CS 754: Programming Language April 16, 2024 PROJECT REPORT

Instructor: Prof. SUJIT KUMAR CHAKRABARTI Mayank Sharma IMT2021086, Aasmaan Gupta IM2021006, Nimish G.S IMT2021077, Rithik Bansal IMT2021099

1 Problem Description: Core Learning Objectives and Outcomes

This project aims to develop an in-memory Relational Database Management System (RDBMS) using a functional programming approach with OCaml. The goal is to implement the foundational features of traditional RDBMS but with a unique twist by leveraging OCaml's rich type system and functional programming paradigms. This approach provides a distinct learning platform for understanding both database management systems and advanced functional programming concepts. Here are the core learning objectives and outcomes envisaged from this project:

1.1 Objectives

The project is designed around several key learning objectives:

1. **Understanding RDBMS Architecture:** We will explore how relational databases utilize tables, rows, and columns to manage data efficiently and how SQL operations such as CREATE, READ (SELECT), UPDATE, and DELETE functions within an RDBMS.
2. **OCaml Programming Proficiency:** The project aims to enhance skills in OCaml, especially for systems programming and data management, including mastering data types, modules, and pattern matching.
3. **Implementing Functional Programming Principles:** The project will apply functional programming principles to typical database operations, utilizing high-order functions for data manipulation tasks.

1.2 Outcomes

Upon completion of the project, the following outcomes are expected:

1. **Functional RDBMS Prototype:** A fully functional RDBMS prototype in OCaml capable of handling basic SQL queries and demonstrating effective in-memory data management.
2. **Comprehensive Documentation:** Detailed documentation that covers system design, implementation details, and usage guidelines, serving as an instructional material for future reference.

2 Solution Outline

This section provides a detailed overview of the current deliverables of our RDBMS project, focusing on the functionalities that have been implemented as reflected in the provided CommandLine.ml and DefTypeAndFunctions.ml files. The key deliverables of the project encompass a sophisticated codebase with well-integrated algorithmic solutions and operational frameworks designed to enhance the efficiency and functionality of our database management system.

- Codebase Core Database Engine: Our current codebase incorporates essential components of the database engine. This includes: Data storage mechanisms capable of handling structured data types and supporting basic data operations such as insert, delete, and update. Indexing mechanisms that optimize data retrieval, enhancing performance for read-heavy operations.

- Interface Modules: We have developed a basic command-line interface that allows users to interact directly with the database. This interface supports straightforward SQL-like commands for manipulating and querying the database.

- Documentation: Our project includes comprehensive documentation that details the operational use and structural configuration of the database, aiming to assist developers and users in navigating and utilizing the system effectively.

- Query Optimization: Basic query optimization is implemented to streamline the execution of queries by simplifying operations and reducing computational overhead. This is achieved through straightforward interpretation and execution strategies outlined in the CommandLine.ml and DefTypeAndFunctions.ml files.

- Join Operations: We support basic join operations (e.g., inner joins), allowing for the combination of data from multiple tables based on related columns. This capability is essential for performing relational database operations where data residing in different tables needs to be correlated and analyzed together.

3 Team's Progress

3.1 Implementation Overview

Our team has successfully implemented a series of foundational functionalities in our in-memory RDBMS project, focusing on establishing a robust framework for database and table management. The core of our development has included the definition of necessary data structures and the creation of several critical operational functions.

3.1.1 Data Type Definitions

We have defined essential data structures to accurately represent various components of a relational database:

- Custom types for column data representation, supporting both integer and character data.
- Structured records for columns, rows, and tables that include all necessary details such as names, types, and the actual data.
- A comprehensive type to encapsulate the entire database with its constituent tables.

I will now explain all the datatypes we have used.

- **type operatorType**: Defines a new type named **operatorType** that represents various relational (comparison) operators commonly used in SQL queries. Each **|** defines a variant of this type, representing different SQL operators such as less than, greater than, equal, not equal, etc. These are used to construct conditions for queries within our database.

- type **columnEntrycontentType**: Defines a variant type that can store data in a column as either a string (Char of string) or an integer (Int of int). This allows columns to support different data types. Used to store data in a column.

- type **joinType**: Defines types of SQL joins. This type supports three kinds of joins: inner, left, and right, essential for relational database queries that involve combining rows from two or more tables.

- type **columnEntryType**: Defines the types of columns that can exist in a table. CHAR of int indicates a character type with a specified length, and INT represents an integer type. Used to define a column.

- type **columnSpecification**: Defines a record type for column specifications in a table, including the type of data the column stores (**columnEntryType**) and its name (**colName**).

- type **transactionStatus**: Represents the status of a database transaction, which can be either pending, committed, or rolled back. This helps manage the state of transactions within our database system.

- type **columnContent**: Defines a record type for storing data content of a column along with its title. This structure is used to hold the actual data (content) and associate it with a column.

- type **filterType** Defines a record for a filter used in database queries. It includes an operator (**filterTypeOperator**) and the data (**filterTypeData**) to which the operator is applied. This is crucial for implementing conditional logic in queries

- type **constraintType**: Defines various constraints that can be applied to columns in a database. These include primary keys, foreign keys, unique constraints, and not-null constraints. These are essential for ensuring data integrity and relational integrity between tables.

- type **singleRowEntry**: Defines a record type for a row in a table, which is composed of a list of **columnContent** records. This represents one row of data in a table.

- type table: Defines a record type for a table in the database, including a list of rows (**rowEntries**), a list of column specifications (**specifyColumn**), and the title of the table (**tbTitle**).

- type **dataBase**: Defines a record type for the entire database, including a list of tables (**dataBaseTableRecord**) and a title for the database (**dataBaseTitle**). This type encapsulates the entire database, organizing it as a collection of tables, each with its own data and schema. These definitions form the foundation of your RDBMS, defining how data is structured and manipulated. Each type and structure is critical for the proper functioning of your database, from storing data to querying and maintaining data integrity.

3.1.2 Helper Functions

A series of helper functions have been meticulously designed to enhance the manipulation and querying capabilities of the database. Key among these are functions that efficiently locate and retrieve tables by name, which are pivotal in ensuring that operations such as additions, deletions, and updates target the correct entities, thus maintaining data integrity. Additionally, to manage the complexity of updates within the immutable structure of OCaml, where function shadowing can occur, functions are crafted to replace old table values with new ones dynamically. This approach avoids the common issue of function shadowing in OCaml by managing the state transitions of tables explicitly, thereby ensuring that changes are made to the latest version of the table list without altering the database's inherent structural integrity. The helper functions within the RDBMS are designed to facilitate various database operations.

1. The **computeBool** function determines the truth value of comparisons between two values based on a specified operator such as equal, not equal, greater than, etc., and is commonly used in query filtering operations to evaluate conditions.

2. The **invertfilterType** function creates a new filter with the logical inverse of the operator specified in the input filter, useful for negating current filter conditions in queries.

3. The **replaceTable** function updates or adds a table in the database's table list with an updated or new table, essential for operations that modify table structures or contents.

4. The **fetchTable** function retrieves a table by its title from the database, critical for operations requiring interaction with specific tables.

5. The **checkfilterType** function checks if any entry in a row satisfies a specified filter condition, used in data selection processes where rows need to be filtered based on certain conditions.

6. Lastly, the **selectEntries** function filters and returns all rows from a table that meet a specific filter condition, utilized for querying data in SELECT operations where specific rows need to be isolated based on query conditions. These functions integrate seamlessly with the database's architecture, using the defined types and structures to maintain consistent and reliable operations.

3.1.3 Database and Table Creation

3.1.3.1 Initializing a New Database Instance

- Mechanism: A new database instance is initiated with an entirely empty set of tables. This functionality is crucial for starting a database management system from scratch, ensuring that there are no pre-existing tables or data that could complicate the initial setup.

- Implementation: The initialization is handled by the 'constructDataBase' function, which takes a name for the database and creates a new database record with an empty list of tables. Here's how the function typically looks:

- ***let constructDataBase name =***
  { dataBaseTitle = name; dataBaseTableRecord = [] };

- dataBaseTitle: Set to the input 'name', this field specifies the name of the newly created database.

- dataBaseTableRecord: Initialized as an empty list, indicating that the database starts with no tables. This approach adheres to functional programming principles where immutability is preferred, allowing the database structure to be explicitly defined and controlled.

3.1.3.2 Adding New Tables to the Database

- Mechanism: New tables can be added to the database, starting with an empty schema and containing no pre-existing data. This capability is essential for dynamically expanding the database as new data requirements arise.

- Implementation: The addition of new tables is managed through the 'constructTable' function. This function modifies the existing database by appending a new table with a specified title and initially empty columns and rows. Here's a simplified version of the function:

- ***let constructTable dataBase tbTitle =***
  { dataBase with dataBaseTableRecord = { tbTitle = tbTitle; specifyColumn = []; rowEntries = [] } :: dataBase.dataBaseTableRecord };
- tbTitle: This parameter specifies the name of the new table.

- specifyColumn: Initialized as an empty list, it indicates that the table starts with no predefined column schema.

- rowEntries: Also an empty list, signifying that the table contains no data upon creation.

- Appending to 'dataBaseTableRecord': The new table is prepended to the existing list of tables within the database, leveraging OCaml's list operations to efficiently manage this insertion.

The functions 'constructDataBase' and 'constructTable' reflect a clear, functional approach to database and table management. By starting with empty structures and providing mechanisms to incrementally add components, the system remains flexible and scalable. Moreover, using immutable data structures ensures that each state of the database is explicit and that previous states are preserved until explicitly changed, which is a core advantage in functional programming environments like OCaml.

These implementations facilitate a robust foundation for building and expanding an in-memory RDBMS, ensuring that each component can be individually tailored and securely managed as the database evolves.

3.1.4 Table Modification

Our RDBMS project includes advanced capabilities for dynamic table modification, which are essential for adapting to the constantly evolving data and business requirements. The functionality to add new columns to existing tables and to remove tables from the database enhances our system's flexibility and ensures that our database schema can adapt to changing needs without compromising data integrity.

3.1.4.1 Adding New Columns to Existing Tables: This feature enables the database to support the ongoing evolution of data structures as new types of data need to be accommodated.

1. The process is managed by the **insertColumnToTable** function. This function integrates seamlessly into our RDBMS architecture by taking the database instance, the title of the table to be modified, and the definition of the new column as inputs. It begins by retrieving the specified table from the database using the fetchTable function, which searches through the database's table list and locates the table by its title.

2. Once the table is retrieved, a new column definition is appended to the table's specifyColumn list. This is achieved by creating a new table structure where the specifyColumn is updated to include the new column. The updated table then replaces the original in the database's list of tables using the replaceTable function.

3. This function iterates over the list of tables, replacing the old table with the new, modified table in a manner that preserves the order and integrity of the database's table list.

3.1.4.2 Removing Tables from the Database: The capability to remove tables is just as crucial as the ability to add new columns. It allows for the removal of obsolete or redundant tables, ensuring that the database schema remains optimized and relevant to current business needs.

The **removeTable** function facilitates this process by filtering the specified table out of the database's list of tables. It takes the database instance and the title of the table to be removed as parameters. Within this function, a helper function, **filterdataBaseTableRecord**, is employed to create a new list of tables that excludes the table designated for removal.

This is done using OCaml's list filtering capabilities, which iterate over the list of tables and exclude the table that matches the specified title. The resulting list, which no longer includes the removed table, is then used to update the database's main table list, thus effectively removing the table from the database. These mechanisms for adding new columns and removing tables are implemented to uphold the principles of immutability and functional programming inherent in OCaml. By ensuring that all changes result in the creation of new instances of data structures rather than modifying existing ones directly, our system guarantees that operations are safe, predictable, and reversible. This architectural choice reduces the risk of data corruption and promotes easier state management, which is particularly beneficial in environments where data integrity is paramount. In conclusion, the dynamic table modification features of our RDBMS significantly contribute to its robustness and adaptability. By allowing for the seamless addition of new columns and the removal of tables, our database can quickly adapt to changes, supporting a dynamic and responsive data management system that meets the demands of modern business environments.

3.2.5 Filtering and Data Retrieval

As our RDBMS evolves, sophisticated filtering capabilities have become integral to efficiently querying and managing the data stored within our system. These functionalities enable our database to support dynamic data retrieval based on specific conditions, crucial for a wide range of applications from simple queries to complex analytical operations.

A. **Data Comparison:** In our system, we have implemented mechanisms to perform data comparisons against specified conditions, a fundamental feature for conducting searches and queries across the database. This is achieved through functions that compare data entries from columns against specified conditions such as equality, inequality, greater than, or less than. The function responsible for this, though not detailed in code snippets here, essentially takes two values and a relational operator as inputs (similar to the computeBool function) and returns a boolean result. This functionality is essential for determining whether data entries meet specific criteria, enabling effective data filtration and retrieval based on user queries.

B. **Predicate Evaluation:** Our RDBMS is capable of evaluating complex predicates on rows of data, pivotal for executing filtering operations where only rows that satisfy certain conditions are needed. This involves checking each piece of data within a row against the given predicate, significantly enhancing the system's ability to handle conditional logic within queries. For example, predicate evaluation might be performed by a function that assesses each row against specified criteria, such as checking if a numeric field is greater than a certain value or if a string field contains a specific substring.

C. **Row Filtering:** We have developed functions that iterate over lists of row data, applying predicate evaluations to determine which rows should be included in the query results. This functionality supports dynamic filtering of data based on user-defined criteria, making our database highly flexible and responsive to user needs. For instance, the selectEntries function filters rows by applying a predicate across each row of a specified table, collecting those that pass the predicate test. This selective retrieval allows users to efficiently extract and work with subsets of data tailored to their specific requirements.

D. **Table-Specific Filtering:** Filtering capabilities are extended to handle entire tables, optimizing performance and streamlining operations. This approach allows datasets within a table to be quickly filtered according to predefined predicates, optimizing data retrieval and manipulation based on specific business logic. This is implemented via a high-level function that applies a filtering predicate across all rows in a table, effectively paring down the dataset to only those entries that meet the defined conditions.

E. **How Filtering Works in Detail:** Filtering in our system is managed through a series of functions designed to apply complex logical conditions to data stored in the database. Here's how a typical filtering process might be conceptualized in the code, although specific functions are not shown here:

- **Filter Construction:** Initially, a filter is defined, which might include specifying the type of comparison (e.g., equals, not equals, greater than), the target column, and the value to compare against. This filter configuration forms the basis of the predicate used in subsequent operations.

- **Applying the Filter:** A function iterates through each row in the target table, applying the filter to each row. This might involve checking each column specified in the filter against the condition defined. For instance, if the filter is set to identify rows where a certain column's value is greater than 10, the function checks this condition for each row.

- **Collecting Results:** Rows that meet the filter criteria are collected and returned as a result set. This result set can then be used for further processing or displayed to the user.

- **Optimization:** For efficiency, filters can be applied as early as possible in data retrieval operations to minimize the volume of data that needs to be processed. This is particularly important in large-scale databases where operations on extensive datasets can become resource-intensive. These filtering functions are integral to ensuring that our database can effectively and efficiently retrieve and manipulate data as per user-defined conditions, providing robust support for a variety of data-driven applications.

3.1.6. Join:

- In our RDBMS project, the implementation of join operations is a critical feature that allows for the relational integration of data across different tables. We have focused on providing efficient and flexible join capabilities that support various types of joins including inner, left, and right joins. The join functionality is designed to merge rows from two or more tables based on a related column between them, effectively allowing for complex queries and data analysis that span multiple data sets.

- The process begins with the identification of the join type and the key columns on which the tables will be joined. For instance, an inner join operation is executed to combine rows from two tables where the join condition is met, and only the matching rows from both tables are returned. This is essential for queries where precise matching is required. On the other hand, left and right joins are used to return all rows from one side of the join, filling in with nulls where no match is found on the other side, which is crucial for comprehensive data analysis that includes records without matching counterparts.

- Our system efficiently manages these operations by creating a temporary data structure that holds the results of the join. This includes combining columns from both tables and computing each row's resultant values based on the specified join condition. The efficiency of the join operation is optimized through the use of indexed structures or hash tables to reduce the computational overhead associated with matching rows across large datasets. Furthermore, our RDBMS ensures that the joins are performed in a way that minimizes the impact on the system's performance. This involves optimizing the order of join operations and selectively loading data into memory to prevent excessive I/O operations.

- Such optimizations are crucial in scenarios involving large-scale data, where performance and speed are paramount. Overall, the join functionality in our RDBMS is designed to be robust and scalable, supporting a wide range of use cases from simple data retrievals to complex analytical queries, ensuring that users can efficiently and effectively analyze their data across multiple relational tables.

7. Future Work:

In the ongoing development of our RDBMS project, we are committed to continuously enhancing its robustness, efficiency, and user interaction capabilities. This section outlines a strategic plan for implementing advanced persistence mechanisms, optimizing concurrency control, and introducing sophisticated lexer and parser components for data input. These enhancements are designed to ensure the system can handle complex scenarios and large-scale operations efficiently.

Enhancing Persistence:

1. *Write-Ahead Logging (WAL)*: Implementing WAL is critical for ensuring data integrity and aiding in disaster recovery. By logging every write operation to a log file before it is committed to the database, WAL allows the system to replay these logs to recover its last known consistent state after a system failure.

2. *Snapshot and Checkpointing:* Regularly capturing snapshots of the database's current state can provide restore points for quick recovery. Checkpointing reduces the amount of log data that must be processed during recovery by periodically saving the state of the database, thereby speeding up the recovery process.

3. *Integration with Distributed File Systems:* Leveraging distributed file systems such as HDFS or Amazon S3 for data storage can enhance the scalability and reliability of the database. These systems offer built-in replication and fault tolerance mechanisms, ensuring data availability and durability across geographically distributed data-centers.

4. *Hybrid Storage Solutions:* Implementing a hybrid approach that combines the high speed of in-memory processing with the durability of on-disk storage can optimize performance and cost. Frequently accessed data can be kept in RAM for fast access, while archival data can be stored on more cost-effective disk storage.

Improving Concurrency:

1. *Multi-version Concurrency Control (MVCC):* By allowing each user connected to the database to see a snapshot of the database at a particular point in time, MVCC enables high concurrency for read operations and provides transaction isolation without requiring read locks.

2. *Fine-Grained Locking:* Implementing more granular locking mechanisms, such as row-level or even field-level locking, can significantly reduce lock contention. This allows more transactions to execute in parallel, improving the system's overall throughput.

3. *Optimistic Concurrency Control:* This method is particularly useful in environments where conflicts are rare. Instead of locking data, transactions are allowed to proceed without checks until commit time, at which point the system checks if other transactions have made conflicting changes.

4. *Lock-Free Data Structures:* Exploring the use of lock-free or minimal-lock data structures and algorithms can help to eliminate bottlenecks associated with traditional locking mechanisms, especially in highly concurrent environments.

Implementing Lexer and Parser for Enhanced Data Input:

1. *ANTLR-Based Lexer and Parser:* Using a powerful tool like ANTLR to generate a lexer and parser allows for precise control over the syntax and structure of input commands. ANTLR uses a defined grammar to generate code that can parse input data according to the specified rules, offering a robust solution for syntax analysis and error handling.

2. *Custom Lexer and Parser Development:* Building a custom lexer and parser from scratch gives complete control over the parsing logic, allowing for optimizations specific to our database's language and syntax. This approach also enables tailored error messaging and recovery strategies, enhancing the user's ability to correct input errors effectively.

3. *Integration with Existing Parsing Libraries:* Incorporating established libraries for parsing SQL, such as JSqlParser or sqlparse, can streamline the development process and reduce potential bugs. These

libraries are well-tested and can parse complex SQL queries, providing a reliable foundation for our input processing.

4. *Error Handling and Syntax Checking:* Enhancing our lexer and parser to include detailed error handling and syntax checking will improve usability by providing clear, actionable feedback. Implementing features like error highlighting and suggestions for corrections can significantly enhance the user experience.

5. *Interactive SQL Editor:* Developing an interactive SQL editor with features like syntax highlighting, auto-completion, and real-time validation could make the database more accessible, especially for less technical users. Such an editor could be part of a larger web-based database management interface, providing a rich user interface for direct interaction with the database.

These proposed enhancements will not only improve the technical capabilities of our RDBMS but also aim to make it more user-friendly and adaptable to the needs of modern database applications. Implementing these features will ensure our system can efficiently handle increased loads, provide robust data integrity, and offer a flexible and intuitive user experience.

3.3 Challenges Encountered

- **Integration of Components:** One of the initial challenges includes ensuring that the lexer, parser, and interpreter work seamlessly together, facilitating smooth data flow and function execution.

- **Performance Optimization:** Another major challenge is optimizing the system for performance, balancing the computational overhead of real-time interpretation against the need for rapid data access and manipulation.