Databend

Distributed and standalone processing in databend

目录
CONTENT

1. Quickly understand by easy cases
2. Weaknesses in design
3. How to better processing one query #3380

⚠️ The query optimizer is not considered in this.

Quickly understand by easy cases

Weaknesses in design

- Talk is cheap. Show me the code

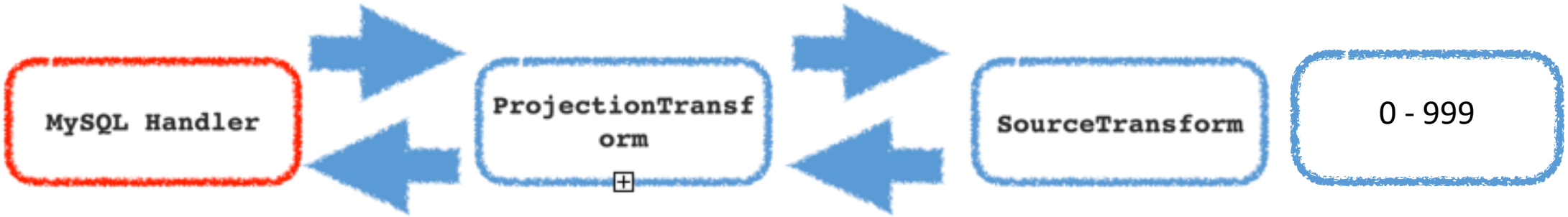
```rust
#[async_trait::async_trait]
pub trait Processor: Sync + Send {
    /// Processor name.
    fn name(&self) -> &str;

    /// Connect to the input processor, add an edge on the DAG.
    fn connect_to(&mut self, input: Arc<dyn Processor>) -> Result<()>;

    /// Inputs.
    fn inputs(&self) -> Vec<Arc<dyn Processor>>;

    /// Reference used for downcast.
    fn as_any(&self) -> &dyn Any;

    /// Execute the processor.
    async fn execute(&self) -> Result<SendableDataBlockStream>;
}
```

```rust
pub type SendableDataBlockStream =
    std::pin::Pin<
        Box<
            dyn futures::stream::Stream<
                Item = Result<DataBlock>
            > + Send
        >
    >;
```

Quickly understand by easy cases

— Standalone mode

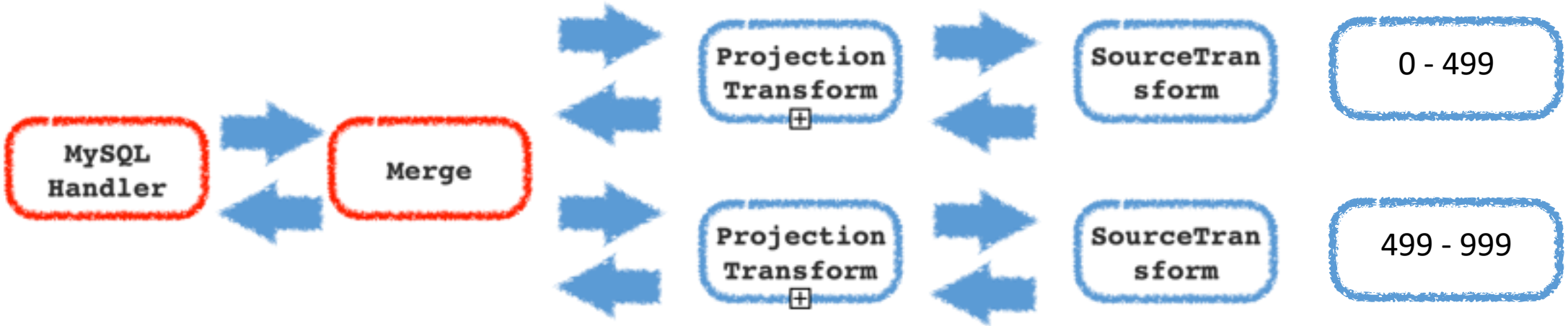
```
SET max_threads = 1;
SELECT * FROM numbers_mt(1000);
```



Quickly understand by easy cases

— Standalone mode

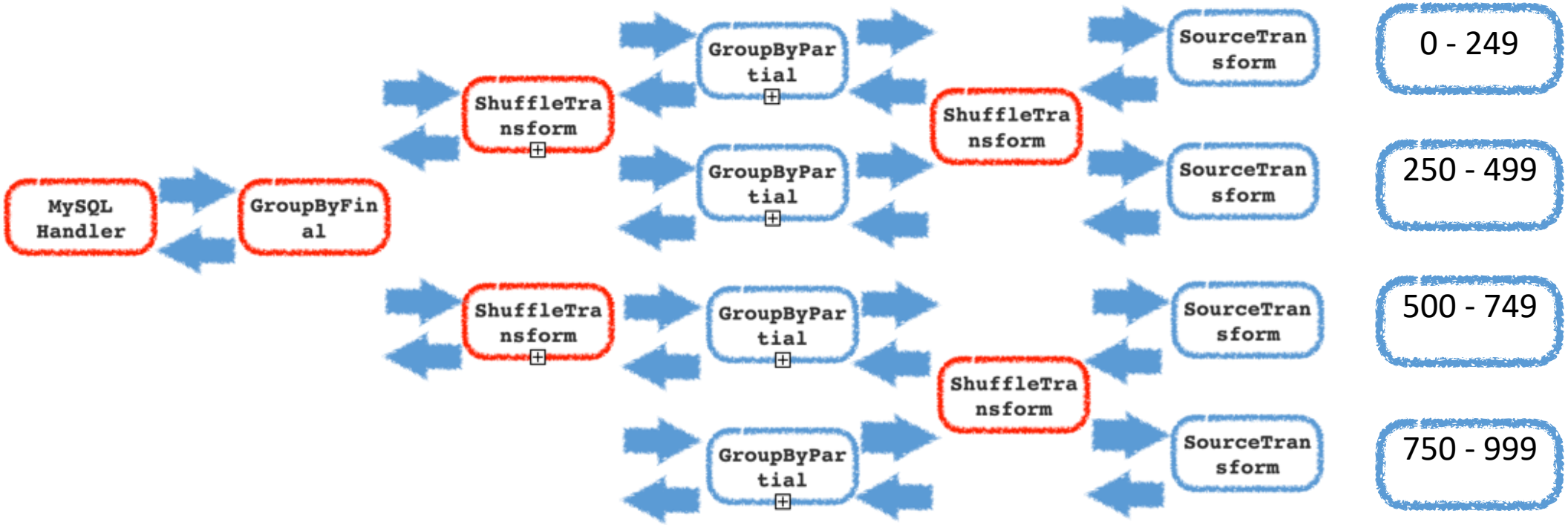
```
SET max_threads = 2;
SELECT * FROM numbers_mt(1000);
```



Quickly understand by easy cases

— Cluster mode(2 nodes)

```
SET max_threads = 2;
SELECT * FROM numbers_mt(1000);
```



Weaknesses in design

Weaknesses in design

- Functional weakness

- Scale in/out only to new queries

Scaling Up vs Scaling Out

Snowflake supports two ways to scale warehouses:

- Scale up by resizing a warehouse.
- Scale out by adding warehouses to a multi-cluster warehouse (requires Snowflake Enterprise Edition or higher).

Warehouse Resizing Improves Performance

Resizing a warehouse generally improves query performance, particularly for larger, more complex queries. It can also help reduce the queuing that occurs if a warehouse does not have enough compute resources to process all the queries that are submitted concurrently. Note that warehouse resizing is not intended for handling concurrency issues; instead, use additional warehouses to handle the workload or use a multi-cluster warehouse (if this feature is available for your account).

Snowflake supports resizing a warehouse at any time, even while running. If a query is running slowly and you have additional queries of similar size and complexity that you want to run on the same warehouse, you might choose to resize the warehouse while it is running; however, note the following:

- As stated earlier about warehouse size, larger is not necessarily faster; for smaller, basic queries that are already executing quickly, you may not see any significant improvement after resizing.
- Resizing a running warehouse does not impact queries that are already being processed by the warehouse; the additional compute resources, once fully provisioned, are only used for queued and new queries.
- Resizing between a 5XL or 6XL warehouse to a 4XL or smaller warehouse, will result in a brief period during which the customer is charged for both the new warehouse and the old warehouse while the old warehouse is quiesced.

https://docs.snowflake.com/en/user-guide/warehouses-considerations.html#scaling-up-vs-scaling-out

Weaknesses in design

- Performance weakness

Weaknesses in design

- Sync tasks also bear asynchronous costs

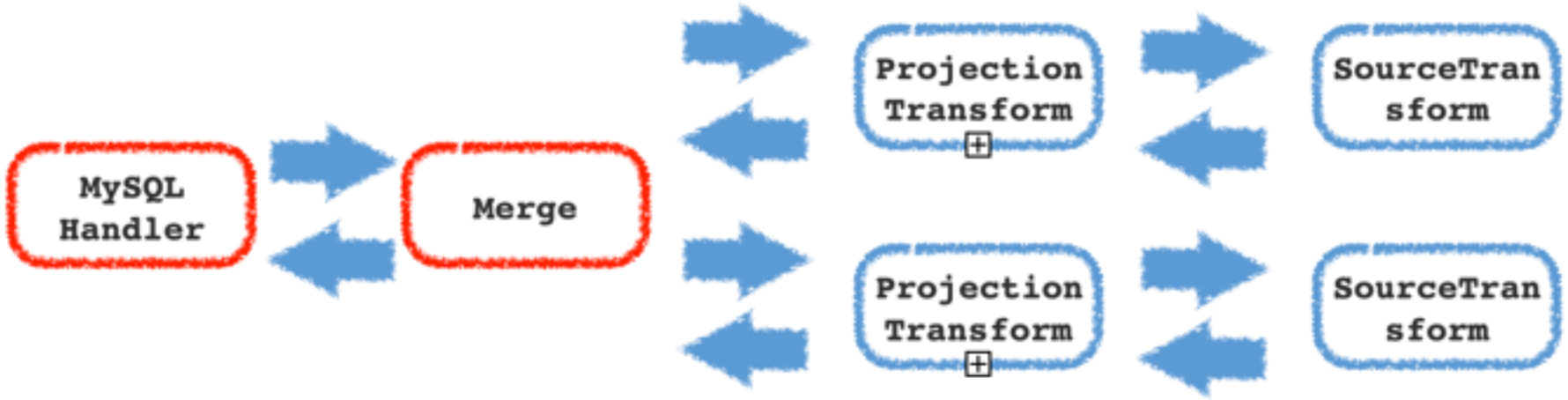
```rust
#[async_trait::async_trait]
pub trait Processor: Sync + Send {
    /// Processor name.
    fn name(&self) -> &str;

    /// Connect to the input processor, add an edge on the DAG.
    fn connect_to(&mut self, input: Arc<dyn Processor>) -> Result<()>;

    /// Inputs.
    fn inputs(&self) -> Vec<Arc<dyn Processor>>;

    /// Reference used for downcast.
    fn as_any(&self) -> &dyn Any;

    /// Execute the processor.
    async fn execute(&self) -> Result<SendableDataBlockStream>;
}
```

```rust
#[must_use = "streams do nothing unless polled"]
pub trait Stream {
    Values yielded by the stream.
    type Item;

    /* ... */
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)

    /* ... */
    #[inline]
    fn size_hint(&self) -> (usize, Option<usize>) { (0, None)
}
```

https://github.com/datafuselabs/databend/blob/main/query/src/pipelines/processors/processor.rs

Weaknesses in design

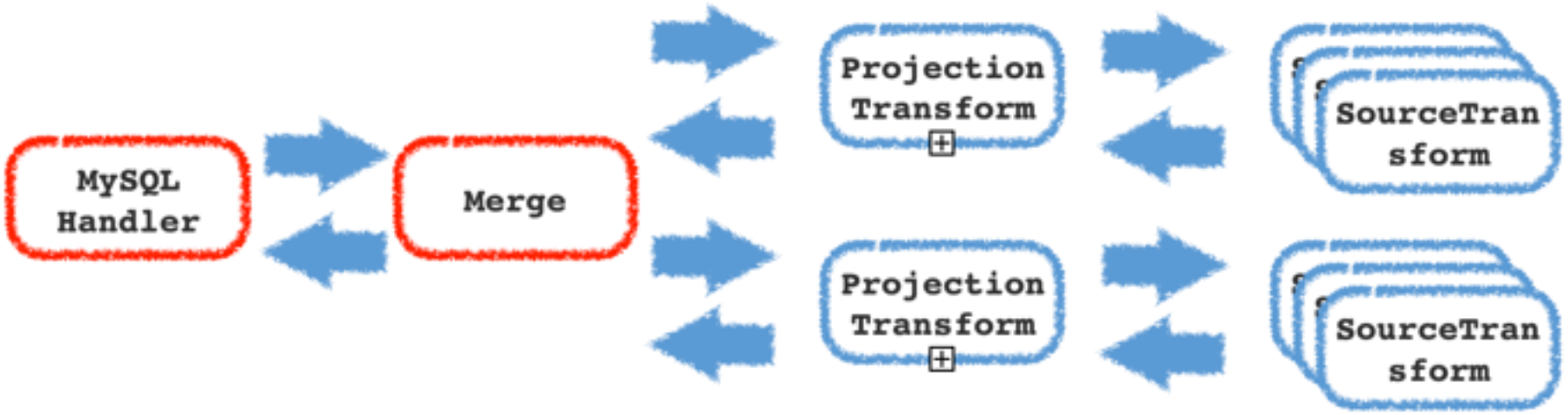
- Asynchronous tasks are not parallel



actual

Weaknesses in design

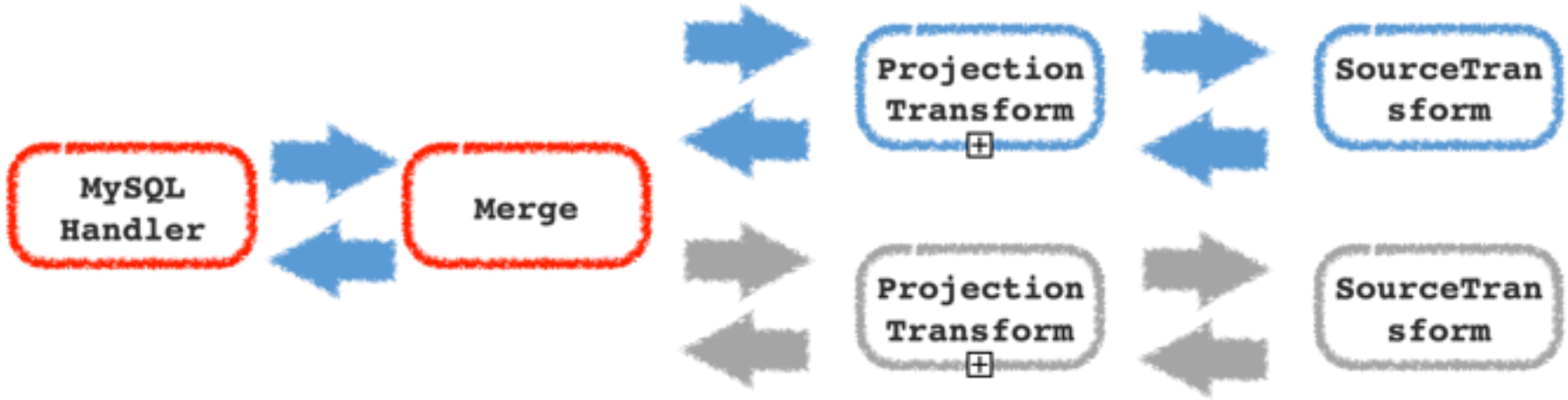
- Asynchronous tasks are not parallel



expect

Weaknesses in design

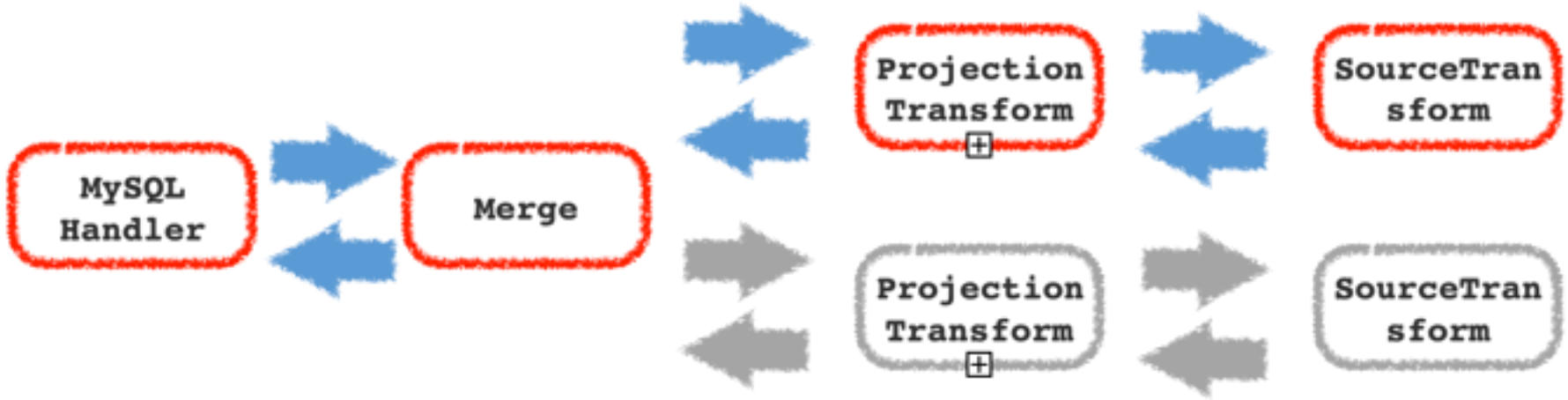
- Not steal tasks between threads



actual

Weaknesses in design

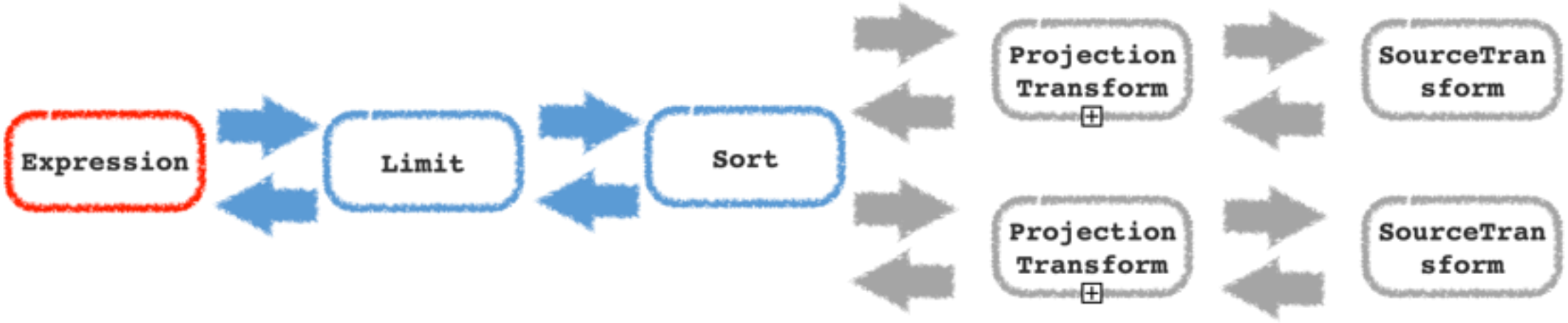
- Not steal tasks between threads



expect

Weaknesses in design

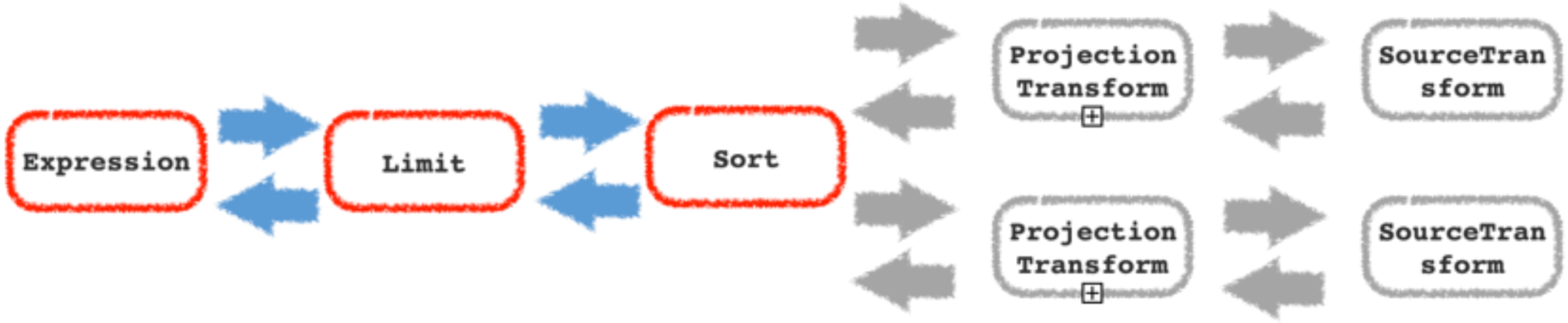
- Not steal tasks between threads



actual

Weaknesses in design

- Not steal tasks between threads



expect

How to better processing one query

https://github.com/datafuselabs/databend/pull/3380

How to better processing one query

- New Processor trait

```
11  + pub enum PrepareState {
12  +     NeedData,
13  +     NeedConsume,
14  +     Sync,
15  +     Async,
16  +     Finished,
17  + }
18  +
19  + // The design is inspired by ClickHouse processors
20  + #[async_trait::async_trait]
21  + pub trait Processor: Send {
22  +     fn prepare(&mut self) -> Result<PrepareState>;
23  +
24  +     // Synchronous work.
25  +     fn process(&mut self) -> Result<()> {
26  +         Err(ErrorCode::UnImplement("Unimplemented process."))
27  +     }
28  +
29  +     // Asynchronous work.
30  +     async fn async_process(&mut self) -> Result<()> {
31  +         Err(ErrorCode::UnImplement("Unimplemented async_process."))
32  +     }
33  + }
```

- **Each processor is a state machine**
- **Which method is executed by state**
- **Difference methods between sync and async**

- Atomic Port

```
+ #[repr(align(8))]
+ struct SharedData(pub Result<DataBlock>);
+
+ pub struct SharedStatus {
+     data: AtomicPtr<SharedData>,
+ }
+
+ impl SharedStatus {
+     pub fn create() -> Arc<SharedStatus> {
+         Arc::new(SharedStatus {
+             data: AtomicPtr::new(std::ptr::null_mut()),
+         })
+     }
+
+     pub fn swap(&self, data: Option<Result<DataBlock>>, flags: usize) -> Option<R
+         let mut expected = std::ptr::null_mut();
+         let desired = match data {
+             None => std::ptr::null_mut(),
+             Some(data) => {
+                 let new_data = Box::into_raw(Box::new(SharedData(data)));
+                 (new_data as usize | flags) as *mut SharedData
+             }
+         };
+
+         loop {
+             match self.data.compare_exchange_weak(
+                 expected,
+                 desired,
+                 Ordering::SeqCst,
+                 Ordering::Relaxed,
+             ) {
+                 Err(new_expected) => {
```

- **State and data in CAS**
- **Support pull and push data**
- **Each processor has inputs and outputs**

- Example SyncSource

```rust
struct ExampleSyncSource {
    pos: usize,
    data_blocks: Vec<DataBlock>,
}

impl ExampleSyncSource {
    pub fn create(
        data_blocks: Vec<DataBlock>,
        outputs: Vec<OutputPort>,
    ) -> Result<ProcessorPtr> {
        SyncSourceProcessorWrap::create(outputs, ExampleSyncSource {
            pos: 0,
            data_blocks,
        })
    }
}

impl SyncSource for ExampleSyncSource {
    fn generate(&mut self) -> Result<Option<DataBlock>> {
        self.pos += 1;
        match self.data_blocks.len() >= self.pos {
            true => Ok(Some(self.data_blocks[self.pos - 1].clone())),
            false => Ok(None),
        }
    }
}
```

```rust
+ #[async_trait::async_trait]
+ impl<T: 'static + SyncSource> Processor for SyncSourceProcessorWrap<T> {
+     fn prepare(&mut self) -> Result<PrepareState> {
+         Ok(match &self.generated_data {
+             None if self.is_finish => self.close_outputs(),
+             None => PrepareState::Sync,
+             Some(_) => match self.push_data_to_outputs() {
+                 true => self.close_outputs(),
+                 false => PrepareState::NeedConsume,
+             },
+         })
+     }
+
+     fn process(&mut self) -> Result<()> {
+         match self.inner.generate()? {
+             None => self.is_finish = true,
+             Some(data_block) => self.generated_data = Some(data_block),
+         };
+
+         Ok(())
+     }
+ }
```

- Execute one processor in executor

```
pub unsafe fn schedule_next(graph: &StateLockGuard, pid: usize) -> Result<ScheduleQueue> {
    let mut need_schedule_nodes = vec![pid];
    let mut schedule_queue = ScheduleQueue::create();

    while !need_schedule_nodes.is_empty() {
        if let Some(need_schedule_pid) = need_schedule_nodes.pop() {
            let mut node = graph.nodes[need_schedule_pid].lock();

            match (*node.processor.get()).prepare()? {
                PrepareState::Finished => {
                    node.state = RunningState::Finished;
                }
                PrepareState::NeedData | PrepareState::NeedConsume => {
                    node.state = RunningState::Idle;
                }
                PrepareState::Sync => {
                    node.state = RunningState::Processing;
                    schedule_queue.push_sync(node.processor.clone());
                }
                PrepareState::Async => {
                    node.state = RunningState::Processing;
                    schedule_queue.push_async(node.processor.clone());
                }
            }

            let need_schedule_inputs = &mut *node.inputs.get();
            let need_schedule_outputs = &mut *node.outputs.get();
```

- **Lock and call Processor::prepare**
- **Which method is executed by state**
- **Add self to schedule queue if ready state**
- **Add neighbor processor to schedule queue**

How to better processing one query

- Execute schedule queue in executor

```rust
pub fn execute_with_single_worker(&self, worker_num: usize) -> Result<()> {
    let mut context = ExecutorWorkerContext::create(worker_num);

    while !self.global_tasks_queue.is_finished() {
        // When there are not enough tasks, the thread will be blocked, so we nee
        while !self.global_tasks_queue.is_finished() && !context.has_task() {
            self.global_tasks_queue.steal_task_to_context(&mut context);
        }

        while context.has_task() {
            let executed_pid = context.execute_task(&self.global_tasks_queue)?;

            // We immediately schedule the processor again.
            let schedule_queue = self.graph.schedule_next(executed_pid)?;
            schedule_queue.schedule(&self.global_tasks_queue, &mut context);
        }
    }

    Ok(())
}
```

- **All workers share a global queue**
- **Each worker has a local task queue**
- **First task is put to local queue without lock**
- **Other tasks put to global queue with lock**

- Execute schedule queue in executor

```rust
pub fn execute_task(&mut self, queue: &ExecutorTasksQueue) -> Result<usize> {
    match std::mem::replace(&mut self.task, ExecutorWorkerTask::None) {
        ExecutorWorkerTask::None => Err(ErrorCode::LogicalError("Execute none task.
        ExecutorWorkerTask::Sync(processor) => self.execute_sync_task(processor),
        ExecutorWorkerTask::Async(processor) => self.execute_async_task(processor,
        ExecutorWorkerTask::AsyncSchedule(boxed_future) => self.schedule_async_task
    }
}

fn execute_sync_task(&mut self, processor: ProcessorPtr) -> Result<usize> {
    unsafe {
        (&mut *processor.get()).process()?;
        unimplemented!()
    }
}
```

- **Call Processor::process if sync state**

- Execute schedule queue in executor

```
fn execute_async_task(&mut self, processor: ProcessorPtr, queue: &ExecutorTasksQueue) -> Result<usize> {
    unsafe {
        let mut boxed_future = (&mut *processor.get()).async_process().boxed();
        self.schedule_async_task(boxed_future, queue)
    }
}

fn schedule_async_task(&mut self, mut boxed_future: BoxFuture<'static, Result<()>>, queue: &ExecutorTasksQueue) -
    loop {
        let finished = Arc::new(AtomicBool::new(false));
        let waker = ExecutingAsyncTaskWaker::create(&finished);

        let waker = futures::task::waker_ref(&waker);
        let mut cx = Context::from_waker(&waker);

        match boxed_future.as_mut().poll(&mut cx) {
            Poll::Ready(Ok(res)) => { return unimplemented!(); }
            Poll::Ready(Err(cause)) => { return Err(cause); }
            Poll::Pending => {
                let async_task = ExecutingAsyncTask { finished: finished.clone(), future: boxed_future };

                match queue.push_executing_async_task(self.worker_num, async_task) {
                    None => { return unimplemented!(); }
                    Some(task) => { boxed_future = task.future; }
                };
            }
        };
    }
}
```

- **Call Processor::async_process if async state**
- **Mock waker with atomic_bool flags**
- **Push to global queue if async not ready**

- Execute worker in executor

```rust
impl PipelineThreadsExecutor {
    pub fn start(&mut self, workers: usize) -> Result<()> {
        if self.is_started {
            return Err(ErrorCode::PipelineAreadlyStarted(
                "PipelineThreadsExecutor already started.",
            ));
        }

        self.is_started = true;
        self.base.initialize_executor(workers)?;
        for worker_num in 0..workers {
            let worker_executor = self.base.clone();
            Thread::spawn(move || {
                if let Err(cause) = worker_executor.execute_with_single_worker(worker_num)
                    // TODO:
                    println!("Executor error : {:?}", cause);
                }
            });
        }
        Ok(())
    }
```

- **Start workers with thread pool**
- **Wait all processor is finished**

Databend

感谢您的观看

THANK YOU FOR WATCHING