OSPP项目申请书

---

1. 项目介绍

1.1 项目概述

- 项目名称：时序存储单机 WAL 实现
- 项目主导师：chunshao90
- 申请人：苏逸钒
- 日期：2024/5/29
- 邮箱：8208220105@csu.edu.cn

1.2 需求概述

1.2.1 基础需求

HoraeDB 单机存储引擎采用基于 LSM 的架构，在 LSM 中，WAL 是保证数据可靠性的重要组件，一个高效的 WAL 实现是实现高吞吐的基础。 WAL 的主要接口有两个

- Append，顺序追加写数据
- Scan，顺序读取成批数据

在目前的实现中，采用的是 KV 存储 RocksDB 作为 WAL 的实现，RocksDB 本身是个复杂的组件，基于它来实现 WAL 会有以下问题：

- 编译困难，在 HoraeDB 开发者邮件列表，编译 RocksDB 失败的邮件经常出现
- RocksDB 自身作为一个完整的 LSM 引擎，本身会有 Compaction、Memtable 等组件，这些组件会带来额外性能损耗，而且调优比较复杂

基于此，我们想**设计一个基于本地磁盘的 WAL 实现**，去掉 RocksDB 这个依赖。

1.2.2 进阶需求

目前WAL实现的抽象设计得较为杂乱，使得为WAL扩展新的底层存储很困难（详情可见此issue）。因此我们想**重构WAL的设计框架**，从而增强代码的可扩展性与可复用性。

更进一步的，我们希望将该项目做成**Rust Wal的最佳实现**。

1.2.3 项目产出

- 完成WAL存储格式设计文档
- 实现WAL读写接口
- 替换基于RocksDB的WAL实现

1.2.4 预期收益

该项目将带来以下收益：

- **增强项目易用性**：解决由于依赖RocksDB所带来的编译困难的问题，同时使开发者能够更有效地参与对WAL组件的调优
- **提高WAL组件的性能与安全性**：借助Rust的语言特性及优秀的异步性能，为HoraeDB带来高吞吐的WAL组件
- **增强代码的可扩展性**：设计优雅的抽象代替现有的WAL代码框架，为后续扩展不同的底层存储、编码方式等带来便利

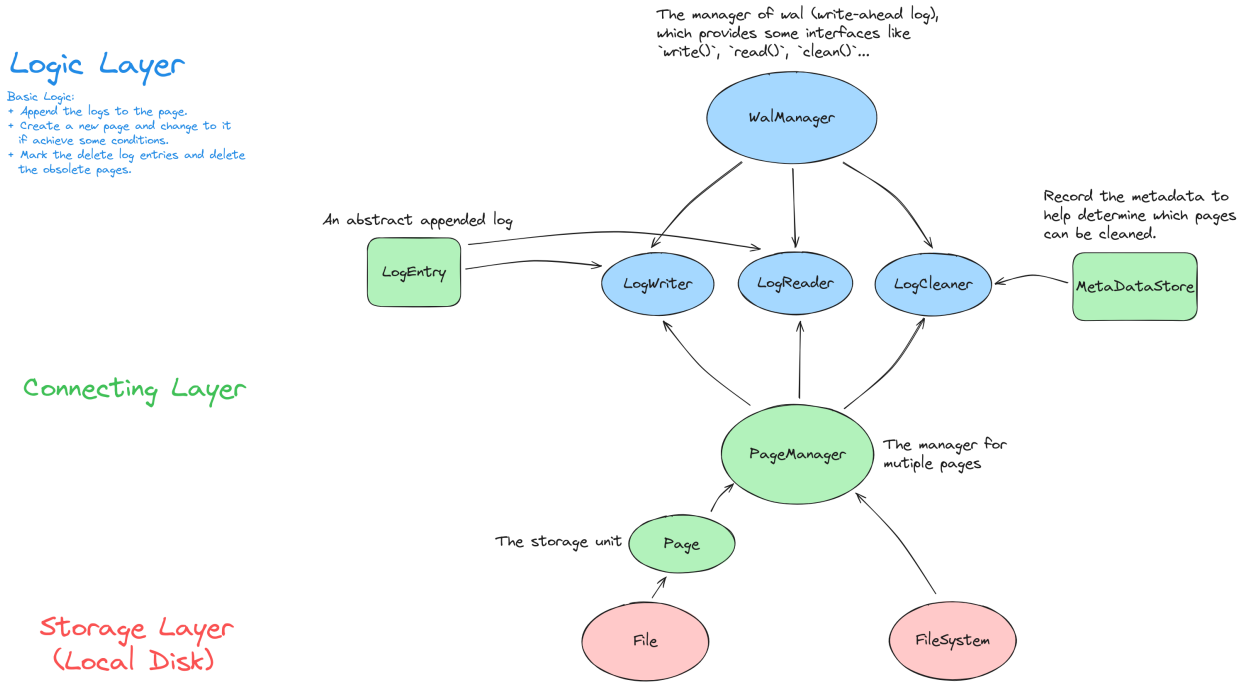
---

2. 技术方案

> *此处所介绍的技术方案为雏形，持续优化中。*

2.1 框架设计

2.1.1 框架图示



2.1.2 设计思路

将Wal module整体框架分为逻辑层、衔接层和存储层。逻辑层为HoraeDB中的其它模块提供简洁的接口；存储层对应不同的underlying storage；衔接层向下对接不同的存储底座，向上为逻辑层的实现提供抽象的接口，或者说衔接层抽象了底层存储、编码方式等的不同实现，为逻辑层提供了一致的接口。

a. **逻辑层设计**：根据Wal的基本逻辑，逻辑层为其它模块提供了三个最基础的抽象接口

- `write()`: Append log到当前页中，并在一定条件下切页
- `read()`: 读取存储在页中的log，主要用于Recovery
- `clean()`: 标识可被移除的LogEntry，后台线程会检查并移除过时的页

这三个接口的功能相对独立，所以将其划分到`LogWriter`，`LogReader`和`LogCleaner`中，使抽象更加简洁清晰

b. **存储层设计**：该框架下存储层可以是各种存储基座。

c. **衔接层设计**：

衔接层为存储基座暴露`Page`和`PageManager`的接口

- `Page`: 对存储基座中的存储单元的抽象
- `PageManager`: 对存储基座中的Manager的抽象，用于管理所有存储单元

存储基座通过这两个接口与逻辑层进行“间接通信”，使得当需要新增存储基座时，只需让其与`Page`和`PageManager`对接，降低了代码实现的复杂度，提高了代码的可扩展性。

同时衔接层为逻辑层实现提供`LogEntry`和`MetaDataStore`的接口

- `LogEntry`: 对追加到`Page`中的log的抽象，包括data和metadata，是`WalManager`读写的对象
- `MetaDataStore`: 对为`Page`记录的metadata的抽象，帮助后台线程决策哪些`Page`可以被移除

这些接口在简化逻辑层实现的同时，也增强了代码的灵活性，使新增Log编码方式、metadata格式等开发工作更加便捷。

2.1.3 代码结构

```
.
└── wal
    ├── cleaner.rs // `LogCleaner`
    ├── log.rs // `LogEntry`
    ├── mod.rs
    ├── manager.rs // `WalManager`
    ├── meta_data.rs // `MetaDataStore`
    ├── reader.rs // `LogReader`
    ├── storage
    │   ├── impls
    │   │   ├── disk.rs // Implementation of `Page`, `PageManager` for local disk
    │   │   └── mod.rs
    │   ├── mod.rs
    │   └── page_manager.rs // `Page` & `PageManager`
    └── writer.rs // `LogWriter`
```

2.2 模块实现

> *此处代码示例较为简略，仅用于传达设计意图。*

2.2.1 实现细节（本地磁盘）

如上所述，该框架的衔接层向下可对接多种不同的具体实现。经过与导师沟通 and 对RocksDB和LevelDB的调研，我为**衔接层**和**基于本地磁盘的Wal实现细节（如何对接衔接层）**做了初步设计。

a. **`Page` & `PageManager`**

- 接口设计

```rust
// page_manager.rs

/// The manager for multiple pages.
pub trait PageManager {
    type Page;
    type Path;

    /// Create a new page.
    fn create(&mut self) -> Self::Page;

    /// Delete the target page.
    fn delete(&mut self, path: Self::Path);
}

/// The storage unit.
pub trait Page {
    /// Read the page data to the `buf`.
    fn read(&self, buf: &mut [u8]);

    /// Write the data to the page.
    fn write(&mut self, data: &[u8]);
```

```
}
```

- `File`对接`Page`，`FileSystem`对接`PageManager`

b. **`LogEntry`**

- 接口设计

```rust
// log.rs

/// An abstract appended log.
pub trait LogEntry {
    type MetaData;

    /// Get the metadata part of the log entry.
    fn metadata(&self) -> Self::MetaData;

    /// Encode the log entry (include the data and metadata) to bytes.
    fn to_bytes(&self) -> Vec<u8>;

    /// Decode from bytes.
    fn from_bytes(raw: Vec<u8>) -> Self;
}
```

- 在基于本地磁盘的Wal实现中，`type MetaData = SequenceNumber;`
- 在对`LogEntry`编码格式的具体实现中，我计划采用较为常见的**Legacy Record Format**：

```rust
/// **Legacy Record Format**
/// ```text
/// +---------+-----------+-----------+--- ... ---+
/// |CRC (4B) | Size (2B) | Type (1B) | Payload   |
/// +---------+-----------+-----------+--- ... ---+
/// ```
/// CRC = 32bit hash computed over the payload using CRC
/// Size = Length of the payload data
/// Type = Type of record
///        (ZeroType, FullType, FirstType, LastType, MiddleType)
///        The type is used to group a bunch of records together to represent
///        blocks that are larger than kBlockSize
/// Payload = Byte stream as long as specified by the payload size
```

- Reference: [RocksDB Wal File Format Doc]

c. **`MetaDataStore`**

- 接口设计

```rust
// meta_data.rs

/// Record the metadata to help determine which pages can be cleaned.
pub trait MetaDataStore {
```

```
    type MetaData;

    /// Update metadata of the page.
    fn update_page_meta(&self, page_id: PageId, update: Self::MetaData);

    /// Flush and persist page meta.
    /// Generally, it will be called before page switching.
    /// If page metadata failed to flush, page switching should fail and retry
later.
    fn flush_page_meta(&self, page_id: PageId);

    /// Update the deleted point, used to determine which pages can be cleaned.
    fn update_deleted_point(&self, deleted_point: Self::MetaData);
}
```

2.2.2 模块间交互

在新的代码框架下，Wal模块向其它模块提供的接口初步设计为

```
// manager.rs

pub struct WalManager<P: Page, M: PageManager, T: MetaDataStore> {
    sequence_num: SequenceNumber,
    writer: LogWriter<P, M>,
    reader: LogReader,
    cleaner: LogCleaner<M, T>,
}

impl<P: Page, M: PageManager, T: MetaDataStore> WalManager<P, M, T> {
    pub fn new() -> WalManager<P, M, T> {
        unimplemented!()
    }

    /// Provide iterator on necessary entries.
    pub fn read(&self) {
        unimplemented!()
    }

    /// Append the logs to the underlying storage.
    // TODO: Implement `write_batch()`.
    pub fn write<L: LogEntry>(&mut self, _log: L) {
        unimplemented!()
    }

    /// Mark the entries whose sequence number is in [0, `_sequence_num`] to
    /// be deleted in the future.
    pub fn mark_entries_up_to(&self, _sequence_num: SequenceNumber) {
        unimplemented!()
    }
}
```

由于与目前实现提供的接口有区别，因此要**替换基于RocksDB的WAL实现**，可能需要修改其它模块的部分代码，其中涉及的代码逻辑主要有

- **Recovery**：负责replay the logs，确保系统发生故障后的数据一致性，主要与`read()`逻辑相关。文件路径为`./src/analytic_engine/src/instance/wal_replayer.rs`
- **Mark Delete Entries**：负责标识哪些Log Entry被"持久化"，主要与`clean()`的逻辑相关。文件路径为`./src/analytic_engine/src/instance/flush_compaction.rs`
- **Manifest**：Manifest模块也涉及了与Wal相关的逻辑用于确保系统的Durability。文件路径为`./src/analytic_engine/src/manifest/details.rs`

（ps: 该链接中有整个Wal模块的代码设计方案，持续开发中）

---

3. 规划

为确保基于本地磁盘的WAL模块的开发能够按时完成，并达到预期的质量和功能要求，需要制定详细的时间规划。本项目**从7月1日开始，至9月30日结束，共计14周**。以下是项目的时间规划，分阶段详细列出了每个阶段的任务及其时间安排。*（每个阶段间有适当的耦合）*

3.1 任务分解

阶段	任务描述
技术设计阶段	深入阅读RocksDB、LevelDB等业界项目的WAL实现
	进一步完善WAL模块整体框架的设计
	调研并选择合适的存储格式（编码方式、metadata格式等）
	调研并选择合适的算法与数据结构
	设计初步的说明文档
核心代码开发阶段	根据设计框架实现衔接层和逻辑层的抽象
	以本地磁盘为底层存储对接衔接层，确保衔接层接口设计的可行性
	处理WAL模块与HoraeDB其它模块的通信
测试开发阶段	为WAL模块开发单元测试/集成测试
	调试代码，修复bug，确保WAL功能的鲁棒性
代码优化阶段	Review代码，使代码更简洁优雅、功能更完善
	跑benchmark并做进一步性能优化
文档实现阶段	编写完整的设计说明与技术实现文档
结项阶段	……

3.2 时间规划

时间	任务描述
第1-2周	技术设计阶段（第一阶段）
第3-6周	核心代码开发阶段（第二阶段）
第7-9周	测试开发阶段（第三阶段）
第10-12周	代码优化阶段（第四阶段）
第13-14周	文档实现阶段（第五阶段）

3.3 期望

过去的绝大多数时间，我都只是从开源项目中“吸收养分”，现在我希望能够更加深度地融入开源社区，一是认识更多优秀的Developers，与他们进行思维的碰撞并提升自己的认知与开发水平；二是为开源社区做贡献，建设这个给予我帮助的地方，也幻想着能够change things，做一些真正有意义的工作。