CMPSC380
Principles of Database Systems, Fall 2014

Final Project: Introduction to Compiler Design

Purpose: Implementation of Query Processing in a Relational Database Management System
Pledge:

Contents

List of listings

1 Background

Relational database management systems (RDBMSs) are everywhere. The relational model is a model of data storage in which data is stored in *tuples*, or rows, which are grouped together to form *relations*, or tables [6,8,15]. The relational model is perhaps the most popular models of data storage currently in use, with Silberschatz, Korth, and Sudarshan calling it "[t]he primary data model for commercial data-processing applications" [15, page 39].

Users and other software typically interact with these systems through the use of a *query language*. A query language is a domain-specific declarative programming language that is used to express database queries, such as selecting data from the database; modifying the database's state by changing, inserting, or deleting records; and specifying the structure of the tables in the database.

Therefore, in order to facilitate users and application programs interaction with the database, it is necessary for an RDBMS to provide a query processing system with some method capable of executing query language statements. Essentially, this means that a compiler or an interpreter is a core feature of any RDBMS. These function similarly to compilers or interpreters which execute general-purpose programming languages.

A majority of modern relational database management systems, from the SQLite embedded database in every Android phone and iPhone [16] to the MySQL databases used in many web applications [1], implement the *Structured Query Language*, or SQL. [15]. SQL consists of two primary components, the *Data Definition Language* or DDL, which is used to define the structures in which data is stored (called the *schema* of the database), and the *Data Manipulation Language*, or DML, which is used for statements which access and modify the data stored in the database.

In order to learn more about how relational databases function, I have developed my own implementation, called DeeBee. DeeBee implements a limited subset of SQL chosen to be expressive enough to allow most relational database functionality. DeeBee is released as open-source software under the MIT license. Current and past releases are available for download at `https://github.com/hawkw/deebee/releases`. The provided JAR file can be included as a library in projects which use the DeeBee API to connect to a DeeBee database; or it may be executed using the *java -jar* command to interact with a DeeBee database from the command line. Finally, DeeBee's ScalaDoc API documentation is available at `http://hawkw.github.io/deebee/api/index.html#deebee.package`.

2 Implementation

2.1 The Scala Programming Language

DeeBee was implemented using the Scala programming language, an object-oriented functional programming language which runs on the Java virtual machine [12–14]. Scala was designed by Martin Odersky of the Programming Methods Laboratory at École Polytechnique Fédérale de Lausanne with the intention of developing a highly scalable programming language, "in the sense that the same concepts can describe small as well as large parts" [13] and in the sense that Scala

should be applicable to tasks of various sizes and complexities, and provide high performance at large scales [14].

Scala was inspired by criticisms of Java and by the recent popularity of functional programming langauges such as Haskell. It aims to provide a syntax that is more expressive than that of Java but is still readily accessable to Java programmers. It is a statically-typed language and was developed with a focus on type safety, immutable data structures, and pure functions [12–14]. Because it compiles to Java bytecode and runs on the JVM, Scala is mutually cross-compatible with Java, meaning that Scala code can natively call Java methods and use Java libraries, and vise versa [14].

A key concept in the Scala design is the idea of an 'embedded domain-specific language (DSL)'. Essentially, this concept suggests that Scala's syntax should be modifiable to the extent that code for specific tasks be expressed with its' own syntax within Scala. These DSLs are still Scala code and can still be compiled by the Scala compiler, but their syntax differs based on the task they are intended for [7, 9, 14]. The Scala parsing library and the ScalaTest testing framework both provide exmaples of embedded DSLs.

Scala was chosen as the ideal language for DeeBee's implementation due to the expressiveness of its syntax, which allows complex systems to be implemented in few lines of code; its performance at scale; the existence of powerful libraries for text parsing (**??**) and concurrent programming using the actors model ; and the cross-platform capabilities of the JVM.

2.2 Query Parsing

DeeBee processes SQL queries by parsing the input query strings and generating an abstract syntax tree (AST) which represents the query. The AST is then interpreted against a context (typically a database or table) to determine the actions necessary to carry out the query. The query is then conducted using the API of the target relations to carry out the necessary operations.

DeeBee's query parser was implemented using the Scala standard library's combinator parsing [11] package. Combinator parsing is a method of parser implementation, popular in functional programming languages such as Scala and Haskell, which provides an alternative to parser generators such as `yacc` or `bison`.

In the parser generator approach, the grammar of a language is described using a syntax similar to Baukus-Naur Form (BNF), and then the parser generator is invoked on that grammar description. The parser generator uses the grammar description to produce the source code for a parser for that language[1]. This parser can be included in a project and used to parse text input.

While the use of these tools is often much less demanding than implementing a parser by hand, the use of parser generators has a number of disadvantages. The generated source code is frequently highly complex and, since it is generated automatically, it may be poorly documented and difficult to understand. This can make debugging and maintaining the generated code very difficult.

[1] In the case of `bison`, these are LALR(1) parsers implemented in C

2.2.1 Combinator Parsing

Combinator parsing provides a method of implementing recursive descent parsers using a BNF-like domain-specific language, but rather than using that grammar description to generate parsing code, the grammar description is the source code of the parser, and can be inspected, maintained, and modified transparently [7, 9, 11].

In this approach, a parser is defined as a function which takes as input a string and either accepts or rejects that string according to some rule. A parser combinator is a higher-order function which takes as parameters two parsers, and produces a new parser that combines the two input parsers. Parser combinators exist which combine the input parsers according to various rules, such as sequencing, alternation, optionality, or repetition. The language specification is written by repeatedly combining primitive parsers for tokens such as keywords, constants, and identifiers into more complex parsers for constructs such as expressions. At runtime, the parser then attempts to parse some input against the start symbol parser, which is composed of multiple smaller parsers, recursively descending through the grammar by calling each component parser. [2, 5, 11, 17]

For example, in the Scala parser combinators library, the `~` parser combinator is the sequencing parser combinator. If `a` and `b` are parser functions, then the expression `a ~b` would create a parser which accepts a string containing a substring accepted by `a` followed by a substring accepted by `b`. Similarly, `|` is the alternation combinator, so the expression `a | b` specifies a parser that accepts either a string accepted by `a` or `b`. The `.?` combinator can be applied to a single parser in order to make that parser optional (one or none), while the `.*` combinator will accept any number of repetitions of the input parser.

Multiple parser combinators can be combined into complex expressions. For example, if `a`, `b`. and `c` are all parser functions, the expression `def d = a ~b | c.*` defines `d` as a parser that accepts a string which contains a substring matching `a` followed by a substring matching either `b` or any number of repetitions of `c`.

2.2.2 Packrat Parsing

A major disadvantage of top-down parsers, such as the recursive descent parsers created through the use of parser combinators, is that they cannot parse left-recursive grammars. Scala's parsing library offers a solution to this problem by providing packrat parsing functionality.

Packrat parsing, first described by Bryan Ford in his masters' thesis [3] is a technique that both improves performance of backtracking parsers (such as recursive descent parsers), guaranteeing linear parse time, and allows the parsing of grammars containing left recursion, as well as any LL(k) or LR(k) language. Packrat parsing works by memoizing the results of each invocation of

This class enhances parser combinators with the addition of a memoization facility. This allows recursive-descent parsing of left-recursive grammars. It also improves performance, providing linear-time parsing for most grammars [3, 4, 10]. I make liberal use of packrat parsers in order take advantage of their improved performance.

Some difficulties were encountered in parsing, mostly related to the parsing of nested predicate expressions in SQL `WHERE` clauses. These issues were eventually resolved by separating the productions for 'leaf' predicates (those consisting of a comparison between an attribute and an ex-

pression) and predicates consisting of multiple predicate expressions, and using the longest-match parser combinator to back-track in order to parse the entire `WHERE` clause.

2.3 Query Interpretation

Queries are interpreted against a table using the `Relation` API. DeeBee defines a core trait for all tables, `Relation`, which defines a number of 'primitive' operations on a table, such as filtering rows by a predicate, projecting a relation by selecting specific columns, and taking a subset number of rows. These in turn rely on two abstract methods, for accessing the table's attribute definitions and the set of the table's rows. These methods are implemented by each concrete class that implements `Relation`.

Relation is extended by two other traits, `Selectable` and `Modifyable`, which provide polymorphic functions for actually processing queries. These can be mixed in as needed to represent tables which support these functionalities.

Queries involving predicates, such as `SELECT` and `DELETE` statements with `WHERE` clauses, must go through an additional step of predicate construction, which converts SQL ASTs for `WHERE` clauses into Scala partial functions which take as a parameter a row from a table and return a Boolean. These functions are emitted by the AST node for `WHERE` clauses, using the target relation as a context for accessing the attributes corresponding to names in the SQL predicate. A similar process is performed for `INSERT` statements, which must be checked against the type and integrity constraints in the attribute corresponding to each value in the statement.

3 Storage

DeeBee currently provides one storage type, `CSVRelation`, which stores each table as a comma-separated values file on disk. While this storage mechanism is not ideal for a relational database, as it provides poor performance, it was used for the first DeeBee storage backend due to ease of implementation and testing. However, the DeeBee storage system is designed to be modular, making it possible to implement other storage backends in the future. I am considering the implementation of a B+ tree backend, similar to those used in 'real-world' RDBMSs, and/or some form of hash bucket based storage.

4 Further Research

While DeeBee currently implements the entirity of the planned subset of SQL, there are a number of additional features that could be implemented. As discussed in Section 3, CSV storage is not as performant as the storage methods used by real RDBMSs, and I am considering implementing a more advanced storage system.

Furthermore, DeeBee SQL does not currently implement a large portion of SQL syntax, such as `JOIN`s, `UPDATE` statements, `VIEW`s, or aggregate functions such as `SUM` and `COUNT`. DeeBee also does not implement all of the data types available in other SQL databases, such as BLOBs (binary large

objects) and CLOBs (character large objects). An implementation of the SQL `DATE` type based on `java.util.date` was prototyped but is currently not yet implemented.

Finally, DeeBee's connections API, used by other programs to interact with the system, could be expanded with support for non-blocking connections and additional options to configure the connected database, similar to those available in the Java DataBase Connection driver (JDBC). I have considered the implementing a JDBC compatible API for DeeBee, but am not currently certain if this is possible, due to the differences between DeeBee and other SQL databases.

References

[1] Dale Dougherty. LAMP: The Open Source Web Platform - O'Reilly Media, 2001. `http://www.onlamp.com/pub/a/onlamp/2001/01/25/lamp.html`.

[2] Jeroen Fokker. Functional parsers. In *Advanced Functional Programming*, pages 1–23. Springer, 1995.

[3] Bryan Ford. *Packrat parsing: a practical linear-time algorithm with backtracking*. PhD thesis, Massachusetts Institute of Technology, 2002.

[4] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ACM SIGPLAN Notices*, volume 37, pages 36–47. ACM, 2002.

[5] Richard A Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In *Practical Aspects of Declarative Languages*, pages 167–181. Springer, 2008.

[6] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. *Database System Implementation*, volume 654. Prentice Hall Upper Saddle River, NJ:, 2000.

[7] Debasish Ghosh. *DSLs in action*. Manning Publications Co., 2010.

[8] Jan L Harrington. *Relational database design and implementation: clearly explained*. Morgan Kaufmann, 2009.

[9] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 137–148. ACM, 2008.

[10] Manohar Jonnalagedda, Martin Odersky, and Tiark Rompf. Packrat Parsing in Scala. Technical report, Ecole Polytechnique Fédérale de Lausanne, 2009.

[11] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in Scala. Technical report, Katholieke Universiteit Leuven, 2008.

[12] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

[13] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[14] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2008.

[15] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Education, 2010.

[16] SQLite. Famous Uses of SQLite, 2014. `https://www.sqlite.org/famous.html`.

[17] S Doaitse Swierstra. Combinator parsers: From toys to tools. *Electronic Notes in Theoretical Computer Science*, 41(1):38–59, 2001.