# P4 - MiniDB

**Deadline**: Sunday, May 5, 2024

**Accept Assignment**: https://classroom.github.com/a/0rlt1cG1

**Submit Assignment**: https://autolab.cse.buffalo.edu/courses/cse410-s24/assessments/P4-MiniDB

In this assignment, you will implement a simple database engine

This assignment is intended to: - Serve as a capstone, integrating P1 to P3. - Provide insight into the implementation of a database's query engine

You should expect to spend approximately 20-30 hours on this assignment. Plan accordingly.

---

To complete this assignment, you should:

1. Accept this assignment through GitHub Classroom.
2. Modify the file `src/query.rs`, implementing the functions labeled `todo!()`. Note that you may need to add additional fields to some structures.
3. Commit your changes and push them to Github.
4. Go to Autolab, select your repository, acknowledge the course AI Policy, and click Submit.

You may repeat steps 2-4 as many times as desired. You may also modify `src/data/database.rs`.

---

## Overview

In this assignment you will implement the core components of a database query engine in a sequence of steps.

Initially, the engine supports queries of the form:

```
SELECT *
SELECT * FROM table
```

#### Composing Operators

Directly creating an iterator that implements **all** of a query will be incredibly complex. Instead, you are encouraged to adopt a compositional model of code based on `Source<Row>`:

- The `FROM` clause is the root
- If a `WHERE` clause is present, transform the root `Source` appropriately
- If the `SELECT` clause is not a wildcard, transform the root `Source` appropriately

In other words, each clause extends the structure of the query, potentially adding another layer over the prior one. Using `Source<Row>` abstracts the lower layers, allowing the code to treat them abstractly.

#### Testing

Tests for each step are available via `cargo test`

Also note that `cargo run` will bring up a simple command-line shell that allows you to enter SQL queries, one per line. You will need to run `cargo test` at least once to initialize the tables `foo` and `bar`.

---

## Documentation

### `MiniValue`

Values in MiniDB are allowed to be one of: - String - Integer - Float - Boolean - Null

MiniValue provides a wrapper around these types, allowing Rust to use them semi-interchangeably (this is referred to as Boxing the type, not unlike Rust's `Box` type).

Many useful operations are defined over MiniValue directly, including arithmetic, coercion and more.

Note the presence of `expr::eval::eval` (or one of the several classes that use it (detailed in `Source<Row>`, below)

### `Row`

A row is a wrapper around a vector of `MiniValue`s. You can access individual fields with `row[index]`. If you have access to the row's `Schema`, you can use `row.get(...)` to retrieve a field by its name.

### `Source<Row>`

As in P3, we will be using `Source<>` classes as a way to compose simple database operations. For this project, we will be focusing on `Source<Row>` (i.e., collections of `Row` records)

See `src/data/row_sources.rs` for several templates that may be useful:

- `TransformRows`: Generate a new sequence of rows by applying a vector of expressions to input rows (c.f. `SELECT`)
- `FilterRows`: Generate a new sequence of rows by filtering rows based on an expression (c.f. `WHERE`)
- `ConcatRows`: Wraps around a Join Source to translate a `(Row, Row)` tuple into a single `Row` object with fields concatenated (c.f. `JOIN`).

### `QueryResult`

Most methods in `src/query.rs` return `QueryResult`. Successful responses consist of: - `Source<Row>`: A collection of rows representing the result of the query - `Schema`: The names of columns in the source

---

## Objectives

You are encouraged to add features incrementally, according to the following order.

#### Step 1

Add support for constant expressions in the `SELECT` clause

```
SELECT 1
SELECT 1+3
SELECT 1 FROM foo
```

Notes: - See notes on `Source<Row>`, below

#### Step 2

Add support for non-constant expressions in the `SELECT` clause

```
SELECT a, a * 2 AS b FROM foo
SELECT a FROM foo
```

Notes: - See notes on `Source<Row>`, below - As a simplifying assumption, you may assume that each field Name appears only once per row. That is, you will never see a table with two identically named fields, or a join of a table with itself.

**Step 3**

Add support for the `WHERE` clause

```
SELECT * FROM foo WHERE a > 10
SELECT a FROM foo WHERE b < 50
```

Notes: - See notes on `Source<Row>` , below

**Step 4**

Add support for `JOIN` terms in the `FROM` clause

```
SELECT * FROM foo JOIN bar
SELECT * FROM foo JOIN bar ON b = c
```

Computing `A JOIN B ON expr` should run in expected O(| `A` |) + O(| `B` |) + O(| `A JOIN B ON expr` |).

**Step 5**

Add support for multiple entries in the `FROM` clause.

```
SELECT * FROM foo, bar
SELECT * FROM foo, bar WHERE b = c
```

---

## Development Ideas / Stretch Goals

- Add support for inserting rows into existing tables by adding support for the `INSERT` statement in `src/data/database.rs`
- Add support for creating new tables by adding support for the `CREATE TABLE` statement in `src/data/database.rs`
- Add support for selecting from CSV files.
- Modify the backing store for tables from `DataFile` to `BPlusTree` from your P2. Note that all tables will need a `key` field if you do this.
- The naive implementation of `SELECT * FROM foo, bar WHERE b = c` produces an iterator that runs in O(| `foo` | * | `bar` |). Modify `simple_select(...)` to produce an iterator based on a hash join.
- The naive implementation of `SELECT * FROM foo WHERE b > 40` produces an iterator

that runs in O(|`foo`|). Modify `simple_select(...)` and use your `BPlusTree` implementation to make it possible to produce an iterator that runs in O(log|`foo`|) + O(|`foo WHERE B > 40`|).

- Add support for qualified field names (e.g., `foo.a` or `bar.c`)