it is critical to be precise.

For example, in this thesis, we use the crash failure model throughout, but by necessity of building a general-purpose system, we must assume a wide variety of applications, desired properties, and metrics. In particular, regarding the application model, we differentiate techniques based on data-intensive vs. not (e.g., varying the

strategy for small vs. large objects in Chapter 4), as well as deterministic vs. nondeterministic computation. Most of the chapters in this thesis aim for exactly-once execution semantics; Exoflow further aims for durability and availability. Consistency is considered out of scope as there is no native support for replication of mutable data. Finally, for metrics, low run-time overhead is overall prioritized, but a constant theme of the thesis is how to achieve this simultaneously with low recovery overhead. This culminates in the Exoflow work, which allows exploring a spectrum of tradeoffs.

Given this huge application variety, tackling all of the desired properties and metrics in one all-encompassing system design is likely impossibly complex in practice. At the same time, the system design has to consider fault tolerance from the start, as fault tolerance properties cannot be easily bolted on after the fact. Thus, one of the key insights that drove this thesis was that fault tolerance for a general-purpose distributed system should be designed through a layered approach.

To illustrate this, we point out that indeed the fault tolerance goal from the lineage stash to the ownership chapters became narrower in scope. Rather than provide transparent exactly-once semantics for all distributed future-based applications, the ownership design only does so for deterministic and idempotent tasks. Critically, however, the ownership design *does* guarantee fast and reliable failure recovery for pure functions, as well as fast and reliable failure detection for all other tasks. These were not easily accomplished in the lineage stash work due to the lack of a centralized task control plane. Meanwhile, speed and reliability made it possible to efficiently build other system layers on top of the ownership work, including Exoflow, which handles failures detected by the ownership layer, including for applications that are nondeterministic and/or that have side effects.

This design is an application of the end-to-end principle: in cases where efficient run time or recovery depends on application-specific features, the endpoints (end applications or Exoflow) should handle recovery rather than the system (the ownership architecture). There is furthermore a practical extensibility argument in this approach. In industrial settings, many users will often share a particular cluster deployment of a system. Thus, upgrading or forking a low-level cluster framework is a nontrivial task, as it requires coordinating among the many different applications built on top. However, if the system core is more minimal in scope, then services traditionally provided by the system can be provided as a library instead and are more easily modified by end users as needed. This is the key idea behind both Exoshuffle and Exoflow, which provide distributed shuffle and recovery as a library, respectively. Thus, the ownership design can be considered an exokernel [81] for distributed applications that exports the distributed future as its primitive.

2. Systems must provide minimum features first, but remain future-proof. The industrial success of the ownership architecture, compared to previous versions of the Ray backend, can be explained by the focus on providing minimum features first. Note that this is not equivalent to giving up on fault tolerance entirely; indeed

it is the opposite. As we learned in the first lesson, there are many types of fault tolerance, and some are more essential than others. Case in point, ownership focused on providing *fault-tolerant* memory safety first, while allowing for the possibility of building more complex recovery optimizations on top.

Another example of this is in the original motivation for the ownership architecture. As described above, reducing performance overheads for small actor tasks was one of the motivating goals for the ownership work. The impact of reducing performance overheads cannot be overstated; reducing performance overheads by a magnitude or more is practically guaranteed to expand the feasible applications for a system. However, doing so at the cost of significant system complexity is also risky because it prevents future optimizations. If we had opted to introduce the special execution path for actor tasks only, it likely would have become increasingly difficult to introduce similar optimizations for non-actor tasks. Meanwhile, the ability to mix actor and non-actor tasks, though unpopular at the beginning of the Ray project, is one of the most powerful features of the system today.

In addition to performance, *flexibility* continues to be a primary goal for the Ray system and of course for OSes in general. In some cases, this was at odds with fault tolerance. For example, early versions of Ray only allowed a single caller for actors, so as to facilitate transparent state recovery for actors. However, this was a significant limitation that made it impossible to support many applications. Meanwhile, the applications in distributed training that we were evaluating at the time all used synchronous checkpointing and did not even need transparent state recovery! Thus, we made the decision to allow multiple callers and forego transparent state recovery for actors, as we believed that this flexibility was the more necessary feature and that state recovery could be reasonably implemented by end applications. Note that the decision was to forego transparent state recovery as part of the ownership architecture; we later adopted the idea in the Exoflow work, but this time at a higher-level API where enforcing a single caller was practical. This again shows the importance of prioritizing minimum features first, but without closing the door to future optimizations.

3. For practical adoption, interoperability over completeness. From its inception, the Ray project had an ambitious goal of becoming the default execution engine of cloud applications. While its success in this ultimate goal is yet to be determined, we have already seen substantial adoption among machine learning and data processing applications. Many of the lessons here may be obvious but cannot be overstated: (1) preserving a simple API, even while the system implementation continues to evolve, and (2) ensuring that these APIs enable specific application niches that have the potential to grow. For example, distributed reinforcement learning was very much a niche application at the start of the Ray project, but it also exhibited potential for growth, had as of yet no canonical framework, and was able to use the early distributed futures APIs to achieve something useful (in this case, scale-out with

distributed execution). Thus RLlib [127] was developed and adopted even while we rewrote the Ray core multiple times.

Another lesson learned on the road to adoption was the value of interoperability with other applications and systems over feature completeness. As Ray was designed as a platform, there was significant overlap with both existing data processing frameworks such as Apache Spark, as well as cluster managers such as Kubernetes. The similarities and differences compared to these systems are discussed in more detail in Sections 7.1.1 and 7.1.7, respectively, but such boundaries were not as well-defined at the beginning of this work.

As evidence of this, Ray had no standard way of interoperating with most other systems until relatively recently. This proved to be a significant obstacle to adoption, especially in the case of Kubernetes, which was and still is the most popular method of deploying cloud applications in industry settings. Without a Ray operator for Kubernetes, many larger-scale users could not easily adopt Ray even if they had wanted to. One of the reasons for this delay was that Ray had not been designed with containerization in mind – autoscaling for example was initially designed at the granularity of VMs.

For data processing systems like Spark and Dask that were already popular with Python users, it also took years to define data processing applications that were: (a) not already well supported, or (b) well-supported but where Ray could achieve a better result. Thus, there was little incentive for existing Spark and Dask users to try out Ray. It was not until projects such as Exoshuffle and Ray Data [193], which could handle last-mile data preprocessing for ML training more efficiently than Spark, and Dask-on-Ray, which could execute Dask Dataframes applications more efficiently than Dask could, that Ray began to gain traction among data processing applications.

What worked in these instances was to define the successful application use cases incrementally, and to prioritize interoperability first, so that at least it was simple for end users to extract and load data between Ray and other more mature data processing systems. For example, the Spark-on-Ray project allowed Spark applications to be executed on Ray clusters. The advantage of such a project was that Ray-based ML applications could take full advantage of Spark's more advanced data processing support, while also sharing resources.

As one more piece of evidence of this, we can contrast RPC vs. actor systems. Compared to RPC, actor systems are certainly more fully featured, but they are also less interoperable, as discussed in Section 7.1.2. This may explain why libraries such as gRPC are the overwhelming choice over actor systems.

One could argue that by prioritizing interoperability and widespread adoption over feature completeness, one runs the danger of never having enough compelling features to add to the table. Among academic circles, there is perhaps a related concern about the novelty and contribution of such a system that appears to not be as comprehensive. But I believe that these concerns can be avoided as long as system designers persist in their aim to enable specific and new end application use

cases. This aim defines the necessary features, the interfaces that should be provided to interoperate with other systems, and often the lasting contributions of the work. Prioritizing adoption ensures that the aim stays true. For me personally, the question of the right way to design something new while remaining practical and interoperable is indeed where some of the most interesting research can be found.

7.2.3 Limitations and future work

Some of Ray's current most successful applications include: reinforcement learning, ML training including hyperparameter search and last-mile data loading and preprocessing, ML inference both batch and online, and large-scale shuffle via Exoshuffle. In all of these cases, Ray's enhanced flexibility proved to be a significant edge, e.g., by enabling expression of complex control flow loops in reinforcement learning, hyperparameter search, and large-scale shuffle. In all but reinforcement learning, the Ray dataplane described throughout this thesis also proved to be critical.

Given the eventual and ambitious goal of building an OS for distributed applications, there are of course several limitations. Some of the application areas where Ray has substantial room to improve are: the composition of different applications both within Ray and with other systems, high-performance ML applications, and online stateful applications. Some shared requirements among these applications include: low-latency distributed execution (tasks that are a few milliseconds or less), efficient memory sharing, and heterogeneity, both in hardware and computation patterns. Here, I will expand upon some of the existing gaps for these applications, with potential approaches emphasized in italics.

Performance. The current performance overheads in Ray are well understood and are described in more detail in Chapter 4. In general, one can expect on average 1-2 round-trips per task execution, with actor tasks requiring fewer than non-actor tasks. This is because each task is dynamically scheduled, and scheduling decisions are usually made one task at a time. Tasks that take distributed futures as arguments further require 0-1 IPCs to get the values, plus additional round-trips for object transfer if the value is not already local. The actual execution latency can therefore vary widely depending on task placement, which affects locality with inputs, queuing delay, and resource interference with other tasks.

The current overhead is reasonable for tasks ranging in the 10s of ms or more and/or for objects that are MBs or larger, but becomes significant for extremely low-latency applications in online web serving or ML inference. For large-scale ML model serving, for example, executing a single forward pass on one layer of a model may take only a few milliseconds. For web serving and microservices, Ray will also share the same performance problems as RPC systems, such as high overheads from serialization and communication [88].

The root cause of these overheads is scheduling and dispatching tasks and their associated object transfers one at a time. This adds to the number of operations per task (i.e. how many protocol messages must be sent to transfer an object), reduces chance of pipelining operations (e.g., overlapping the transfer of an object dependency with the scheduling decision for the task), and requires more metadata per task.

A promising approach to reduce these overheads is to employ *a combination of distributed system optimizations and JIT compilation*. System-level optimizations may be taken to mitigate execution overheads, but they are difficult to apply in general given the variety of applications and the dynamic nature of both the application logic and the cluster configuration. For example, one could preallocate mutable object buffers that could be reused for multiple DAG executions, but this is only likely to improve run time if the task dependencies and object sizes remain the same. JIT compilation could therefore be used to identify such cases where system-level optimizations are more likely to succeed. In particular, one could imagine JIT-compiling DAG fragments that are repeatedly generated by a program, with fallback to a completely dynamic path if a new fragment is generated.

Another future direction is to improve scheduler extensibility. Variation in task execution overhead often comes from the schedluer policy for task placement. Unfortunately, simultaneously achieving policy completeness, performance, and evolvability for the system scheduler is a difficult problem, much as we saw for fault tolerance in this thesis. Works such as ESCHER [43] for distributed futures or DCM [189] for container placement provide some ideas, but the fine granularity and breadth of distributed applications continue to be challenges. For example, both ML and web serving often have application-specific request routing and scaling policies. Some of these policies can be expressed in Ray with significant manual effort, using the low-level scheduling primitives that were introduced in the Exoshuffle work, such as specifying hard or soft affinity to a particular node. However, ensuring that such scheduling primitives are specified correctly and will not interact poorly with other concurrent resource requests or cluster changes at run time is a difficult problem.

A promising approach for enabling scheduler extensibility is to apply *domain-specific languages for controlling task scheduling and placement*. Currently, the scheduling primitives exposed by Ray and other lower-level cluster frameworks such as Kubernetes are analogous to an "assembly language": they offer basic functionality such as node affinity for tasks or hinting when data can be freed. Specifying application- or cluster-wide policies, such as guaranteeing a certain level of resource utilization, can require significant application or framework developer effort and complexity. Introducing DSLs for scheduling or object management policies could allow for greater extensibility and reduced developer effort.

The remaining Ray overheads come mainly from the dataplane (the object store). First, objects must be entirely in memory before they can be read, which can add undue memory pressure and prevent pipelining. Exoshuffle works around this by slicing task inputs and outputs into small enough objects that the execution's pipeline

granularity can maximize disk and network bandwidth. However, at large enough scale, this will eventually lead to excessive object metadata, which itself adds memory pressure. Here again, we may explore a range of solutions spanning systems and programming language design. To support more flexibility in object reads, one could introduce *programming interfaces for large objects that support slicing and streaming*. With these additional semantics, opportunities for system-level optimizations include *the object transfer policy*, e.g., prefetching data for streamed objects, and *leveraging specialized hardware*, e.g., remote memory or programmable networks.

Finally, objects are always immutable, which simplifies the design but can be inefficient for applications requiring low latency and fine-grained mutations. Interestingly, because distributed futures are immutable, we can consider the initial distributed futures program to be an intermediate representation that follows static single assignment (SSA). Thus, we can *preserve reference immutability but apply static or dynamic program analysis* to determine when physical object buffers may be reused.

Fault tolerance. For offline workloads, we believe that the combination of fault tolerance provided by ownership and Exoflow is generally sufficient. However, this is not yet the case for online workloads, where Ray will again share many of the same limitations as RPC and actor systems, namely the lack of state persistence and efficient and transparent exactly-once semantics.

Thus, one of the main extensions of the current work will involve enhancing fault tolerance properties for online and stateful workloads. In particular, there are open questions on what the interface should be between a workflow orchestrator such as Exoflow and database systems that are commonly used to persist application state. Currently, orchestration systems treat such databases as a black box. While Exoflow additionally supports annotation for application semantics that are relevant to recovery strategy, the annotation process is manual and potentially error-prone.

Therefore, an interesting direction for the future is exploring how we can more *deeply integrate application semantics into a distributed programming language for workflows*, to unlock new run-time and recovery optimizations. Exoflow's annotations already rely on compile-time checks to optimize the recovery approach. In the future, stronger guarantees in fault tolerance and performance could be guaranteed through additional compile-time checks. For example, instead of marking all tasks that interact with an application database as having a side effect, one could imagine representing a database with an interface that captures the properties of different queries. Then, Exoflow could automatically induce a finer-grained task DAG based on which interfaces a task uses, instead of requiring the developer to specify the DAG explicitly and with coarse-grained semantics.

Interoperability and emerging hardware. While Ray's dataplane is critical to the success of many of its current applications, it is currently targeted to commodity

hardware, i.e. tasks executing on the CPU and objects passed through host memory. Although Ray supports heterogeneous resources, and indeed execution across heterogeneous CPU-GPU clusters is one of its key motivations and advantages, it does not for example have GPU-native execution: application-level code is required to schedule individual kernels to GPUs and to pass distributed future values to and from a GPU. GPU-GPU communication is typically offloaded to a high-performance ML-specific framework such as Distributed TensorFlow.

The main benefit of supporting heterogeneous memory is to make it simpler to develop more complex distributed and heterogeneous applications. Among distributed GPU applications, for example, developers are currently limited by restrictive SPMD programming models, which make it difficult to support elasticity, fault tolerance, dynamic scheduling, GPU multiplexing, and GPU heterogeneity (e.g., network asymmetry or use of different GPU types) [39].

Another benefit is to share common functionality around memory management and task scheduling in a common "operating system". This will become especially important in the future as we continue to develop a greater variety of accelerators and advanced memory technologies to deal with hardware limitations in CPU performance and memory bandwidth, respectively. Adoption of such technologies is often a chicken-and-egg problem, in that ease of application development is necessary for adoption, but adoption in turn depends on building a critical mass of target applications. Thus, a common operating system that accelerates distributed application development will be critical for leveraging new hardware technologies.

A first step towards this goal is to *expose pluggable APIs for supporting heterogeneous memory in distributed futures systems*. However, simply exposing a file interface as a traditional OS would is insufficient. The power of the distributed futures model relies on specifying data and tasks *together*. Thus, plugging in heterogeneous memory requires integration with both the virtual memory and task scheduling systems. For example, in some cases, a task may be able to run on different kinds of hardware, and the best choice depends on availability at run time. To unlock this use case, the system must be aware of cost tradeoffs between moving the data vs. the computation.

In a distributed memory system, the transport mechanism is also important and often should be customized to the application. Some cases may require a choice between multiple transports, e.g., if there is network asymmetry or multiple paths available. Heterogeneous hardware also often requires specialized transports that may not be easily captured with a typical `send/recv` API. For example, communication collectives such as allreduce operations in ML have unique properties that are not possible to express in the distributed futures API as described in this thesis: they involve both communication and computation that mutates the data, and they require gang scheduling to avoid deadlock.

A related problem in building dataplanes for distributed futures is interoperability. Previous work provides key pieces of a solution to data interoperability:

RPC provides communication while Apache Arrow provides zero-copy deserialization and language interoperability. This thesis partially addresses the problem of reducing overheads between *distributed applications that share large data*, by providing a common system for distributed memory management. However, the problem of interoperability with *other* systems persists.

Currently, data passing between a distributed futures system and a third-party framework is likely to incur serialization and/or copying overheads. While zero-copy and language-interoperable serialization formats such as Apache Arrow alleviate this problem, they are not sufficient. For example, many data processing jobs must often convert data between other formats, e.g., to support UDFs and because different frameworks will often use different internal serialization formats. While individual frameworks can use query optimizers to elide unnecessary format conversions, there is no way to do so across frameworks. One approach to solve this problem is to have *frameworks externalize their internal serialization steps*, e.g., through a DAG intermediate representation, and then optimize across different frameworks to determine the best format to use at the boundaries.

Another problem is the ownership and mutability of the data being passed. As data is passed out of one system and into another, ideally we would like the latter system to have full control over the physical memory: it should be able to mutate the data as it wants and be able to decide when to deallocate it. This is true for frameworks that communicate via RPC but only because copies are made. This is also true for distributed futures systems, but only for applications running on the same distributed futures cluster. Supporting it for arbitrary compositions of distributed applications likely requires *ownership semantics for distributed futures* (e.g., moveable references) and *coordinated memory management across frameworks*, possibly via a separately managed distributed memory pool. It is also interesting to consider how to leverage trends such as disaggregated memory, as they can help to avoid copying and serialization overheads, and how to scale their benefits to larger clusters.

Debugging. In most respects, distributed futures systems will have the same challenges and opportunities in distributed debugging tools as other distributed programming systems such as actor runtimes. One of the unique opportunities in distributed futures systems is that the system handles both the compute and the data, which opens an opportunity for *performance debugging in data-intensive applications*. In actor and RPC systems, the system controls compute and network (for small messages), but has little visibility into or control over memory or disk usage between messages. Meanwhile, in Exoshuffle (Chapter 5), the distributed futures system is responsible for pipelining data movement across the producing and consuming tasks, memory, disk, and network. One could then adapt this system, for example, to probe for performance bottlenecks, e.g., by virtually speeding up one of the resources [68]. This decomposition of distributed applications into separate resource units has been

proposed before by the monotasks work [156], which rearchitected the Spark RDD engine to execute decomposed tasks. Distributed futures can be seen as a more generic realization of the same idea that bakes resource decomposition directly into the programming API.

Security and isolation. The security model used in all works in this thesis is that the cluster operator and all other users of the same cluster are trusted. Of course, this model is not sufficient for multitenant clusters that are shared by larger organizations or by multiple distinct organizations.

Regarding confidentality for distributed futures, an interesting avenue is to *augment distributed futures with capabilities*. Distributed futures already have an advantage over raw pointers in distributed shared memory in that they are first-class references and typically a task will only access distributed futures that are explicitly passed to it. Enforcing access is therefore more straightforward than for raw pointers.

Resource isolation is also a highly relevant issue for multitenant clusters. As described, the evaluations in this thesis assume that each application has full access to all resources in the cluster. If multiple applications share the same cluster, it can lead to resource starvation and interference. Thus, a future direction of work is to provide *distributed resource sandboxes for applications*, analogous to a cgroup in Linux. In fact, such an approach is even useful for resource management of single applications that are made up of multiple distinct workloads, such as an online training loop consisting of both training and inference, or a distributed hyperparameter search with a shared data loader.

Broader themes. A core thesis of this work is that the set of distributed applications will continue to expand in scale and diversity. Thus, a common theme in some of the above challenges is in maintaining performance and especially evolvability with an increasingly complex system core and expanding application set. Evolvability will become particularly important as so far the applications we have targeted have had similar latency budgets (>10ms per task) and thus passing data through shared host memory carried sufficiently low overhead. Extending the system to push this envelope further will likely require significantly different scheduling and memory management policies, as well as more control over accelerator memory and execution.

Therefore, when we compare Ray to modern OSes, one of the most significant gaps we have seen is *pluggability*. For example, Ray's dataplane currently supports a "virtual memory" stack, consisting of worker heap memory, shared object store memory, and swap space on disk. It does not support any kind of "device" memory, such as GPUs. Similarly, there is no way to safely hand control of the core scheduling and memory management policies to an application, analogous to eBPFs, FUSE filesystems, or user-level scheduling and networking [106, 134].

A promising broader direction towards tackling this challenge of supporting

increasing application and hardware heterogeneity is to *leverage programming languages*. Many of the performance limitations discussed here are evidence that we are reaching the boundaries of an eagerly executed and general-purpose distributed execution system. Thus, I believe that the way forward towards greater generality and evolvability is to leverage programming languages and their interface with the distributed execution system.

So far, this has been a little explored area of distributed futures systems. CIEL introduces a compiler to support lazy evaluation, but it requires a special-purpose language and is limited to stateless applications. JIT compilation is heavily used in distributed ML frameworks, but for ML-specific distributed futures systems such as PyTorch RPC, the operations are still eagerly executed and require the user to manually specify scheduling and placement. There are many ways in which programming language techniques could be applied, and above I have discussed a few:

1. Applying ownership types [66] to enhance fault tolerance in both single- and cross-application settings (see Section 4.7).

2. Just-in-time compilation to preserve the dynamicity of distributed futures but expand to lower-latency settings and execution on heterogeneous hardware.

3. Domain-specific languages to enable system extensibility in a safe and correct manner, with low developer effort required.

4. Integrating application semantics such as nondeterminism and side effects into the programming language, to provide more automatic and efficient recovery as compared to Exoflow (see Section 6.7).

5. Designing intermediate representations for distributed frameworks, to improve efficiency for cross-framework workflows.

7.3 Conclusion

This thesis presents the first steps towards an "operating system" for distributed and data-intensive applications. We introduce the distributed futures API and a system architecture that implements "virtual memory" for distributed futures applications. Through this architecture, we demonstrate an end-to-end approach to fault tolerance for distributed futures. This architecture has reached broad impact through the open-source distributed futures system Ray, which is being used today as a platform for ML and data processing applications. We conclude with directions for future work that aim to grow the generality and evolvability of the operating system, with a focus on composite distributed applications, increasingly heterogeneous clusters, and codesign of the programming language with the execution system.

Bibliography

[1] Airflow XComs. https://airflow.apache.org/docs/apache-airflow/stable/concepts/xcoms.html. Accessed: 2022-12-13.

[2] Akka. https://akka.io/.

[3] Akka Persistence. https://doc.akka.io/docs/akka/current/typed/index-persistence.html.

[4] An Overview of End-to-End Exactly-Once Processing in Apache Flink (with Apache Kafka, too!). https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html.

[5] Apache Airflow. https://airflow.apache.org/.

[6] Apache thrift. https://thrift.apache.org/.

[7] End-to-end mlops pipeline example on azure. https://github.com/microsoft/MLOps/tree/master/examples/KubeflowPipeline.

[8] Google Cloud Composer. https://cloud.google.com/composer.

[9] gRPC. https://grpc.io.

[10] Improved Fault-tolerance and Zero Data Loss in Apache Spark Streaming. https://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html.

[11] Kubeflow. https://www.kubeflow.org/.

[12] PyTorch - Remote Reference Protocol. https://pytorch.org/docs/stable/notes/rref.html.

[13] Ray. https://github.com/ray-project/ray.

[14] Ray v1.0. https://github.com/ray-project/ray/releases/tag/ray-1.0.0.

[15] Taming the OOM killer. https://lwn.net/Articles/317814/.

[16] Temporal. `https://temporal.io/`.

[17] Understanding Ownership - Rust. `https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html`.

[18] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[19] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT press, 1986.

[20] Marcos K Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. Memory disaggregation: why now and what are the challenges. *ACM SIGOPS Operating Systems Review*, 57(1):38–46, 2023.

[21] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[22] Remzi Can Aksoy and Manos Kapritsos. Aegean: replication beyond the client-server model. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 385–398, 2019.

[23] Alibaba. Emr remote shuffle service: A powerful elastic tool of serverless spark - alibaba cloud community. `https://www.alibabacloud.com/blog/emr-remote-shuffle-service-a-powerful-elastic-tool-of-serverless-spark_597728`, may 2021. (Accessed on 02/01/2022).

[24] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M Hellerstein, and Russell Sears. BOOM Analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European conference on Computer systems*, pages 223–236. ACM, 2010.

[25] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260. www.cidrdb.org, 2011.

[26] Lorenzo Alvisi and Keith Marzullo. Trade-offs in implementing causal message logging protocols. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 58–67. Citeseer, 1996.

[27] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[28] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14, 2009.

[29] Michael Armbrust. SPARK-20928: Continuous Processing Mode for Structured Streaming. `https://issues.apache.org/jira/browse/SPARK-20928`, 2017.

[30] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. Delta lake: high-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, 2020.

[31] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Mikroelektronik och informationsteknik, 2003.

[32] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[33] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[34] Anubhav Awasthi, Rajendra Gujja, and Mohit Saxena. Introducing amazon s3 shuffle in aws glue. `https://aws.amazon.com/blogs/big-data/introducing-amazon-s3-shuffle-in-aws-glue/`, Nov 2021. (Accessed on 10/16/2022).

[35] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[36] Henry C Baker Jr and Carl Hewitt. The incremental garbage collection of processes. *ACM SIGART Bulletin*, (64):55–59, 1977.

[37] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. {CORFU}: A shared log design for flash clusters. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 1–14, 2012.

[38] Mayank Bansal and Bo Yang. Zeus: Uber's highly scalable and distributed shuffle as a service - databricks. `https://databricks.com/session_na20/zeus-ubers-highly-scalable-and-distributed-shuffle-as-a-service`, July 2020. (Accessed on 02/01/2022).

[39] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems*, 4:430–449, 2022.

[40] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[41] John K Bennett, John B Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 168–176, 1990.

[42] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.

[43] Romil Bhardwaj, Alexey Tumanov, Stephanie Wang, Richard Liaw, Philipp Moritz, Robert Nishihara, and Ion Stoica. Escher: expressive scheduling with ephemeral resources. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 47–62, 2022.

[44] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[45] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[46] Robert D Blumofe, Philip A Lisiecki, et al. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, 1997.

[47] Dmitry Borovsky and Brian Cho. Cosco: An efficient facebook-scale shuffle service - databricks. `https://databricks.com/session/cosco-an-efficient-facebook-scale-shuffle-service`, May 2019. (Accessed on 01/19/2022).

[48] Maarten A. Breddels and Jovan Veljanoski. Vaex: big data exploration in the era of gaia. *Astronomy & Astrophysics*, 618:A13, oct 2018.

[49] Benjamin Brock, Aydın Buluç, and Katherine Yelick. Bcl: A cross-platform distributed data structures library. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[50] Jyothi Prasad Buddha and Reshma Beesetty. Step functions. In *The Definitive Guide to AWS Application Integration*, pages 263–342. Springer, 2019.

[51] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proceedings of the VLDB Endowment*, 15(8):1591–1604, 2022.

[52] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021.

[53] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[54] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 16. ACM, 2011.

[55] Lingfeng Cai, Xianglin Wei, Changyou Xing, Xia Zou, Guomin Zhang, and Xiulei Wang. Failure-resilient dag task scheduling in edge computing. *Computer Networks*, 198:108361, 08 2021.

[56] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, August 2017.

[57] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.

[58] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[59] Rick Cattell. Scalable sql and nosql data stores. *Acm Sigmod Record*, 39(4):12–27, 2011.

[60] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[61] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[62] DeQing Chen, Sandhya Dwarkadas, Srinivasan Parthasarathy, Eduardo Pinheiro, and Michael L Scott. Interweave: A middleware system for distributed shared state. In *International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 207–220. Springer, 2000.

[63] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[64] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98, 2012.

[65] Alvin Cheung, Natacha Crooks, Joseph M Hellerstein, and Matthew Milano. New directions in cloud programming. 2021.

[66] David G Clarke, John M Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64, 1998.

[67] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI '10, page 21, USA, 2010. USENIX Association.

[68] Charlie Curtsinger and Emery D Berger. Coz: Causal profiling.

[69] Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, et al. Bigdl: A distributed deep learning framework for big data. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 50–60, 2019.

[70] Pritam Damania, Shen Li, Alban Desmaison, Alisson Azzolini, Brian Vaughan, Edward Yang, Gregory Chanan, Guoqiang Jerry Chen, Hongyi Jia, Howard Huang, et al. Pytorch rpc: Distributed deep learning built on tensor-optimized remote procedure calls. *Proceedings of Machine Learning and Systems*, 5, 2023.

[71] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

[72] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[73] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[74] Polars Developers. Polars – user guide. `https://pola-rs.github.io/polars-book/user-guide/index.html`, 2022. (Accessed on 10/16/2022).

[75] Spark developers. Spark release 3.2.0. `https://spark.apache.org/releases/spark-release-3-2-0.html`, October 2021. (Accessed on 01/26/2022).

[76] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 401–414, 2014.

[77] Alexei A Efros. *Data-driven approaches for texture and motion*. University of California, Berkeley, 2003.

[78] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[79] Elmootazbellah N Elnozahy and Willy Zwaenepoel. On the use and implementation of message logging. In *Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing*, pages 298–307. IEEE, 1994.

[80] Elmootazbellah Nabil Elnozahy. *Manetho: fault tolerance in distributed systems using rollback-recovery and process replication*. PhD thesis, Rice University, 1994.

[81] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 251–266, New York, NY, USA, 1995. Association for Computing Machinery.

[82] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, apr 2014.

[83] Yuan Yu Michael Isard Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, and Pradeep Kumar Gunda Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. *Proc. LSDS-IR*, 8, 2009.

[84] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, 2019.

[85] Apache Software Foundation. Apache arrow datafusion documentation. `https://arrow.apache.org/datafusion/`, 2022. (Accessed on 10/16/2022).

[86] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[87] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[88] Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. Towards modern development of cloud applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 110–117, 2023.

[89] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 213–231, Carlsbad, CA, October 2018. USENIX Association.

[90] Ionel Gog, Michael Isard, and Martín Abadi. Falkirk wheel: Rollback recovery for dataflow systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 373–387, 2021.

[91] Jonathan Goldstein, Ahmed Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Ghosh Peshawaria, Tal Zaccai, and Irene Zhang. A.m.b.r.o.s.i.a: Providing performant virtual resiliency for distributed applications. *Proc. VLDB Endow.*, 13(5):588–601, jan 2020.

[92] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review*, 23(5):202–210, 1989.

[93] Robbie Gruener, Owen Cheng, and Yevgeni Litvin. Introducing petastorm: Uber atg's data access library for deep learning. `https://eng.uber.com/petastorm/`, September 2018. (Accessed on 01/19/2022).

[94] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[95] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 443–462, 2020.

[96] Yanfei Guo, Jia Rao, and Xiaobo Zhou. iShuffle: Improving hadoop performance with Shuffle-on-Write. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 107–117, San Jose, CA, June 2013. USENIX Association.

[97] Ajay Gupta. Revealing apache spark shuffling magic. `https://medium.com/swlh/revealing-apache-spark-shuffling-magic-b2c304306142`, may 2020. (Accessed on 02/01/2022).

[98] Robert H Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[99] Brandon Haynes, Amrita Mazumdar, Armin Alaghi, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. Lightdb: A DBMS for virtual reality video. *Proc. VLDB Endow.*, 11(10):1192–1205, 2018.

[100] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[101] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *9th Biennial Conference on Innovative Data Systems Research (CIDR 2019)*, 2019.

[102] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, jun 1997.

[103] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[104] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[105] 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, Berkeley, CA, USA, 2011. USENIX Association.

[106] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 588–604, 2021.

[107] J-H Hwang, Magdalena Balazinska, Alex Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *21st International Conference on Data Engineering (ICDE'05)*, pages 779–790. IEEE, 2005.

[108] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[109] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[110] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 691–707, 2021.

[111] Pete Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. *Distributed Shared Memory: Concepts and Systems*, pages 211–227, 1994.

[112] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[113] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering*, (1):23–31, 1987.

[114] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.

[115] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *Proc. VLDB Endow.*, 1(1):574–585, August 2008.

[116] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[117] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 71–86, 2015.

[118] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[119] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15, 2014.

[120] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[121] Jingui Li, Xuelian Lin, Xiaolong Cui, and Yue Ye. Improving the shuffle of hadoop mapreduce. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 1, pages 266–273, Bristol, UK, 2013. IEEE.

[122] Kai Li. *Shared virtual memory on loosely coupled multiprocessors*. Yale University, 1986.

[123] Kai Li. Ivy: A shared virtual memory system for parallel computing. *ICPP (2)*, 88:94, 1988.

[124] 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, 2014.

[125] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[126] Tianyu Li, Badrish Chandramouli, Sebastian Burckhardt, and Samuel Madden. Darq matter binds everything: Performant and composable cloud programming via resilient steps. In *Proceedings of the ACM on Management of Data*, 2023.

[127] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.

[128] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[129] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *ACM SIGPLAN Notices*, 23(7):260–267, 1988.

[130] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, Sangbin Cho, Eric Liang, and Ion Stoica. Exoshuffle: An extensible shuffle architecture. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 564–577, 2023.

[131] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, SangBin Cho, Eric Liang, and Ion Stoica. Exoshuffle-cloudsort. *arXiv preprint arXiv:2301.03734*, 2023.

[132] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory oltp recovery. In *2014 IEEE 30th International Conference on Data Engineering*, pages 604–615. IEEE, 2014.

[133] Simon Marlow. *Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming.* " O'Reilly Media, Inc.", 2013.

[134] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[135] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 513–526, 2017.

[136] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.

[137] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, mar 2008.

[138] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[139] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. Convergence analysis of distributed stochastic gradient descent with shuffling. *Neurocomput.*, 337(C):46–57, apr 2019.

[140] Konstantin Mishchenko, Ahmed Khaled, and Peter Richtarik. Random reshuffling: Simple analysis with vast improvements. In H. Larochelle, M. Ranzato,

R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 17309–17320, Virtual, 2020. Curran Associates, Inc.

[141] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[142] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. Ludwig: a type-based declarative deep learning toolbox, 2019.

[143] Luc Moreau. Hierarchical distributed reference counting. In *Proceedings of the 1st international symposium on Memory management*, pages 57–67, 1998.

[144] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051*, 2015.

[145] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[146] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[147] Derek G. Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM*, 59(10):75–83, September 2016.

[148] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[149] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. Tf.data: A machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, jul 2021.

[150] D.G. Murray. *A Distributed Execution Engine Supporting Data-dependent Control Flow*. University of Cambridge, 2012.

[151] Bruce Jay Nelson. Remote procedure call. 1981.

[152] Robert Nishihara, Philipp Moritz, Stephanie Wang, Alexey Tumanov, William Paul, Johann Schleier-Smith, Richard Liaw, Mehrdad Niknami, Michael I. Jordan, and Ion Stoica. Real-time machine learning: The missing pieces. In *Workshop on Hot Topics in Operating Systems*, 2017.

[153] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[154] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus: an architecture for extensible distributed systems. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 58–68, 1993.

[155] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):7, 2015.

[156] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 184–200, 2017.

[157] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, Red Hook, NY, USA, 2019. Curran Associates, Inc.

[158] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith S. Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. Velox: Meta's unified execution engine. *Proc. VLDB Endow.*, 15(12):3372–3384, 2022.

[159] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. Towards scalable dataframe systems, 2020.

[160] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *International Workshop on Memory Management*, pages 211–249. Springer, 1995.

[161] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. Scanner: Efficient video analysis at scale. *ACM Trans. Graph.*, 37(4):138:1–138:13, July 2018.

[162] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *Peer-to-Peer Systems IV: 4th International Workshop, IPTPS 2005, Ithaca, NY, USA, February 24-25, 2005. Revised Selected Papers 4*, pages 205–216. Springer, 2005.

[163] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 193–206, 2019.

[164] Hang Qu, Omid Mashayekhi, David Terei, and Philip Levis. Canary: A scheduling architecture for high performance cloud computing. *arXiv preprint arXiv:1602.01412*, 2016.

[165] Mark Raasveldt and Hannes Mühleisen. Duckdb: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery.

[166] Sriram Rao, Lorenzo Alvisi, and Harrick M Vin. The cost of recovery in message logging protocols. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):160–173, 2000.

[167] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[168] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. Themis: An i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[169] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. Tritonsort: A balanced large-scale sorting system. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 29–42, USA, 2011. USENIX Association.

[170] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {ZeRO-Offload}:

Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[171] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 130 – 136, 2015.

[172] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. {AIFM}: High-performance, application-integrated far memory. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 315–332, 2020.

[173] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[174] Salvatore Sanfilippo. Redis: An open source, in-memory data structure store. `https://redis.io/`, 2009.

[175] Danilo Sato, Arif Wider, and Windheuser Christoph. Continuous delivery for machine learning, Sep 2019.

[176] Peter Sbarski and Sam Kroonenburg. *Serverless architectures on AWS: with examples using Aws Lambda*. Simon and Schuster, 2017.

[177] Richard D Schlichting and Fred B Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3):222–238, 1983.

[178] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[179] Lior Shabtay and Adrian Segall. On the memory overhead of distributed snapshots. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 401–, New York, NY, USA, 1994. ACM.

[180] Mehul A. Shah, Amiato, and Chris Nyberg. Cloudsort: A tco sort benchmark. `http://sortbenchmark.org/2014_06_CloudSort_v_0_4.pdf`, June 2014. (Accessed on 01/24/2022).

[181] Min Shen. Rpc implementation to support pushing and merging shuffle blocks. `https://issues.apache.org/jira/browse/SPARK-32915`, Sep 2020. (Accessed on 10/16/2022).

[182] Min Shen, Ye Zhou, and Chandni Singh. Magnet: Push-based shuffle service for large-scale data processing. *Proc. VLDB Endow.*, 13(12):3382–3395, aug 2020.

[183] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[184] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: a high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[185] Sergei Sokolenko. How distributed shuffle improves scalability and performance in cloud dataflow pipelines. `https://cloud.google.com/blog/products/data-analytics/how-distributed-shuffle-improves-scalability-and-performance-cloud-dataflow-pipelines`, September 2018.

[186] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[187] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E Gonzalez, and Joseph M Hellerstein. Optimizing prediction serving on low-latency serverless dataflow. *arXiv preprint arXiv:2007.05832*, 2020.

[188] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E Gonzalez, Joseph M Hellerstein, and Jose M Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[189] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. Building scalable and flexible cluster managers using declarative programming. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 827–844, 2020.

[190] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[191] Andrew Stuart Tanenbaum and Robbert Van Renesse. A critique of the remote procedure call paradigm. In *Proc. Euteco'88 IR 123*, pages 1–11. 1988.

[192] PyTorch Team. torch.utils.data – pytorch documentation. `https://pytorch.org/docs/stable/data.html`, 2022. (Accessed on 10/16/2022).

[193] Ray Team. Ray datasets: Distributed data preprocessing. `https://docs.ray.io/en/latest/data/dataset.html`, 2022. (Accessed on 10/16/2022).

[194] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Ali Ghodsi, Michael Armbrust, Benjamin Recht, Michael Franklin, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the Twenty-Sixth ACM Symposium on Operating Systems Principles*, SOSP '17. ACM, 2017.

[195] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall International (UK) Ltd., 1996.

[196] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *International Workshop on Mobile Object Systems*, pages 49–64. Springer, 1996.

[197] Qian Wang, Rong Gu, Yihua Huang, Reynold Xin, Wei Wu, Jun Song, and Junluan Xia. Nadsort. `http://sortbenchmark.org/NADSort2016.pdf`, 2016. (Accessed on 01/26/2022).

[198] Stephanie Wang. Analyzing memory management and performance in dask-on-ray. `https://medium.com/distributed-computing-with-ray/analyzing-memory-management-and-performance-in-dask-on-ray-930a2236b70d`, June 2021. (Accessed on 01/26/2022).

[199] Stephanie Wang, Benjamin Hindman, and Ion Stoica. In reference to rpc: It's time to add distributed memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 191–198, New York, NY, USA, 2021. Association for Computing Machinery.

[200] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. Lineage stash: fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 338–352, 2019.

[201] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for Fine-Grained tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 671–686, Virtual, April 2021. USENIX Association.

[202] Stephanie Wang, Edward Oakes, and Frank Luan. Ownership nsdi'21 artifact. `https://github.com/stephanie-wang/ownership-nsdi2021-artifact`.

[203] Yandong Wang, Cong Xu, Xiaobing Li, and Weikuan Yu. Jvm-bypass for efficient hadoop shuffling. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 569–578, Cambridge, MA, USA, 2013. IEEE.

[204] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, Austin, Texas, 2010. Python in Science Conference.

[205] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[206] Chenggang Wu, Jose M. Falerio, Yihan Lin, and Joseph M. Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[207] Reynold Xin. Apache spark the fastest open source engine for sorting a petabyte. `https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html`, October 2014. (Accessed on 01/19/2022).

[208] Reynold Xin. Project hydrogen: Unifying state-of-the-art ai and big data in apache spark. Spark + AI Summit, 2018.

[209] M. Young, A. Tevanian, R. Rashid, D. Golub, and J. Eppinger. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, page 63–76, New York, NY, USA, 1987. Association for Computing Machinery.

[210] Peifeng Yu and Mosharaf Chowdhury. Fine-grained gpu sharing primitives for deep learning applications. *Proceedings of Machine Learning and Systems*, 2:98–111, 2020.

[211] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *In Correct Hardware Design and Verification Methods (CHARME '99), Laurence Pierre and Thomas Kropf editors. Lecture Notes in Computer Science, Springer-Verlag.*, volume 1703, pages 54–66, June 1999.

[212] 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*, pages 2–2. USENIX Association, 2012.

[213] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[214] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[215] Kai Zeng, Sameer Agarwal, Ankur Dave, Michael Armbrust, and Ion Stoica. G-ola: Generalized on-line aggregation for interactive analysis on big data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 913–918, New York, NY, USA, 2015. Association for Computing Machinery.

[216] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204, 2020.

[217] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. Riffle: Optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[218] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[219] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: Efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 641–656, New York, NY, USA, 2021. Association for Computing Machinery.

[220] Siyuan Zhuang, Stephanie Wang, Eric Liang, Yi Cheng, and Ion Stoica. {ExoFlow}: A universal workflow system for {Exactly-Once}{DAGs}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 269–286, 2023.

Appendix A

Ownership system protocols

The following appendix is adapted from the previously published paper *Ownership: A Distributed Futures System for Fine-Grained Tasks* [201].

A.1 Distributed Reference Counting

Type	Description
Local reference	A flag indicating whether the `DFut` has gone out of the process's scope.
Submitted task count	Number of tasks that depend on the object that were submitted by this process and that have not yet completed execution.
Borrowers	The set of worker IDs of the borrowers created by this process, by passing the `DFut` as a first-class value.
Nested `DFuts`	The set of `DFuts` that are in scope and whose values contain this `DFut`.
Lineage count	Number of `Tasks` that depend on this `DFut` that may get re-executed. This count only determines when the lineage (the `Task` field) should be released; the *value* can be released even when this count is nonzero.

Table A.1: Full description of the `References` field in Table 4.2. Every process with an instance of the `DFut` (either the owner or a borrower) maintains these fields.

If a `DFut` never leaves the scope of its owner, it does not require a distributed reference count. This is because the owner always has full information about which pending tasks require the object. However, since our API allows passing `DFuts` to other tasks as first-class values, we use a distributed reference count to decide when the object is out of scope.

Our reference counting protocol is similar to existing solutions [160, 143]. As explained in Section 4.4.2, the reference count is maintained with a tree of processes. Each process keeps a local set of borrower worker IDs, i.e. its children nodes in the tree. Most of the messages needed to maintain the tree are piggy-backed on existing protocols, such as for task scheduling.

A borrower is created when a task returns a `SharedDFut` to its parent task, or passes a `SharedDFut` to a child task. In both cases, the process executing the task adds the ID of the worker that executes the parent or child task to its local borrower set.

In many cases, a child task will finish borrowing the `DFut` by the time it has finished execution. Concretely, this means that the worker executing the child task will no longer have a local reference to the `DFut`, nor will it have any pending dependent tasks. Thus, when the worker returns the task's result to its owner, the owner can remove the worker from its local set of borrowers, with no additional messages needed. This optimization is important for distributing load imposed by reference counting among the borrowers, rather than requiring all reference holders to be tracked by the owner.

However, in some cases, the worker may borrow the `DFut` past the duration of the child task. There are two cases: (1) the worker passed the `DFut` as an argument to a task that is still pending execution, or (2) the worker is an actor and stored the `DFut` in its local state. In these cases, the worker notifies the owner that it is still borrowing the `DFut` when replying with the task's return value.

Eventually, the owner must collect all of the borrowers in its local set. It does this by sending a request to each borrower to reply once the borrower's reference count has gone to zero. Borrowers themselves never delete from their local set of borrowers. Once a borrower no longer has a reference or any pending dependent tasks, it replies to the owner with its accumulated local borrower set. The owner then removes the borrower, merges the received borrowers into its local set and repeats the same process with any new borrowers. If a borrower dies before it can be removed, the owner removes it upon being notified of the borrower's death.

When a `DFut` is *returned* by a task, it results in a nested `DFut`. Nested `DFuts` can be automatically flattened, e.g., when submitting a dependent task, but we must still account for nesting during reference counting. We do this by keeping a set of `DFuts` whose values contain the `DFut` in question in the ownership table (Table 4.2). The `DFut`'s value is pinned if its nested set is non-empty.

A.2 Formal Specification

We developed a formal specification for the ownership-based system architecture [202]. It models the system state transitions of the ownership table for task scheduling, garbage collection, and worker failures. The goal is to check the correctness of the

system design, which is manifested in the following properties:

- Safety: A future's lineage information is preserved as long as a task exists that depends on the value of the future. This is defined recursively: at any time, either the value of a future is stored inline (thus cannot be lost), or all futures that this future depends on for computing its value must be safe. Formally, it means the following invariant holds at any given time: $\forall x$,

$$\begin{aligned}&\text{LineageInScope}(x) \triangleq \\ &\quad \lor\, x = \texttt{INLINE\_VALUE} \\ &\quad \lor\, \forall arg \in x.args : \text{LineageInScope}(arg)\end{aligned}$$

- Liveness: The system will eventually execute all tasks and resolve all future values, even in case of failures, i.e., all `Get` calls eventually return.

- No Resource Leakage: The system will eventually clean up all task states and future values, after the all references to futures become out-of-scope.

We checked the model using the TLA$^+$Model Checker [211] for up to 3 levels of recursive remote function calls, where each function creates up to 3 futures, and verified that the safety and liveness properties hold in more than 44 million distinct states. Currently, the model does not include first-class futures or actors; we plan to include these and open-source the full TLA$^+$specification in the future.

Appendix B

Exoshuffle libraries, applications, and evaluation

The following appendix is adapted from the previously published paper *Exoshuffle: An Extensible Shuffle Architecture* [130] and includes additional contributions of Frank Sifei Luan et al.

B.1 Expressing Shuffle Strategies with Distributed Futures

B.1.1 Pre-Shuffle Merge

Riffle [217] is a specialized shuffle system built for Spark. Its key optimization is merging small map output blocks into larger blocks, thereby converting small, random disk I/O into large, sequential I/O before shuffling over the network to the reducers. The merging factor F is either pre-configured, or dynamically decided based on a block size threshold. As soon as F map tasks finish on an executor node, their output blocks ($F \times R$) are merged into R blocks, each consisting of F blocks of data from the map tasks. This strategy, illustrated in Figure 5.2b, is implemented in Listing 1 (`shuffle_riffle`). The code additionally takes `F` as the merging factor, and a merge function which combines multiple map outputs into one.

Riffle's key design choice is to merge map blocks *locally* before they are pulled by the reducers, as shown in the highlighted lines. For simplicity, the code assumes that the first F map tasks are scheduled on the first worker, the next F map tasks on the second worker, etc. In reality, the locality can be determined using scheduling placement hints or runtime introspection (§5.4.2) Section 5.5.1 shows that this implementation of Riffle-style shuffle improves the job completion time over simple shuffle.

B.1.2 Push-based Shuffle

Push-based shuffle (Fig 5.2c) is an optimization that pushes shuffle blocks to reducer nodes as soon as they are computed, rather than pulling blocks to the reducer when they are required. Magnet [182] is a specialized shuffle service for Spark that performs this optimization by merging intermediate blocks on the reducer node before the final reduce stage. This improves I/O efficiency and data locality for the final reduce tasks. `shuffle_magnet` in Listing 1 implements this design.

B.1.3 Straggler Mitigation

Distributed futures enable dynamic task graphs by nature, making it ideal for detecting and reacting to stragglers during runtime.

Speculative Execution One way to handle stragglers is through speculative execution. Tasks that are suspected to be stragglers can be duplicated, and the system chooses whichever result is available first. This can be accomplished with distributed futures using the `ray.wait` primitive, as shown in Listing 4.

```
map_out = ...
_, timeout_tasks = ray.wait(map_out, timeout=TIMEOUT)
duplicates = []
for task in timeout_tasks:
  duplicates.append(map.remote(task.args))
for t1, t2 in zip(timeout_tasks, duplicates):
  t, _ = ray.wait([t1, t2], num_returns=1)
  map_out[t1.id] = t
```

Listing 4: Mitigating stragglers with speculative execution.

Best-effort Merge Shuffle systems including Riffle and Magnet also implement "best-effort merge", where a timeout can be set on the shuffle and merge phase [217, 182]. If some merge tasks are cancelled due to timeout, the original map output blocks will be fetched instead. This ensures straggler merge tasks will not block the progress of the entire system. Best-effort merge can be implemented in Exoshuffle as shown in Listing 5 using an additional `ray.cancel()` API which cancels the execution of a task. The cancelled task's input, which are the original map output blocks, will then be directly passed to the reducers. This way, the task graph is dynamically constructed as the program runs, adapting to runtime conditions while still enjoying the benefits of transparent fault tolerance provided by the system.

```
map_out = ...
merge_out = ...
_, timeout_tasks = ray.wait(merge_out, timeout=TIMEOUT)
for task in timeout_tasks:
  ray.cancel(task)
  merge_out[task.id] = task.args
out = [reduce.remote(merge_out[:, r]) for r in range(R)]
ray.wait(out)
```

Listing 5: Mitigating stragglers via task cancellation.

B.1.4 Data Skew

Data skew can be prevented at the data management level using techniques such as key salting, or periodic repartitioning. However, it is still possible for skews to occur during ad-hoc query processing, especially for those queries involving joins and group-bys. Data skew during runtime can cause the working set of a reduce task to be too large to fit into executor memory.

Dynamic repartitioning solves this problem by further partitioning a large reducer partition into smaller ones. This is straightforward to implement since the distributed futures programming model enables dynamic tasks by nature. Listing 6 shows that we can recursively split down a reducer's working set until it fits into a predefined memory threshold.

```
@ray.remote
def reducer(*parts):
  total_size = [part.size() for part in parts]
  if total_size > THRESHOLD:
    L = len(parts) // 2
    return flatten([
      reducer.remote(*parts[:L]),
      reducer.remote(*parts[L:])])
```

Listing 6: Dynamic repartitioning for skewed partitions.

B.2 Expressing Shuffle Applications

Because Exoshuffle implements shuffle at the application level, it can easily interoperate with other applications. Here, we demonstrate two example applications that use fine-grained pipelining with shuffle to improve end-to-end performance. These applications are evaluated in Appendix B.3.3.

B.2.1 Online Aggregation with Streaming Shuffle

Online aggregation [102] is an interactive query processing mode where partial results are returned to the user as soon as some data is processed, and are refined as progress continues. This is especially useful when the query takes a long time to complete. Online aggregation is difficult to implement in MapReduce systems because they require all outputs to be materialized before being consumed. Past work made in-depth modifications to Hadoop and Spark to support online aggregation [67, 215].

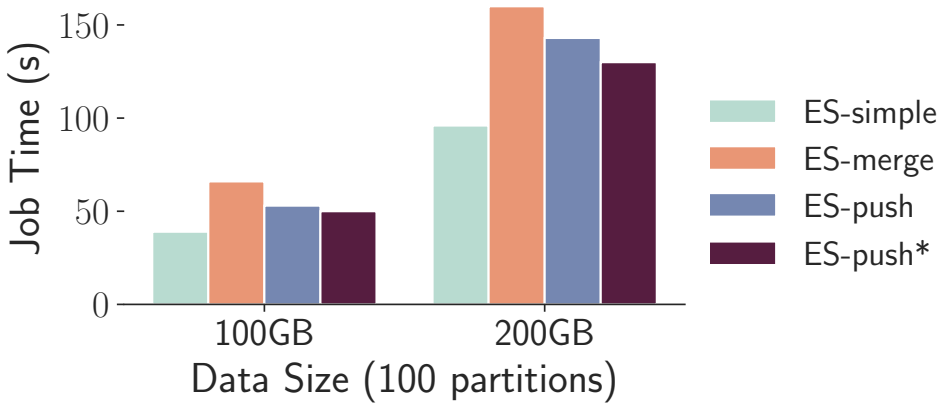
```
1  def streaming_shuffle(map, reduce, print_aggregate):
2    reduce_states = [None] * R
3    for rnd in range(N):
4      map_results = [map.remote(M*rnd+i) for i in range(M)]
5      ray.wait(reduce_states)
6      reduce_states = [
7        reduce.remote(reduce_state, *map_results[:, r])
8        for r, reduce_state in enumerate(reduce_states)]
9      print_aggregate.remote(reduce_states)
10   return ray.get(reduce_states)
11
12 def model_training(trainer, data):
13   shuffle_out = shuffle(data, ...)
14   for epoch in range(EPOCHS):
15     next_shuffle_out = shuffle(data, ...)
16     for block in shuffle_out:
17       trainer.train(ray.get(block))
18     shuffle_out = next_shuffle_out
```

Listing 7: Streaming shuffle and pipelined data loading for ML.

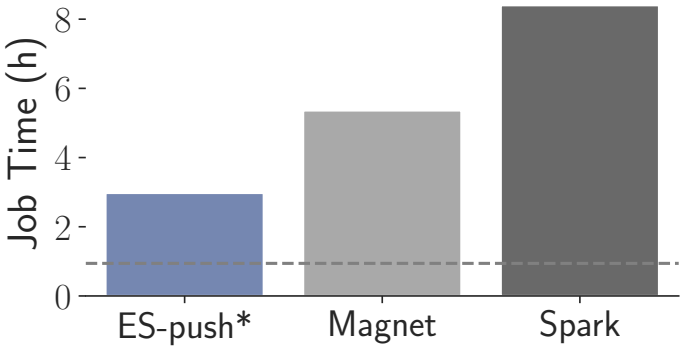
Online aggregation is straightforward to implement in Exoshuffle without the need to modify the underlying distributed futures system. Listing 7 shows the `streaming_shuffle` routine. It requires a modified `reduce` function that takes a reducer state and a list of map outputs and returns an updated state, and an `aggregate` function which combines the reducer states to produce aggregate statistics. Shuffle is executed in rounds. At the end of each round, the aggregation function is invoked with the reducer outputs, and will asynchronously print an aggregate statistic (e.g. sum) to the user. Note that the Exoshuffle user can simply swap between `simple_shuffle` and `streaming_shuffle` to get the semantics they desire.

B.2.2 Distributed ML Training with Pipelined Shuffle

Exoshuffle also enables fine-grained pipelining for ML training, as illustrated in Section 5.1. In Listing 7, `model_training` shows the code skeleton. On line 13, the `shuffle` function (could be any in Listing 1) returns a set of distributed futures pointing to reducer outputs. They are passed immediately to the model trainer while



(a) In-memory sort on 10 SSD nodes.



(b) 100 TB on 100 HDD nodes.

Figure B.1: Comparing job completion times on the Sort Benchmark. The dashed lines indicate the theoretical baseline (§5.5.1.1). Exoshuffle is abbreviated as ES.

shuffle executes asynchronously. As soon as a reducer block becomes available, the model trainer acquires it (line 17) and send it to the GPU for training. This achieves the fine-grained pipelining described in Section 5.1.

B.3 Evaluation

B.3.1 Performance Comparison of Shuffle Libraries

In-memory Performance. Figure B.1a shows that when data fits in memory, ES-simple is actually the fastest algorithm compared to all other variants. This is because the other algorithms create copies of data by merging them, triggering unnecessary

disk spilling. Magnet observes similar behavior for small datasets[1].

Conclusion. These experiments show that the shuffle algorithms provided by Exoshuffle offer the same performance benefits as their monolithic counterparts. Furthermore, the most performant shuffle algorithm depends on the data size and hardware configuration, and Exoshuffle offers the flexibility to choose the most suitable algorithm at the application level, without having to deploy multiple systems.

B.3.2 Shuffle Scalability

To test performance at large scale, we run the Sort Benchmark on 100 TB data with 50 000× 2 GB input partitions on a cluster of 100× d3.2xlarge VMs. For Exoshuffle, we run the ES-push* variant since it is the most optimized for scale. For baselines, we run both Spark's native shuffle (Spark) and its push-based shuffle service (Magnet). We run both baselines with compression on because Spark without compression becomes unstable at this scale.

Figure B.1b shows the results. Exoshuffle outperforms both native Spark shuffle and the push-based shuffle service Magnet, despite Spark's compression reducing total bytes spilled by 40%. Magnet improves shuffle performance by 1.6× because it reduces random disk I/O. Exoshuffle further improves performance over push-based Spark by 1.8×. This difference comes from reduced write amplification in ES-push*, which spills only the merged map outputs, while Magnet also spills the un-merged map outputs. These additional writes provide faster failure recovery through improved durability, albeit at the cost of performance. Exoshuffle allows the application to choose between these tradeoffs by using ES-push vs. ES-push*.

B.3.3 Shuffle Applications

Next, we show how Exoshuffle can extend distributed shuffle support for a broader set of applications.

B.3.4 CloudSort

The CloudSort competition [180] calls for the most cost-efficient way to sort 100 TB of data on the public cloud. We ran Exoshuffle-CloudSort on a cluster of 40× i4i.4xlarge nodes with input and output data stored on Amazon S3, and set a new world record of $0.97/TB [131]. This is 33% more cost-efficient than the previous world record set in 2016. The previous entry used a heavily modified version of Spark for the CloudSort workload [197]. In contrast, Exoshuffle-CloudSort is only hundreds of lines of application code running on a release version of Ray.

[1] The Spark 3.3.1 documentation states: "Currently [Magnet] is not well suited for jobs/queries which runs quickly dealing with lesser amount of shuffle data."

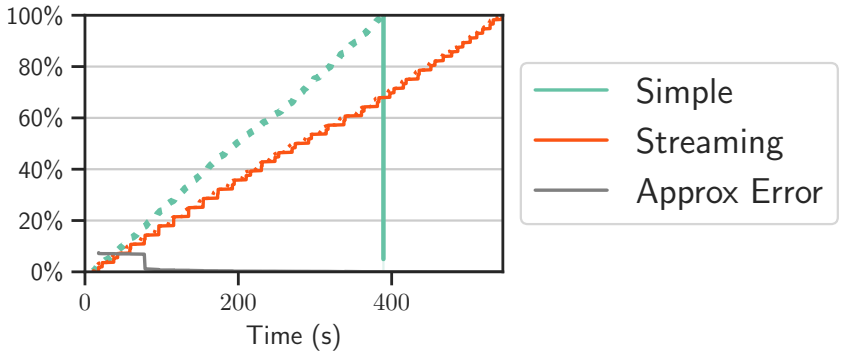


Figure B.2: Online aggregation. Dotted lines show map progress; solid, reduce progress.

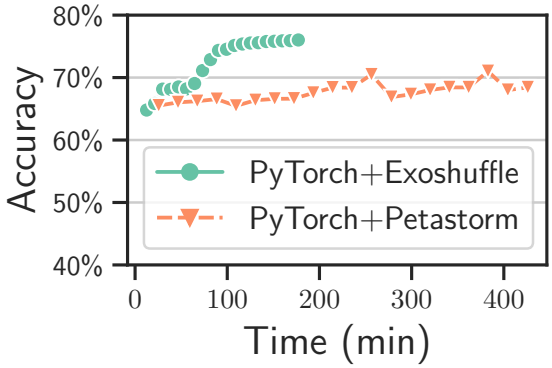


Figure B.3: Single-node ML training for 20 epochs.

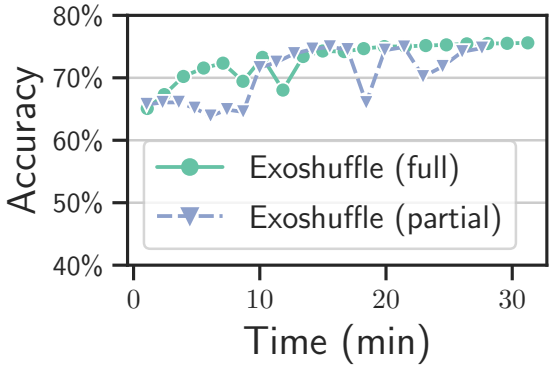


Figure B.4: 4-node, distributed ML training for 20 epochs.

To account for the fact that the cloud hardware costs have lowered since 2016, we take the setup from the previous record-winning entry and look up its cost on today's Alibaba Cloud. Table B.1 shows that the same amount of cloud resources would cost $115 today. Still, Exoshuffle-CloudSort achieves another 15% cost reduction beyond this result. We calculate another theoretical baseline of simply shuffling 100 TB data through the AWS network (without sorting), which would cost $74. This puts our record within 31% of the theoretical limit. This result demonstrates that Exoshuffle can achieve state-of-the-art performance and cost-efficiency for large-scale shuffle.

System	Cost
NADSort (2016)	$1.44/TB
NADSort (2022, extrapolated)	$1.15/TB
Exoshuffle-CloudSort (2022)	**$0.97/TB**

Table B.1: CloudSort costs over years.

B.3.5 Online Aggregation with Streaming Shuffle

We use a 1 TB dataset containing 6 months of hourly page view statistics on Wikipedia. We run an aggregation to get the ranking of the top pages by language on 10× r6i.2xlarge nodes with data loaded from S3. Figure B.2 shows the difference between regular and streaming shuffle. The streaming shuffle takes 1.4× longer to run in total due to the extra computation needed to produce partial results. However, with streaming shuffle, the user can get partial aggregation results within 8% error[2] of the final result in 18 seconds, 22× faster than regular shuffle. Exoshuffle makes it easy to switch between `simple_shuffle` and `streaming_shuffle` to choose between partial result latency and total query run time.

B.3.6 Distributed ML Training

Many distributed training frameworks already run on top of Ray. By offering Exoshuffle as a library, we enable these workloads to leverage scalable shuffle. We demonstrate Exoshuffle's ability to support fine-grained pipelining for ML training using the Ludwig framework [142] to train a deep classification model TabNet on the HIGGS dataset (7.5 GB). Ludwig integrates ML data loaders with the PyTorch training framework [157]. Efficient training requires randomly shuffling the data per epoch before sending it into the GPU for training.

We first run the ML training on a single g4dn.4xlarge instance. We compare two versions of Ludwig: Ludwig 0.4.0 uses Petastorm [93], which prefetches data

[2]Error is computed using the KL-divergence $D_{KL} = \sum p \log(p/\hat{p})$ where p is the true statistic and $\hat{p}$ is the sample statistic.

in batches into a per-process memory buffer and performs random shuffle in the buffer. This approach makes the shuffle window size limited by the memory buffer size (§5.2.2). In this experiment, we set the shuffle window size to 9% of the total data size to avoid OOM errors. In comparison, Ludwig 0.4.1 uses Exoshuffle offered through Ray Data [193]. It pipelines data loading and shuffling with GPU training (§5.1), and supports full shuffle across loading batches by storing data in the shared-memory object store. Figure B.3 shows that model training with Exoshuffle is 2.4× faster end-to-end thanks to the fine-grained pipelining. The model also converges faster per-epoch and to a higher accuracy, because Exoshuffle performs complete random shuffling between epochs, whereas Petastorm's random shuffle is limited to subsets of the data.

Next, we run the training on 4 g4dn.xlarge nodes to show the distributed shuffle performance. Ludwig 0.4.x has known bugs with distributed training, so we could not compare Petastorm with Exoshuffle. Instead, we use the latest Ludwig 0.6.0 and compares two shuffle strategies with the Exoshuffle-based data loader: full shuffle (the default) and partial shuffle. For partial shuffle, we emulate the Petastorm behavior and perform random shuffling only in each in-memory batch. Figure B.4 shows that per-epoch time is slightly faster with partial shuffle, since it is fully local, but the convergence accuracy is slightly lower because of the less random shuffling of training data. This example demonstrates that Exoshuffle gives the developer the flexibility to choose the best shuffle strategy based on their training needs, while providing high-throughput data loading and shuffling.

Appendix C

Exoflow system design and evaluation

The following appendix is adapted from the previously published paper *ExoFlow: A Universal Workflow System for Exactly-Once DAGs* [220] and includes additional contributions of Siyuan Zhuang et al.

C.1 Architecture

We further describe the Exoflow design and the requirements of the pluggable execution backend and persistent storage.

C.1.1 Workflow execution

The workflow control layer is implemented using the system Ray [145]. Ray provides remote task invocation, distributed immutable memory, and distributed actors. However, Ray only provides at-most-once or at-least-once guarantees and lacks built-in persistence for memory and actors. Thus, Ray tasks and actors are distinct from workflow tasks and actors, which execute exactly-once and can be natively checkpointed.

We use Ray actors to implement the workflow controller and task executors (Figure 6.5). The controller uses Ray's *distributed futures* [201] to coordinate task execution and checkpointing. Distributed futures are an asynchronous extension of RPC where each invocation returns a future pointing to the eventual and possibly remote return value. Ray actors and distributed futures also directly implement application-facing references (Section 6.3).

We build on Ray for three reasons: (1) futures make it simple for the controller to manage concurrent task execution and checkpointing, (2) passing remote values by reference avoids bottlenecks from large task outputs being passed directly through the

centralized controller, and (3) the RPC-like interface straightforwardly and efficiently wraps other execution backends. For example, the Lambdas backend is implemented by wrapping a synchronous Lambda invocation in a Ray task.

The controller is a state machine where the state describes the current execution status of a workflow DAG and is persisted in storage. On `run`, the controller logs the workflow DAG specification (arguments, `Opts`, etc.) to durable storage and triggers execution. On each iteration of the event loop, the controller may select a workflow task whose inputs are ready and submit the task to an executor. For example, in Figure 6.5, the controller submits `C` to executor 1 and immediately receives back the distributed future `Ref(1bf)`. The controller uses this system-internal `Ref` to wait for task completion, and then passes it to downstream workflow tasks (e.g., `D`).

Checkpointing is carried out asynchronously by background threads on the executors, enabling parallel and distributed checkpoints that are not bottlenecked by the centralized controller. To checkpoint an output, the executor asynchronously writes a copy to a deterministic storage location (e.g., `w0/B/output` in Figure 6.5). The controller considers the checkpoint done once it is fully written. For convenience, the controller can also synchronize the checkpoint by requesting a signal from the executor (controller to executor 2 in Figure 6.5).

Checkpoint synchronization is required: (1) at the end of a workflow, (2) before executing a task with `can_rollback=False`, and (3) before executing a task with a `rollback` option. Section 6.5 evaluates a simple policy that synchronizes all pending checkpoints for a workflow in any of these cases and shows that this provides sufficient performance for key applications. A more sophisticated policy may synchronize only the minimum necessary.

Exoflow handles passing and checkpointing application references (Section 6.3.4). When a task finishes, the executor replaces any `Refs` and `ActorRefs` appearing in the task's output with placeholders, e.g., `x` in Figure 6.5. When passing the output to another task, the controller also passes a list of concrete references (`Ref(e02)` for `x`) used by the executor to fill the placeholders. Task checkpoints include a list of `Ref` checkpoint locations, which are written in parallel and distributed fashion. The controller restores and swaps `Refs` after a failure.

If a workflow task returns a `WorkflowDAG` as its output, the controller simply records the sub-workflow (if `checkpoint=True`), points the output of the parent task to the output of the sub-workflow, then resumes execution.

C.1.2 Workflow recovery

The controller handles task and checkpoint failures. In both cases, the protocol rolls back any previous outputs as needed, then rolls "forward" by re-executing workflow tasks.

The first step is to determine the re-execution task frontier. For example, suppose `C` in Figure 6.5 fails because we lost `A`'s cached output `Ref(be5)`. Then, we walk

the DAG backwards from `C` and add each visited task node to the re-execution set. For each task, we check argument availability, i.e. whether the value has a checkpoint or a live `Ref`. If all arguments are available, then we terminate. Else, we add the tasks that create the arguments (`A`) to the re-execution set. If a visited task has `deterministic=False`, then we also add all tasks downstream to the re-execution set. Thus, if `C` fails and we need to re-execute A, we also re-execute B, even though it has a checkpoint.

From the re-execution task set, we carry out rollback. In reverse-topological order of the re-execution set, we first clear any cached output `Refs` and output checkpoints, e.g., `/w0/B/output` and `/w0/B/x` for `B`. If it has a `rollback` task, then we re-execute this task, using the same protocol as normal task execution. Finally, we resume workflow execution as normal, starting from the earliest task frontier of the re-execution set.

Critical controller state is persisted, so recovering from controller failure is straightforward. On failure, all in-memory controller state (the table in Figure 6.5) is wiped, including any `Refs`. On restart, the controller simply scans persistent storage for incomplete workflows, rebuilds its in-memory table, then re-executes them using the described protocol.

Correctness. We provide informal proofs that the final outputs are consistent (Definition 6.1). During normal execution, this follows from the execution protocol: starting from a consistent prefix of outputs, executing a task will produce another consistent prefix.

For recovery, we first consider reconstruction of internal outputs, i.e. values returned by workflow tasks. If the task is deterministic, then the reconstructed output will match the original. If the task is nondeterministic, then the described rollback procedure returns execution to a consistent prefix that does not include any results downstream to the original output.

Next, we consider external outputs: tasks with `can_rollback=False` or `rollback` defined. For a task T with `can_rollback=False`, the application guarantees idempotence, so it is enough to show that once T begins, the failure-free execution will include the same inputs for T. To show this, we rely on Invariant 6.1 (Section 6.3.3) and checkpoint synchronization (Appendix C.1.1). The system synchronizes the partition provided by Invariant 6.1 before submitting T; thus once T begins, any future recovery procedure will never add T to the rollback set.

If T instead has `rollback` defined, we must show that if T fails, `rollback` will complete with the same view of inputs as T's previous execution, before re-executing T. Invariant 6.2 and checkpoint synchronization guarantee that we can deterministically and idempotently recreate `rollback`'s original inputs.

Correctness also requires preventing conflicts between different executions of the same task. For task checkpoints, if the backend's failure detection for executors is reliable, then by the time we re-execute T, we can be sure that there is no concurrent

checkpoint in progress. Under unreliable failure detection, the Exoflow controller assigns unique checkpoint locations to prevent races between concurrent executions. This requires one extra durable write before each task execution to record the expected checkpoint location.

For a task that returns `Refs` or `ActorRefs`, the execution backend can provide reliable failure detection for references by killing all copies of a `Ref` *before* reporting failure to Exoflow. Alternatively, a safe and efficient method that works for both crash and fail-stop failures is to generate unique references for each execution.

C.1.3 Execution backends

Integration. Exoflow references are compatible with existing third-party mechanisms for task communication and recovery. For example, Ray does not provide exactly-once semantics, but it does automatically reconstruct `Refs` created by deterministic (at-least-once) tasks [201]. Exoflow encourages hierarchical recovery, wherein the execution backend can attempt to handle `Ref` failures first, then throw unrecoverable errors up to the workflow controller.

Exoflow is compatible with backends that use logging and checkpointing. In general, log-based tasks would use `deterministic=True` and `can_rollback=False` annotations, while checkpoint-based tasks would use `deterministic=False` and `can_rollback=True`. The backend can also directly leverage Exoflow for checkpointing instead of supplying a user-defined `rollback` function; this shifts the responsibility of checkpoint coordination to Exoflow and automatically enables optimizations such as overlapping with execution.

Preventing leaks. The workflow layer ensures that previous `Refs` and pending checkpoints do not leak; invalid `Refs` and checkpoints are dropped during rollback. The execution backend must additionally prevent resource leaks for dead `Refs`. Dead `Refs` can be deleted via reference counting (the controller calls back to the backend once a `Ref` goes out of scope) or garbage collection (the backend scans the controller's in-memory state for dead `Refs`).

C.2 Implementation

Exoflow is built on Ray v2.0.1, which uses gRPC [9] for tasks and actors and a custom shared-memory object store for `Ref` storage [145]. Exoflow is implemented as a Ray Python program in 4k LoC.

We implemented two execution backends for Exoflow: Ray itself ("Exoflow-Ray") and the serverless FaaS offering AWS Lambdas ("Exoflow-Lambdas"). In each case, a typical deployment would use one Ray node to host the Exoflow controller.

In Exoflow-Lambdas, the controller node takes the place of the gateway provided by AWS for their proprietary serverless workflow offering (Step Functions).

We chose to implement Exoflow on Ray for three reasons:

1. Support for first-class references to immutable data, which we use to implement `Refs`.
2. Support for actors (stateful workers), which we use to implement `ActorRefs`.
3. Low task and actor overhead, similar to pure RPC.

We also use Ray actors to implement executors. Workflow tasks are stateless, but we use actors to store execution state about checkpoints that are pending after task completion.

To build Exoflow on another actor system such as Akka [2] or Orleans [42], we must implement `Refs`. This is straightforward for workloads that only pass small data. For data-intensive workflows, one can build a custom in-memory store that is tightly coupled to executors, as in Ray, or use an external key-value store. The latter requires low implementation effort, but may result in poor locality. It is ideal if the execution backend cannot be modified, e.g., to support values larger than the Lambdas response size in Exoflow-Lambdas.

C.3 Evaluation

C.3.1 Online-offline graph processing

Distributed graph processing systems can be generally divided into stream vs. batch processing [136]. Streaming systems can handle continuous updates and produce timely results, but may not offer the same precision as batch systems.

We use references in Exoflow to link stream and batch graph processing, producing a single application that can both handle online queries and produce periodic exact results. We use Ray actors to implement a version of Kineograph [64], a streaming graph processing system that uses distributed snapshots for consistency. Each workflow task ingests one epoch of incoming graph updates to compute a graph snapshot and an online approximate result, and we periodically pass the snapshot in-memory to another workflow task that uses Spark to compute the full result.

We evaluate on the SNAP Twitter follower network dataset [118] (41M nodes and 1.5B edges), with each input record representing an edge insert event. We run the push-model TunkRank algorithm used by Kineograph to compute Twitter user influences on a 3-node r3.8xlarge cluster, 1 for streaming and 2 for the Spark cluster. We use two Ray actors to process the input stream and use Exoflow to checkpoint and pass the `ActorRefs` between streaming tasks. Each streaming task represents a 10-second epoch and also returns 4 `Refs` that represent the partitioned graph snapshot. These `Refs` are passed to a Spark task every 20 epochs. Latency is reported for

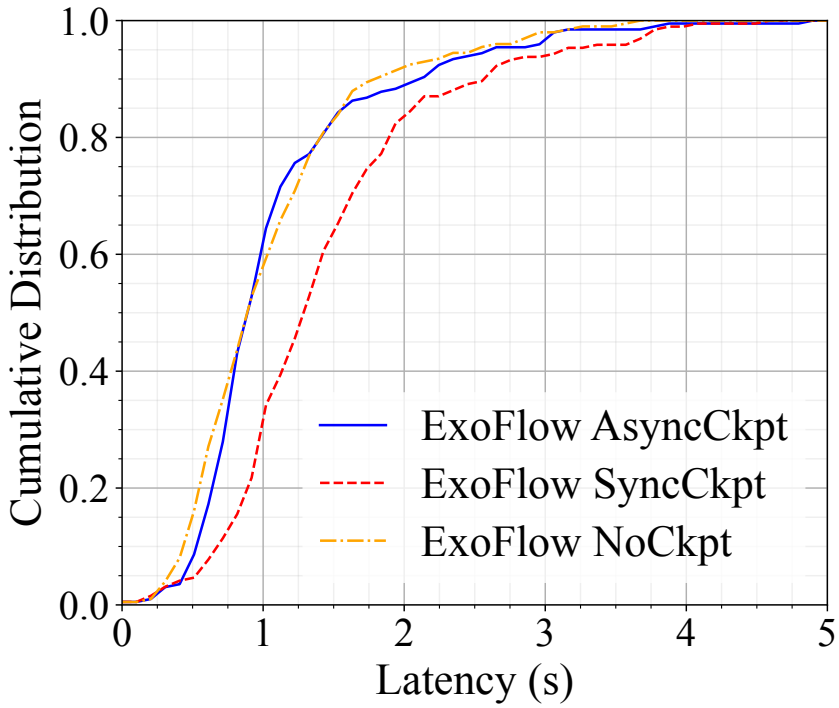


Figure C.1: **(c)** Latency CDF of online-offline graph processing.

200 epochs, after an initial warmup of 150 epochs. The average digestion rate is 44.94k tweets per second with our dataset. Kineograph achieves about 40k tweets per second with 2 ingest node + 48 graph nodes with a similar setting. We outperform Kineograph likely because we utilize shared memory for data passing, with more powerful hardware, which significantly reduces overhead of data pushing.

Figure C.1 shows a CDF for latency from input event to the earliest time that the event is reflected in a streaming task's output (although inconsistent results can be returned earlier by querying the ingest actors directly). `AsyncCkpt` allows the snapshot to be viewed before it is checkpointed. `NoCkpt` has impractical recovery overhead, but we use it as a performance baseline. `AsyncCkpt` achieves similar latency as `NoCkpt`, meaning that checkpointing overhead remains stable as the graph grows larger; this is because streaming tasks pass through previous `Refs` that are already checkpointed, so Exoflow only checkpoints new data on each epoch. `SyncCkpt` is similar to Kineograph, checkpointing the snapshot before making it visible, and adds less than 1s latency. Finally, the error rate of the online results and the batch processing task duration both grow linearly over time, confirming the tradeoffs between batch vs. stream processing.

C.3.2 Microbenchmarks

Latency. With equivalent backends, Exoflow matches or reduces execution overheads of existing workflow systems while enabling more flexible inter-task communication. Figure C.2a (1 m5.8xlarge instance) shows the latency of workflow execution ("Trigger") and task execution with different size arguments. We use exactly-once systems (Airflow [5], AWS Standard Step Function [50]) and at-least-once systems (AWS Express Step Function [50], Ray [145]) as baselines. Airflow is an industrial custom-built workflow system while Step Functions are the AWS-native workflow offering for Lambdas.

First, with the Lambdas backend, Exoflow has similar trigger latency as AWS Standard Step Function. Airflow has generally high overhead due to coordinating execution through a database, which can easily lead to inefficient scans.

"1B" in Figure C.2a compares minimum task execution latency. Exoflow-Lambdas achieves comparable latency as AWS Step Functions, as the primary overheads for exactly-once and at-least-once execution come from durability and Lambdas invocation, respectively. Exoflow-Ray improves upon the latter as it uses Ray for execution.

Finally, we compare the ability to pass large data between tasks. AWS Step Functions limit data passing to 256KB, but plain Lambdas have a size limit of 6MB. Thus, Exoflow-Lambdas can actually support larger data sizes. This could be improved further with `Refs`, e.g., with Redis [174] for distributed memory. Airflow's XCom [1] can support slightly larger data but is fundamentally limited by its database-centric design. Meanwhile, Exoflow-Ray uses Ray `Refs` for efficient data passing. The gap between `AsyncCkpt` and `NoCkpt` latency is small but grows with

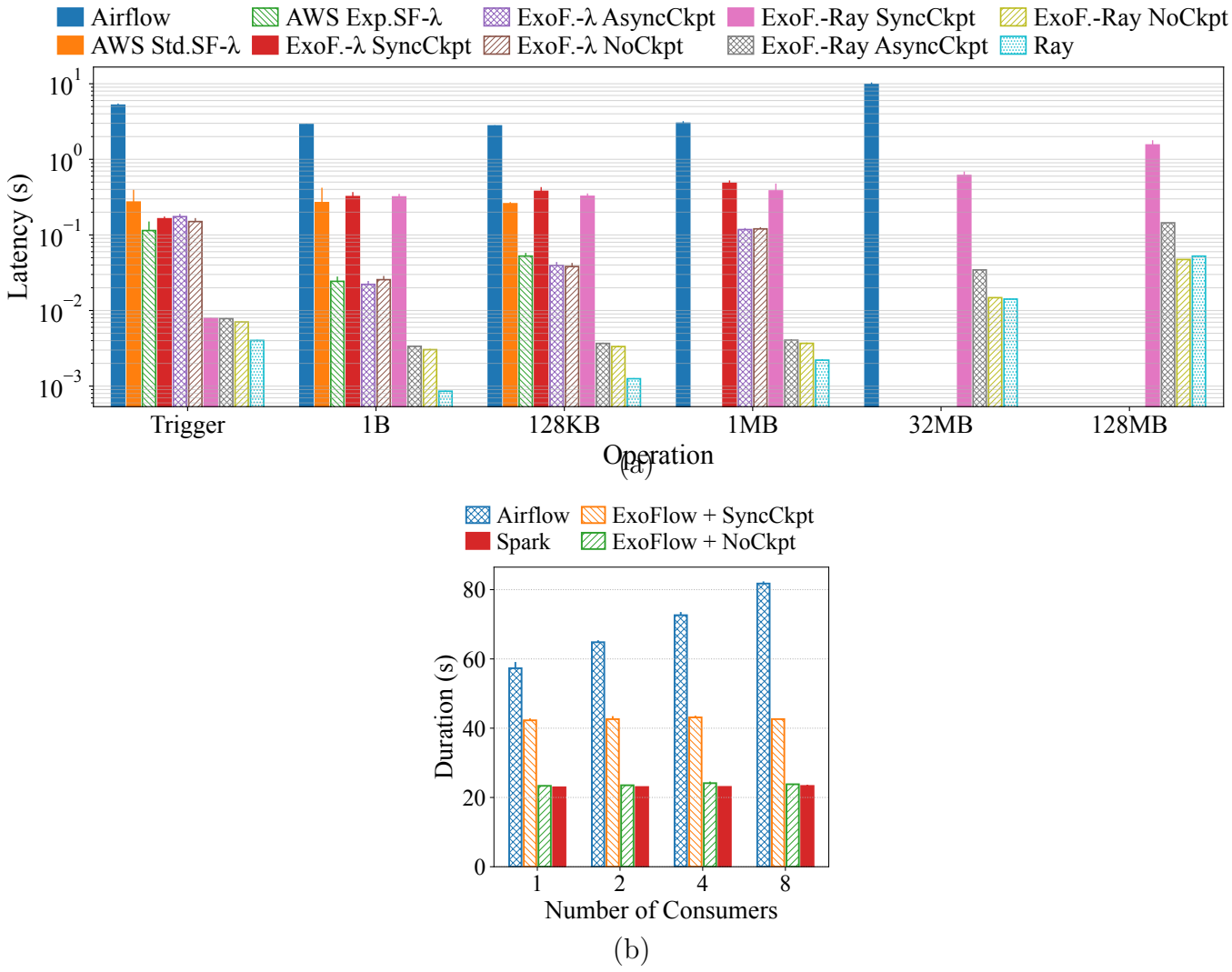
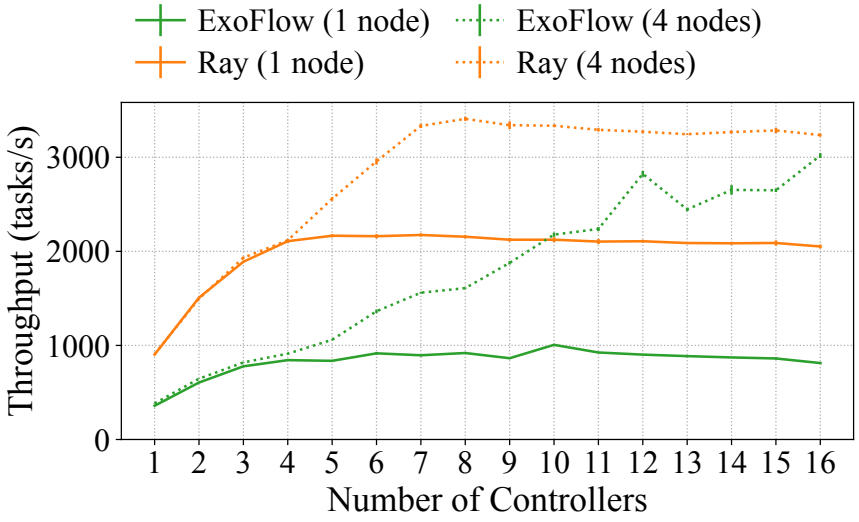
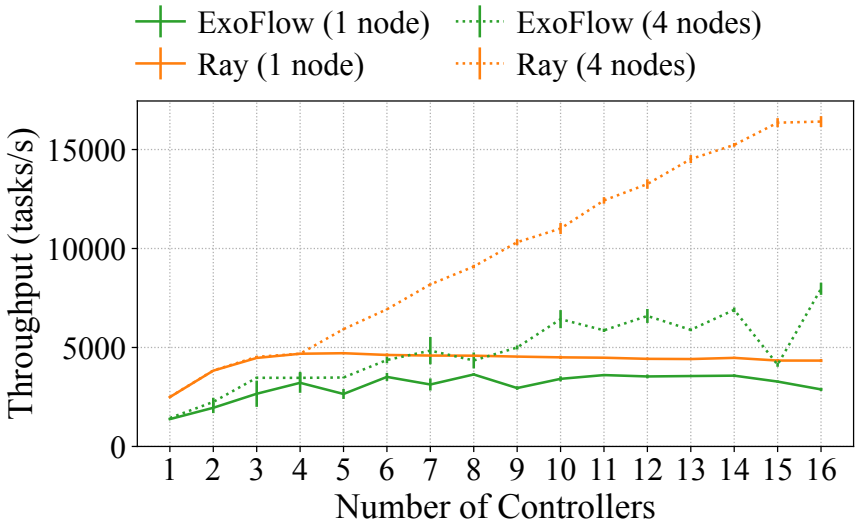


Figure C.2: Microbenchmarks. **(a)** Triggering and data passing latency of Exoflow and other workflow systems, using AWS Lambda (λ) and Ray as execution backends. Missing bars indicate limitations in inter-task communication. **(b)** End-to-end run time for the ETL workflow shown in Figures 6.2b and 6.4c, compared with Airflow and native Spark.



(a)



(b)

Figure C.3: Microbenchmarks, cont. Maximum task throughput (a: 1 task/DAG; b: 100 tasks/DAG) of 10k tasks, compared against Ray as an optimal baseline, on 1 node and 4 nodes.

data size; although the checkpoint is asynchronous, Exoflow synchronously copies the data to guard against concurrent writes.

In summary, Exoflow's low execution overheads make it a practical replacement for existing workflow systems, and it enables greater flexibility in task communication and recovery.

Data sharing for ETL. We evaluate Exoflow against Airflow for a Spark workflow similar to Figure 6.2b (1 m5.8xlarge instance, 4GB Spark memory). Figure C.2b measures total run time for a workflow that uses Spark to generate a 1GB random dataset, followed by multiple downstream tasks that consume the data with data sampling Spark jobs. Such a workflow requires orchestration across Spark jobs, which Spark does not provide, and is therefore often run on a workflow orchestrator such as Airflow.

Airflow run time grows proportionately with the number of consumers because they cannot share data in-memory. Meanwhile, Exoflow scales well even with synchronous checkpointing because consumers share data via Spark's native cache. Furthermore, Exoflow runs as fast as native Spark alone, while facilitating composition with other systems as well.

Throughput and Scalability. We measure maximum throughput with varying numbers of controllers, (AWS m5.2xlarge) nodes, and tasks per DAG. We use Ray as the optimal baseline, as Ray is also the execution backend.

Figure C.3a (1 task/DAG) shows that Exoflow and Ray both reach saturation after 4 controllers on one node. With 4 nodes, scalability continues, and the gap between Exoflow and Ray narrows at around 16 controllers. Figure C.3b (100 parallel tasks/DAG) shows that throughput overall improves via task batching. Again, with four nodes, both Exoflow and Ray scale linearly with the number of controllers. Exoflow achieved roughly 50% of Ray's throughput, due to additional overhead from workflow orchestration and ensuring exactly-once semantics.

Appendix D

Some random walks

Here are some nice walks in the Bay Area, walked by the author over the course of her thesis. These are inspired by [77] (although sadly several of these require a car). Stay hydrated!

Soda Hall to Fire Trails (4-8 miles) This walk goes right from campus, is very nice for sunset views, and has lots of options ranging from a couple hours to a full day. The Fire Trails have many entrypoints, but the one closest to Soda is about 1 mile away, on Centennial Drive. The Fire Trails are popular with grad students and fun to explore on your own, so I won't describe them too much here.

I will, however, give you a nice shortcut to a great view! The Fire Trail starting from the Centennial Drive trailhead winds gently uphill, until about 1 mile in, when the trail turns sharply to the left and goes up a short but steep hill. This leads to the Upper Fire Trails, and there are great views from here. However, if you aren't feeling the hill, you can also continue straight, and turn left onto the residential road Panoramic Way. Continue for another 0.5 miles, past houses that seem to be on stilts, to reach the intersection with Dwight Way. There is a wonderful sunset view of Berkeley and the Bay at this corner.

From here, you can turn back, head to Southside via Dwight Way, continue up Panoramic Way to intersect with the Upper Fire Trails, or call for a ride!

Tilden Park Tilden is a delight, and one of my regrets in grad school is not going more often and earlier on. There are again many trails here to explore, but here are two good options.

Tilden Park: Meadows Canyon to Curran to Wildcat Gorge Trail (3 miles) This short and moderate loop is a great introduction to Tilden Park, winding through chaparral-covered hills, down through a eucalyptus forest, then running along a shaded creek. The short distance combined with the nature and variety per mile

made this a great option for whenever I felt I had no time or energy for a walk (but in fact I nearly always did).

The directions here are straightforward with a map of Tilden so I will just give some tips. It is possible to start this trail from at least three different points, but I like to start from the Lone Oak Road parking area, where there never seems to be as many people as in other parts of Tilden. This way, you can also go clockwise to get the unshaded climb along Meadows Canyon out of the way first, although it does mean you will go downhill on the slightly steeper Curran Trail. I also feel that Wildcat Gorge is the nicest section, so this direction leaves the best for last.

There are some benches near the top of Meadows Canyon Trail that will give you the nicest views of this loop, especially at sunset. You can also add a short detour to see Lake Anza, and/or add the trail that goes around the lake. Keep an eye out for one of my favorite trees ever along the Wildcat Gorge section.

Tilden Park: Lone Oak to Wildcat Peak (3-4 miles) This steeper loop goes up to one of the nicest panoramic views in Tilden. As it is for most Tilden points of interest, there are multiple ways up. I like to start from the Lone Oak parking area and walk Loop Road to Laurel Canyon Trail to Laurel Canyon Road, to Wildcat Peak Trail. There is also a redwood grove with some picnic tables right before the last climb to the peak, if you're looking for a nice lunch spot. On the way down, take Wildcat Peak Trail and Sylvan Trail back to Loop Road, because everyone likes a loop more than an out-and-back. You can also take the slightly longer but much flatter approach from Inspiration Point along Nimitz Way.

Mt Tam State Park: Stinson Beach to Matt Davis to Steep Ravine to Dipsea Trail (7-9+ miles) This is my favorite hike in the Bay Area. It feels like it has practically every ecosystem in California that isn't the desert or mountains: waterfalls, forests, golden (i.e. brown) hills, redwoods, and the beach, all in one loop. Also, with a little bit more effort than the standard trail calls for, you get excellent views of the Bay Area in addition to the ocean.

Start at Stinson Beach, where there is plenty of free parking and public bathrooms. Walk towards and past the Stinson Fire Station, where you will see the Matt Davis trailhead on your right. From here, take the switchbacks up through your first ecosystem of the day, a mossy forest with waterfalls at your side. Eventually you will leave the forest cover and enter the prototypical California hills. This area is nicest in the winter and spring, when the hills are green.

You can continue on the Matt Davis Trail, or for more adventure and fewer crowds, take this alternative. Turn left onto the Coastal Trail. About a few hundred feet after the intersection, you will find a use trail on your right that climbs up the hill. The climb is very steep but short. Take the trail up and over to the ridgeline, where you will find a network of use trails. Explore! Eventually, find another use

trail that climbs back down on the other side of the hill and meet up with the Matt Davis Trail to continue.

From here, the route is standard. Take Steep Ravine Trail through a redwood forest to Dipsea Trail through beach shrubbery back to Stinson Beach. Alternatively, for an extra long walk, you can continue from Matt Davis all the way up to the peak of Mt Tam, although you will find that most people just drive up to avoid this 15mi round-trip hike. End with a foot soak in the frigid Pacific waters! That is the other reason for starting at Stinson Beach.