CMPSC380
Principles of Database Systems, Fall 2014

Final Project Status Update

Pledge:

Hand in: Wednesday, December 3rd, 2014

1 Introduction

As of Wednesday, December 3rd, I have implemented a significant amount of the planned functionality for DeeBee. I have used the ScalaTest [8] framework to construct multiple specifications which describe DeeBee's functionality prior to its' implementation. Currently, a general overview of DeeBee's development status is as follows:

- The SQL parser has been completed.
- A majority of the query-processing subsystem has been implemented. Some functionality is still being developed, but a basic subset of the core SQL queries can be processed.
- Databases consisting of multiple tables have yet to be implemented.
- Storage on disk has yet to be implemented.
- A connection API for querying DeeBee from an external program has yet to be implemented.

2 Query Parsing

DeeBee's query parsing subsystem is almost fully completed. The parser was implemented using the Scala standard library's parser-combinators [1] 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. [1–3, 7].

Following the Scala philosophy of embedded domain-specific languages [1,4,5], 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. Listing 1 provides a small sample of the source code for the DeeBee SQL parser, demonstrating the use of parser combinators.

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 [6]. I make liberal use of packrat parsers in order take advantage of their improved performance.

```
lazy val query: P[Node] = (
    createTable
      | select
      | delete
      | insert
      | dropTable
    ) <~ ";"
  lazy val createTable: P[CreateStmt] =
    ("create" ~ "table") ~> identifier ~ "(" ~ rep1sep(attr | refConstraint, ",") <~ ")"
    ^^ {
        case name ~ "(" ~ contents => new CreateStmt(
        name,
        contents.filter {
            _.isInstanceOf[Attribute]
        }.asInstanceOf[List[Attribute]],
        contents.filter {
            _.isInstanceOf[Constraint]
        }.asInstanceOf[List[Constraint]]
        )
    }
  lazy val insert: P[InsertStmt] =
    "insert" ~> "into" ~> identifier ~ ("values" ~> "(" ~> repsep(literal, ",") <~ ")")
    ^^ {
        case into ~ values => InsertStmt(into, values)
    }

  lazy val dropTable: P[DropStmt] = "drop" ~> "table" ~> identifier ^^{
    case i => DropStmt(i)
    }
  lazy val attr: P[Attribute] = ident ~ typ ~ inPlaceConstraint.* ^^{
    case name ~ dt ~ cs => Attribute(name, dt, cs)
    }
  lazy val typ: P[Type] = (
    ("int" | "integer") ^^^ IntegerType
      | "char" ~> "(" ~> int <~ ")" ^^{
            case i => CharType(i)
        }
      | "varchar" ~> "(" ~> int <~ ")" ^^{
            case n => VarcharType(n)
        }
      | ("numeric" | "decimal") ~> "(" ~> int ~ "," ~ int <~ ")" ^^{
            case p ~ "," ~ d => DecimalType(p,d)
        }
    )
```

Listing 1: An excerpt from the DeeBee SQL parser source code, demonstrating the use of parser-combinators.

The SQL parser has a comprehensive ScalaTest test suite. These tests specify the entire subset of the structured-query language which I intend for DeeBee to initially support. These tests function by parsing a SQL statement,attempting to reconstruct the statement from the abstract syntax tree produced by the parser, and then checking that the reconstructed statement and the parsed statement are the same. All of these tests have been implemented.

3 Query Processing

The query-processing subsystem, including an abstract syntax tree for representing SQL statements, a system for interpreting SQL `WHERE` clauses into Scala predicates, and systems for evaluating SQL queries against the general relation API, have been implemented. Currently, the query-processing system is capable of processing `SELECT`, `DELETE`, and `INSERT` statements.

Some additional work is still necessary in order to complete the query processing system. The primary missing feature is related to the enforcement of integrity constraints. Currently, when values are `INSERT`ed into a table, they are checked against the schema's type constraints, but not against other integrity constraints such as `UNIQUE` and `NOT NULL`.

Furthermore, additional SQL features may be implemented, if there is enough time remaining once the core functionality is completed. Aggregate functions, such as `SUM`, joins, views, triggers, and `CHECK` constraints, are all not considered 'core' functionality and have been excluded from DeeBee's requirements, but if there is sufficient extra time, I may attempt to implement some of these features.

The processing of SQL queries is implemented primarily in the `Relation` API. `Relation` provides 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 methods, for accessing the table's attribute definitions and the set of the table's rows, which will be provided by each concrete implementation of the `Relation` trait. The source code for `Relation` is provided in Listing 2.

```scala
trait Relation {

    def rows: Set[Row]

    def attributes: Seq[Attribute]

    def project(names: Seq[String]): Relation with Selectable = new View(
        rows.map(
            r => names.map(
                name => {
                    r(attributes.indexWhere(a => a.name.name == name) match {
                        case n if n < 0 => throw new QueryException(
                            s"Could not process projection," +
                            s" $this did not contain $name"
                        )
                        case n: Int => n
                    })
                }
            )
        ),
        attributes.filter(names contains _.name.name)
    )
    protected def filter(predicate: Row => Boolean): Relation = new View(
        rows.filter(predicate),
        attributes
    )
    protected def add(row: Row): Try[Relation with Modifyable]
    protected def filterNot(predicate: Row => Boolean): Try[Relation with Modifyable]
    protected def drop(n: Int): Try[Relation with Modifyable]

    def iterator = rows.toIterator

    def take(n: Int) = new View(rows.take(n), attributes)
    override def toString = rows.map(r => r.foldLeft("")(
        (acc, thing) => acc + "|" + thing)
    ).mkString("\n")
}
```

Listing 2: An excerpt from the DeeBee `Relation` source code, showing the definition of the core `Relation` API.

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. The source code of these traits is presented in Listing 3 and

Listing 4.

```
trait Selectable extends Relation {

  def process(select: SelectStmt): Try[Relation] = {
    val predicate = select.where
      .map(clause => clause.emit(this))
    (select.projections match {
      case GlobProj :: Nil => Success(if (predicate.isDefined) {
        this.filter(predicate.get.get)
      } else this)
      case Nil => Failure(new QueryException(
        "Received a SELECT statement with no projections.")
        )
      case p: Seq[Proj] if p.length > 0 => Success((if (predicate.isDefined) {
        this.filter(predicate.get.get)
      } else this).project(p.map(_.emit)))
    }).map(results =>
      results.take(
        select.limit
          .map(_
          .emit(this)
          .get // this will be Success because it's a constant.
          )
          .getOrElse(results.rows.size)
      )
      )
  }

}
```

Listing 3: An excerpt from the DeeBee `Relation` source code, showing the definition of the `Selectable` trait.

```
trait Modifyable extends Relation with Selectable {
    def process(insert: InsertStmt):
        Try[Relation with Selectable with Modifyable] = insert match {
        case InsertStmt(_, vals: List[Const[_]]) if vals.length == attributes.length =>
        add(
            Try(
                (
                    for { i <- 0 until vals.length } yield {
                        attributes(i)
                            .apply(vals(i)
                            .emit(this)
                            .get)
                        }
                    ).map{t: Try[Entry[_]] => t.get}
                )
            .get
        )
        case InsertStmt(_, vals) =>
            Failure(new QueryException(
            s"Could not insert (${vals.mkString(", ")}):\n" +
            s"Expected ${attributes.length} values, but received ${vals.length}."))
      }
  def process(delete: DeleteStmt):
    Try[Relation with Selectable with Modifyable] = delete match {
        case DeleteStmt(_, None, None) => drop(rows.size)
        case DeleteStmt(_, Some(comp), None) =>
            (
                for (pred <- comp.emit(this)) yield filterNot(pred)
            ).flatten
        case DeleteStmt(_, None, Some(limit)) =>
            (
                for (n <- limit.emit(this)) yield drop(n)
            ).flatten
        case DeleteStmt(_, Some(comp), Some(limit)) => ??? //TODO: Implement
  }

}
```

Listing 4: An excerpt from the DeeBee `Relation` source code, showing the definition of the `Modifyable` trait.

All of this functionality is tested by the ScalaTest test suite, at both the unit testing and integration testing levels.

4 Storage Management

The storage managmenet component of DeeBee is the one that currently requires the most work, as a majority of its' functionality is still under development.

Right now, there exists no mechanism for persisting data on disk, a core RDBMS feature. However, I intend to implement a concrete `Relation` implementation that stores data on disk as a comment-separated values file. Additionally, I intend to investigate the implementation of hashing or B+ tree based storage mechanisms, similar to those used in real-world databases. While there may not be enough time to implement this functionality before the end of classes this year, I intend to continue to maintain DeeBee as an open-source project.

Furthermore, the database management system, which is responsible for creating and deleting tables, and dispatching queries to the targeted tables, has yet to be implemented. I am considering the use of the Akka framework, which provides an actors model for Scala similar to that of the Erlang programming language, for implementing this functionality. The use of the actors model would provide additional fault-tolerance and improved concurrent operation.

5 API

Currently, there is no external API for opening connections to a DeeBee database, or for shipping queries to the database. I am considering attempting to implement a JDBC driver class for DeeBee. However, this may not be possible, due to DeeBee not supporting some functionality, such as query compilation, that other SQL databases support. If this is not possible, I will implement my own connection API, instead.

References

[1] Parser combinators in Scala, author=Moors, Adriaan and Piessens, Frank and Odersky, Martin, journal=CW Reports, year=2008, institution=Katholieke Universiteit Leuven. Technical report.

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

[3] 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.

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

[5] 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.

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

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

[8] Bill Venners. Scalatest.