Execution Engine: KCS

`Authors: Connor, Kyle, Sarvesh`

Vectorized push-based velox inspired execution engine

Step 1: Finalize Interfaces

Finalize API with other teams:

- I/O Service
- Catalog
- Scheduler

Step 1.1: Potential StorageClient API

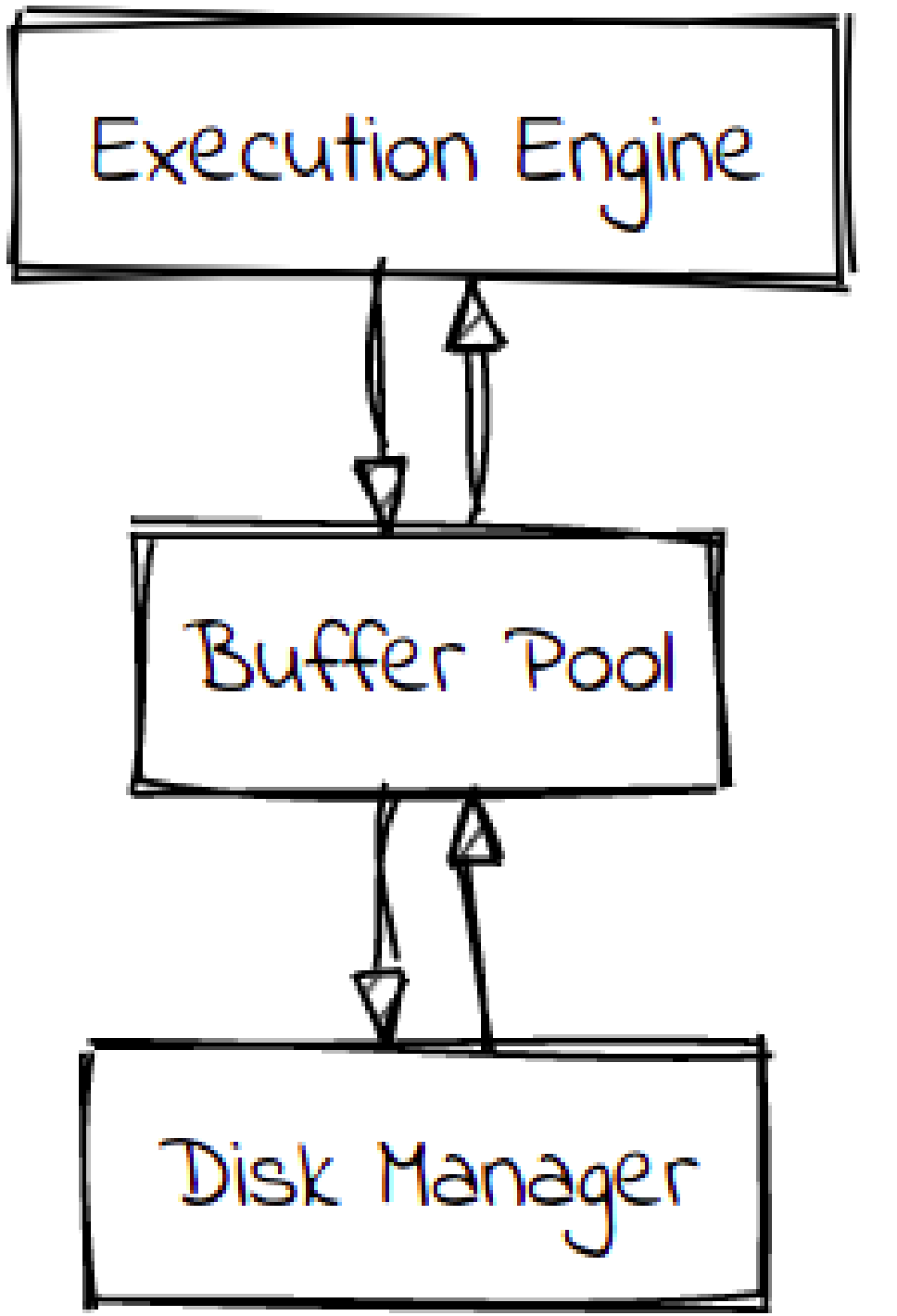
```rust
impl StorageClient {
    /// Have some sort of way to create a `StorageClient` on our local node.
    pub fn new(_id: usize) -> Self {
        Self
    }

    /// The only other function we need exposed would be a way to actually get data.
    /// What we should get is a stream of `Recordbatch`s, which is just Apache Arrow
    /// data in memory.
    ///
    /// The executor node really should not know what the underlying data is on the Blob data store.
    /// In our case it is Parquet, but since the Execution Engine is not in charge or loading
    /// those Parquet files, it should just receive it as in-memory Arrow data
    ///
    /// Note that we will likely re-export the `SendableRecordBatchRecord` from DataFusion
    /// and use that as the return type instead
    pub async fn request_data(
        &self,
        _request: BlobData,
    ) -> Result<Box<dyn Stream<Item = RecordBatch>>> {
        todo!()
    }
}
```

Step 1.2: Example usage of the storage client

```rust
//! Example `main` function for the EE teams

use testing_721::operators;
use testing_721::operators::Operator;
use testing_721::storage_client;

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    // Initialize a storage client
    let sc = storage_client::StorageClient::new(42);

    // Formualte a request we want to make to the storage client
    let request = create_column_request();

    // Request data from the storage client. Note that this request could fail
    let stream = sc.request_data(request).await?;

    // Theoretically, there could be a pipeline breaker somehwere that turns the asynchronous
    // flow into a synchronous one, turning the Stream into an Iterator

    // Executor node returns a future containing another stream that can be sent to another operator
    let table_scan_node = operators::TableScan::new();
    let result = table_scan_node.execute_with_stream(stream);

    Ok(())
}

/// Just formulate a toy example of a request we could make to the `StorageClient`
fn create_column_request() -> storage_client::BlobData {
    let columns = vec!["grades".to_string(), "name".to_string(), "gpa".to_string()];
    storage_client::BlobData::Columns("students".to_string(), columns.into_boxed_slice())
}
```

Step 2: Buffer Pool Manager

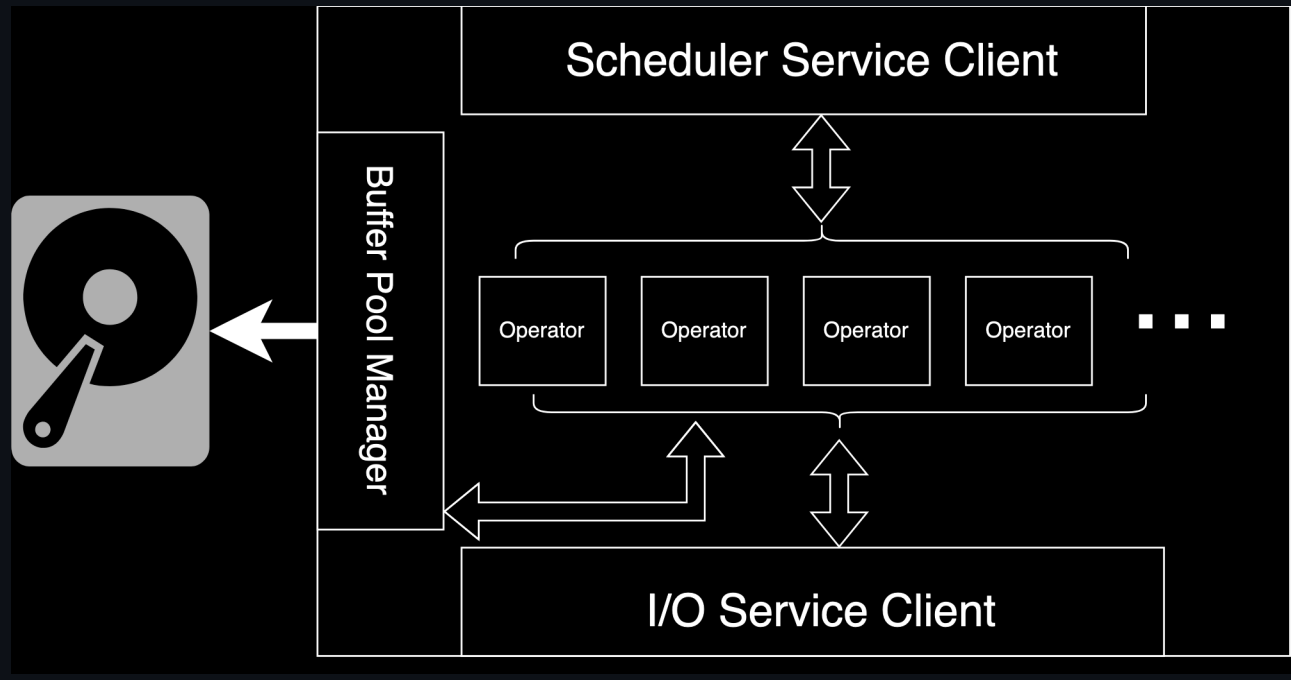
Need to spill the data to local disk.



Step 3: Implement operators

- TableScan
- FilterProject
- HashAggregation
- HashProbe + HashBuild
- MergeJoin
- NestedLoopJoin
- OrderBy
- TopN
- Limit
- Values
- More may be added as a stretch goal.

Final Design



Our Design Rationale

- Robust
- Forward Compatibility

For the sake of code quality...

- Pair programming
- Unit testing

Stretch Goal

- Integrating with a DBMS
- Testing against TPC-H or TPC-H like workload

List of rust crates we plan to use

- `arrow` : for handling the Apache Arrow format
- `tokio` : high performance async runtime
- `rayon` : data parallelism crate