PROJECT – 3

→ First, I had opened the Ubuntu command prompt and in that I had created a new project called ‘sql_query_validator’ by using the command:

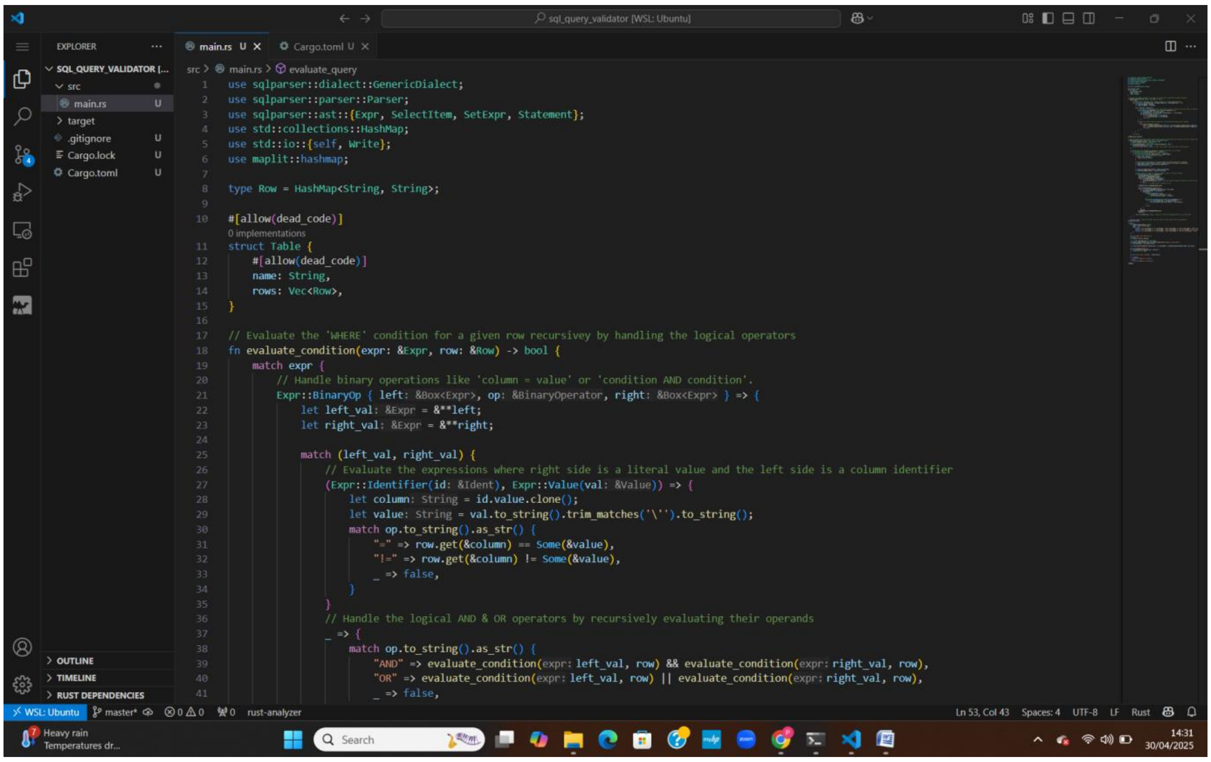
‘cargo new sql_query_validator’

and next I redirected into that project by using the command:

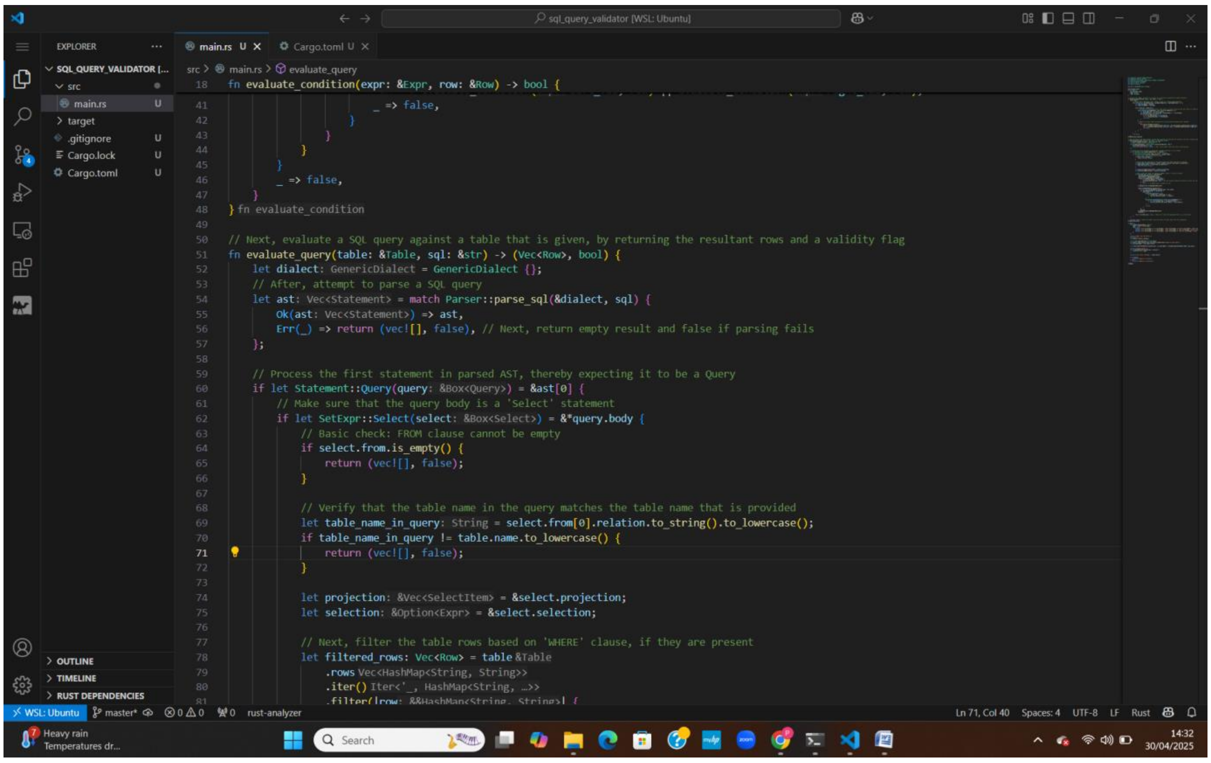
‘cd sql_query_validator’.

→ Next, I had opened VS code and connected my VS code to WSL and opened the project that I had created and in that I had opened the ‘main.rs’ rust file and entered the code for sql query validation as below:

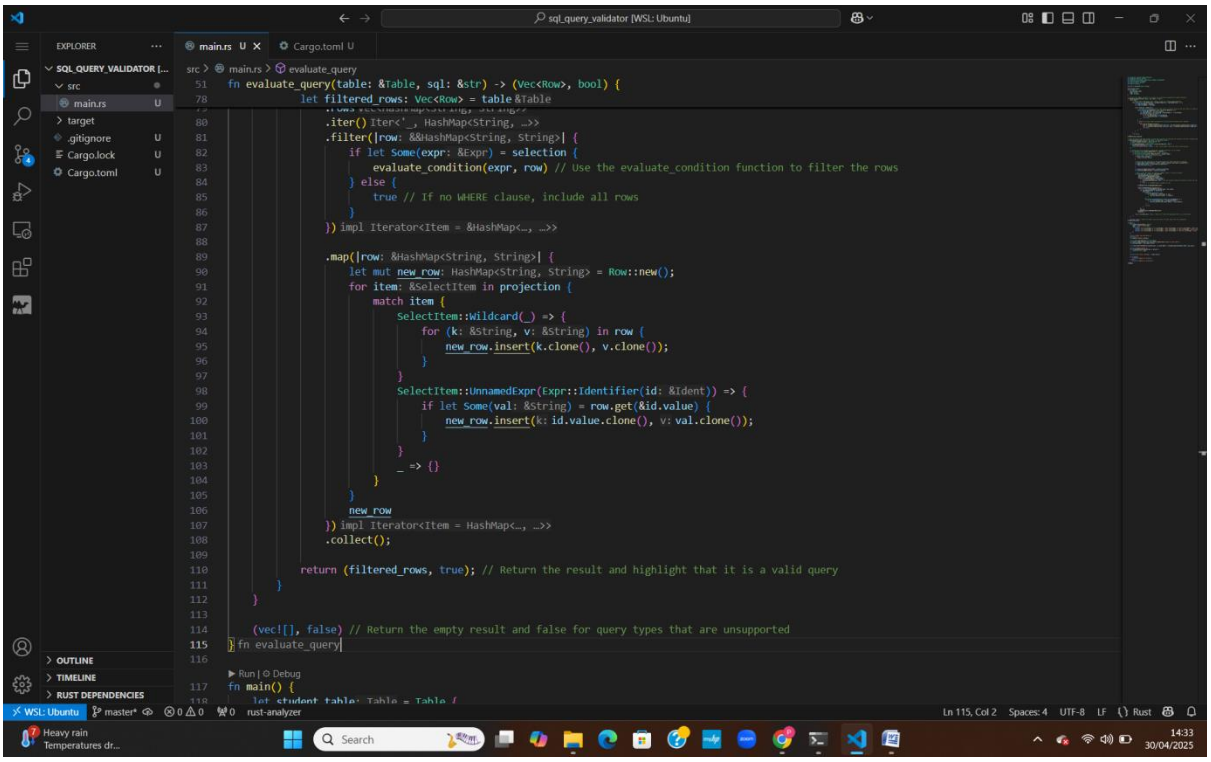
SQL QUERY VALIDATOR CODE:



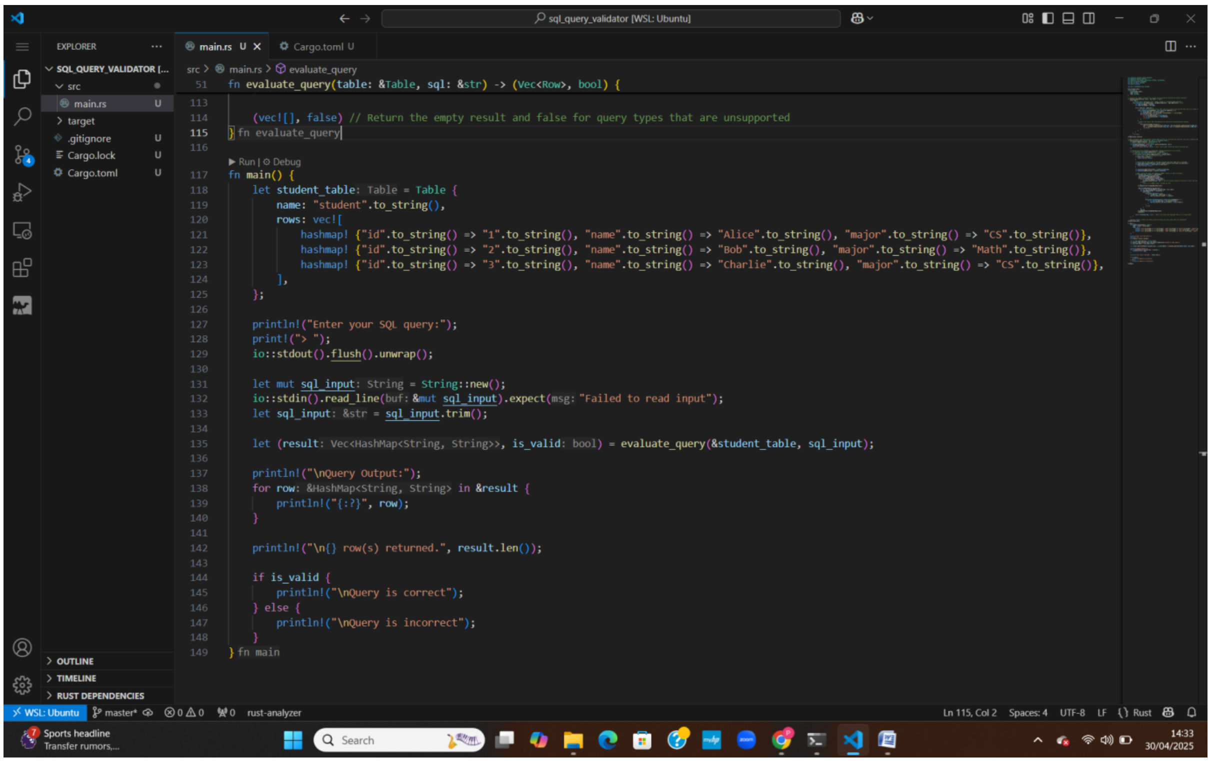
```rust
use sqlparser::dialect::GenericDialect;
use sqlparser::parser::Parser;
use sqlparser::ast::{Expr, SelectItem, SetExpr, Statement};
use std::collections::HashMap;
use std::io::{self, Write};
use maplit::hashmap;

type Row = HashMap<String, String>;

#[allow(dead_code)]
0 implementations
struct Table {
    #[allow(dead_code)]
    name: String,
    rows: Vec<Row>,
}

// Evaluate the 'WHERE' condition for a given row recursivey by handling the logical operators
fn evaluate_condition(expr: &Expr, row: &Row) -> bool {
    match expr {
        // Handle binary operations like 'column = value' or 'condition AND condition'.
        Expr::BinaryOp { left: &Box<Expr>, op: &BinaryOperator, right: &Box<Expr> } => {
            let left_val: &Expr = &**left;
            let right_val: &Expr = &**right;

            match (left_val, right_val) {
                // Evaluate the expressions where right side is a literal value and the left side is a column identifier
                (Expr::Identifier(id: &Ident), Expr::Value(val: &Value)) => {
                    let column: String = id.value.clone();
                    let value: String = val.to_string().trim_matches('\'').to_string();
                    match op.to_string().as_str() {
                        "=" => row.get(&column) == Some(&value),
                        "!=" => row.get(&column) != Some(&value),
                        _ => false,
                    }
                }
                // Handle the logical AND & OR operators by recursively evaluating their operands
                _ => {
                    match op.to_string().as_str() {
                        "AND" => evaluate_condition(expr: left_val, row) && evaluate_condition(expr: right_val, row),
                        "OR" => evaluate_condition(expr: left_val, row) || evaluate_condition(expr: right_val, row),
                        _ => false,
```



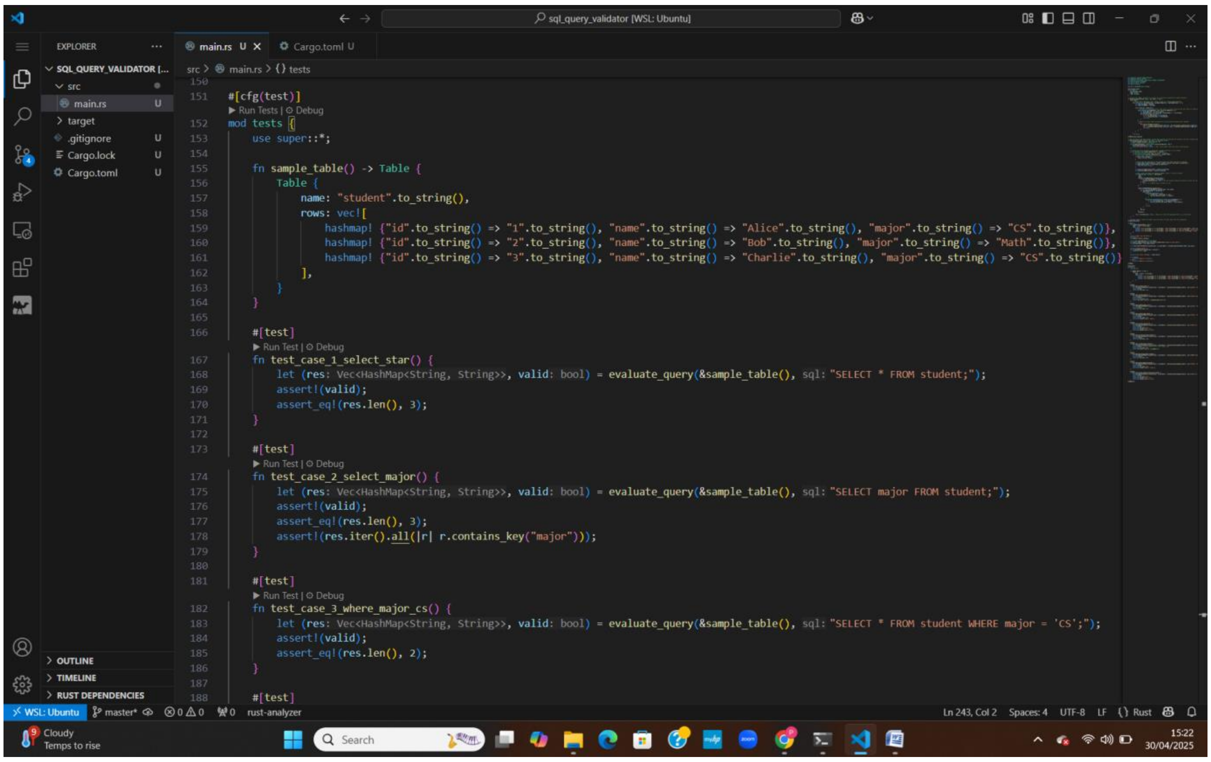
```
fn evaluate_condition(expr: &Expr, row: &Row) -> bool {
                        _ => false,
                    }
                }
            }
        }
        _ => false,
    }
} fn evaluate_condition

// Next, evaluate a SQL query against a table that is given, by returning the resultant rows and a validity flag
fn evaluate_query(table: &Table, sql: &str) -> (Vec<Row>, bool) {
    let dialect: GenericDialect = GenericDialect {};
    // After, attempt to parse a SQL query
    let ast: Vec<Statement> = match Parser::parse_sql(&dialect, sql) {
        Ok(ast: Vec<Statement>) => ast,
        Err(_) => return (vec![], false), // Next, return empty result and false if parsing fails
    };

    // Process the first statement in parsed AST, thereby expecting it to be a Query
    if let Statement::Query(query: &Box<Query>) = &ast[0] {
        // Make sure that the query body is a 'Select' statement
        if let SetExpr::Select(select: &Box<Select>) = &*query.body {
            // Basic check: FROM clause cannot be empty
            if select.from.is_empty() {
                return (vec![], false);
            }

            // Verify that the table name in the query matches the table name that is provided
            let table_name_in_query: String = select.from[0].relation.to_string().to_lowercase();
            if table_name_in_query != table.name.to_lowercase() {
                return (vec![], false);
            }

            let projection: &Vec<SelectItem> = &select.projection;
            let selection: &Option<Expr> = &select.selection;

            // Next, filter the table rows based on 'WHERE' clause, if they are present
            let filtered_rows: Vec<Row> = table &Table
                .rows Vec<HashMap<String, String>>
                .iter() Iter<'_, HashMap<String, …>>
```



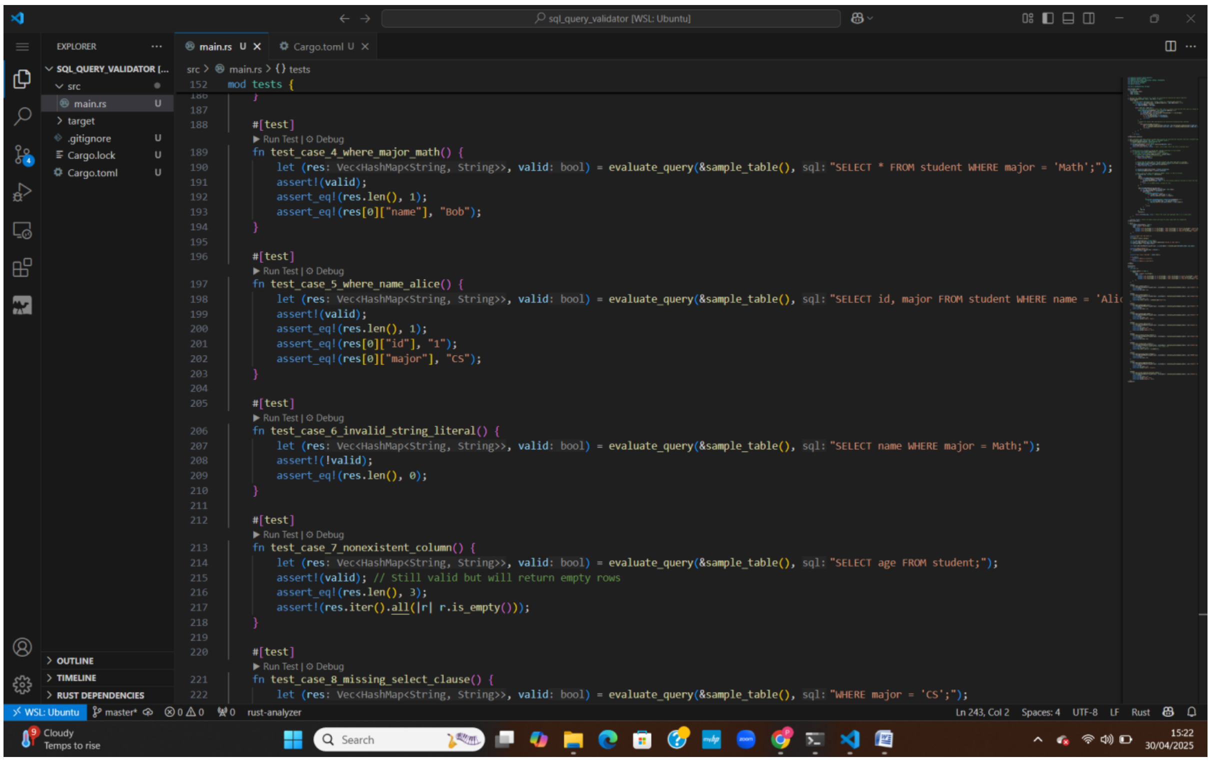
```
fn evaluate_query(table: &Table, sql: &str) -> (Vec<Row>, bool) {
            let filtered_rows: Vec<Row> = table &Table
                .iter() Iter<'_, HashMap<String, …>>
                .filter(|row: &&HashMap<String, String>| {
                    if let Some(expr: &Expr) = selection {
                        evaluate_condition(expr, row) // Use the evaluate_condition function to filter the rows
                    } else {
                        true // If no WHERE clause, include all rows
                    }
                }) impl Iterator<Item = &HashMap<…, …>>

                .map(|row: &HashMap<String, String>| {
                    let mut new_row: HashMap<String, String> = Row::new();
                    for item: &SelectItem in projection {
                        match item {
                            SelectItem::Wildcard(_) => {
                                for (k: &String, v: &String) in row {
                                    new_row.insert(k.clone(), v.clone());
                                }
                            }
                            SelectItem::UnnamedExpr(Expr::Identifier(id: &Ident)) => {
                                if let Some(val: &String) = row.get(&id.value) {
                                    new_row.insert(k: id.value.clone(), v: val.clone());
                                }
                            }
                            _ => {}
                        }
                    }
                    new_row
                }) impl Iterator<Item = HashMap<…, …>>
                .collect();

            return (filtered_rows, true); // Return the result and highlight that it is a valid query
        }
    }

    (vec![], false) // Return the empty result and false for query types that are unsupported
} fn evaluate_query

fn main() {
```



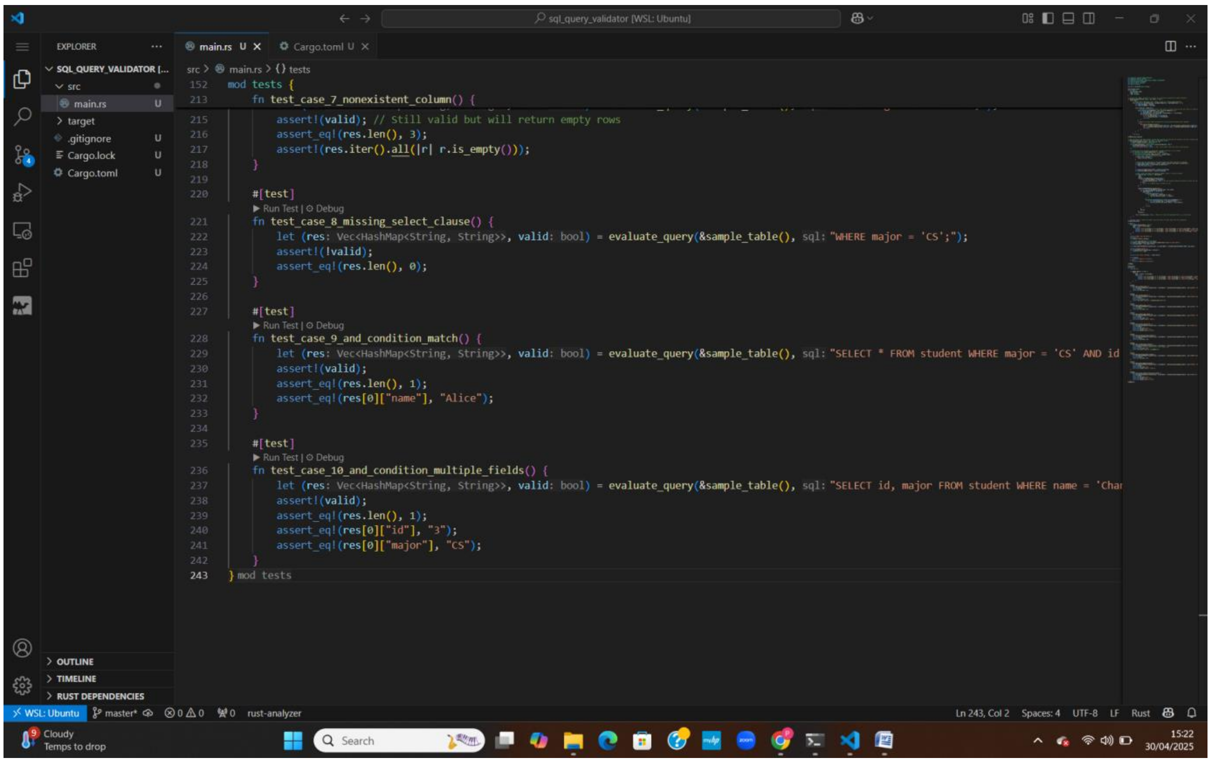
```rust
fn evaluate_query(table: &Table, sql: &str) -> (Vec<Row>, bool) {

    (vec![], false) // Return the empty result and false for query types that are unsupported
} fn evaluate_query

fn main() {
    let student_table: Table = Table {
        name: "student".to_string(),
        rows: vec![
            hashmap! {"id".to_string() => "1".to_string(), "name".to_string() => "Alice".to_string(), "major".to_string() => "CS".to_string()},
            hashmap! {"id".to_string() => "2".to_string(), "name".to_string() => "Bob".to_string(), "major".to_string() => "Math".to_string()},
            hashmap! {"id".to_string() => "3".to_string(), "name".to_string() => "Charlie".to_string(), "major".to_string() => "CS".to_string()},
        ],
    };

    println!("Enter your SQL query:");
    print!("> ");
    io::stdout().flush().unwrap();

    let mut sql_input: String = String::new();
    io::stdin().read_line(buf: &mut sql_input).expect(msg: "Failed to read input");
    let sql_input: &str = sql_input.trim();

    let (result: Vec<HashMap<String, String>>, is_valid: bool) = evaluate_query(&student_table, sql_input);

    println!("\nQuery Output:");
    for row: &HashMap<String, String> in &result {
        println!("{:?}", row);
    }

    println!("\n{} row(s) returned.", result.len());

    if is_valid {
        println!("\nQuery is correct");
    } else {
        println!("\nQuery is incorrect");
    }
} fn main
```



```rust
#[cfg(test)]
mod tests {
    use super::*;

    fn sample_table() -> Table {
        Table {
            name: "student".to_string(),
            rows: vec![
                hashmap! {"id".to_string() => "1".to_string(), "name".to_string() => "Alice".to_string(), "major".to_string() => "CS".to_string()},
                hashmap! {"id".to_string() => "2".to_string(), "name".to_string() => "Bob".to_string(), "major".to_string() => "Math".to_string()},
                hashmap! {"id".to_string() => "3".to_string(), "name".to_string() => "Charlie".to_string(), "major".to_string() => "CS".to_string()}
            ],
        }
    }

    #[test]
    fn test_case_1_select_star() {
        let (res: Vec<HashMap<String, String>>, valid: bool) = evaluate_query(&sample_table(), sql: "SELECT * FROM student;");
        assert!(valid);
        assert_eq!(res.len(), 3);
    }

    #[test]
    fn test_case_2_select_major() {
        let (res: Vec<HashMap<String, String>>, valid: bool) = evaluate_query(&sample_table(), sql: "SELECT major FROM student;");
        assert!(valid);
        assert_eq!(res.len(), 3);
        assert!(res.iter().all(|r| r.contains_key("major")));
    }

    #[test]
    fn test_case_3_where_major_cs() {
        let (res: Vec<HashMap<String, String>>, valid: bool) = evaluate_query(&sample_table(), sql: "SELECT * FROM student WHERE major = 'CS';");
        assert!(valid);
        assert_eq!(res.len(), 2);
    }

    #[test]
```

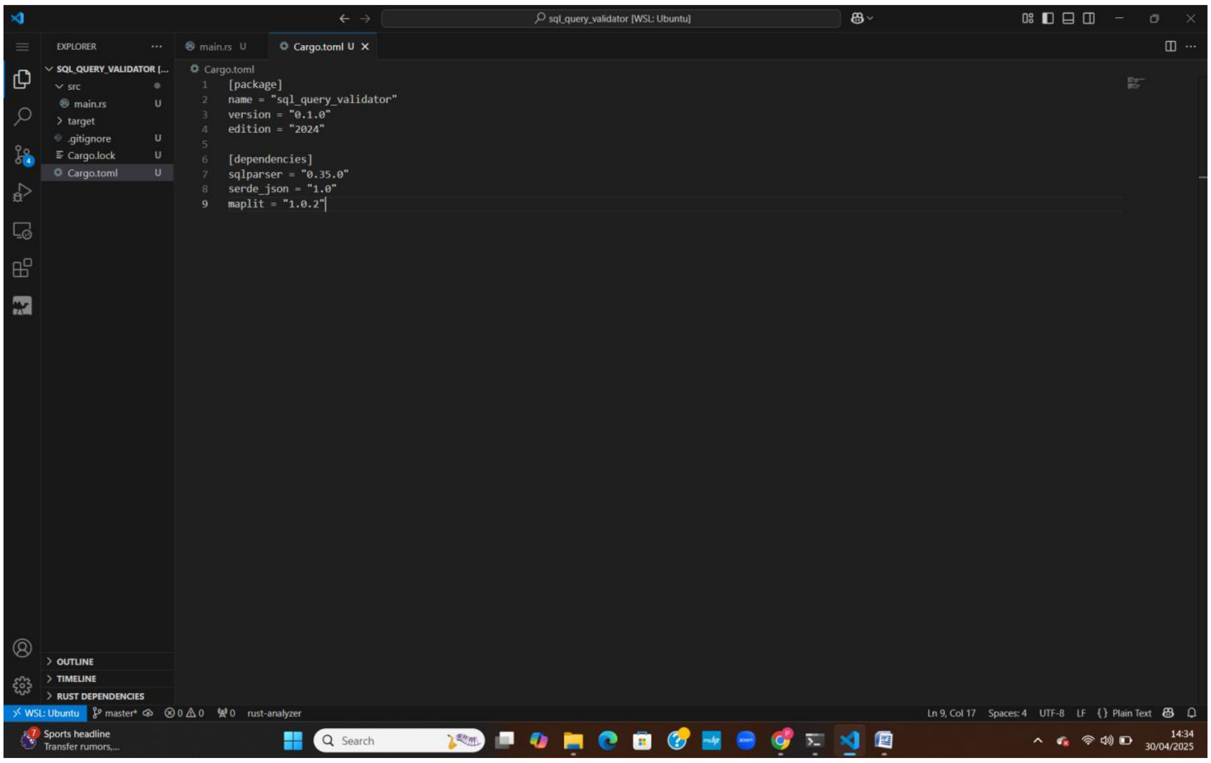


```rust
mod tests {
    }

    #[test]
    ▶ Run Test | ⚙ Debug
    fn test_case_4_where_major_math() {
        let (res: Vec<HashMap<String, String>>, valid: bool) = evaluate_query(&sample_table(), sql: "SELECT * FROM student WHERE major = 'Math';");
        assert!(valid);
        assert_eq!(res.len(), 1);
        assert_eq!(res[0]["name"], "Bob");
    }

    #[test]
    ▶ Run Test | ⚙ Debug
    fn test_case_5_where_name_alice() {
        let (res: Vec<HashMap<String, String>>, valid: bool) = evaluate_query(&sample_table(), sql: "SELECT id, major FROM student WHERE name = 'Alic
        assert!(valid);
        assert_eq!(res.len(), 1);
        assert_eq!(res[0]["id"], "1");
        assert_eq!(res[0]["major"], "CS");
    }

    #[test]
    ▶ Run Test | ⚙ Debug
    fn test_case_6_invalid_string_literal() {
        let (res: Vec<HashMap<String, String>>, valid: bool) = evaluate_query(&sample_table(), sql: "SELECT name WHERE major = Math;");
        assert!(!valid);
        assert_eq!(res.len(), 0);
    }

    #[test]
    ▶ Run Test | ⚙ Debug
    fn test_case_7_nonexistent_column() {
        let (res: Vec<HashMap<String, String>>, valid: bool) = evaluate_query(&sample_table(), sql: "SELECT age FROM student;");
        assert!(valid); // Still valid but will return empty rows
        assert_eq!(res.len(), 3);
        assert!(res.iter().all(|r| r.is_empty()));
    }

    #[test]
    ▶ Run Test | ⚙ Debug
    fn test_case_8_missing_select_clause() {
        let (res: Vec<HashMap<String, String>>, valid: bool) = evaluate_query(&sample_table(), sql: "WHERE major = 'CS';");
```



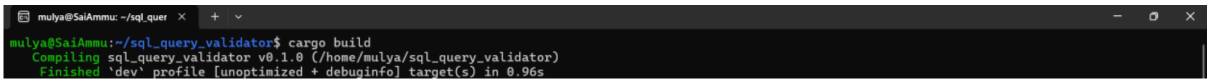
```rust
mod tests {
    fn test_case_7_nonexistent_column() {
        assert!(valid); // Still valid but will return empty rows
        assert_eq!(res.len(), 3);
        assert!(res.iter().all(|r| r.is_empty()));
    }

    #[test]
    ▶ Run Test | ⚙ Debug
    fn test_case_8_missing_select_clause() {
        let (res: Vec<HashMap<String, String>>, valid: bool) = evaluate_query(&sample_table(), sql: "WHERE major = 'CS';");
        assert!(!valid);
        assert_eq!(res.len(), 0);
    }

    #[test]
    ▶ Run Test | ⚙ Debug
    fn test_case_9_and_condition_match() {
        let (res: Vec<HashMap<String, String>>, valid: bool) = evaluate_query(&sample_table(), sql: "SELECT * FROM student WHERE major = 'CS' AND id
        assert!(valid);
        assert_eq!(res.len(), 1);
        assert_eq!(res[0]["name"], "Alice");
    }

    #[test]
    ▶ Run Test | ⚙ Debug
    fn test_case_10_and_condition_multiple_fields() {
        let (res: Vec<HashMap<String, String>>, valid: bool) = evaluate_query(&sample_table(), sql: "SELECT id, major FROM student WHERE name = 'Char
        assert!(valid);
        assert_eq!(res.len(), 1);
        assert_eq!(res[0]["id"], "3");
        assert_eq!(res[0]["major"], "CS");
    }
} mod tests
```

→ Next, I had opened the 'Cargo.toml' file to add required dependencies to run my code as below.



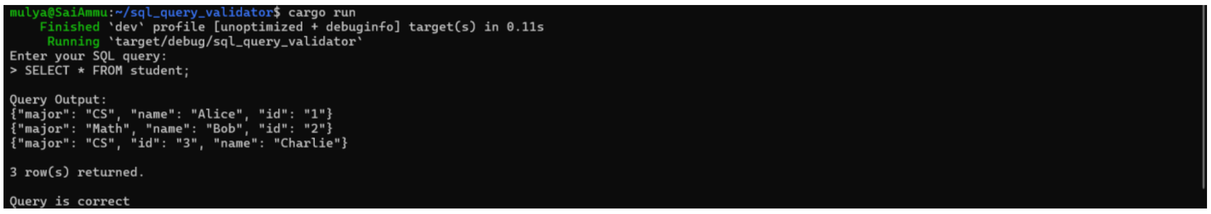
```
[package]
name = "sql_query_validator"
version = "0.1.0"
edition = "2024"

[dependencies]
sqlparser = "0.35.0"
serde_json = "1.0"
maplit = "1.0.2"
```

→ After, I had opened my Ubuntu command prompt and given the below command to compile my rust code that is given in VS code.



```
mulya@SaiAmmu:~/sql_query_validator$ cargo build
   Compiling sql_query_validator v0.1.0 (/home/mulya/sql_query_validator)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.96s
```

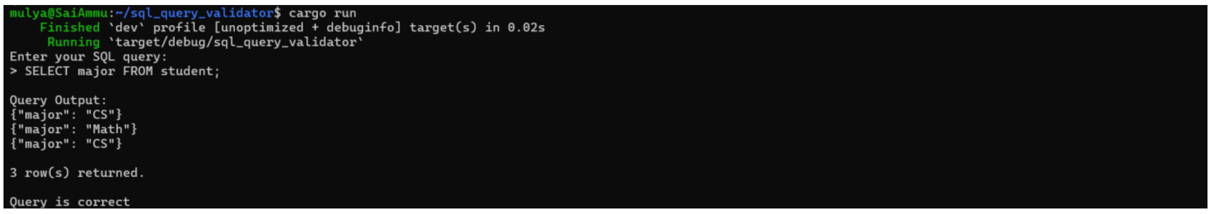
→ Next, I had given the below command to run my rust code to validate the SQL queries and I had entered a SQL query to test my code whether it is giving correct output or not by validating the SQL query.



```
mulya@SaiAmmu:~/sql_query_validator$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.11s
     Running `target/debug/sql_query_validator`
Enter your SQL query:
> SELECT * FROM student;

Query Output:
{"major": "CS", "name": "Alice", "id": "1"}
{"major": "Math", "name": "Bob", "id": "2"}
{"major": "CS", "id": "3", "name": "Charlie"}

3 row(s) returned.

Query is correct
```

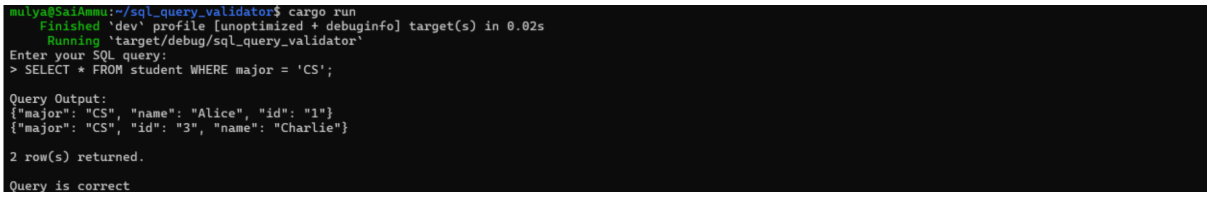
→ Next, I had given some more queries to check / validate whether my code will be passing other SQL queries which contains basic SELECT, WHERE and AND mainly and also made sure whether they are returning an output properly or not and whether the query is correct or not.

Test case 1:



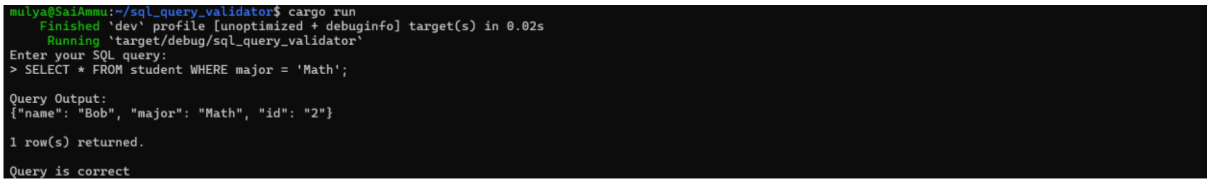
```
mulya@SaiAmmu:~/sql_query_validator$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/sql_query_validator`
Enter your SQL query:
> SELECT major FROM student;

Query Output:
{"major": "CS"}
{"major": "Math"}
{"major": "CS"}

3 row(s) returned.

Query is correct
```

Test case 2:



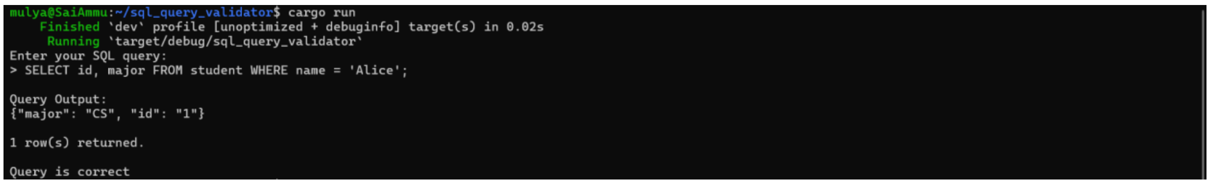
```
mulya@SaiAmmu:~/sql_query_validator$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/sql_query_validator`
Enter your SQL query:
> SELECT * FROM student WHERE major = 'CS';

Query Output:
{"major": "CS", "name": "Alice", "id": "1"}
{"major": "CS", "id": "3", "name": "Charlie"}

2 row(s) returned.

Query is correct
```

Test case 3:



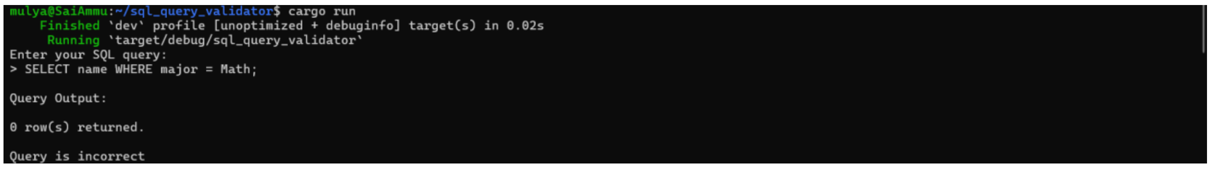
```
mulya@SaiAmmu:~/sql_query_validator$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/sql_query_validator`
Enter your SQL query:
> SELECT * FROM student WHERE major = 'Math';

Query Output:
{"name": "Bob", "major": "Math", "id": "2"}

1 row(s) returned.

Query is correct
```

Test case 4:



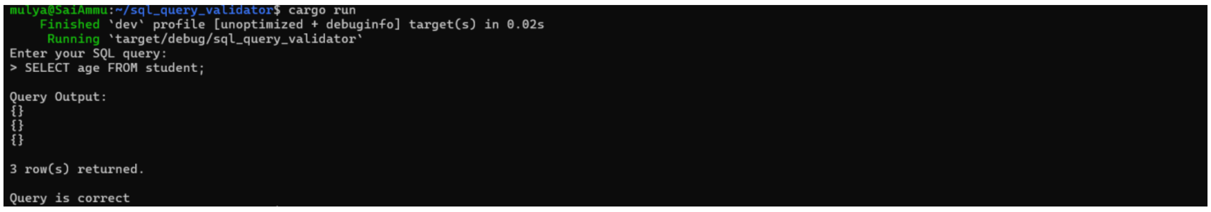
```
mulya@SaiAmmu:~/sql_query_validator$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/sql_query_validator`
Enter your SQL query:
> SELECT id, major FROM student WHERE name = 'Alice';

Query Output:
{"major": "CS", "id": "1"}

1 row(s) returned.

Query is correct
```

Test case 5:



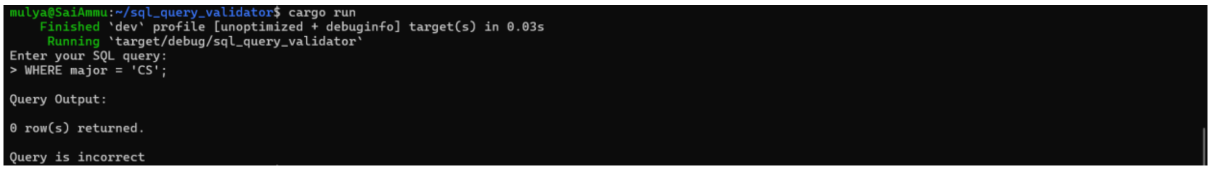
```
mulya@SaiAmmu:~/sql_query_validator$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/sql_query_validator`
Enter your SQL query:
> SELECT name WHERE major = Math;

Query Output:

0 row(s) returned.

Query is incorrect
```

Test case 6:



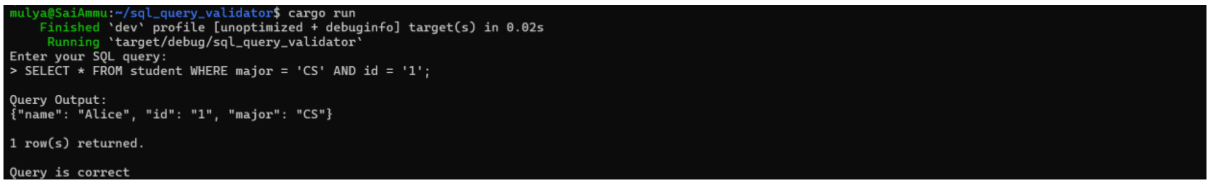
```
mulya@SaiAmmu:~/sql_query_validator$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/sql_query_validator`
Enter your SQL query:
> SELECT age FROM student;

Query Output:
{}
{}
{}

3 row(s) returned.

Query is correct
```

Test case 7:



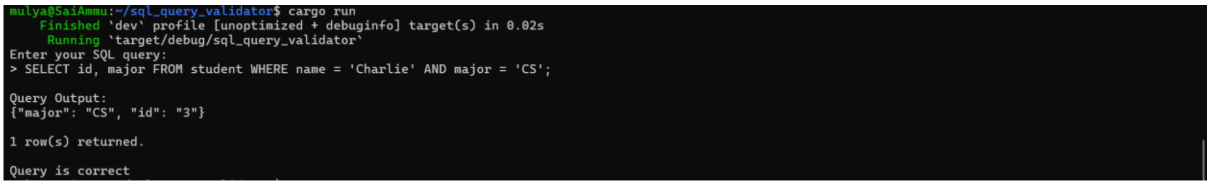
```
mulya@SaiAmmu:~/sql_query_validator$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.03s
     Running `target/debug/sql_query_validator`
Enter your SQL query:
> WHERE major = 'CS';

Query Output:

0 row(s) returned.

Query is incorrect
```

Test case 8:



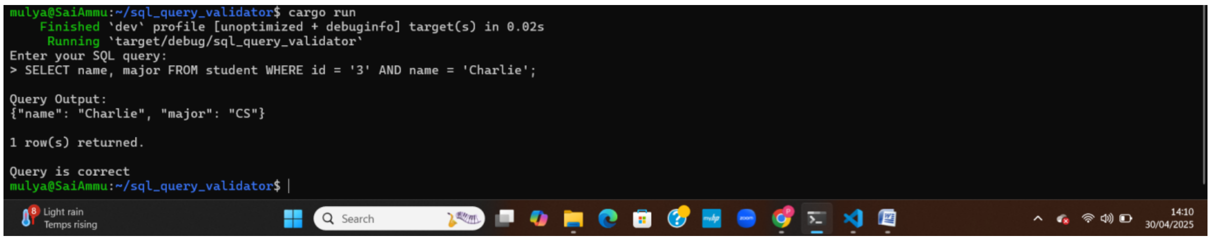
```
mulya@SaiAmmu:~/sql_query_validator$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/sql_query_validator`
Enter your SQL query:
> SELECT * FROM student WHERE major = 'CS' AND id = '1';

Query Output:
{"name": "Alice", "id": "1", "major": "CS"}

1 row(s) returned.

Query is correct
```

Test case 9:



```
mulya@SaiAmmu:~/sql_query_validator$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/sql_query_validator`
Enter your SQL query:
> SELECT id, major FROM student WHERE name = 'Charlie' AND major = 'CS';

Query Output:
{"major": "CS", "id": "3"}

1 row(s) returned.

Query is correct
```

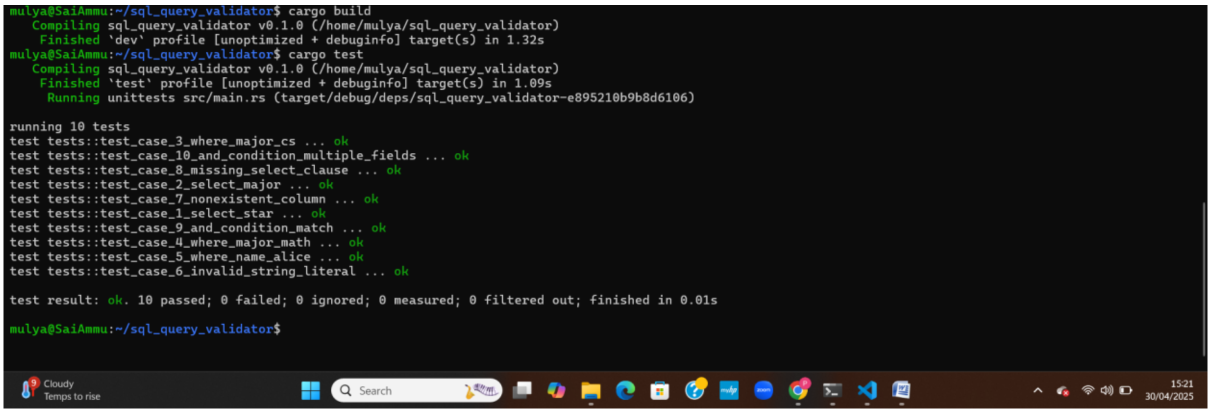
Test case 10:



```
mulya@SaiAmmu:~/sql_query_validator$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/sql_query_validator`
Enter your SQL query:
> SELECT name, major FROM student WHERE id = '3' AND name = 'Charlie';

Query Output:
{"name": "Charlie", "major": "CS"}

1 row(s) returned.

Query is correct
mulya@SaiAmmu:~/sql_query_validator$ |
```

→ Next, I had given 'unit tests' in my code for the SQL queries that I need to validate and all the unit tests are passed as below by giving the command:

'cargo test'



```
mulya@SaiAmmu:~/sql_query_validator$ cargo build
   Compiling sql_query_validator v0.1.0 (/home/mulya/sql_query_validator)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 1.32s
mulya@SaiAmmu:~/sql_query_validator$ cargo test
   Compiling sql_query_validator v0.1.0 (/home/mulya/sql_query_validator)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 1.09s
     Running unittests src/main.rs (target/debug/deps/sql_query_validator-e895210b9b8d6106)

running 10 tests
test tests::test_case_3_where_major_cs ... ok
test tests::test_case_10_and_condition_multiple_fields ... ok
test tests::test_case_8_missing_select_clause ... ok
test tests::test_case_2_select_major ... ok
test tests::test_case_7_nonexistent_column ... ok
test tests::test_case_1_select_star ... ok
test tests::test_case_9_and_condition_match ... ok
test tests::test_case_4_where_major_math ... ok
test tests::test_case_5_where_name_alice ... ok
test tests::test_case_6_invalid_string_literal ... ok

test result: ok. 10 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s

mulya@SaiAmmu:~/sql_query_validator$
```