Proposal: Introduce process framework to Amoro

Author: nathan ma

Background

Amoro 是面向多 format 的湖仓管理系统，需要解决两个核心问题：

1. 多 format 的抽象适配，并且在 format 之上建立湖仓管理的统一模型
2. 基于第一个点，有个高度可扩展的调度框架，通过数据驱动的方式自发管理不同 format 文件的合并，清理，格式转换以及未来有可能的任务类型

对于第一个点，在 2024 年已经通过模块拆分和重构，将不同 format 的适配抽到了 sub module 级别，并且抽象了 admin api 以扩展管理服务对不同 format 元数据的适配（将 format 各自不同的 POJO 转化成 Amoro 管理服务上看到的页面内容），目前 hudi 和 paimon 都走了这套接口，但统一模型这块还没有落地，目前 AMS 内部的 TableRuntime 与 Iceberg 以及 Iceberg 的优化流程深度绑定，以至多 format 的托管陷入瓶颈。这篇 proposal 将引入统一 TableRuntime 概念，并提出 Process 框架来扩展不同的调度需求，这套 process api 的应用除了系统已经支持的任务调度之外，也会用于将 AMS 内定时触发的孤儿文件清理，expire 等任务提交到外部，解决 AMS 在这块的瓶颈和可观测性问题。进一步说，process 框架未来可以定制各种任务的串联，形成天然数据驱动，状态可恢复的引擎能力。

Motivation

1. 通过统一 TableRuntime 模型，让不同 format 的任务调度成为可能
2. 基于统一的 TableRuntime，构建 process 框架，将 AMS 定时调度的内部任务扩展为向外部系统提交的任务
3. 基于 process 框架，可以将 zorder 等不同的引擎任务提交到外部
4. 基于 process 框架，可以将当前 Iceberg 优化任务的不同阶段扩展到 optimizer 中

Goals

本方案提出通过引入：

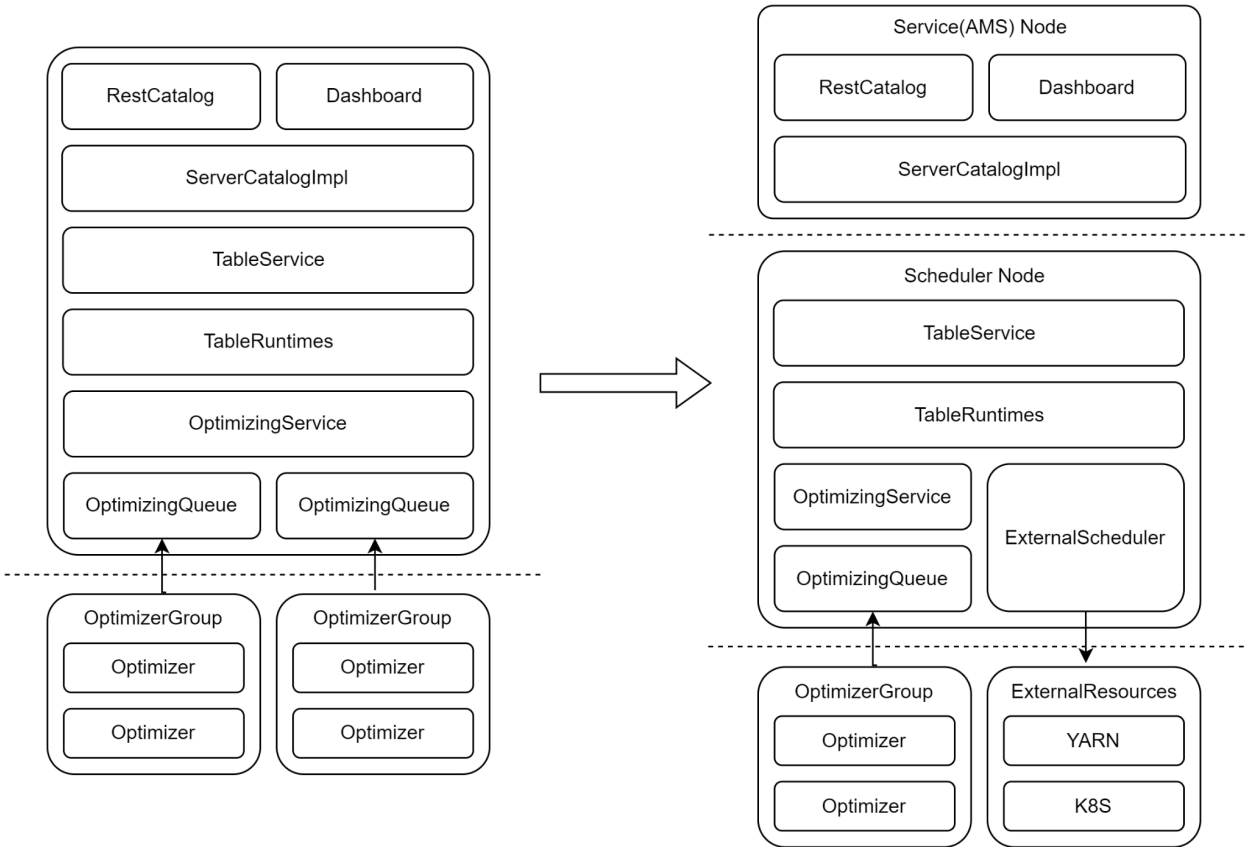
1. 统一 TableRuntime 模型：解耦 Iceberg 优化任务与相关状态机
2. 引入 Process Framework：构建可扩展的任务调度体系

主要解决以下问题：

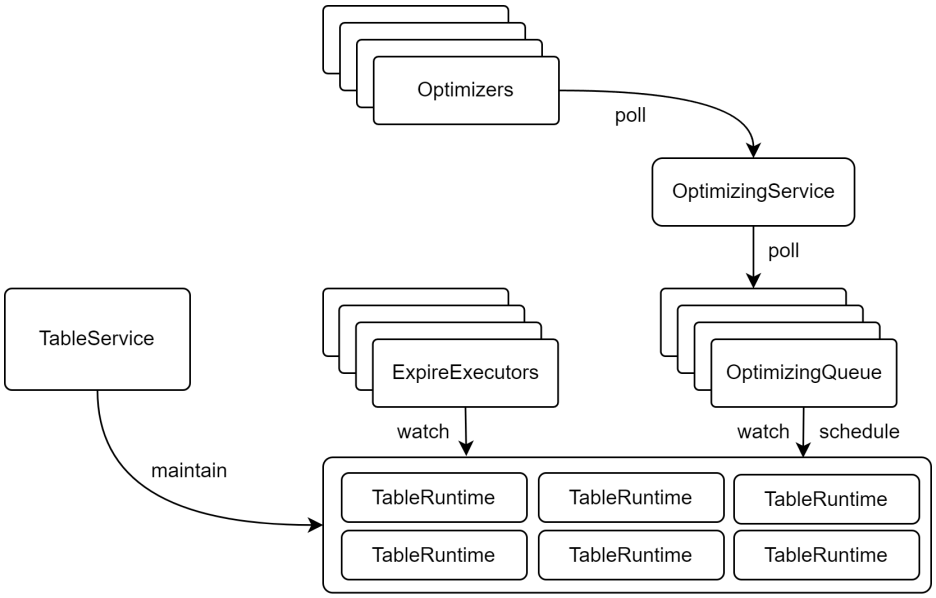
- 实现跨格式的任务调度能力
- 将 AMS 内部任务(如文件清理) 外部化
- 为未来功能 (zorder等) 提供扩展基础

Architecture

当前 Amoro 是单节点的主从架构，内部服务可以划分为两大类：元数据服务（如 RestCatalog，dashboard）和调度服务（对应当前 OptimizingService），前者是无状态服务，只涉及元数据（包含表元数据和任务元数据）的访问，可以从 AMS 中抽离以实现无状态扩展。
对调度服务而言，因为维护了表的调度元数据，天然持有调度状态，长期来看可以通过分区，或者和 OptimizerGroup 绑定等策略，实现分布式调度，调度器之间可以接管和迁移，单个调度节点也可以有冷备。
在生产中，元数据服务和调度服务的分离并不意味着两者必须分开部署，在架构上应当支持在一个进程中拉起两个服务的灵活性：



Process framework 以及统一 TableRuntime 的引入，是解决调度服务在多 format 扩展，多调度框架适配以及多语言的方案，所以我们这里只关注调度服务本身的架构演进。当前调度服务涉及的模块如下图所示：

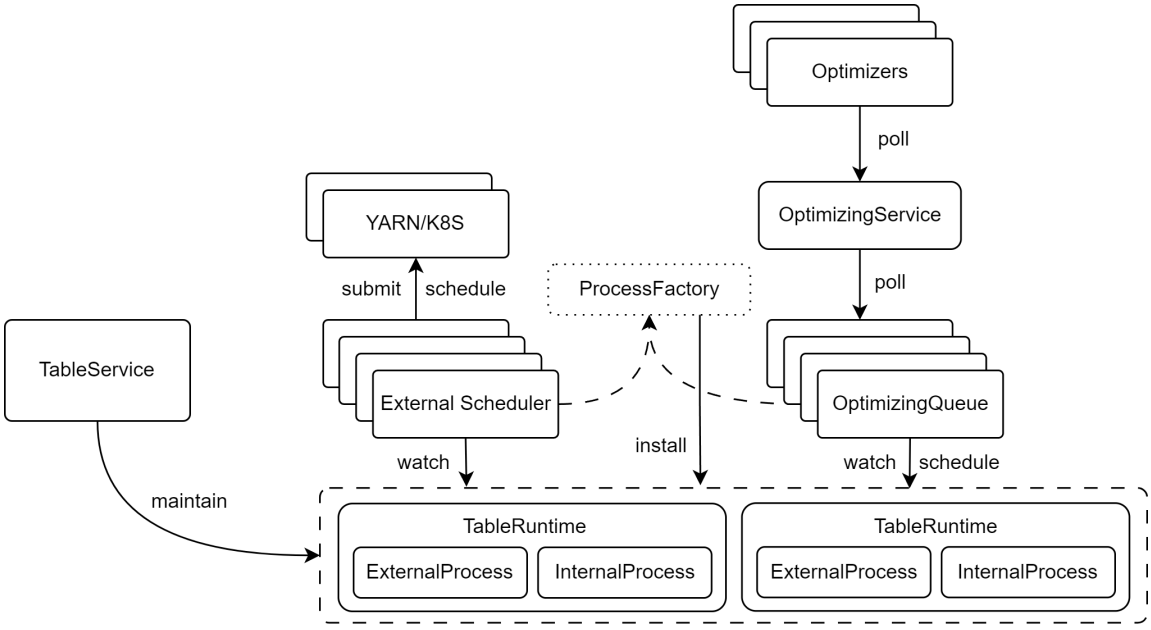


TableService 探查和维护 TableRuntime 列表，并触发 TableRuntime 变更时的回调。Executors 和 OptimizingQueue 则监听 TableRuntime 的变化以执行相应的本地调度和分布式调度。当前的 TableRuntime 包含了 TableConfiguration 和任务执行状态，并且维护了任务状态机的转化流程。和 Iceberg 任务状态的绑定，导致了和 Iceberg 默认优化任务的深度耦合，难以泛化和扩展。

新的调度服务整体框架基本保持不变，需要做的是：

1. TableRuntime 的泛化，与 OptimizingState 的解耦
2. Executors 通过适配 process framework 向调度器转化
3. OptimizingQueue 适配 process framework，这块工作会相对独立，会前置依赖将 OptimizingState 从 TableRuntime 抽离

引入 process framework 之后调度服务的整体框架：



Key interfaces

新的 TableRuntime 首先解决和默认优化任务的耦合:

```
public interface TableRuntime {


 List<? extends TableProcessState> getProcessStates();


 ServerTableIdentifier getTableIdentifier();


 TableConfiguration getTableConfiguration();
}
```

新的 TableRuntime 主要提供三块:

1. 表标志，包含三元组和 format 类型
2. 表配置，TableConfiguration
3. 当前正在进行的任务状态列表，这里泛化成一个 State，原先的 TableRuntime 中和默认优化任务有关的部分会转化成 State 的一种实现，State 同时也是对状态持久化的描述，未来新的 TableProcess 可以定义不同的 State 并通过调度服务实现可持久化的保存和恢复

围绕泛化后的 TableRuntime，不同的调度器依然需要通过 TableService 监听 TableRuntime 的添加，删除以及配置更新，以触发调度器的变更，但是不再会监听优化任务状态的变化，这些优化任务的状态可以各自维护在调度器内部，也可以通过 TableRuntime 获取 State。
在 TableRuntime 内部，维护了不同 Process 的工厂类，以可插拔的方式让不同 format 的表，以及可能经过特殊配置的表，使用不同的调度器，以及相应的 Process 生成规则：

```
public interface ManagedTableRuntime extends TableRuntime {

  AmoroProcess<? extends TableProcessState> trigger(Action action);

  void install(Action action, ProcessFactory<? extends TableProcessState>
processFactory);

  ProcessFactory<? extends TableProcessState> getProcessFactory(Action
action);
}
```

ProcessFactory 定义了怎么基于 TableRuntime 生成一个 Process，以及相应的恢复方式，在工厂方法中，实现者可以通过 TableRuntime.getProcessStates 获取当前正在进行的 process，以实现 process 互斥或其他验证逻辑，TableRuntime 会保障 Process 的创建和恢复具备原子性。

```
public interface ProcessFactory<T extends ProcessState> {

  /**
   * Create a process for the action.
   *
   * @param tableRuntime table runtime
   * @param action action type
   * @return target process which has not been submitted yet.
   */
  AmoroProcess<T> create(TableRuntime tableRuntime, Action action);

  /**
   * Recover a process for the action from a state.
   *
   * @param tableRuntime table runtime
   * @param state state of the process
   * @return target process which has not been submitted yet.
   */
  AmoroProcess<T> recover(TableRuntime tableRuntime, T state);
}
```

最后，我们让老的 AMS 内部调度的 Executors 和生成默认优化任务的 OptimizingQueue 分别实现各自的 ProcessFactory，即可基于统一的 TableRuntime 模型，实现多 format 以及多调度框架的不同适配，Executors 则向 ExternalScheduler 演进，从而将内部任务提交到外部，提升 AMS 稳定性。
调度器通常实现了 ProcessFactory，或者维护了一种或多种 ProcessFactory 实现。Process 的接口定义参考代码库即可

Concepts related

Process 相关概念：

- **Action**: Process 类型，例如 expire, clean, optimizing, 每种 format 可以定义自己的 Action
- **Process**: 一个执行过程称之为 Process, 是对 Amoro 托管任务的一种泛化和抽象，它可以是提交到外部的一个任务，也可以拆分成很多个 task 由默认优化器执行
- **ProcessFactory**: Process 工厂类，可通过 SPI 的方式加载，用于定制 Process 流程
- **ProcessState**: Process 的状态抽象，会持久化到系统库并在重启时恢复

ResourceContaine 扩展：

- **InternalResourceContainer**：对标前 ResourceContainer，定义申请资源和销毁资源
- **ExternalResourceContainer**：提交 process 到外部资源组，kill 资源以及查询资源状态

ResourcedGroup 扩展：

- OptimizerGroup: 对标当前的 OptimizerGroup
    - **ManagedOptimizerGroup**: 支持 request 和 release resource 的 OptimizerGroup
    - **UnmanagedOptimizerGroup**: 对标之前的 **ExternalOptimizerGroup**
- **ExternalResourceGroup**: 对标队列的概念，Container 是 ExternalResourceContainer

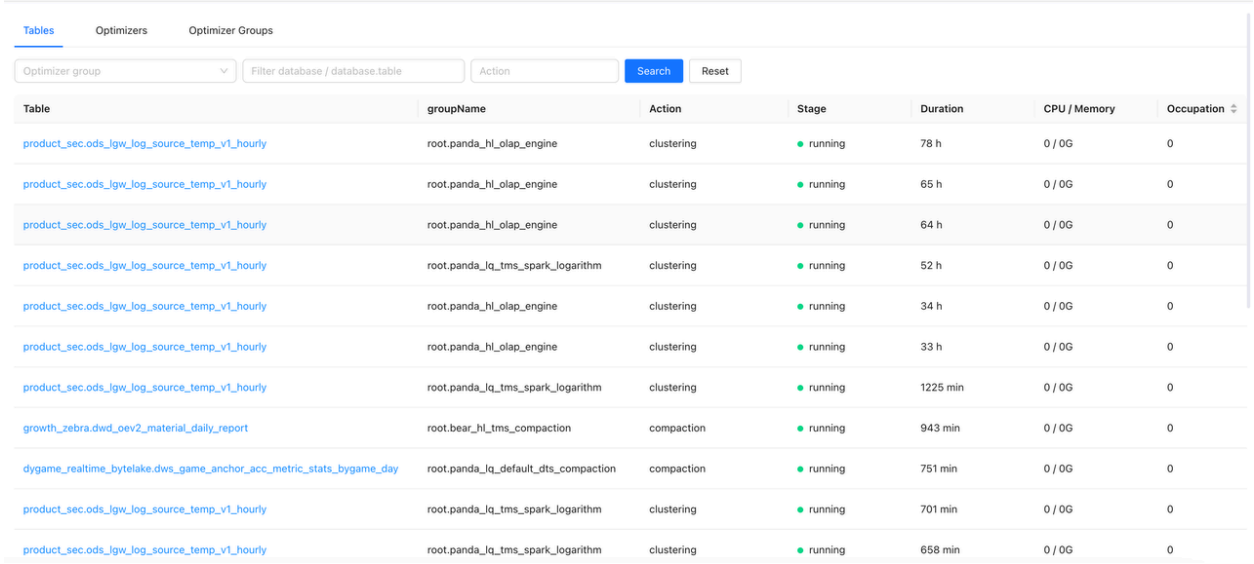
ExternalResourceContainer 接口如下所示：

```
public interface ExternalResourceContainer extends ResourceContainer {

  Resource submit(AmoroProcess<? extends TableProcessState> process);

  void release(String resourceId);
}
```

New UI



Roadmap

- [ ] 引入 process framework 实现 ExternalScheduler 代替内部调度（本次 PR）
- [ ] 基于 process framework 实现自定义 ExternalScheduler，扩展 ORDER 等功能
- [ ] 通过 process framework 重构默认优化任务，默认优化的不同阶段定义为不同 process，并实现不同 process 的串联调度