FINAL PROJECT - ~~SIMPLE~~ SQL DATABASE -

GENERAL INFORMATION

Author	Michael Romashov
JDK Version	Oracle OpenJDK 21
IDE Version	IntelliJ IDEA 2023.2.5
Operating System	Microsoft Windows 11 Professional

MOTIVATION

While I was scanning through the list of projects, I saw many options that only required a couple of simple features, well-suited for beginner software engineers, but not the challenge I was looking for.

The "simple" text database stood out as a project where I could go far beyond the basic requirements, and add functionality through which I could learn something new. My immediate thought was to write a more feature-rich database that could support multiple data-types and could be queried through SQL.

I’ve had plenty of opportunities to work with databases and SQL through coursework, personal projects, and work, so I had a good idea of what would be required. This gave me the confidence that I would be able to finish the project in a reasonable amount of time while implementing all of the features that I wanted.

OVERVIEW

The application I created is a relatively simple database that can be queried using SQL. The application provides a text box where you can type semicolon-terminated queries, and the output is shown in an adjacent grid.

The only internal state managed by the application is the database itself, which consists of tables that each contain a specified column definition and several rows of data.

The program performs simple disk I/O when importing and exporting SQL script files through the *File* menu option. When importing, a series of SQL commands is read from a file and exected to create a new database. When exporting, the current state is serialized into a series of SQL queries that create and populate the tables that currently exist within the database.

MODULES

engine.db

This module contains classes used to model the database and the data stored within. The three key components of this module are the **[Value]** algebraic data type and the **[Table]** and **[Database]** classes.

The **[Value]** algebraic data type, modeled using a sealed interface of records, wraps a string, integer, or boolean value in a variant of the interface so that they can be stored together in the same **[List]**. The interface also defines **[toString]** methods that allow values to be shown in the frontend.

The **[Table]** class represents a singular table within our database. It contains column definitions that specify the names and types of columns and a 2-dimensional array of **[Value]** objects that store our data. The class also provides methods to retrieve, insert, and update rows from the table.

The **[Database]** class represents the current state of the database. It contains several **[Table]** objects and provides a method that takes a query and performs the respective operation.

engine.io

This module contains a singular class **[Serde]**, which has two static methods for serializing and deserializing a database.

The **[Serde.serialize]** method takes a file output stream and a database and outputs SQL commands that would generate an equivalent database. It includes commands to both create and populate each table, as well as comments separating them for readability.

The **[Serde.deserialize]** method simply takes in a string from a SQL script file, parses the input, and executes each query on a clean database. The newly created database is then returned so that the current application state can be overwritten.

engine.sql

This module contains data types and utilities that allow for a SQL query entered in the user interface to be converted into a format that the program can understand. There are two crucial components to this process: the **[Tokenizer]** and the **[Parser]**.

The **[Tokenizer]** reads the input string directly and splits it up into "tokens", which are an enumerated set of keywords, symbols, and values that can be represented with a Java record. The different types of tokens are:

- **statements** - reserved keywords in the SQL language
- **identifiers** - table and column names
- **literals** - string, integer, and boolean values
- **operators** - comparisons used for filtering
- **punctuation** - symbols such as commas, semicolons, and parentheses

The **[Parser]** then takes this list of tokens and attempts to interpret them as full queries. The parser returns one of the following query types along with any parameters:

- **SHOW TABLES** - lists all tables in the database
- **CREATE TABLE** - creates a new table with a given schema
- **DROP TABLE** - deletes a table
- **INSERT INTO** - adds data to a table as a new row
- **SELECT FROM** - retrieves data with optional filters and orderings
- **DELETE FROM** - removes a row from the table
- **UPDATE SET** - updates a row with new data

gui

This module contains the Java Swing interface definitions, along with code to attach event listeners to various interactive elements. The form was created using IntelliJ's built-in form editor, which generates an XML file containing the UI elements and any related attributes.

tests

This module contains unit tests for the **[Tokenizer]** and **[Parser]** classes. This was used to develop the two classes by employing test-driven development.

ALGORITHMS

The core algorithm behind the SQL parser is the tokenizer. Broadly speaking, the tokenizer attempts multiple strategies to consume one or more characters from the input stream and convert them to **[Token]**s. If at any point, the tokenizer loops through all of the strategies without consuming a character, then the string does not conform to our language specification and cannot be tokenized.

One such strategy is to test if the current character is one of the punctuation symbols { ( ) , ; } and to return the respective **[Token.Punctuation]** object. Another strategy involves testing characters against the regex pattern [_A-Za-z] and to collect them into a single **[Token.Identifier]**.

The following is an example of how the tokenizer interprets a simple SQL query.

```
SELECT first_name FROM users
```

→

```
Keyword[SELECT],
Identifier['first_name'],
Keyword[FROM],
Identifier['users'],
Punctuation[;]
```

The parser then takes this sequence of tokens and create a **[Query]** to be executed by the database.

```
Keyword[SELECT],
Identifier['first_name'],
Keyword[FROM],
Identifier['users'],
Punctuation[;]
```

→

```
SELECT[
   table: 'users',
   columns: 'first_name'
]
```

CONCLUSIONS

In the end, I managed to implement a database that supports CRUD (Create, Read, Update, Delete) functionality, as well as provides utilities to export and import the database to be saved or sent to somebody else. Some features of SQL databases that I did not implement are foreign relations, subqueries, and stored procedures. These are left as an exercise to the reader, of course :).

Throughout the entire process of writing this application, I honestly grew to resent Java. I particularly did not like how Java does not distinguish between a nullable and non-nullable type (without using meta-programming constructs such as annotations), which makes it very annoying to represent "Optional" data. Paradigms such as algebraic data-types, which I used a lot, are also haphazardly shoved into an existing language and feel that way when you use them. The language forces you to jump through many hoops in order to use newer, more exotic features. I also did not like being forced to put everything into a class, as there were many functions that were not necessarily associated with any state.

If I were to revisit this project in the future, I think I would switch to using the Rust language for it's first-class support of strongly-typed algebraic data types and powerful pattern matching functionality. This would greatly simplify any logic that needs to account for multiple types of data, such as tokens, queries, and table values.

Thinking more about features, I think that the next improvement would be to add indexing to speed up ordering and to implement foreign key relations.