Execution Engine: KCS

Authors: Connor, Kyle, Sarvesh

Vectorized Push-Based inspired Execution Engine

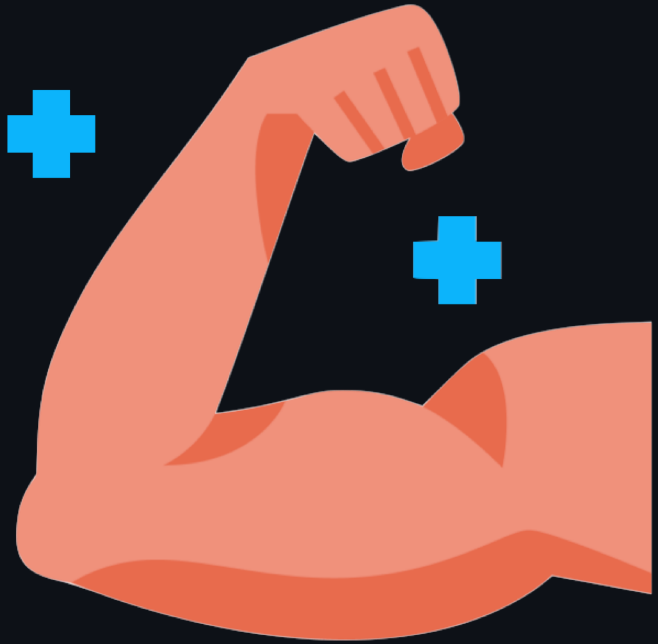
Overview

We will be taking heavy inspiration from:

- DataFusion
- Velox
- InfluxDB
    - which is built on top of DataFusion

Our Design Goals

- Robustness
- Modularity
- Extensibility
- Forward Compatibility



Features

- Encode behavior in the type system
- Provide bare minimum statistics the optimizer needs
  - Timing
  - Cardinality

List of rust crates we plan to use

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

Design Rationale

Push vs Pull Based

Push	Pull
Improves cache efficiency by removing control flow logic	Easier to implement
Forking is efficient: You push a thing only once	Operators like LIMIT make their producers aware of when to stop running (Headache for the optimizer)
Parallelization is easier	Parallelization is harder

Step 1: Finalize Interfaces

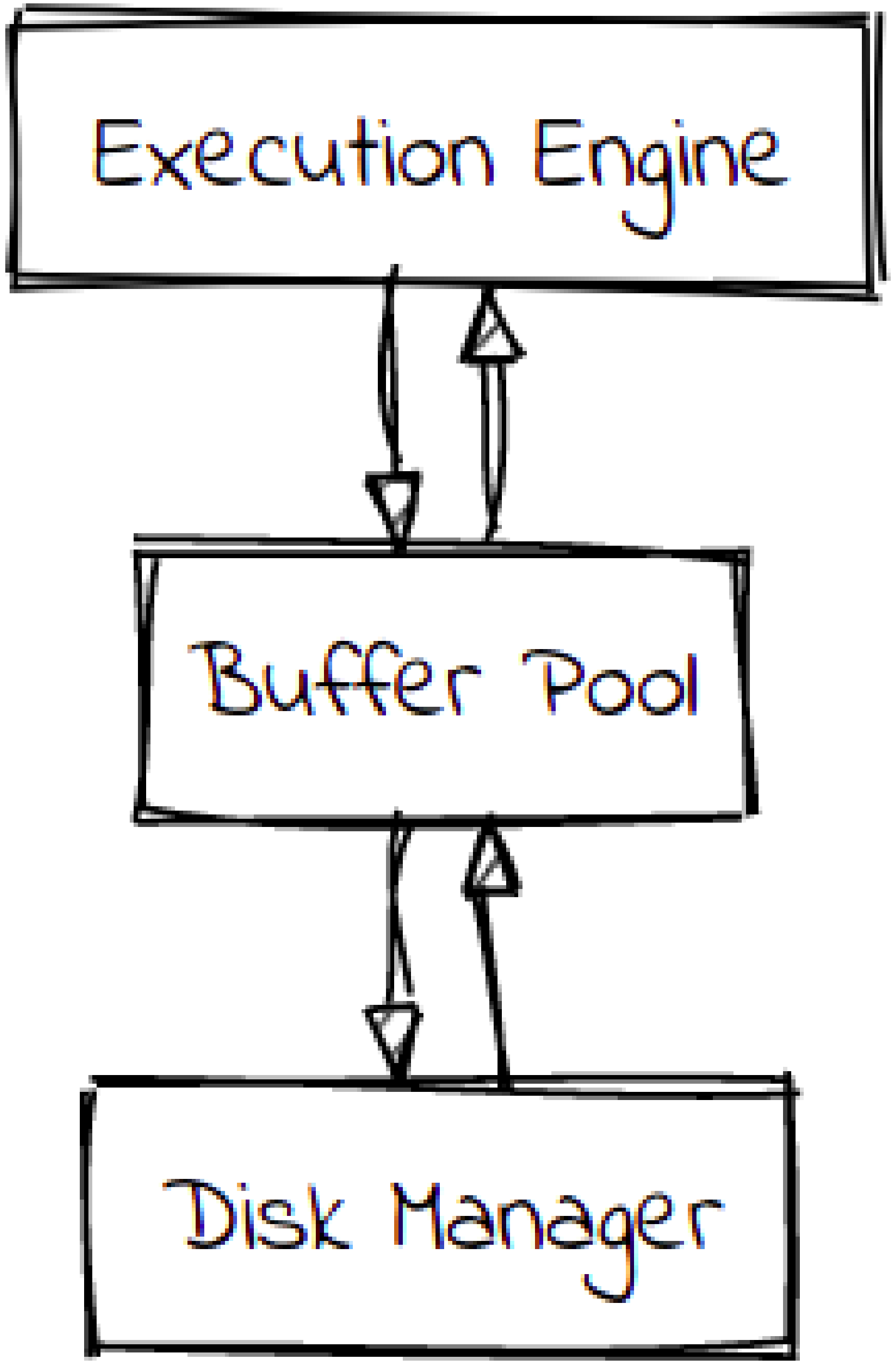
Finalize API with other teams:

- I/O Service
- Catalog
- Scheduler

Step 2: Buffer Pool Manager

Need to spill the data to local disk.

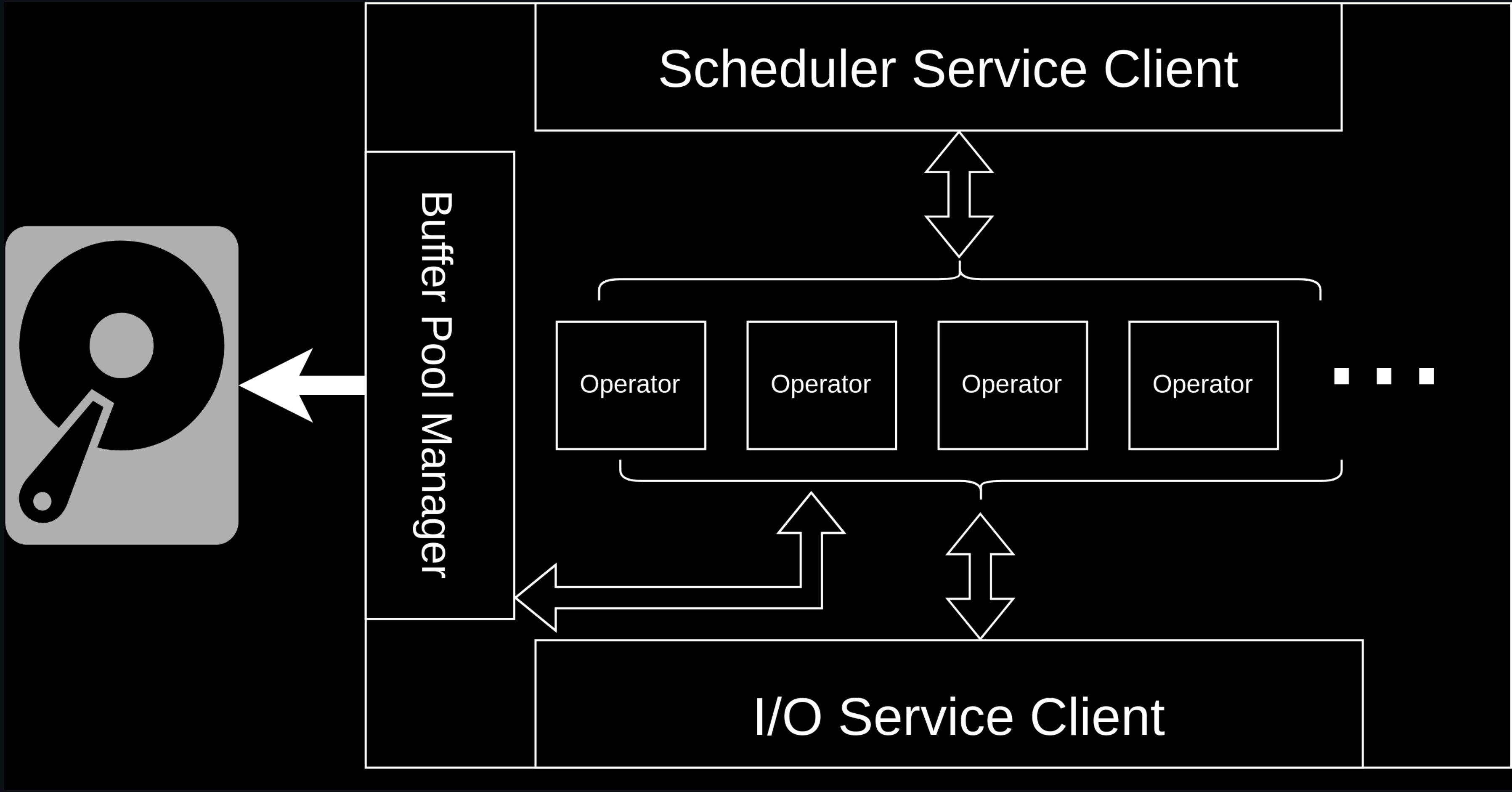
- Can potentially rip out the `memory_pool`



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



Testing

- Unit tests for each operator
- Timing each operator's performance to benchmark our code

For the sake of code quality...

- Pair programming (all combinations: KC, KS, CS)
- Unit testing for each operator
- Integrated tests across mutliple operators

Stretch Goal

- Integrating with a DBMS
- Testing against TPC-H or TPC-H like workload
- Add a lot of statistics and timers to each operator (for optimizer's sake)

Potential `StorageClient` API

```rust
/// Will probably end up re-exporting this type:
pub type SendableRecordBatchStream =
    Pin<Box<
        dyn RecordBatchStream<Item =
            Result<RecordBatch, DataFusionError>
        > + Send
    >>;

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

    pub async fn request_data(
        &self,
        _request: BlobData,
    ) -> SendableRecordBatchStream {
        todo!()
    }
}
```

Example usage of the storage client

```rust
#[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?;

    // 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(())
}
```