Implementation of a Basic RDBMS in OCaml

Mayank Sharma IMT2021086, Aasmaan Gupta IMT2021006,
Nimish G.S IMT2021077, Rithik Bansal IMT2021099

Overview of Basic Functionalities (Key Functions)

List the main operations supported-

1) Create Table

2) Drop Table

3) Add Column and row

4) Filter Rows and Update Rows

5) Delete Rows

6) Join Tables

Type Definitions

Type Definitions

1) **SQL Column Types**

type colEntryType = INT | CHAR of int;;

- `INT`: Represents an integer.
- `CHAR of int`: Represents a character array/string with a specified length.

2) **Relational Operators**

type operatorType = OperatorEqual | OperatorNotEqual | OperatorLessThan | OperatorGreaterThan | OperatorGreaterEqual | OperatorLessEqual;;

OperatorEqual: Equality, OperatorNotEqual: Inequality ,OperatorLessThan: Less than, OperatorGreaterThan: Greater than OperatorGreaterEqual: Greater than or equal to ,OperatorLessEqual: Less than or equal to

Type Definitions

3) **Column Value Types**

type colEntryValueType = Int of int | Char of string;;

Variant type to store values of columns:

- `Int of int`: Stores integer values.
- `Char of string`: Stores string values.

4) **Column Specification**

type colSpecification = {colName : string; colEntryType : colEntryType};;

Defines a record for storing column definitions:

- `colName`: Name of the column.
- `colEntryType`: Data type of the column as defined by `colEntryType`.

Type Definitions

5) **Column Data**

type colData = {dataColName : string; value : colEntryValueType};;

Record type to store values for a given column in a table:

- `dataColName`: The name of the column.
- `value`: The value of the column, defined by `colEntryValueType`.

6) Row Entry

type singleRowEntry = {data : colData list};;

Models a row in a table as a list of `colData` records, where each `colData` represents a column with its value.

Type Definitions

7) Table Structure

type table = {tableName : string; coldeflist : colSpecification list; rowEntries : singleRowEntry list};;

- `tableName`: Name of the table.
- `coldeflist`: List of column specifications.
- `rowEntries`: List of rows, each row containing data for the columns.

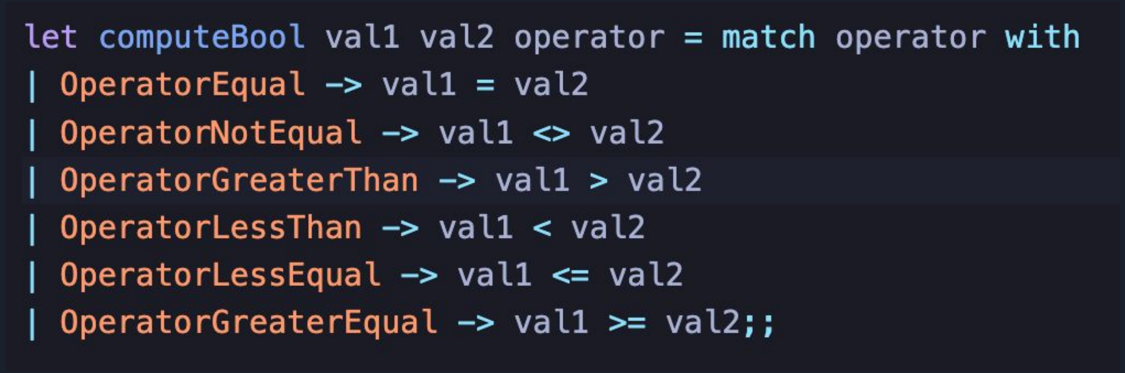
8) Database Structure

type dbms = {dbmsName : string; tableRecord : table list};;

- `dbmsName`: Name of the database.
- `tableRecord`: List of tables contained in the database.

Functions

Functions

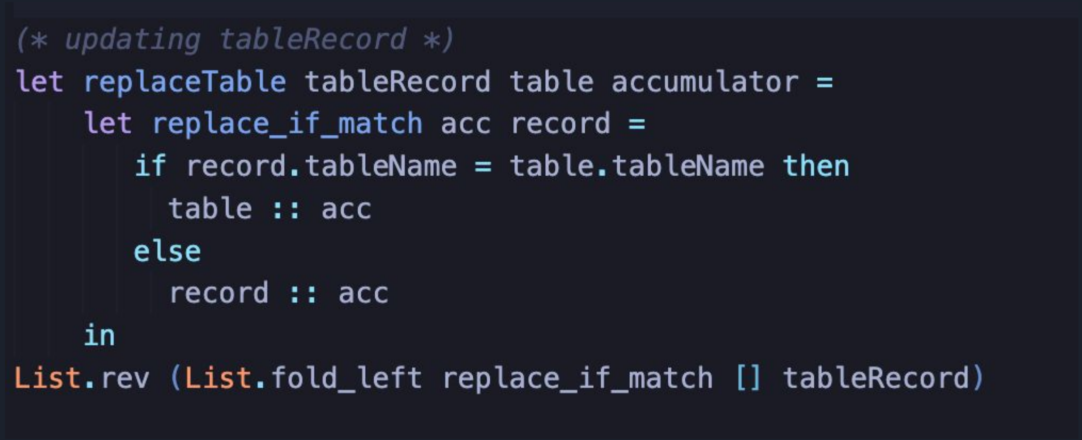


```
let computeBool val1 val2 operator = match operator with
| OperatorEqual -> val1 = val2
| OperatorNotEqual -> val1 <> val2
| OperatorGreaterThan -> val1 > val2
| OperatorLessThan -> val1 < val2
| OperatorLessEqual -> val1 <= val2
| OperatorGreaterEqual -> val1 >= val2;;
```

val1, val2: The values to be compared.

operator: A variant from operatorType that defines the type of comparison.

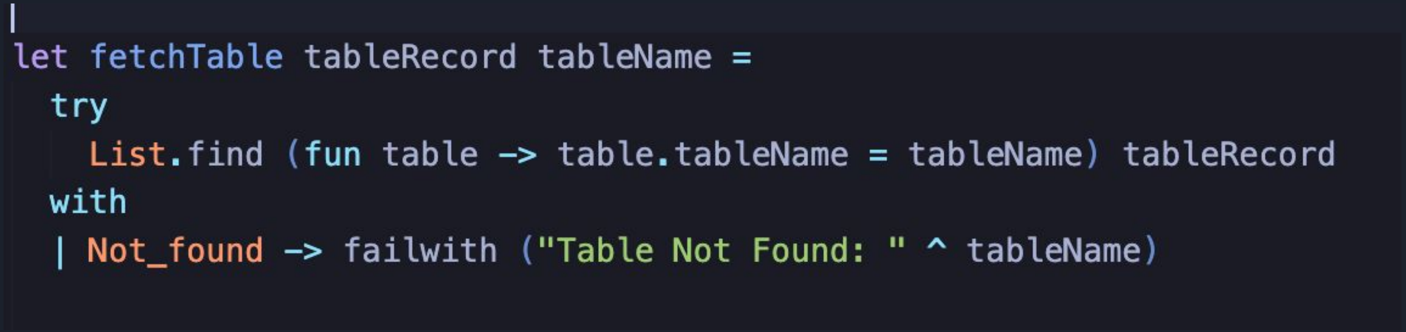
Functions



```
(* updating tableRecord *)
let replaceTable tableRecord table accumulator =
    let replace_if_match acc record =
       if record.tableName = table.tableName then
         table :: acc
       else
         record :: acc
    in
List.rev (List.fold_left replace_if_match [] tableRecord)
```

Updates or replaces a table within the database's table record based on matching table names.

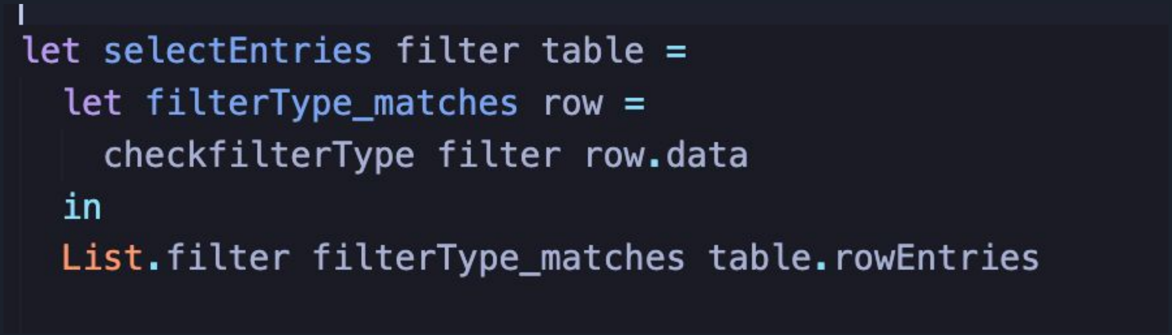
Functions



```
let fetchTable tableRecord tableName =
  try
    List.find (fun table -> table.tableName = tableName) tableRecord
  with
  | Not_found -> failwith ("Table Not Found: " ^ tableName)
```

Fetches a table from the database using the table name. It throws an error if the table is not found, ensuring the operations are performed on existing tables only.

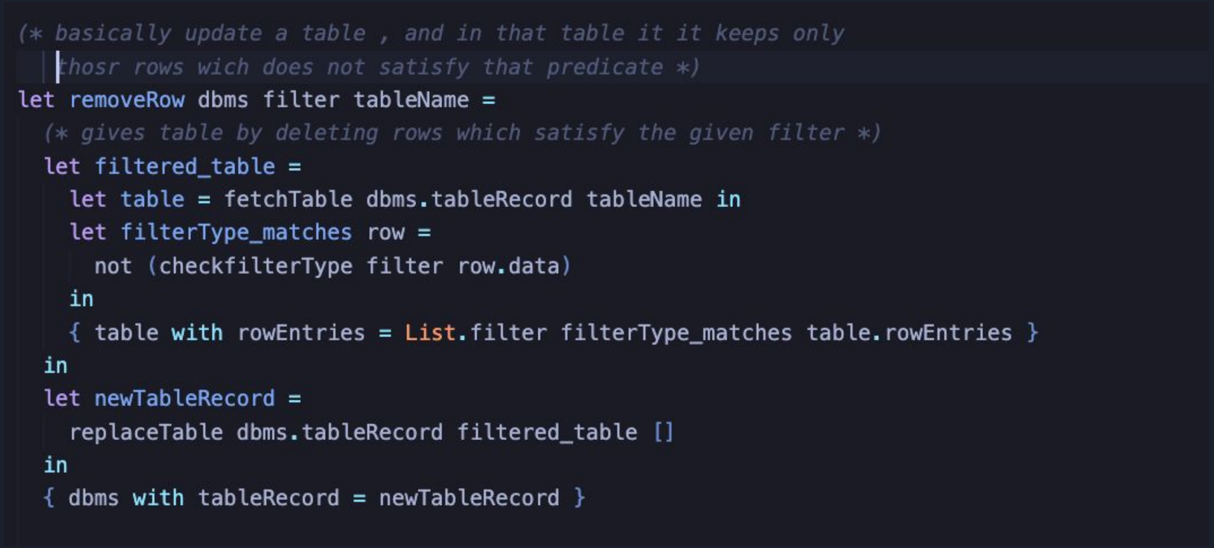
Functions



```
let selectEntries filter table =
  let filterType_matches row =
    checkfilterType filter row.data
  in
  List.filter filterType_matches table.rowEntries
```

Applies a given filter to all rows in a table, returning only those rows that meet the filter conditions.

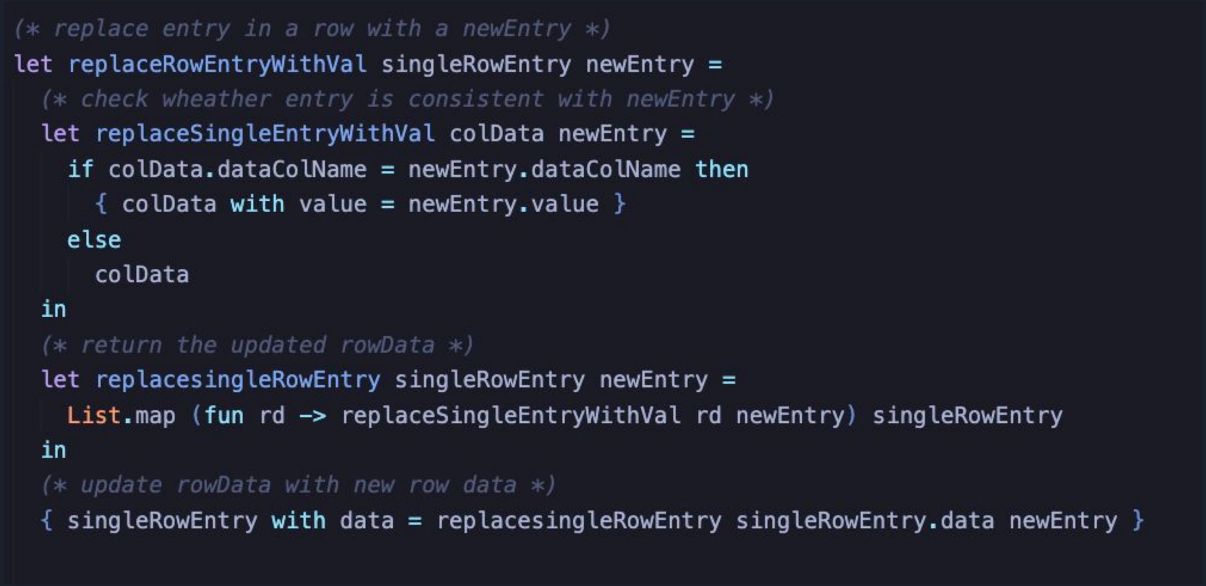
Functions



```
(* basically update a table , and in that table it it keeps only
   thosr rows wich does not satisfy that predicate *)
let removeRow dbms filter tableName =
  (* gives table by deleting rows which satisfy the given filter *)
  let filtered_table =
    let table = fetchTable dbms.tableRecord tableName in
    let filterType_matches row =
      not (checkfilterType filter row.data)
    in
    { table with rowEntries = List.filter filterType_matches table.rowEntries }
  in
  let newTableRecord =
    replaceTable dbms.tableRecord filtered_table []
  in
  { dbms with tableRecord = newTableRecord }
```

Removes rows from a specified table that satisfy a given condition (filter). This operation is akin to the SQL DELETE statement but is conditional based on the specified filter.

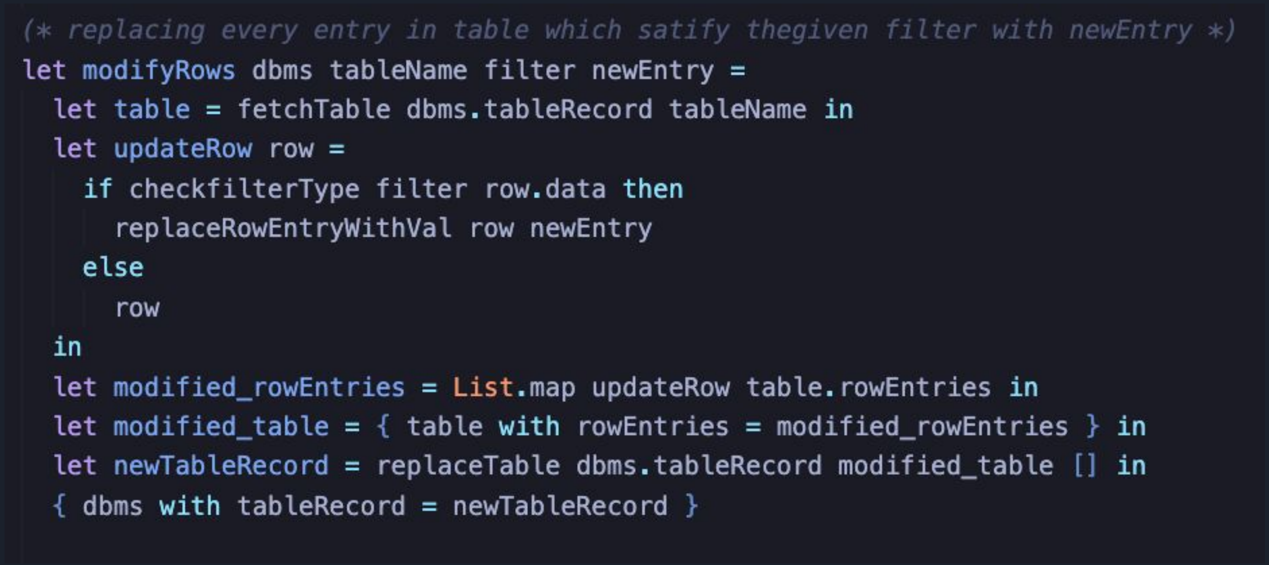
Functions



```
(* replace entry in a row with a newEntry *)
let replaceRowEntryWithVal singleRowEntry newEntry =
  (* check wheather entry is consistent with newEntry *)
  let replaceSingleEntryWithVal colData newEntry =
    if colData.dataColName = newEntry.dataColName then
      { colData with value = newEntry.value }
    else
      colData
  in
  (* return the updated rowData *)
  let replacesingleRowEntry singleRowEntry newEntry =
    List.map (fun rd -> replaceSingleEntryWithVal rd newEntry) singleRowEntry
  in
  (* update rowData with new row data *)
  { singleRowEntry with data = replacesingleRowEntry singleRowEntry.data newEntry }
```

Updates specific entries within a row, based on matching column names. This is useful for updating specific fields within records.

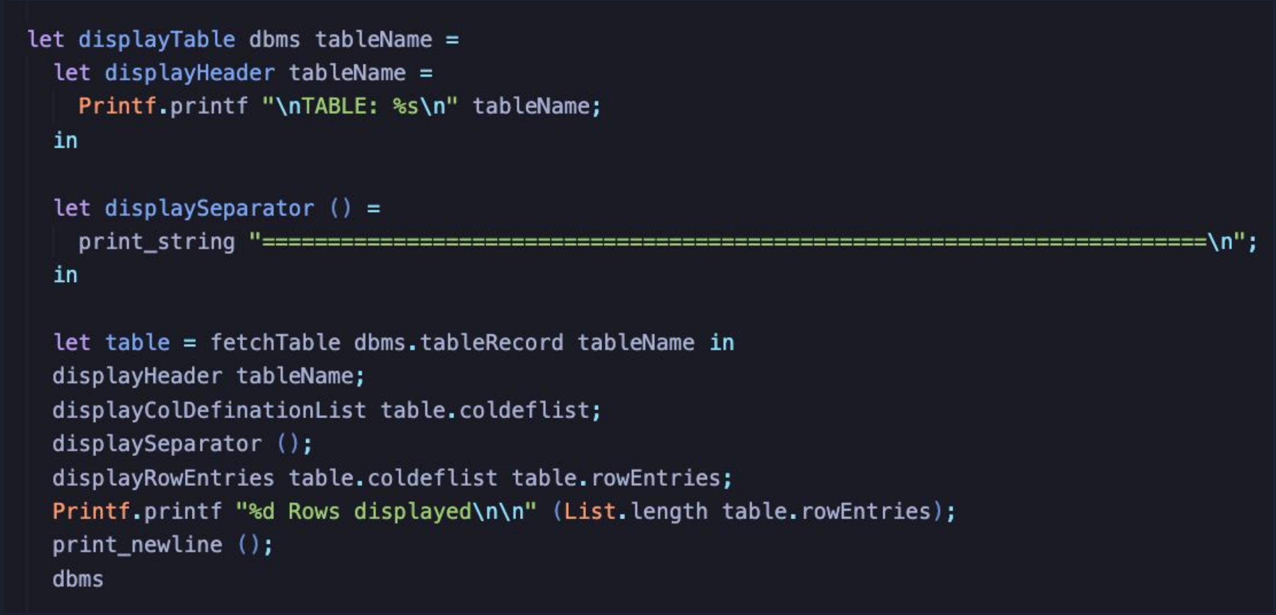
Functions



```
(* replacing every entry in table which satify thegiven filter with newEntry *)
let modifyRows dbms tableName filter newEntry =
  let table = fetchTable dbms.tableRecord tableName in
  let updateRow row =
    if checkfilterType filter row.data then
      replaceRowEntryWithVal row newEntry
    else
      row
  in
  let modified_rowEntries = List.map updateRow table.rowEntries in
  let modified_table = { table with rowEntries = modified_rowEntries } in
  let newTableRecord = replaceTable dbms.tableRecord modified_table [] in
  { dbms with tableRecord = newTableRecord }
```

Applies updates to all rows in a table that meet specific conditions. This function modifies rows in bulk, using a filter to identify rows that need updating and applying the changes as specified by newEntry.

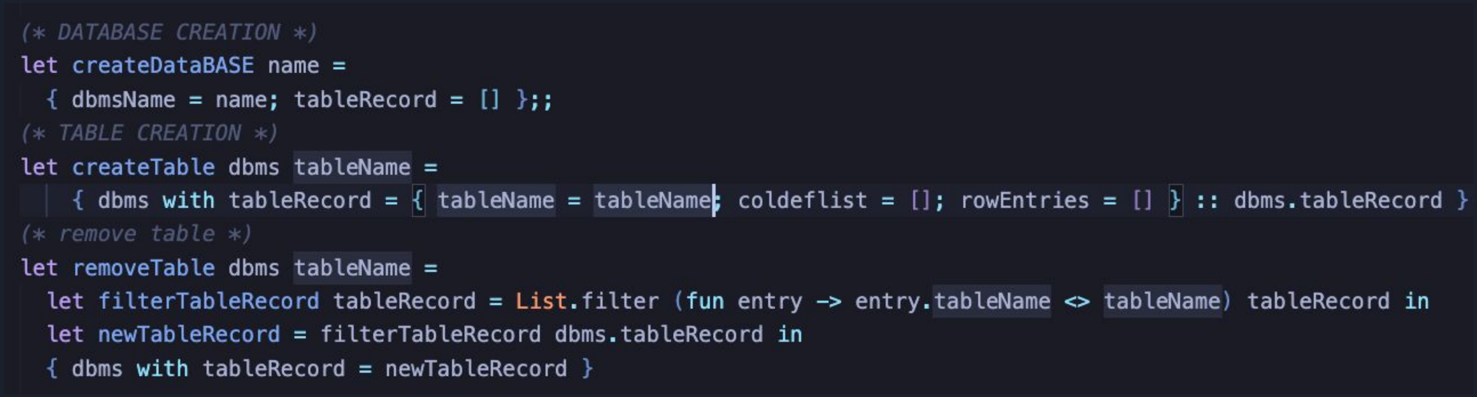
Functions



```
let displayTable dbms tableName =
  let displayHeader tableName =
    Printf.printf "\nTABLE: %s\n" tableName;
  in

  let displaySeparator () =
    print_string "========================================================================\n";
  in

  let table = fetchTable dbms.tableRecord tableName in
  displayHeader tableName;
  displayColDefinationList table.coldeflist;
  displaySeparator ();
  displayRowEntries table.coldeflist table.rowEntries;
  Printf.printf "%d Rows displayed\n\n" (List.length table.rowEntries);
  print_newline ();
  dbms
```

Visualizing Tables and Their Data

Functions



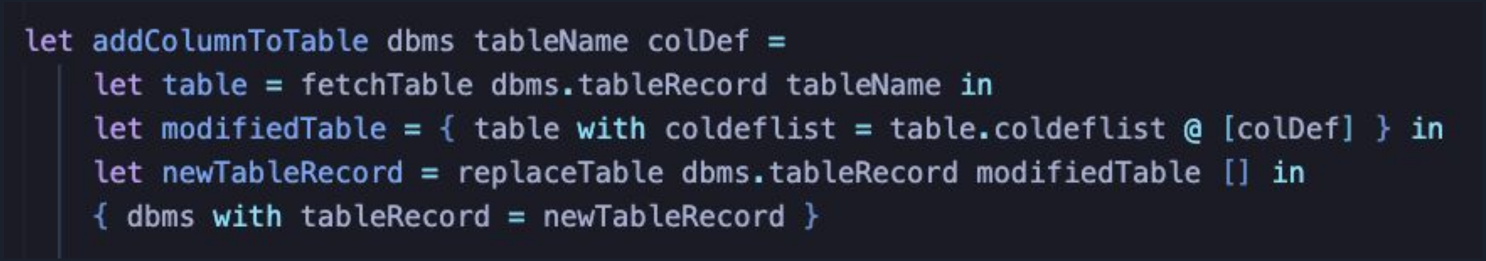
```
(* DATABASE CREATION *)
let createDataBASE name =
  { dbmsName = name; tableRecord = [] };;
(* TABLE CREATION *)
let createTable dbms tableName =
    { dbms with tableRecord = { tableName = tableName; coldeflist = []; rowEntries = [] } :: dbms.tableRecord }
(* remove table *)
let removeTable dbms tableName =
  let filterTableRecord tableRecord = List.filter (fun entry -> entry.tableName <> tableName) tableRecord in
  let newTableRecord = filterTableRecord dbms.tableRecord in
  { dbms with tableRecord = newTableRecord }
```

first function initializes a new database with a specified name.

Second function adds a new table with the specified name to an existing database.

Third function removes a specified table from the database.

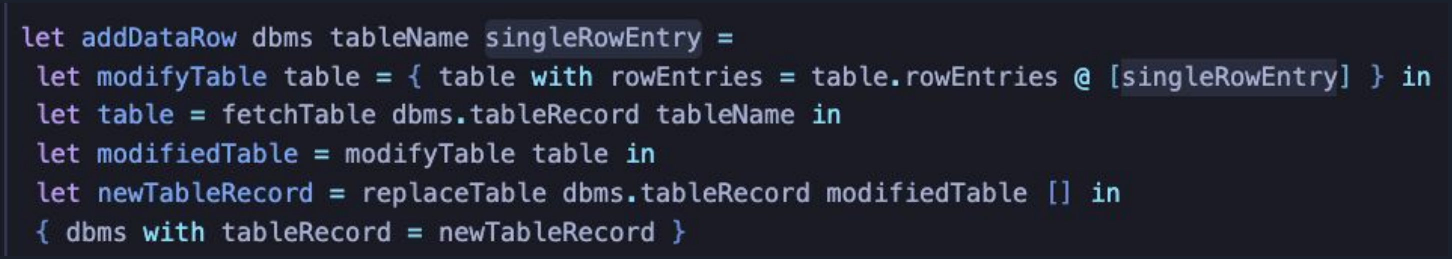
Functions



```
let addColumnToTable dbms tableName colDef =
    let table = fetchTable dbms.tableRecord tableName in
    let modifiedTable = { table with coldeflist = table.coldeflist @ [colDef] } in
    let newTableRecord = replaceTable dbms.tableRecord modifiedTable [] in
    { dbms with tableRecord = newTableRecord }
```

This function adds a new column to an existing table within the database

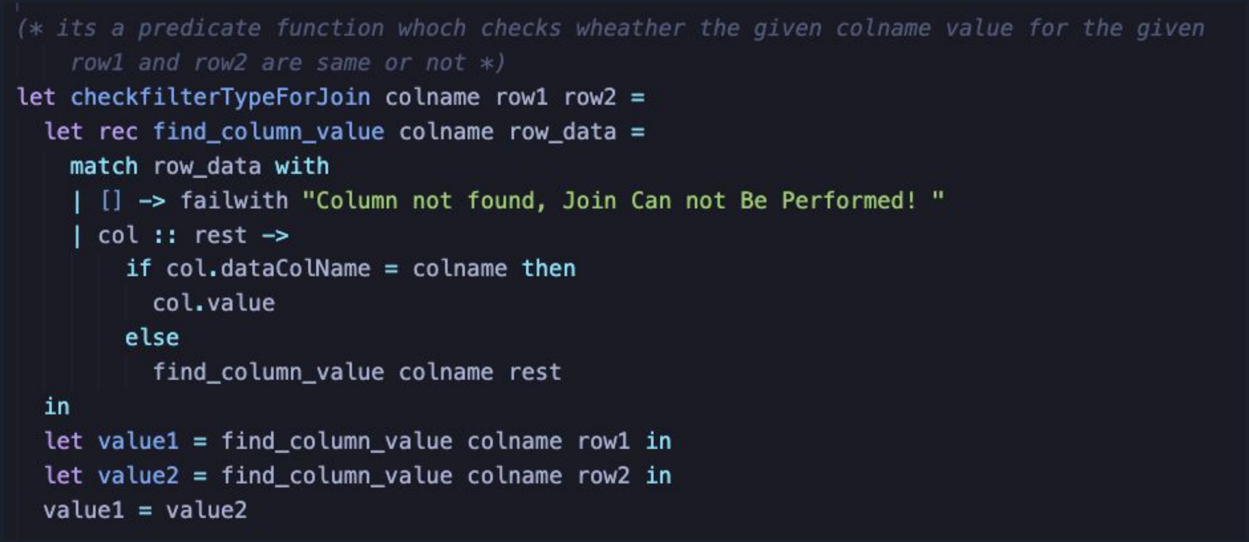
Functions



```
let addDataRow dbms tableName singleRowEntry =
 let modifyTable table = { table with rowEntries = table.rowEntries @ [singleRowEntry] } in
 let table = fetchTable dbms.tableRecord tableName in
 let modifiedTable = modifyTable table in
 let newTableRecord = replaceTable dbms.tableRecord modifiedTable [] in
 { dbms with tableRecord = newTableRecord }
```

Appends a new row of data to a specified table

Functions

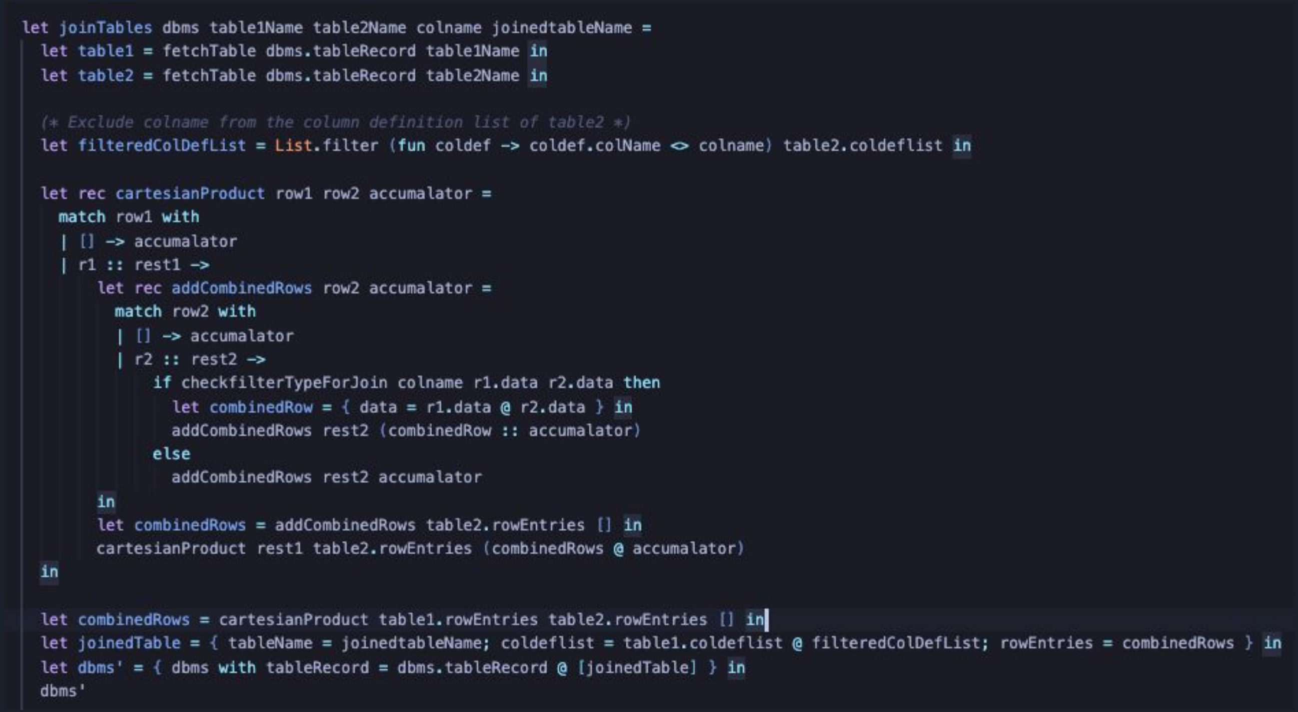


```
(* its a predicate function whoch checks wheather the given colname value for the given
    row1 and row2 are same or not *)
let checkfilterTypeForJoin colname row1 row2 =
  let rec find_column_value colname row_data =
    match row_data with
    | [] -> failwith "Column not found, Join Can not Be Performed! "
    | col :: rest ->
        if col.dataColName = colname then
          col.value
        else
          find_column_value colname rest
  in
  let value1 = find_column_value colname row1 in
  let value2 = find_column_value colname row2 in
  value1 = value2
```

Validates Join Conditions Between Rows

Determines whether two rows from different tables can be joined based on the value of a specified column..

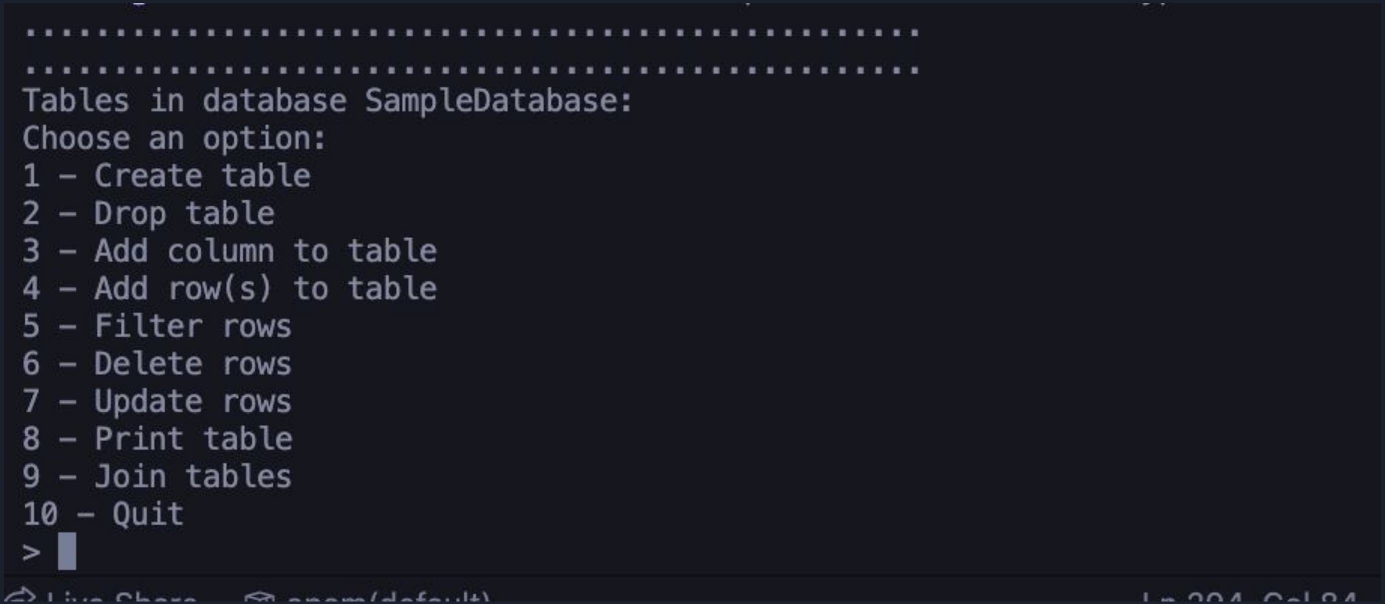
Functions



```
let joinTables dbms table1Name table2Name colname joinedtableName =
  let table1 = fetchTable dbms.tableRecord table1Name in
  let table2 = fetchTable dbms.tableRecord table2Name in

  (* Exclude colname from the column definition list of table2 *)
  let filteredColDefList = List.filter (fun coldef -> coldef.colName <> colname) table2.coldeflist in

  let rec cartesianProduct row1 row2 accumalator =
    match row1 with
    | [] -> accumalator
    | r1 :: rest1 ->
        let rec addCombinedRows row2 accumalator =
          match row2 with
          | [] -> accumalator
          | r2 :: rest2 ->
              if checkfilterTypeForJoin colname r1.data r2.data then
                let combinedRow = { data = r1.data @ r2.data } in
                addCombinedRows rest2 (combinedRow :: accumalator)
              else
                addCombinedRows rest2 accumalator
        in
        let combinedRows = addCombinedRows table2.rowEntries [] in
        cartesianProduct rest1 table2.rowEntries (combinedRows @ accumalator)
  in

  let combinedRows = cartesianProduct table1.rowEntries table2.rowEntries [] in
  let joinedTable = { tableName = joinedtableName; coldeflist = table1.coldeflist @ filteredColDefList; rowEntries = combinedRows } in
  let dbms' = { dbms with tableRecord = dbms.tableRecord @ [joinedTable] } in
  dbms'
```

Executes a join operation between two tables on a specified column, creating a new table that combines rows from both tables where the join condition is met.

Command Line Based Interface



```
..................................................
..................................................
Tables in database SampleDatabase:
Choose an option:
1 - Create table
2 - Drop table
3 - Add column to table
4 - Add row(s) to table
5 - Filter rows
6 - Delete rows
7 - Update rows
8 - Print table
9 - Join tables
10 - Quit
> 
```

Demo

Key Takeaways

Key Takeaways

Functional Programming Strengths (Higher-Order Functions Utilizes higher-order functions extensively to manipulate data structures)

Type Safety (Strong Typing in OCaml, explicit type definitions for columns, rows, tables, and databases, prevent errors like mismatched data types and operations on undefined table structures)

Modular Design (Reusability and Extensibility)

Understanding OCaml's Capabilities (demonstrating its utility in real-world applications.)

Challenges Faced and How we dealt with them

1) State Management in Immutable Context -

OCaml's immutable data structures initially pose challenges for tasks typically associated with mutable states, like updating a database.

The solution lies in using functional patterns such as returning new instances of data with the required modifications, as demonstrated in the project where each update or deletion operation returns a new modified state of the database.

2) Debugging and Testing

Thank You