CMPSC380
Principles of Database Systems, Fall 2014

Final Project: Advanced Topics in Data Management

Purpose: DeeBee: Implementation of A Relational Database Management System

Pledge:

Hand in: Friday, December 12th, 2014

Abstract: DeeBee is a small relational database management system implemented for educational purposes. It implements a subset of the structured query language, enough to support simple database operations; and is designed for modularity, so that additional advanced database features can be added in the future.

Contents

List of listings

1 Introduction

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 [5,8,16]. 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" [16, page 39].

A majority of modern relational database management systems, from the SQLite embedded database in every Android phone and iPhone [17] to the MySQL databases used in many web applications [2], implement the *Structured Query Language*, or SQL. SQL is a *query language*; domain-specific declarative programming language that is used by database administrators, users, and application software to interact with a database [16].

In order to learn more about how such SQL databases function 'under the hood', I have implemented my own small RDBMS, called DeeBee. DeeBee implements a small subset of the SQL, chosen to be expressive enough to allow basic database operations to be performed but minimal enough to allow DeeBee to be implemented within the constraints of the Computer Science 380 Final Project. Implementing DeeBee has yielded many insights into the challenges, techniques, and patterns involved in the design and implementation of an RDBMS.

At the time of writing, the DeeBee codebase comprises almost 1700 lines of Scala source code and over 500 lines of comments. While this is small compared to many 'real-world' systems, it still represents a significant undertaking for a single individual over a 24-day period. Therefore, I have made use of a number of software engineering tools and practices in order to best maximize my productivity. While these techniques are not the focus of this assignment, I will touch on them briefly as well.

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 [6, 9, 14]. The Scala parsing library (Section 4.1) and the ScalaTest testing framework (Section 2.3) 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 (Section 4.1) and concurrent programming using the actors model (Sections 3 and 4.2); and the cross-platform capabilities of the JVM.

2.2 Tools

DeeBee builds were conducted using SBT, the Simple Build Tool. SBT is a build tool similar to Maven or Gradle, that is configured using a Scala-based domain-specific language. Like Maven and Gradle, it is capable of automatically managing dependencies using Ivy repositories [15]. SBT was chosen due to improved compatibility with Scala-focused plugins, its' high performance incremental Scala compiler, and my desire to try out a new build tool.

Continuous Integration Each push to the DeeBee GitHub repository was built using the Travis continuous integration service (`https://travis-ci.org/hawkw/deebee`), which ran tests, collected coverage information, generated and archived API documentation, and deployed to GitHub releases (when a new Git tag was pushed). Codacy (`https://www.codacy.com/public/hawkweisman/deebee`), a service which performs static analysis of a codebase and assesses it according to various quality criteria such as code complexity, code style, performance, and compatibility, was used to ensure that the codebase maintained an overall high quality.

2.3 Testing

Testing was conducted using the ScalaTest testing framework. ScalaTest provides an embedded DSL for conducting many different types of testing, including configuring test reports that automatically output specification information [19]. Three specifications were written to describe DeeBee's behaviour: an integration testing specification that deals with database-level behaviour, a specification for the SQL parser, and a specification for query processing at the table level.

Code coverage data was collected using the Scoverage SBT plugin, and was archived on Coveralls (`https://coveralls.io/r/hawkw/deebee?branch=master`), a service which tracks coverage for open-source projects. Working with Coveralls caused some issues, as files that were not targets

for code coverageanalysis were tracked by Coveralls despire being excluded from coverage in the buildfile. This lead to coverage reports that were consistently lower than the actual recorded coverage percentage.

3 Architecture

DeeBee makes use of the actor model for concurrent programming. In this model, actors are independant objects capable of carrying out specific operations and communicating through asynchronous, anonymous passing of immutable message objects. When an actor recieves a message, that message enters its' queue (called a *mailbox*), and it is capable of responding to the message by sending a message to the sender, sending messages to other actors, changing its' internal state, or carrying out some process [1, 7, 13]. Essentially, actors are finite-state machines with mailboxes.

The asynchronous and anonymous nature of actor message-passing makes actors a highly useful way to compose concurrent systems. The anonymity (an actor cannot access any fields or methods of another actor) and decoupled nature of actors allows for a high level of fault tolerance, as if the process corresponding to one actor crashes, it has little to no effect on the rest of the system. Furthermore, since actors communicate through immutable message objects, actors can run on separate computers on a network, and communication overhead (both over the network and locally) is as low as possible. Finally, the actors model is almost entirely reactive in nature — since computation happens only as a response to messages, the system only uses computational resources when fulfulling a request, and does not use CPU time while 'idle' [1, 7, 13]. This property makes the actors model ideal for databases. In Scala, functionality related to actors is provided by the Akka framework.

DeeBee models each database as an actor, and each table in the database as a child actor. The database actor is responsible for receiving queries and handling them by dispatching DML statements to the target tables or by creating and deleting table actors in response to DDL statements. The table actors can respond to queries by updating their state, or by sending result sets to the querying connection.

4 Query Processing

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.

4.1 Query Parsing

DeeBee's query parser was implemented using the Scala standard library's parser-combinators [11] package. Combinator parsing represents a functional programming approach to text parsing. In this approach, a parser combinator is a higher-order function which takes as a parameter two parsers (here defined as functions which accept some strings and reject others) and produces a new

parser that combines the two input parsers according to some rule, such as sequencing, repetition, or disjunction. The repeated combination of smaller primitive parsers through various combinators constructs a recursive-descent parser for the specified language. [3, 4, 11, 18].

Following the Scala philosophy of embedded domain-specific languages [6, 9, 11], the Scala parsing library represents these combinators as symbols similar to those found in the Bauckus-Naur Form, a common symbolic representation of a language grammar. Using the Scala parser-combinators, then, is almost as simple as constructing the BNF for the language to be parsed.

The 'Packrat Parser' class contained within the Scala parsing library 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 [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 expression) 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.

4.2 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.

5 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 im-

plementation 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.

6 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 5, 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] Gul Abdulnabi Agha. Actors: a model of concurrent computation in distributed systems. 1985.

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

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

[4] 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.

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

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

[7] Philipp Haller. On the integration of the actor model in mainstream technologies: the scala perspective. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, pages 1–6. ACM, 2012.

[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] Shiti Saxena. *Getting Started with SBT for Scala*. Packt Publishing Ltd, 2013.

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

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

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

[19] Bill Venners. ScalaTest.