EXOFlow 综述:满足精确一次语义的通用分布式工作流系统

刘轩[1)]

[1)](华中科技大学计算机学院 武汉市 中国 430074)

摘 要 对于运行时和恢复性能之间的基本权衡，当前的分布式系统通常构建特定的应用程序的恢复策略以最小化开销。然而，将不同的应用程序组合成异构管道这一行为正变得越来越普遍。在同一系统中实现多个互操作恢复技术少见且困难。因此，用户必须在以下之间进行选择：1.在单个系统上构建，并面临运行性能与恢复开销的选择，2.将多个能够提供特定应用程序的权衡的系统拼接在一起，这非常具有挑战性。本文提出了 ExoFlow，一种通用工作流系统，即使在相同的应用程序中也可以灵活选择恢复与性能权衡。本文解决方案的关键是将执行与恢复，并提供 Exactly-Once 语义作为执行的单独层。一般而言，工作流任务可以返回捕获任意任务间通信的引用。为了实现工作流系统，并且最终用户可以控制应用程序的恢复，文献设计了任务注释以指定执行语义，例如任务是否是非确定性。ExoFlow 阐述了从 ETL 管道到有状态无服务工作流之类的工作流应用程序的恢复，同时在任务通信和恢复方面实现了进一步优化。。

关键词 工作流系统 精确一次语义 任务恢复 检查点

An Overview of Efficient Merge of LSM Trees

Dingxin Wang[1)]

[1)](Huazhong University of Science and Technology, Wuhan, China,430074)

Abstract Given the fundamental tradeoff between run-time and recovery performance, current distributed systems often build application-specific recovery strategies to minimize overheads. However, it is increasingly common for different applications to be composed into heterogeneous pipelines. Implementing multiple interoperable recovery techniques in the same system is rare and difficult. Thus, today's users must choose between: (1) building on a single system, and face a fixed choice of performance vs. recovery overheads, or (2) the challenging task of stitching together multiple systems that can offer application-specific tradeoffs. We present ExoFlow, a universal workflow system that enables a flexible choice of recovery vs. performance tradeoffs, even within the same application. The key insight behind our solution is to decouple execution from recovery and provide exactly-once semantics as a separate layer from execution. For generality, workflow tasks can return references that capture arbitrary inter-task communication. To enable the workflow system and therefore the end user to take control of recovery, we design task annotations that specify execution semantics such as nondeterminism. ExoFlow generalizes recovery for existing workflow applications ranging from ETL pipelines to stateful serverless workflows, while enabling further optimizations in task communication and recovery.

Key words Workflow System Exactly-once Task Recovery Checkpointing

1 引言

分布式应用程序的一个关键要求是容错，即使发生故障，从外部看来执行也不会失败。一般来说，恢复和运行时开销之间存在权衡。例如，日志记录（logging）通常会产生更高的执行开销，但通过允许系统只重新执行失败的任务来减少恢复时间。同时，检查点（checkpointing）减少了执行开销，但可能会产生更高的恢复开销，因为系统必须在失败后回滚（rollback）并需要额外的计算。

当前的分布式系统通常基于应用程序在恢复和性能之间选择不同的权衡方式。例如，Apache Spark 使用基于谱系的日志进行批处理，Apache Flink 使用检查点进行流处理。

然而，将不同的应用程序组合成异构流水线这一行为正变得越来越普遍。例如，机器学习流水线可能会使用批量摄取来构建训练数据集，然后将数据流式传输到批量分布式训练作业以减少延迟和内存开销。如果我们对整个流水线使用单一恢复策略，性能和恢复可能是次优的，因为不同的恢复策略适用于不同的应用程序。因此，为了优化端到端性能和恢复，我们需要组合不同的恢复策略。

在同一系统中实现多个互操作恢复技术非常具有挑战性。例如，Spark 引入了“连续处理”来减少流处理应用程序的性能开销，但该模式尚未在故障期间提供精确一次语义 。另一方面，Flink 增加了批处理模式，但这需要从流路径构建一个完全独立的恢复系统。

总体而言，这些问题导致具有不同恢复性能权衡要求的领域中的不同应用程序得到不均衡的支持。本文中提出了一个通用工作流系统，即使在相同的应用程序中，也可以灵活选择恢复与性能权衡。工作流是任务的有向无环图 (DAG)，其中每个任务封装了一个函数调用，任务之间的边表示数据依赖关系。工作流用于协调跨系统的执行，从而优先考虑通用性。DAG API 允许每个任务中的任意应用程序代码，从提交 Spark 作业到调用微服务。

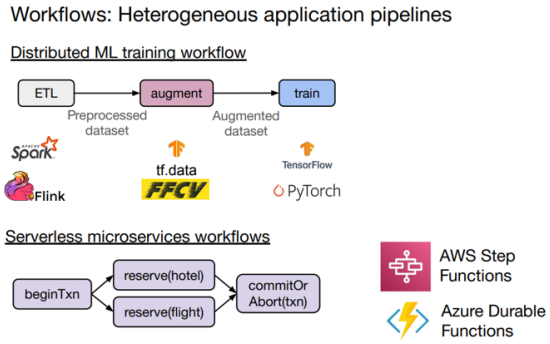

图 1.1 工作流系统的应用

2 问题背景与动机

2.1 问题背景

容错恢复是分布式系统中的重要问题，一般来说，系统设计者需要在性能与恢复开销中进行权衡，例如持续的 logging 可以实现快速的恢复但是会引入大量的性能开销，而定期的 checkpointing 则可以减少性能开销但是会导致修复时重新执行计算任务。

本工作旨在为分布式应用提供一个灵活的进行恢复和性能之间权衡的选择，而为了尽量提高通用性，本工作选择 DAG 模型来描述一个分布式计算任务。

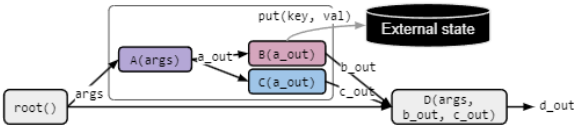

图 2.1 工作流 DAG 图模型

如图 2.1 是一个有向无环图（Workflow DAG）任务的例子，其中整个任务的输入是 args，共包含 A/B/C/D 四个子任务，子任务之间的边代表任务的依赖关系，而边上的 a_out/ b_out 等代表子任务的输出。

- 可见性（visibility）

有内部（internal）和外部 external）两种。内部输出（图中的 a_out/b_out/c_out）只有 DAG 内部的子任务可见，而外部输出（图中的 key, val）对任意组件（包含系统外部其他组件）可见。

- 决定性（determinism）

有决定性与非决定性两种。例如某个输出依赖于当前的时间或者某个随机生成的数值，那么这个输出就是非决定性的。

可见性与决定性对于容错恢复方法非常重要，

考虑一下两个场景：

假设当前已经执行完 A (生成 a_out1)和 C（生成 c_out1) 两个子任务，此时在执行 B 任务前，a_out1 由于出现了错误而丢失了。

假设 a_out1(或者子任务 A)是非决定性的：若重新执行子任务 A 来恢复 a_out，由于它是 非决定性的，得到的输出可能是与之前不同的 a_out2。此时为了使得 B/C 两个任务看到的是一致的 a_out2，必须回滚 c_out1，并重新使用 a_out2 来执行 B/C 两个任务。

假设 a_out1(或者子任务 A)是决定性的：重新执行 A 可以得到与之前一致的 a_out1，因此不需要回滚 c_out1，只需要重新执行 A，再执行 B 即可。

决定性（determinism）

假设 a_out1(或者子任务 A)是非决定性的: 同理，重新执行子任务 A 会得到与之前不同的 a_out2，因此需要回滚 B 任务，从而使得 B/C 两个任务看到的是一致的 a_out2。但是由于 B 产生了外部输出 (key, val)，必须想办法回滚这个外部输出，否则只能预先对 a_out1 做插入检查点。

假设 a_out1 (或者子任务 A)是决定性的：重新执行 A 可以得到与之前一致的 a_out1，再执行 C 即可。

2.2 Exactly once语义

流处理（streaming process），有时也被称为事件处理（event processing），可以被简洁地描述为对于一个无限的数据或事件序列的连续处理。一个流，或事件，处理应用可以或多或少地由一个有向图，通常是一个有向无环图（DAG），来表达。在这样一个图中，每条边表示一个数据或事件流，而每个顶点表示使用应用定义好的逻辑来处理来自相邻边的数据或事件的算子。其中有两种特殊的顶点，通常被称作 sources 与 sinks。Sources 消费外部数据/事件并将其注入到应用当中，而 sinks 通常收集由应用产生的结果。图 1 描述了一个流处理应用的例子。

一个执行流/事件处理应用的流处理引擎通常允许用户制定一个可靠性模式或者处理语义，来标示引擎会为应用图的实体之间的数据处理提供什么样的保证。由于你总是会遇到网络、机器这些会导致数据丢失的故障，因而这些保证是有意义的。有三种模型/标签，at-most-once、at-least-once 以及 exactly-once，通常被用来描述流处理引擎应该为应用提供的数据处理语义。

Exactly-once：

倘若发生各种故障，事件也会被确保只会被流应用中的所有算子“恰好”处理一次。拿来实现“exactly-once”的有两种受欢迎的典型机制：

1. 分布式快照/状态检查点（checkpointing）
2. At-least-once 的事件投递加上消息去重

用来实现“exactly-once”的分布式快照/状态检查点方法是受到了 Chandy-Lamport 分布式快照算法 1 的启发。在这种机制中，流处理应用中的每一个算子的所有状态都会周期性地 checkpointed。倘若系统发生了故障，每一个算子的所有状态都会回滚到最近的全局一致的检查点处。在回滚过程中，所有的处理都会暂停。Sources 也会根据最近的检查点重置到正确到 offset。整个流处理应用基本上倒回到最近的一致性状态，处理也可以从这个状态重新开始。

2.3 现有方法的不足

现有的工作流系统（例如 Apache Airflow）在不知道 DAG 输出的语义下，只能悲观的假设所有的子任务都是非决定性的并且会产生外部输出，在这种情况下，为了实现一致性语义，只能同步地对每个输出做检查点插入，无法为用户提供修复与性能的权衡选择。

2.4 挑战

挑战 1：如图 2.2，现有的工作流系统必须同步地将所有内部输出进行检查点的插入。

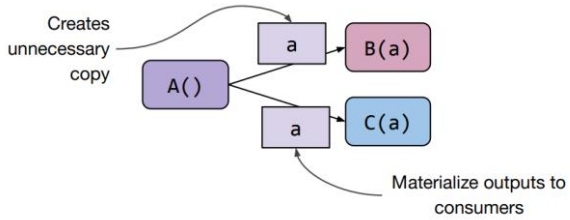

图 2.2 挑战 1

解决方法：通过传递引用来使得后端执行引擎来决定如何传递实际的值。

挑战 2：对于一 serverless 的工作流系统，如何对外部输出进行纠正。

由于任务是非决定性的，导致其必须在下游任务开始前设置输出的检查点。

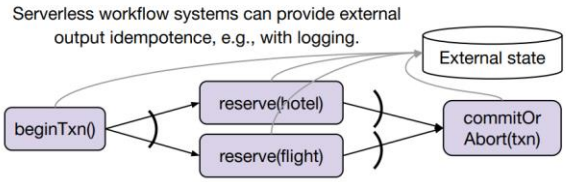

图 2.3 挑战 2

解决方法：通过任务注释来获取语义。

3 ExoFlow 设计与实现

针对这些问题，本工作设计了 ExoFlow，一个能够满足一致性语义的通用 DAG 工作流系统。

ExoFlow 的核心设计是将任务的执行与 修复进行解耦，并允许用户对任务的决定性以及输出可见性进行注释，从而利用应用的语义，在满足一致性语义的同时，实现灵活的修复与性能权衡选择。

3.1 API

图 3.1 展示了 ExoFlow 提供的 API，主要包含了。

DAG interface：用于创建和执行 DAG。

Task annotation：用于标记任务的语义。

Internal output (Ref)：用于在 task 之间进行传递的引用。Ref 是每个执行后端实现的接口，例如对于 AWS Lambda 来说，Ref 可能是一个保存在 key-value store 中一组 K-V Pair 的 Key。

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.

图 3.1 API

3.2 ExoFlow实现

为了实现一致性语义，ExoFlow 在用户提交 DAG 以及任务的语义注释后，会检查这个 DAG 是否满足两个不变式：

不变式 1(External output commit)：对于 deterministic=False（即非决定性）的每个任务 vi，假设 G 是包含 vi 和所有下游任务的最小子图。那么，对于 G 中每个 can_rollback=False（即可回滚）的任务 vj，必须存在一个顶点切分，将 vi 与 vj 分割开来，使得切分中的所有任务都具有 checkpoint=True（具有检查点）。

不变式 2(Rollback durability)：对于从具有 deterministic=False 的任务 vi 开始到具有 rollback function Rj 的任务 vj 结束的每条路径，沿路径必须至少存在一个具有 checkpoint=True 的顶点。

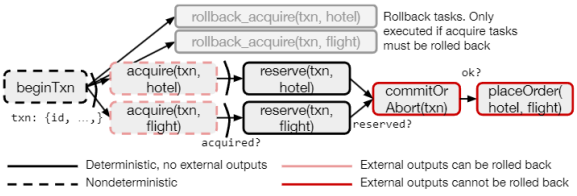

图 3.2 样例模型

利用图 3.2 的例子（一个简化过的 transaction processing 的 workflow）解释这两个不变式。

首先，假设 commitOrAbort 执行出现了失败，由于 commitOrAbort 设置了 can_rollback=False（因为 transaction commit 后无法 rollback），此时只能想办法获得 commitOrAbort 原先的输出重新执行（commitOrAbort 满足幂等性）。而由于 acquire 是非决定性的，因此 acquire 或者 reserve 的输出必须有 checkpoint=True（满足不变式 1）。

其次，假设 acquire 出现了失败且需要执行 回滚操作，回滚操作需要使用 acquire 任务的输出(例如 txn id)来执行，但是由于 beginTxn 是非决定性的，因此为了获取这个输出，只能对 beginTxn 设置 checkpoint=True（满足不变式 2）。

4 ExoFlow 性能测试

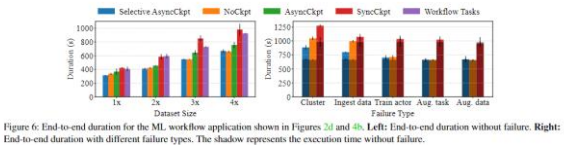

Figure 6: End-to-end duration for the ML workflow application shown in Figures 2d and 4b. **Left:** End-to-end duration without failure. **Right:** End-to-end duration with different failure types. The shadow represents the execution time without failure.

图 4.1 性能测试

ExoFlow 首先使用了一个 ML workflow 来测试系统性能，测试的容错恢复方式包含：

Selective AsyncCkpt: 只对部分内部输出进行异步检查点

NoCkpt: 只对最终 ML 模型进行检查点，达到理论的最优性能

AsyncCkpt: 对内部输出进行异步的 检查点

SyncCkpt: 对内部输出进行同步的检查点，代表了现有系统的方法

Workflow Tasks：不使用 Ref 进行内部输出的传递，会产生额外的数据持久化开销

通过上图的测试结果可以得到结论：

ExoFlow 提供的 Ref 接口可以避免内部输出的持久化，提高数据传递性能

ExoFlow 可以有效的应对多种失败

ExoFlow 可以提供灵活的修复与性能的权衡

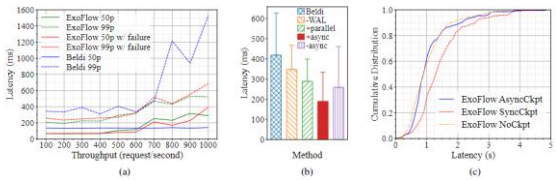

图 4.2 方法比较

本工作还对比了 ExoFlow 与 Beldi[OSDI'20]处理 stateful serverless workflow 的性能。

上图 4.2(a) 展示了 ExoFlow 与 Beldi 在处理执行一个 serverless transaction 时的延迟。在低吞吐(<700)的情况下，ExoFlow 相比 Beldi 有绝对的优势，主要原因是 ExoFlow 可以避免对一些 决定性的任务进行日志点插入。

图 4.2（b）进一步展示了利用应用的决定性等语义信息可以优化性能，其中的测试是对 Beldi 系统进行了修改，使其能够进行异步、并行检查点等，并且得到相当的性能收益。

图 4.2（c）展示 ExoFlow 在一个 graph processing 任务上使用不同容错恢复测量的性能，进步证明了 ExoFlow 的通用性以及灵活性。

5 总结

本篇综述关注于任务流系统的修复与性能权衡问题，介绍了一系列当前技术背景与问题，如一致性语义，检查点和 logging 技术等。同时关注技术的通用性，使其能在后端使用任意的执行引擎

现有的分布式系统（如 Airflow，Kubeflow，AWS Step Function 等）大多为特定应用领域提供专门的、高效和透明的恢复。ExoFlow 有一个正交互补的目标。为了将异构的应用程序统一一个通用的方案，我们必须提供通用和可互操作的恢复方法。其中最大的挑战是在不牺牲灵活性的情况下获得足够的应用程序语义。

本文提出了一种在可用性（最小注释、编译安全检查）和功能（灵活参考、自动恢复）之间取得平衡的方法。在此过程中提供与通用 API 匹配的通用恢复的工作流 DAG。

首先，我们介绍了有关的工作流系统在现实的应用，分析各个技术的不足之处。之后对各个容错恢复技术进行讨论，日志容错恢复方面较好，但是不断地进行 logging 会导致性能方面的损失。检查点机制减轻了性能方面的负担，但也导致出现错误时需要回滚且重新计算，因此在与此相关的研究领域中，都不可避免地需要对这两项操作进行权衡选择。

接下来描述了在工作流系统中，通过进行任务注释的方式，来进行性能与修复权衡的机制。如，注释任务的可见性和决定性，将任务的执行与恢复解耦，简述了 Exactly-once 语义的具体含义。提出设计 ExoFlow 的目标，即通用性，可插拔性和执行/恢复解耦。

之后介绍了 ExoFlow 的具体设计，建立任务流的 DAG 模型图，在此之上定义两个不变式，对任务的执行进行注释，可以判断在何时应合适地插入检查点。

最后，用一个机器学习的工作流系统测试性能，结果显示 ExoFlow 可以很好地满足提出的目标。其次与其他工作流系统在 stateful severless 任务上的执行效果上对比，结果证明 ExoFlow 有相当的性能收益，以及有很好的通用性和灵活性。

综上，ExoFlow 是一项通过任务注释与数据输出引用来实现的通用任务流系统，其将任务的修复与执行解耦，使得其能对性能与修复灵活地权衡。

参 考 文 献

[1] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, Vincent Liu. Fault-tolerant and transactional stateful serverless workflows.//USENIX Symposium on Operating Systems Design and Implementations, 2020

[2] Robert Ryan McCune, Tim Weninger, Gregory Madey. Thinking Like a Vertex: a Survey of Vertex-Centric Frameworks for

Distributed Graph Processing. ACM Computing Surveys Volume 48Issue 2Article No.: 25pp 1–39

[3] Philipp Moritz, Robert Nishihara, Stephanie Wang. Ray: A Distributed Framework for Emerging AI Applications//Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, 2018

[4] Tianyu Li, Badrish Chandramouli, Sebastian Burckhardt, Samuel Madden. DARQ Matter Binds Everything: Performant and Composable Cloud Programming via Resilient Steps// Proceedings of the ACM on Management of Data. Volume 1 Issue 2 Article No.: 117 pp 1–27

[5] Derek Murray, Frank Mcsherry, Rebecca Isaacs, Michael Isard, Paul Barham, Martín Abadi. Naiad: A Timely Dataflow System. Proceedings of the ACM Symposium on Operating Systems Principles. 2013: Pages 439–455

附录 1.

问题 1：这项工作是如何判断任务的如决定性和非决定性的任务注释的，是用户自己判定还是 ExoFlow 自行来判断。

回答：我认为这一个性质是由 ExoFlow 自己判断的，首先工作在构建 DAG 图的过程中，只需检查任务是否与其他任务进行通信，若有且是与外部任务进行通信，则这个任务的输出就是外部性的。在决定性方面，只需对这个任务进行简单的检测即可判断这个任务的输出是否依赖于一个不确定值，如在前文提到的，acquire 任务的实现需要依赖时间等因素进行处理，因此很容易判断其是非决定性。

问题 2：什么是一致性语义

通俗地说，数据分析过程中需要满足精确一次处理的条件，即一个任务的总体输出总是会和未发生失败的情况下输出的效果一致，就是 Exactly-Once，这对于很多分布式多系统来说其实是个很大的考验。因为分布式系统天生具有跨网络、多节点、高并发、高可用等特性，难免会出现节点异常、线程死亡、网络传输失败、并发阻塞等非可控情况，从而导致数据丢失、重复发送、多次处理等异常接踵而至。如何保持系统高效运行且数据仅被精确处理一次是很大的挑战。分布式系统 Exactly-Once 的一致性保障，不是依靠某个环节的强一致性，而是要求系统的全流程均保持 Exactly-Once 一致性。需要：

1：支持可靠的数据源接入(例如 Kafka), 源数据可重读，此过程仅可实现 At least once（至少一次），也就是说数据可能会被重复读取

2：在消费端采用分布式、容错、不可变的数据集。其本身是只读的，不存储真实的数据，当结构更新或者丢失时可对其进行重建，不会发生变化。且通过 checkpoint 记录了数据的所有依赖过程，通过“血脉追溯”可重构计算过程且保证多次计算结果相同。

3: 输出端保持 Exactly-Once 一致性，其输出源需要满足一定条件：支持幂等写入、事务写入机制