FD-DBMS - Design & Implementation

Project for CS387 : Database & Information Systems Lab

May 1, 2023

1 Team Members

- Harshvardhan Agarwal, 200050050
- Pulkit Agarwal, 200050113
- Shashwat Garg, 200050130
- Vaibhav Raj, 200050148

2 Introduction

Functional Dependencies form an important component of Database and Information Systems. However, SQL does not provide a direct method of specifying functional dependency other than super-keys. We aim to develop a new database system that includes this functionality along with the other common components of SQL.

3 Scope of the Project

1. **Terminal Interface :** Developing an interface for a client to interact with the database system creating, deleting and updating databases.
2. **Query Processor :** The basic query parsing mechanism that identifies various components of a query and breaks it into an executable form.
3. **Database Storage Management :** Our aim is to handle file-based storage of data for insertion, updation, deletion and querying in a manner that allows us to include functional dependencies over existing features like foreign key, data range etc. Because our aim is to create a usual row-oriented database and not a columnar one, we plan to store entire rows of the table together in server-side files. This makes retrieval easy from an implementation point-of-view.
4. **Metadata Information :** Ideating a suitable structure for storing metadata of tables which can hold information on columns and incorporate constraints like Primary Key,

Foreign Key and Assert Statements; and a common metadata for the database holding functional dependencies and information on tables.

5. **Atomicity and Consistency Maintenance :** Using write-ahead logging before making updates to the database so that sudden interruptions don't leave the snapshot inconsistent & ensuring atomicity of updates so that checks for foreign keys and functional dependencies are performed only after completion of all operations in a given command set. Thus, integrity is maintained.

6. **Isolation :** Using the concept of readers-writer lock to ensure parallel access and updates can be made to the database from separate clients.

4 Design Decisions

1. **Scanner**
   We have **Flex** (a C++ Scanner Generator) and **Python** at our disposal for scanning of queries. A major part of this decision depends on how we introduce commands for functional dependencies.
   Here are the various pros & cons of these alternatives -

   - Python has an easy-to-interface for the *string* data type and various utilities like string slicing, checking for substring etc. come quite handy. If the scanner rules are simple, writing a scanner following logical steps is straightforward. Meanwhile, Flex just requires the scanner rules expressed as a set of Regular Expressions - this can come in handy when the structure of the language is too complicated to manually code it up.
   - The scanner we write in Python would run slower compared to the scanner generated using Flex because Python is interpreted.

   In our design, we have kept the scanner rules extremely simple and this allows us to manually handle the scanning process. Because of the simplicity of these expressions, the time overhead would not be a big problem either. Considering these, **Python** turns out to be a better alternative.

2. **Server**
   Bulk of the code base will use C++ for Database Management and to meet other requirements like consistency and isolation.
   Below, we describe various components of the server -

   - Interaction with the client has been established using Socket Programming principles (that we usually employ in C). The main server thread will use the **select()** call to monitor connections and spawn new threads as and when required to interact with the clients individually.
   - C++ **Semaphores** are used to establish the **Many Readers - Single Writer** lock that allows access to the database.

3. **Client**
   The user is provided with a terminal interface to interact with the database. Further work can be done on the creation of a Python client along the lines of psycopg2, which requires the ideation of server-client exchange through custom protocols.
   Terminal - The **REPL** (Read Eval Print Loop) has been written in Python using the socket library.
   Python Client (Future Work) - The Python library will also use the socket library but provide further utilities like maintenance of state, conversion of output data to CSV etc.

4. **Server-side File Structure**
   Various approaches are possible while creating the server-side file system. We could store the constraints for tables in one place or split it across tables. Moreover, columns could be stored separately.
   After due consideration, we found the below directory structure to be ideal for our use -

   - global_constraints.txt will store constraints that are applicable on the whole database & can span multiple tables like functional dependencies & foreign keys.
   - local_constraints.txt will store constraints applicable to specific tables like primary key, uniqueness or range specifications.

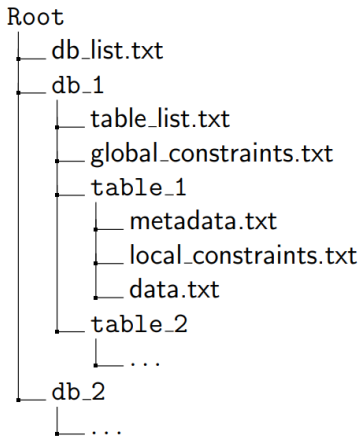

Figure 1: Directory Structure for FD-DBMS

5. **Query Structure**
   The only special command that we are adding is addition of functional dependencies. We tackle it as shown below -
   ADD FUNCDEP [db1.table1,...]—[db2.table2,...]

6. **Data Types**
   For keeping the implementation simple, we allow just 3 basic data types - **INT**, **FLOAT** & **STRING**.

5 Implementation Details

In this section, we discuss the implementation details for various structures which we used extensively throughout the project.

5.1 Data Units

Data units need to be prepared for use as arguments and return values for important functions which perform selection, insertion, update & deletion. We create the following structures for this purpose -

- **Value** - This is the smallest chunk of data in our database. It is a single *value*, along with its *type*. For reference, this will represent one attribute stored in a single record.
- **Record** - It is a collection of attributes and their corresponding values for this record. We store data here as a C++ map with the attribute name being the key. We store the attributes with every record so that functions involving individual records can be processed without the need to pass extra arguments.

5.2 SELECT Query

Unarguably, SELECT is the most important query in a general-purpose database management system. We create the following structure for a SELECT query -

SELECT {expr_1,...} AS {alias_1,...} FROM {query/join}

WHERE {filter} GROUPBY {group_expr_1} HAVING {group_filter}

To implement SELECT, we make use of the following structures -

- **Expression** - This is an arithmetic/string operation on top of other expressions. We give support for arithmetic addition (+), subtraction (-), multiplication (*) & division (/), alongside string concatenation (:).
  The recursive definition terminates with either constants or attribute names.
- **Comparison** - This class implements a binary comparison between two expressions. Usual operators like $[=, !=, <, <=, >, >=]$ are included. Another operator for strings is included which works similar to LIKE - "><". The RHS of this comparison is expected to be a C++-style regex string here.
- **Grouped Expression & Comparison** - These are grouped versions of the above two structures which support calls to aggregate functions - max, min, sum, mean, count.

These structures can be used to allow a recursive approach to solving the query. However, a small deviation has to be made to allow JOIN's in this framework.

To implement JOIN, we have another structure which itself has two SELECT's as its children with some filters in ON. The standardized form of a JOIN is as follows -

JOIN {select_1} {select_2} {on_filter}

Below is an example of the path a query takes through its evaluation.

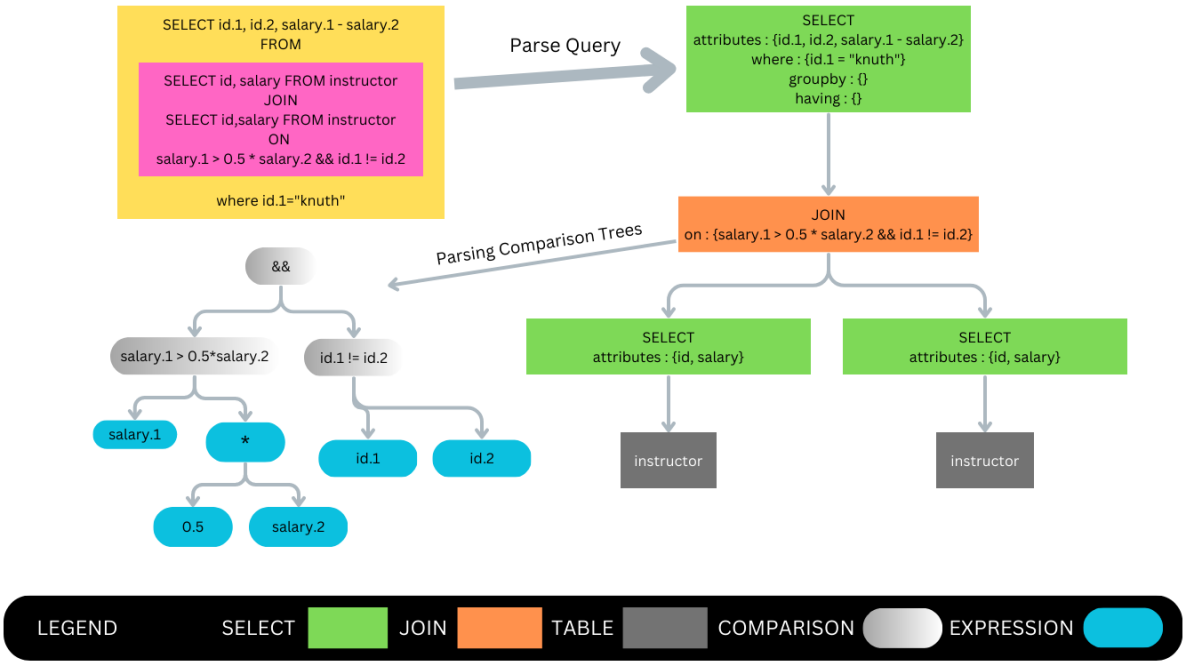

Figure 2: Analysis of a SELECT query during evaluation

5.3 Functional Dependencies

We store functional dependencies in the global constraints file because it can span multiple tables. Checking for a functional dependency is done using the following steps -

- Consider the tuple for the LHS of a functional dependency for all records already in the table.
- If the LHS for this record (to be inserted) matches any of these & the corresponding RHS doesn't, we have an error in insertion. Otherwise, we can add this to the database.
- If the LHS doesn't match, we can again safely add this data point and the RHS for this becomes the unique RHS corresponding to this LHS.