Gemfile

```
source "https://rubygems.org"

git_source(:github) {|repo_name| "https://github.com/#{repo_name}" }

# Specify your gem's dependencies in sql_assess.gemspec
gemspec
```

Gemfile.lock

```
PATH
  remote: .
  specs:
    sql_assess (0.1.0)
      activesupport
      mysql2
      rgl
      sql-parser-vlad (~> 0.0.15)

GEM
  remote: https://rubygems.org/
  specs:
    activesupport (5.2.0)
      concurrent-ruby (~> 1.0, >= 1.0.2)
      i18n (>= 0.7, < 2)
      minitest (~> 5.1)
      tzinfo (~> 1.1)
    ast (2.4.0)
    codecov (0.1.10)
      json
      simplecov
      url
    coderay (1.1.2)
    concurrent-ruby (1.0.5)
    diff-lcs (1.3)
    docile (1.3.0)
    i18n (1.0.0)
      concurrent-ruby (~> 1.0)
    json (2.1.0)
    lazy_priority_queue (0.1.1)
    method_source (0.9.0)
    minitest (5.11.3)
    mysql2 (0.5.1)
    parallel (1.12.1)
    parser (2.5.0.5)
      ast (~> 2.4.0)
    powerpack (0.1.1)
    pry (0.11.3)
      coderay (~> 1.1.0)
      method_source (~> 0.9.0)
    racc (1.4.12)
    rainbow (3.0.0)
    rake (10.5.0)
    rgl (0.5.3)
      lazy_priority_queue (~> 0.1.0)
      stream (~> 0.5.0)
    rspec (3.7.0)
      rspec-core (~> 3.7.0)
      rspec-expectations (~> 3.7.0)
      rspec-mocks (~> 3.7.0)
    rspec-core (3.7.1)
      rspec-support (~> 3.7.0)
    rspec-expectations (3.7.0)
```

```
      diff-lcs (>= 1.2.0, < 2.0)
      rspec-support (~> 3.7.0)
    rspec-mocks (3.7.0)
      diff-lcs (>= 1.2.0, < 2.0)
      rspec-support (~> 3.7.0)
    rspec-support (3.7.1)
    rubocop (0.54.0)
      parallel (~> 1.10)
      parser (>= 2.5)
      powerpack (~> 0.1)
      rainbow (>= 2.2.2, < 4.0)
      ruby-progressbar (~> 1.7)
      unicode-display_width (~> 1.0, >= 1.0.1)
    ruby-progressbar (1.9.0)
    simplecov (0.16.1)
      docile (~> 1.1)
      json (>= 1.8, < 3)
      simplecov-html (~> 0.10.0)
    simplecov-html (0.10.2)
    sql-parser-vlad (0.0.15)
      racc (= 1.4.12)
    stream (0.5)
    thread_safe (0.3.6)
    timecop (0.9.1)
    tzinfo (1.2.5)
      thread_safe (~> 0.1)
    unicode-display_width (1.3.0)
    url (0.3.2)

PLATFORMS
  ruby

DEPENDENCIES
  bundler (~> 1.16)
  codecov
  pry
  rake (~> 10.0)
  rspec (~> 3.0)
  rubocop (~> 0.54.0)
  sql_assess!
  timecop

BUNDLED WITH
   1.16.1
```

LICENSE.txt

README.md

```
[![Build
 ↪  Status](https://travis-ci.com/vladstoick/fyp_sql_assess.svg?token=eN2aKbJE6VKG7yYiEpbw&branch=master)](https:/
[![codecov](https://codecov.io/gh/vladstoick/fyp_sql_assess/branch/master/graph/badge.svg?token=M4iqHixHb7)](https

# SqlAssess

Source code for the grading library. Please refer to the table of contents from
the appendix for explanation of various files.
```

Rakefile

```
require "bundler/gem_tasks"
require "rspec/core/rake_task"

RSpec::Core::RakeTask.new(:spec)

task :default => :spec
```

codecov.yml

```
ignore:
  - spec/**/*
```

lib/sql_assess.rb

```
# frozen_string_literal: true

require 'sql_assess/version'
require 'sql_assess/error'
require 'sql_assess/assesor'
require 'active_support/all'

# The namespace of the library. The public interface is provided by #{Assesor}
module SqlAssess; end
```

lib/sql_assess/assesor.rb

```
# frozen_string_literal: true

require 'sql_assess/database_connection'
require 'sql_assess/runner'
require 'sql_assess/query_comparator'
require 'sql_assess/query_transformer'
require 'sql_assess/data_extractor'
require 'sql_assess/query_attribute_extractor'

module SqlAssess
  # Public interface of the library
  # @author
  class Assesor
    attr_reader :connection

    # @raise [DatabaseSchemaError] if any MySQL errors are encountered
    def initialize(database_host: '127.0.0.1', database_port: '3306', database_username: 'root',
     ↪  database_password: '')
      @connection = SqlAssess::DatabaseConnection.new(
        host: database_host,
        port: database_port,
        username: database_username,
        password: database_password
```

```ruby
    )
  end

  # Compile an assignment
  # @param [String] create_schema_sql_query
  # @param [String] instructor_sql_query
  # @param [String] seed_sql_query
  # @return [Hash] see {DataExtractor#run}
  # @raise [DatabaseSeedError]
  #   if any MySQL errors are encountered while seeding the database
  # @raise [DatabaseSchemaError] if any MySQL errors are encounted
  #   while creating the schema
  # @raise [DatabaseQueryExecutionFailed] if any MySQL errors are
  #   encountered while running the instructor query
  def compile(create_schema_sql_query:, instructor_sql_query:, seed_sql_query:)
    create_database(create_schema_sql_query, seed_sql_query)

    Runner.new(@connection).execute_query(instructor_sql_query)

    QueryTransformer.new(@connection).transform(instructor_sql_query)

    DataExtractor.new(@connection).run
  ensure
    clear_database
  end

  # Assess an assignment
  # @param [String] create_schema_sql_query
  # @param [String] instructor_sql_query
  # @param [String] seed_sql_query
  # @return [QueryComparisonResult]
  # @raise [DatabaseSeedError]
  #   if any MySQL errors are encountered while seeding the database
  # @raise [DatabaseSchemaError] if any MySQL errors are encounted
  #   while creating the schema
  # @raise [DatabaseQueryExecutionFailed] if any MySQL errors are
  #   encountered while running the instructor query or student's query
  def assess(create_schema_sql_query:, instructor_sql_query:, seed_sql_query:, student_sql_query:)
    create_database(create_schema_sql_query, seed_sql_query)

    # Try to compile
    Runner.new(@connection).execute_query(student_sql_query)

    query_result_match = QueryComparator.new(@connection)
                                         .compare(instructor_sql_query, student_sql_query)

    transformer = QueryTransformer.new(@connection)
    instructor_sql_query = transformer.transform(instructor_sql_query)
    student_sql_query = transformer.transform(student_sql_query)

    attributes = QueryAttributeExtractor.new.extract(
      instructor_sql_query, student_sql_query
    )

    QueryComparisonResult.new(
      success: query_result_match,
      attributes: attributes
    )
  ensure
    clear_database
  end

  private
```

```
    def create_database(create_schema_sql_query, seed_sql_query)
      SqlAssess::Runner.new(@connection).create_schema(
        create_schema_sql_query
      )

      SqlAssess::Runner.new(@connection).seed_initial_data(
        seed_sql_query
      )
    end

    def clear_database
      @connection.delete_database
    end
  end
end
```

lib/sql_assess/data_extractor.rb

```
# frozen_string_literal: true

require 'mysql2'

module SqlAssess
  # Class for handling the extraction of data and schema from a database
  # @author Vlad Stoica
  class DataExtractor
    def initialize(connection)
      @connection = connection
    end

    # Extract data from the current connection
    # @return [Hash] data from the table. The format of the hash is { table_name: [rows] }
    def run
      result = []
      tables = @connection.query('SHOW tables;')

      tables.each do |table|
        table_name = table.first.last

        data = @connection.query("SELECT * from #{table_name}")
        columns = @connection.query("SHOW columns from #{table_name}").to_a.map do |column|
          {
            name: column.fetch('Field'),
            type: column.fetch('Type'),
          }
        end

        result << {
          name: table_name,
          columns: columns,
          data: data.to_a,
        }
      end

      result
    end
  end
end
```

lib/sql_assess/database_connection.rb

```
# frozen_string_literal: true

require 'mysql2'
```

```ruby
module SqlAssess
  # Class for handling database connection and securely executing queries
  # @author Vlad Stoica
  class DatabaseConnection
    def initialize(host: '127.0.0.1', port: '3306', username: 'root', database: nil, password: '')
      @client = Mysql2::Client.new(
        host: host,
        port: port,
        username: username,
        password: password,
        flags: Mysql2::Client::MULTI_STATEMENTS
      )

      if database.present?
        @parent_database = true
        @database = database
        @client.query("CREATE DATABASE IF NOT EXISTS `#{@database}`")
      else
        success = false
        attempt = 0
        until success
          if attempt.positive?
            @database = "#{Time.now.strftime('%H%M%S')}_#{attempt}"
          else
            @database = Time.now.strftime('%H%M%S').to_s
          end

          begin
            @client.query("CREATE DATABASE `#{@database}`")
            success = true
          rescue Mysql2::Error => exception
            raise exception unless exception.message.include?('database exists')
            success = false
            attempt += 1
          end
        end
      end

      @client.query("CREATE USER IF NOT EXISTS `#{@database}`;")
      @client.query("GRANT ALL PRIVILEGES ON `#{@database}`.* TO `#{@database}` WITH GRANT OPTION;")

      @restricted_client = Mysql2::Client.new(
        host: host,
        port: port,
        username: @database,
        flags: Mysql2::Client::MULTI_STATEMENTS
      )

      @restricted_client.select_db(@database)
      @client.select_db(@database)
    rescue Mysql2::Error => exception
      raise DatabaseConnectionError, exception.message
    end

    # Execute queries as restricted user
    # @param [String] query
    # @return [Hash] the results of the query
    def query(query)
      @restricted_client.query(query)
    end

    # Drop the temporary database and the temporary user
    def delete_database
      if @parent_database
        # disable foreign key checks before dropping the database
        @client.query('SET FOREIGN_KEY_CHECKS = 0')

        tables = query('SHOW tables')

        tables.each do |table|
          table_name = table['Tables_in_local_db']
          @client.query("DROP table #{table_name}")
        end

        @client.query('SET FOREIGN_KEY_CHECKS = 1')
      else
        @client.query("DROP DATABASE `#{@database}`")
        @client.query("DROP USER IF EXISTS `#{@database}`")
      end
    end

    # Execute a multi statement query as restricted user
    # @param [String] query
    # @return [Array<Hash>] the results of each statement
    def multiple_query(query)
      result = []

      result << @restricted_client.query(query)

      while @restricted_client.next_result
        result << @restricted_client.store_result
      end

      result
    end
  end
end
```

lib/sql_assess/error.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  # Base class for errors from the library
  # @author Vlad Stoica
  class Error < StandardError
  end

  # Error thrown when the library can't connect to the database
  # @author Vlad Stoica
  class DatabaseConnectionError < SqlAssess::Error
  end

  # Error thrown when the library encounters an error while executing the schema query
  # @author Vlad Stoica
  class DatabaseSchemaError < SqlAssess::Error
  end

  # Error thrown when the library encounters an error while executing the seed query
  # @author Vlad Stoica
  class DatabaseSeedError < SqlAssess::Error
  end

  # Error thrown when the library encounters an error while executing the instructor's or student's query
  # @author Vlad Stoica
  class DatabaseQueryExecutionFailed < SqlAssess::Error
  end

  # Error thrown when the library cannot canonicalize a query
  # @author Vlad Stoica
  class CanonicalizationError < SqlAssess::Error
  end
end
```

lib/sql_assess/grader/base.rb

```
# frozen_string_literal: true

require 'rubygems/text'

module SqlAssess
  # Namespace that handles the grading part of the library
  module Grader
    # Base class for the grader
    # @author Vlad Stoica
    class Base
      # Returns the grade for a certain attribute given a list of attributes
      # @param [String] attribute component name (e.g. columns)
      # @param [Hash] student_attributes student's attributes for that component
      # @param [Hash] instructor_attributes instructor's attributes for that component
      def self.grade_for(attribute:, student_attributes:, instructor_attributes:)
        "SqlAssess::Grader::#{attribute.to_s.camelcase}".constantize.new(
          student_attributes: student_attributes,
          instructor_attributes: instructor_attributes
        ).rounded_grade
      end

      def initialize(student_attributes:, instructor_attributes:)
        @student_attributes = student_attributes
        @instructor_attributes = instructor_attributes
      end

      # The levenshtein distance between two strings
      # @param [String] string1
      # @param [String] string2
      # @return [Integer] the distance
      def levenshtein_distance(string1, string2)
        ld = Class.new.extend(Gem::Text).method(:levenshtein_distance)

        ld.call(string1, string2)
      end

      # Rounds the grade to two decimals. The subclasses must implement the
      # grade method.
      # @return [Double] rounded grade to two decimals
      def rounded_grade
        grade.round(2)
      end

      private

      def grade_for_array(instructor_attributes = @instructor_attributes, student_attributes =
      ↪  @student_attributes)
        max_grade = (student_attributes.length + instructor_attributes.length).to_d
        return 1 if max_grade.zero?

        instructor_unmatched_attributes = instructor_attributes.dup
        student_unmatched_attributes = student_attributes.dup

        student_unmatched_attributes = student_unmatched_attributes.keep_if do |student_unmatched_attribute|
          next 0 if instructor_unmatched_attributes.empty?

          match_score = instructor_unmatched_attributes.map do |instructor_unmatched_attribute|
            match_score(student_unmatched_attribute, instructor_unmatched_attribute)
          end

          best_match_score = match_score.each_with_index.max

          if best_match_score[0] == 1
            instructor_unmatched_attributes.delete_at(best_match_score[1])
            false
          else
            true
          end
        end

        matched_attributes = array_difference(
          student_attributes,
          student_unmatched_attributes
        )

        matched_grade = matched_attributes.length * 2.0

        unmatched_grade = student_unmatched_attributes.sum do |student_unmatched_attribute|
          next 0 if instructor_unmatched_attributes.empty?

          match_score = instructor_unmatched_attributes.map do |instructor_unmatched_attribute|
            match_score(student_unmatched_attribute, instructor_unmatched_attribute)
          end

          best_match_score = match_score.each_with_index.max

          if best_match_score[0].positive?
            instructor_unmatched_attributes.delete_at(best_match_score[1])
          end

          best_match_score[0]
        end

        (matched_grade + unmatched_grade) / max_grade
      end

      # Difference with removing only once solution obtained from
      # https://stackoverflow.com/questions/30429659/ruby-difference-in-array-including-duplicates
      def array_difference(array1, array2)
        array1 = array1.dup
        array2.each { |del| array1.slice!(array1.index(del)) if array1.include?(del) }
        array1
      end
    end
  end
end

require_relative 'columns'
require_relative 'order_by'
require_relative 'where'
require_relative 'distinct_filter'
require_relative 'limit'
require_relative 'tables'
require_relative 'group'
require_relative 'having'
```

lib/sql_assess/grader/columns.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Grader
    # Grader for columns
    # @author Vlad Stoica
    class Columns < Base
      private

      def grade
        grade_for_array
      end

      def match_score(column1, column2)
        table_name1, column_name1 = column1.split('.')
        table_name2, column_name2 = column2.split('.')

        if table_name1 == table_name2
          1.0 / (levenshtein_distance(column_name1, column_name2) + 1)
        else
          0
        end
      end
    end
  end
end
```

lib/sql_assess/grader/distinct_filter.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Grader
    # Grader for distinct filter
    # @author Vlad Stoica
    class DistinctFilter < Base
      def initialize(student_attributes:, instructor_attributes:)
        @student_distinct = student_attributes
        @instructor_distinct = instructor_attributes
      end

      private

      def grade
        if @student_distinct == @instructor_distinct
          1.0
        elsif @student_distinct == 'DISTINCT' && @instructor_distinct == 'DISTINCTROW'
          0.5
        elsif @student_distinct == 'DISTINCTROW' && @instructor_distinct == 'DISTINCT'
          0.5
        else
          0
        end
      end
    end
  end
end
```

lib/sql_assess/grader/group.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Grader
    # Grader for GROUP clause
    # @author Vlad Stoica
    class Group < Base
      private

      def grade
        grade_for_array
      end

      def match_score(column1, column2)
        table_name1, column_name1 = column1.split('.')
        table_name2, column_name2 = column2.split('.')

        if table_name1 == table_name2
          1.0 / (levenshtein_distance(column_name1, column_name2) + 1)
        else
          0
        end
      end
    end
  end
end
```

lib/sql_assess/grader/having.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Grader
    # Grader for HAVING clause
    # @author Vlad Stoica
    class Having < Base
      def initialize(student_attributes:, instructor_attributes:)
        @student_having = student_attributes
        @instructor_having = instructor_attributes
      end

      private

      def grade
        return 1 if @student_having == @instructor_having

        return 0 if @student_having == {} || @instructor_having == {}

        # Partial grading

        student_leaves = [get_leaves(@student_having)].flatten
        instructor_leaves = [get_leaves(@instructor_having)].flatten

        conditions_grade = grade_for_array(student_leaves, instructor_leaves)

        internal_nodes = internal_count(@student_having) + internal_count(@instructor_having)

        if internal_nodes.positive?
          tree_grade = grade_for_tree(@student_having, @instructor_having).to_d / internal_nodes
          (conditions_grade + tree_grade) / 2
        else
          conditions_grade
        end
      end

      def grade_for_tree(student_tree, instructor_tree)
        if student_tree && student_tree[:is_inner] && instructor_tree && instructor_tree[:is_inner]
          current_grade = grade_for_node(student_tree, instructor_tree)
          child_node_grade_as_normal = grade_for_tree(student_tree[:left_clause],
           ↪  instructor_tree[:left_clause]) +
                                       grade_for_tree(student_tree[:right_clause],
                                        ↪  instructor_tree[:right_clause])

          child_node_grade_as_reversed = grade_for_tree(student_tree[:left_clause],
           ↪  instructor_tree[:right_clause]) +
                                         grade_for_tree(student_tree[:right_clause],
                                          ↪  instructor_tree[:left_clause])

          child_grade = [
            child_node_grade_as_normal,
            child_node_grade_as_reversed,
          ].max

          current_grade + child_grade
        else
          0
        end
      end

      def internal_count(having_clause)
        if having_clause && having_clause[:is_inner]
          1 + internal_count(having_clause[:left_clause]) + internal_count(having_clause[:right_clause])
        else
          0
        end
      end

      def grade_for_node(student_tree, instructor_tree)
        if student_tree[:type] == instructor_tree[:type]
          2
        else
          0
        end
      end

      def get_leaves(having_clause)
        if having_clause.nil?
          nil
        elsif having_clause[:is_inner] == false
          having_clause
        else
          [
            get_leaves(having_clause[:left_clause]),
            get_leaves(having_clause[:right_clause]),
          ].flatten
        end
      end

      def match_score(having_clause1, having_clause2)
        if having_clause1 == having_clause2
          2
        else
          0
        end
      end
    end
  end
end
```

lib/sql_assess/grader/limit.rb

```
# frozen_string_literal: true
```

```ruby
module SqlAssess
  module Grader
    # Grader for LIMIT clause
    # @author Vlad Stoica
    class Limit < Base
      def initialize(student_attributes:, instructor_attributes:)
        @student_limit = student_attributes
        @instructor_limit = instructor_attributes
      end

      private

      def grade
        grade = 0

        grade += 0.5 if @student_limit[:limit] == @instructor_limit[:limit]

        grade += 0.5 if @student_limit[:offset] == @instructor_limit[:offset]

        grade
      end
    end
  end
end
```

lib/sql_assess/grader/order_by.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Grader
    # Grader for ORDER BY clause
    # @author Vlad Stoica
    class OrderBy < Base
      private

      def grade
        grade_for_array
      end

      def match_score(order_by1, order_by2)
        column1, order1 = order_by1[:column].split(' ')
        column2, order2 = order_by2[:column].split(' ')
        position_difference = (order_by1[:position] - order_by2[:position]).abs + 1

        if column1 == column2
          if order1 == order2
            1.0 / position_difference
          else
            0.5 / position_difference
          end
        else
          0
        end
      end
    end
  end
end
```

lib/sql_assess/grader/tables.rb

```ruby
# frozen_string_literal: true

module SqlAssess
```

```ruby
  module Grader
    # Grader for FROM clause
    # @author Vlad Stoica
    class Tables < Base
      private

      def grade
        if @instructor_attributes.length == 1 && @student_attributes.length == 1
          compare_base_grade
        else
          joins_grade = grade_for_array(
            @instructor_attributes.drop(1),
            @student_attributes.drop(1)
          )

          (joins_grade + compare_base_grade) / 2
        end
      end

      def compare_base_grade
        instructor_condition = @instructor_attributes.first
        student_condition = @student_attributes.first

        if instructor_condition == student_condition
          1
        else
          0
        end
      end

      def match_score(instructor_join, student_expressions)
        if instructor_join == student_expressions
          1
        elsif instructor_join[:table] == student_expressions[:table]
          if instructor_join[:join_type] == student_expressions[:join_type]
            0.75
          elsif instructor_join[:condition] == student_expressions[:condition]
            0.75
          else
            0.5
          end
        else
          0
        end
      end
    end
  end
end
```

lib/sql_assess/grader/where.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Grader
    # Grader for WHERE clause
    # @author Vlad Stoica
    class Where < Base
      def initialize(student_attributes:, instructor_attributes:)
        @student_where = student_attributes
        @instructor_where = instructor_attributes
      end

      private
```

```ruby
def grade
  return 1 if @student_where == @instructor_where

  return 0 if @student_where == {} || @instructor_where == {}

  # Partial grading

  student_leaves = [get_leaves(@student_where)].flatten
  instructor_leaves = [get_leaves(@instructor_where)].flatten

  conditions_grade = grade_for_array(student_leaves, instructor_leaves)

  internal_nodes = internal_count(@student_where) + internal_count(@instructor_where)

  if internal_nodes.positive?
    tree_grade = grade_for_tree(@student_where, @instructor_where).to_d / internal_nodes
    (conditions_grade + tree_grade) / 2
  else
    conditions_grade
  end
end

def grade_for_tree(student_tree, instructor_tree)
  if student_tree && student_tree[:is_inner] && instructor_tree && instructor_tree[:is_inner]
    current_grade = grade_for_node(student_tree, instructor_tree)

    child_node_grade_as_normal = grade_for_tree(student_tree[:left_clause],
     ↪  instructor_tree[:left_clause]) +
                                 grade_for_tree(student_tree[:right_clause],
                                  ↪  instructor_tree[:right_clause])

    child_node_grade_as_reversed = grade_for_tree(student_tree[:left_clause],
     ↪  instructor_tree[:right_clause]) +
                                   grade_for_tree(student_tree[:right_clause],
                                    ↪  instructor_tree[:left_clause])

    child_grade = [
      child_node_grade_as_normal,
      child_node_grade_as_reversed,
    ].max

    current_grade + child_grade
  else
    0
  end
end

def internal_count(where_clause)
  if where_clause && where_clause[:is_inner]
    1 + internal_count(where_clause[:left_clause]) + internal_count(where_clause[:right_clause])
  else
    0
  end
end

def grade_for_node(student_tree, instructor_tree)
  if student_tree[:type] == instructor_tree[:type]
    2
  else
    0
  end
end
```

```
      def get_leaves(node)
        if node.nil?
          nil
        elsif node[:is_inner] == false
          node
        else
          [
            get_leaves(node[:left_clause]),
            get_leaves(node[:right_clause]),
          ].flatten
        end
      end

      def match_score(where_clause1, where_clause2)
        if where_clause1 == where_clause2
          2
        else
          0
        end
      end
    end
  end
end
```

lib/sql_assess/parsers/base.rb

```
# frozen_string_literal: true

module SqlAssess
  # Namespace that handles the components extraction
  module Parsers
    # Base class for the parsers
    # @author Vlad Stoica
    class Base
      def initialize(query)
        @parsed_query = SQLParser::Parser.new.scan_str(query)
      end
    end
  end
end

require_relative 'columns'
require_relative 'order_by'
require_relative 'where'
require_relative 'tables'
require_relative 'distinct_filter'
require_relative 'limit'
require_relative 'group'
require_relative 'having'
```

lib/sql_assess/parsers/columns.rb

```
# frozen_string_literal: true

module SqlAssess
  module Parsers
    # @author Vlad Stoica
    # Parser for the columns
    class Columns < Base
      # @return [Array<String>] the list of columns selected
      def columns
        @parsed_query.query_expression.list.columns.map(&:to_sql)
      end
    end
```

```ruby
  end
end
```

lib/sql_assess/parsers/distinct_filter.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Parsers
    # Parser for the distinct filter
    # @author Vlad Stoica
    class DistinctFilter < Base
      # @return [String] distinct filter or ALL if no distinc filter is mentioned
      def distinct_filter
        @parsed_query.query_expression.filter || 'ALL'
      end
    end
  end
end
```

lib/sql_assess/parsers/group.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Parsers
    # Parser for the GROUP clause
    # @author Vlad Stoica
    class Group < Base
      # @return [Array<String>] the list of columns in the group clause
      def group
        if @parsed_query.query_expression.table_expression.group_by_clause.nil?
          []
        else
          @parsed_query.query_expression.table_expression.group_by_clause.columns.map(&:to_sql)
        end
      end
    end
  end
end
```

lib/sql_assess/parsers/having.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Parsers
    # @author Vlad Stoica
    # Parser for the HAVING clause
    class Having < Base
      # @return [Hash] the binary expression tree of the HAVING clause
      def having
        if @parsed_query.query_expression.table_expression.having_clause.nil?
          {}
        else
          self.class.transform(@parsed_query.query_expression.table_expression.having_clause.search_condition)
        end
      end

      # @return [Hash] the expression tree (not binary tree) of the HAVING clause
      def having_tree
        if @parsed_query.query_expression.table_expression.having_clause.nil?
          {}
        else
          transform_tree(
            @parsed_query.query_expression.table_expression.having_clause.search_condition
          )
        end
      end

      # Transform a clause to a tree
      # @param [SQLParser::Statement] clause current node
      # @return [Hash] tree version the clause
      def self.transform(clause)
        if clause.is_a?(SQLParser::Statement::ComparisonPredicate)
          {
            type: clause.class.name.split('::').last.underscore.humanize.upcase,
            left: clause.left.to_sql,
            right: clause.right.to_sql,
            sql: clause.to_sql,
          }
        elsif clause.is_a?(SQLParser::Statement::SearchCondition)
          type = clause.class.name.split('::').last.underscore.humanize.upcase
          transform_left = merge(type, transform(clause.left))
          transform_right = merge(type, transform(clause.right))
          {
            type: type,
            clauses: [
              transform_left,
              transform_right,
            ].flatten,
          }
        end
      end

      def self.merge(type, clause)
        if clause[:type] == type
          clause[:clauses]
        else
          clause
        end
      end
      private_class_method :merge

      private

      def transform_tree(clause)
        if clause.is_a?(SQLParser::Statement::ComparisonPredicate)
          {
            is_inner: false,
            type: clause.class.name.split('::').last.underscore.humanize.upcase,
            left: clause.left.to_sql,
            right: clause.right.to_sql,
            sql: clause.to_sql,
          }
        elsif clause.is_a?(SQLParser::Statement::SearchCondition)
          type = clause.class.name.split('::').last.underscore.humanize.upcase

          {
            is_inner: true,
            type: type,
            left_clause: transform_tree(clause.left),
            right_clause: transform_tree(clause.right),
          }
        end
      end
    end
  end
```

```
end
```

lib/sql_assess/parsers/limit.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Parsers
    # Parser for the limit clause
    # @author Vlad Stoica
    class Limit < Base
      # @return [Hash{limit:, offset:}]. If offset is not present then return 0,
      #   if limit is not present then return inf.
      def limit
        if @parsed_query.query_expression&.table_expression&.limit_clause.present?
          {
            limit: @parsed_query.query_expression&.table_expression&.limit_clause&.limit,
            offset: @parsed_query.query_expression&.table_expression&.limit_clause&.offset || 0,
          }
        else
          {
            limit: 'inf',
            offset: 0,
          }
        end
      end
    end
  end
end
```

lib/sql_assess/parsers/order_by.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Parsers
    # Parser for the Order BY clause
    # @author Vlad Stoica
    class OrderBy < Base
      # @return [Array<Hash{column:, position:}>]
      def order
        if @parsed_query.order_by.nil?
          []
        else
          @parsed_query.order_by.sort_specification.each_with_index.map do |column, i|
            {
              column: column.to_sql,
              position: i,
            }
          end
        end
      end
    end
  end
end
```

lib/sql_assess/parsers/tables.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Parsers
    # Parser for the FROM clause
    # @author Vlad Stoica
    class Tables < Base
```

```ruby
      # @return [Array<Hash{type:, table:, sql:}, Hash{join_type:,table: Hash{type:, table:, sql:}, sql:}>]
      def tables
        if @parsed_query.query_expression&.table_expression&.from_clause.nil?
          []
        else
          @parsed_query.query_expression.table_expression.from_clause.tables.map do |expression|
            transform(expression)
          end.flatten
        end
      end

      private

      def transform(query)
        if query.is_a?(SQLParser::Statement::Table)
          {
            type: 'table',
            table: query.to_sql,
            sql: query.to_sql,
          }
        elsif query.is_a?(SQLParser::Statement::JoinedTable)
          hash = {
            join_type: query.class.name.split('::').last.underscore.humanize.upcase,
            table: transform(query.right),
            sql: "#{query.class.name.split('::').last.underscore.humanize.upcase} #{query.right.to_sql}",
          }

          if query.is_a?(SQLParser::Statement::QualifiedJoin)
            hash[:condition] = Where.transform(
              query.search_condition.search_condition
            )
            hash[:sql] = "#{query.class.name.split('::').last.underscore.humanize.upcase}
             ↪  #{query.right.to_sql} #{query.search_condition.to_sql}"
          end

          [transform(query.left), hash].flatten
        elsif query.is_a?(SQLParser::Statement::Subquery)
          {
            type: 'Subquery',
            sql: query.to_sql,
            attributes:
             ↪  SqlAssess::QueryAttributeExtractor.new.extract_query(query.query_specification.to_sql),
          }
        end
      end
    end
  end
end
```

lib/sql_assess/parsers/where.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Parsers
    # @author Vlad Stoica
    # Parser for the WHERE clause
    class Where < Base
      # @return [Hash] the binary expression tree of the WHERE clause
      def where
        if @parsed_query.query_expression.table_expression.where_clause.nil?
          {}
        else
          self.class.transform(@parsed_query.query_expression.table_expression.where_clause.search_condition)
        end
      end

      # @return [Hash] the expression tree (not binary tree) of the WHERE clause
      def where_tree
        if @parsed_query.query_expression.table_expression.where_clause.nil?
          {}
        else
          transform_tree(
            @parsed_query.query_expression.table_expression.where_clause.search_condition
          )
        end
      end

      # Transform a clause to a tree
      # @param [SQLParser::Statement] clause current node
      # @return [Hash] tree version the clause
      def self.transform(clause)
        if clause.is_a?(SQLParser::Statement::ComparisonPredicate)
          {
            type: clause.class.name.split('::').last.underscore.humanize.upcase,
            left: clause.left.to_sql,
            right: clause.right.to_sql,
            sql: clause.to_sql,
          }
        elsif clause.is_a?(SQLParser::Statement::SearchCondition)
          type = clause.class.name.split('::').last.underscore.humanize.upcase
          transform_left = merge(type, transform(clause.left))
          transform_right = merge(type, transform(clause.right))
          {
            type: type,
            clauses: [
              transform_left,
              transform_right,
            ].flatten,
          }
        end
      end

      def self.merge(type, clause)
        if clause[:type] == type
          clause[:clauses]
        else
          clause
        end
      end
      private_class_method :merge

      private

      def transform_tree(clause)
        if clause.is_a?(SQLParser::Statement::ComparisonPredicate)
          {
            is_inner: false,
            type: clause.class.name.split('::').last.underscore.humanize.upcase,
            left: clause.left.to_sql,
            right: clause.right.to_sql,
            sql: clause.to_sql,
          }
        elsif clause.is_a?(SQLParser::Statement::SearchCondition)
          type = clause.class.name.split('::').last.underscore.humanize.upcase

          {
            is_inner: true,
            type: type,
            left_clause: transform_tree(clause.left),
            right_clause: transform_tree(clause.right),
          }
        end
      end
    end
  end
end
```

lib/sql_assess/query_attribute_extractor.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  # Class for handling the attribute extraction process
  # @author Vlad Stoica
  class QueryAttributeExtractor
    # Extract the attributes of both the instructor's and student's queries
    #
    # @param [String] instructor_sql_query
    # @param [String] student_sql_query
    # @return [Hash] with two keys student and instructor. Each value has the format
    #   returned by {#extract_query}
    def extract(instructor_sql_query, student_sql_query)
      {
        student: extract_query(student_sql_query),
        instructor: extract_query(instructor_sql_query),
      }
    end

    # Extract the attributes of a query
    # @param [String] query
    # @return [Hash] that contains all attributes of a query.
    def extract_query(query)
      {
        columns: Parsers::Columns.new(query).columns,
        order_by: Parsers::OrderBy.new(query).order,
        where: Parsers::Where.new(query).where,
        where_tree: Parsers::Where.new(query).where_tree,
        tables: Parsers::Tables.new(query).tables,
        distinct_filter: Parsers::DistinctFilter.new(query).distinct_filter,
        limit: Parsers::Limit.new(query).limit,
        group: Parsers::Group.new(query).group,
        having: Parsers::Having.new(query).having,
        having_tree: Parsers::Having.new(query).having_tree,
      }
    end
  end
end
```

lib/sql_assess/query_comparator.rb

```ruby
# frozen_string_literal: true

require 'sql_assess/query_comparison_result'
require 'sql_assess/parsers/base'

module SqlAssess
  # Class for handling the comparison of results between two queries
  # @author Vlad Stoica
  class QueryComparator
    def initialize(connection)
      @connection = connection
    end

    # Compares the results of two queries
    #
    # @param [String] instructor_sql_query
    # @param [String] student_sql_query
    # @return [Boolean] whether the result matches
    def compare(instructor_sql_query, student_sql_query)
      instructor_result = @connection.query(instructor_sql_query).to_a
      student_result = @connection.query(student_sql_query).to_a

      success?(instructor_result, student_result)
    end

    private

    def success?(instructor_result, student_result)
      return false if instructor_result.count != student_result.count

      (0..instructor_result.count).all? do |i|
        instructor_result[i] == student_result[i]
      end
    end
  end
end
```

lib/sql_assess/query_comparison_result.rb

```ruby
# frozen_string_literal: true

require 'sql_assess/grader/base'

module SqlAssess
  # @author Vlad Stoica
  # The final result of an assesment
  # @!attribute [r] success
  #   @return [Boolean] whether the query returned the same results
  # @!attribute [r] attributes
  #   @return [Hash] The extracted attributes of the two queries. See
  #     {QueryAttributeExtractor}
  # @!attribute [r] attributes_grade
  #   @return [Hash] The grade for each component
  # @!attribute [r] grade
  #   @return [Double] The overall grade
  # @!attribute [r] message
  #   @return [String] Hint
  class QueryComparisonResult
    attr_reader :success, :attributes, :grade, :message

    def initialize(success:, attributes:)
      @success = success
      @attributes = attributes

      attributes_grade

      if @success == true
        @grade = calculate_grade * 100.00
      else
        @grade = calculate_grade * 90.00
      end

      @message = determine_hints
    end
```

```ruby
def attributes_grade
  @attributes_grade ||= grade_components_percentages.keys.map do |key|
    key_hash = key == :where ? :where_tree : key
    [
      key,
      SqlAssess::Grader::Base.grade_for(
        attribute: key,
        student_attributes: attributes[:student][key_hash],
        instructor_attributes: attributes[:instructor][key_hash]
      ).to_d,
    ]
  end.to_h
end

private

def calculate_grade
  attributes_grade.sum do |attribute, grade|
    grade * grade_components_percentages[attribute]
  end
end

def grade_components_percentages
  {
    tables: 1 / 8.0,
    columns: 1 / 8.0,
    group: 1 / 8.0,
    where: 1 / 8.0,
    distinct_filter: 1 / 8.0,
    limit: 1 / 8.0,
    order_by: 1 / 8.0,
    having: 1 / 8.0,
  }
end

def determine_hints
  if @grade == 100.00
    'Congratulations! Your solution is correct'
  else
    "Your query is not correct. #{message_for_attribute(first_wrong_component)}"
  end
end

def first_wrong_component
  comp = grade_components_percentages.detect do |component, _|
    attributes_grade[component].to_d != 1
  end

  comp.present? ? comp.first : nil
end

def message_for_attribute(attribute)
  case attribute
  when :columns then 'Check what columns you are selecting.'
  when :tables then 'Are you sure you are selecting the right tables?'
  when :order_by then 'Are you ordering the rows correctly?'
  when :where then 'Looks like you are selecting the right columns, but you are not selecting only the
   ↪  correct rows.'
  when :distinct_filter then 'What about duplicates? What does the exercise say?'
  when :limit then 'Are you selecting the correct number of rows?'
  when :group then 'Are you grouping by the correct columns?'
  end
end
```

```
  end
end
```

lib/sql_assess/query_transformer.rb

```ruby
# frozen_string_literal: true

require 'sql_assess/transformers/base'

module SqlAssess
  # Class for handling the canonicalization process
  # @author Vlad Stoica
  class QueryTransformer
    # The ordered list of transformers applied
    TRANSFORMERS = [
      # Subquery
      Transformers::FromSubquery,
      # Predicate
      Transformers::Not::Base.transformers,
      Transformers::BetweenPredicate::Base.transformers,
      Transformers::ComparisonPredicate::Base.transformers,
      # Columns
      Transformers::AllColumns,
      Transformers::AmbigousColumns::Base.transformers,
      Transformers::EquivalentColumns::Base.transformers,
    ].flatten.freeze

    def initialize(connection)
      @connection = connection
    end

    # Apply sequentially all transformations to a query
    #
    # @param [String] query input query
    # @return [String] canonicalized query
    # @raise [CanonicalizationError] if any parsing errors are encountered
    def transform(query)
      TRANSFORMERS.each do |transformer_class|
        query = transformer_class.new(@connection).transform(query)
      end

      query
    rescue SQLParser::Parser::ScanError, Racc::ParseError
      raise CanonicalizationError
    end
  end
end
```

lib/sql_assess/runner.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  # @author Vlad Stoica
  # A class for executing various types of queries. By providing a method
  # for each type of query, an appropiate error can be returned.
  class Runner
    def initialize(connection)
      @connection = connection
    end

    # Execute the create schema SQL query
    #
    # @param [String] create_schema_sql_query
    # @return [Hash] the results of the query
    # @raise [DatabaseSchemaError] if any MySQL errors are encountered
    def create_schema(create_schema_sql_query)
      @connection.multiple_query(create_schema_sql_query)
    rescue Mysql2::Error => exception
      raise DatabaseSchemaError, exception.message
    end

    # Execute the seed SQL query
    #
    # @param [String] seed_sql_query
    # @return [Hash] the results of the query
    # @raise [DatabaseSeedError] if any MySQL errors are encountered
    def seed_initial_data(seed_sql_query)
      @connection.multiple_query(seed_sql_query)
    rescue Mysql2::Error => exception
      raise DatabaseSeedError, exception.message
    end

    # Execute student's or instructors' query
    #
    # @param [String] sql_query
    # @return [Hash] the results of the query
    # @raise [DatabaseQueryExecutionFailed] if any MySQL errors are encountered
    def execute_query(sql_query)
      @connection.query(sql_query)
    rescue Mysql2::Error => exception
      raise DatabaseQueryExecutionFailed, exception.message
    end
  end
end
```

lib/sql_assess/transformers/all_columns.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    # @author Vlad Stoica
    # Transformer for transforming * to the full list of qulified columns.
    class AllColumns < Base
      # Transforms the query
      #
      # @param [String] query the initial query
      # @return [String] the transformed query
      #
      # @example
      #   With tables: t1(id1), t2(id3);
      #   "SELECT * FROM `t1`, `t2`"
      #   is transformed
      #   to "SELECT `t1`.`id1`, `t2`.`id3` FROM `t1`, `t2`"
      #
      # @example
      #   With tables: t1(id1), t2(id3);
      #   "SELECT `t1`.`id1` FROM `t1`, `t2`"
      #   is transformed
      #   to "SELECT `t1`.`id1` FROM `t1`, `t2`"
      def transform(query)
        @parsed_query = @parser.scan_str(query)

        if @parsed_query.query_expression.list.is_a?(SQLParser::Statement::All)
          transform_star_select
        end

        @parsed_query.to_sql
      end

      private

      def transform_star_select
        table_list = tables(@parsed_query.to_sql)

        new_columns = table_list.map do |table|
          columns_query = "SHOW COLUMNS from #{table}"
          columns = @connection.query(columns_query).map { |k| k['Field'] }

          columns.map do |column_name|
            SQLParser::Statement::QualifiedColumn.new(
              SQLParser::Statement::Table.new(table),
              SQLParser::Statement::Column.new(column_name)
            )
          end
        end.flatten

        @parsed_query.query_expression.instance_variable_set(
          '@list',
          SQLParser::Statement::SelectList.new(new_columns)
        )
      end
    end
  end
end
```

lib/sql_assess/transformers/ambigous_columns/base.rb

```
# frozen_string_literal: true

module SqlAssess
  module Transformers
    # Module for ambigous columns transformers
    module AmbigousColumns
      # @author Vlad Stoica
      # Base class for transformers for ambiguous column. Provides implementation
      # for transforming columns
      class Base < SqlAssess::Transformers::Base
        # The list of ambiguous columns transformers
        def self.transformers
          [Select, From, Where, Group, OrderBy, Having]
        end

        private

        def transform_column(column)
          if column.is_a?(SQLParser::Statement::Column)
            table = find_table_for(column.name)

            SQLParser::Statement::QualifiedColumn.new(
              SQLParser::Statement::Table.new(table),
              column
            )
          elsif column.is_a?(SQLParser::Statement::Aggregate) &&
           ↪  column.column.is_a?(SQLParser::Statement::Column)
            column.class.new(transform_column(column.column))
          elsif column.is_a?(SQLParser::Statement::Arithmetic) ||
           ↪  column.is_a?(SQLParser::Statement::ComparisonPredicate)
            column.class.new(
              transform_column(column.left),
              transform_column(column.right)
            )
```

```ruby
          else
            column
          end
        end

        def transform_column_integer(column)
          if column.is_a?(SQLParser::Statement::Integer)
            @parsed_query.query_expression.list.columns[column.value - 1]
          else
            transform_column(column)
          end
        end

        def transform_tree(node)
          if node.is_a?(SQLParser::Statement::SearchCondition)
            node.class.new(
              transform_tree(node.left),
              transform_tree(node.right)
            )
          else
            transform_column(node)
          end
        end

        def find_table_for(column_name)
          table_list = tables(@parsed_query.to_sql)

          table_list.detect do |table|
            columns_query = "SHOW COLUMNS from #{table}"
            columns = @connection.query(columns_query).map { |k| k['Field'] }
            columns.map(&:downcase).include?(column_name.downcase)
          end
        end
      end
    end
  end
end

require_relative 'from'
require_relative 'group'
require_relative 'order_by'
require_relative 'select'
require_relative 'where'
require_relative 'having'
```

lib/sql_assess/transformers/ambigous_columns/from.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module AmbigousColumns
      # @author Vlad Stoica
      # Ambiguous columns transformer for the FROM clause
      class From < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        #
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT `id1` FROM `t1` LEFT JOIN `t2` on `id1` = `id3`
        #   is transformed to
        #   SELECT `id1` FROM `t1` LEFT JOIN `t2` on `t1`.`id1` = `t2`.`id3`
        def transform(query)
          @parsed_query = @parser.scan_str(query)

          join_clause = @parsed_query.query_expression&.table_expression&.from_clause

          return query if join_clause.nil?

          new_tables = join_clause.tables.map do |table|
            traverse_from(table)
          end

          @parsed_query.query_expression.table_expression.from_clause.instance_variable_set(
            '@tables', new_tables
          )

          @parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/ambigous_columns/group.rb

```
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module AmbigousColumns
      # @author Vlad Stoica
      # Ambiguous columns transformer for the GROUP clause
      class Group < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        #
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT `id1`, SUM(`id3`) FROM `t1`, `t2` GROUP BY `id1`
        #   is transformed
        #   SELECT `id1`, SUM(`id3`) FROM `t1`, `t2` GROUP BY `t1`.`id1`
        #
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT `id1`, SUM(`id3`) FROM `t1`, `t2` GROUP BY 1
        #   is transformed to
        #   SELECT `id1`, SUM(`id3`) FROM `t1`, `t2` GROUP BY `t1`.`id1`
        def transform(query)
          @query = query

          @parsed_query = @parser.scan_str(query)

          if @parsed_query.query_expression.table_expression.group_by_clause.nil?
            return @parsed_query.to_sql
          end

          columns = @parsed_query.query_expression.table_expression.group_by_clause.columns.map do |column|
            transform_column_integer(column)
          end

          @parsed_query.query_expression.table_expression.group_by_clause.instance_variable_set(
            '@columns',
```

```ruby
            columns
          )

          @parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/ambigous_columns/having.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module AmbigousColumns
      # @author Vlad Stoica
      # Ambiguous columns transformer for the Having clause
      class Having < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        #
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT `id1`, SUM(`id3`) FROM `t1`, `t2` HAVING SUM(`id3`) > 3 GROUP BY 1
        #   is transformed to
        #   SELECT `id1`, SUM(`id3`) FROM `t1`, `t2` HAVING SUM(`t2`.`id3`) > 3 GROUP BY 1
        def transform(query)
          @parsed_query = @parser.scan_str(query)

          having_clause = @parsed_query.query_expression.table_expression.having_clause

          return query if having_clause.nil?

          transformed_having_clause = transform_tree(having_clause.search_condition)

          @parsed_query.query_expression.table_expression.having_clause.instance_variable_set(
            '@search_condition', transformed_having_clause
          )

          @parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/ambigous_columns/order_by.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module AmbigousColumns
      # @author Vlad Stoica
      # Ambiguous columns transformer for the ORDER BY clause
      class OrderBy < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        #
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT `id1` FROM `t1`, `t2` ORDER BY 1
        #   is transformed to
        #   SELECT `id1` FROM `t1`, `t2` ORDER BY `t1`.`id1`
        def transform(query)
          @parsed_query = @parser.scan_str(query)

          return @parsed_query.to_sql if @parsed_query.order_by.nil?

          sort_specification = @parsed_query.order_by.sort_specification.map do |specification|
            specification.class.new(
              transform_column_integer(specification.column)
            )
          end

          @parsed_query.order_by.instance_variable_set(
            '@sort_specification',
            sort_specification
          )

          @parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/ambigous_columns/select.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module AmbigousColumns
      # @author Vlad Stoica
      # Ambiguous columns transformer for the Select clause
      class Select < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT `id1` FROM `t1`, `t2`
        #   is transformed to
        #   SELECT `t1`.`id1` FROM `t1`, `t2`
        def transform(query)
          @query = query
          @parsed_query = @parser.scan_str(query)

          columns = @parsed_query.query_expression.list.columns.map do |column|
            transform_column(column)
          end

          @parsed_query.query_expression.list.instance_variable_set(
            '@columns',
            columns
          )

          @parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/ambigous_columns/where.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module AmbigousColumns
      # @author Vlad Stoica
      # Ambiguous columns transformer for the WHERE clause
      class Where < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        #
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT `id1` FROM `t1`, `t2` WHERE `id3` > 3
        #   is transformed to
        #   SELECT `id1` FROM `t1`, `t2` WHERE `t2`.`id3` > 3
        def transform(query)
          @parsed_query = @parser.scan_str(query)

          where_clause = @parsed_query.query_expression.table_expression.where_clause

          return query if where_clause.nil?

          transformed_where_clause = transform_tree(where_clause.search_condition)

          @parsed_query.query_expression.table_expression.where_clause.instance_variable_set(
            '@search_condition', transformed_where_clause
          )

          @parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/base.rb

```ruby
# frozen_string_literal: true

require 'sql-parser'

module SqlAssess
  # Module for canonicalization transformers
  module Transformers
    # Base transformer. Provides implementation for traversing from, for
    # getting the list of tables.
    # @abstract
    # @author Vlad Stoica
    class Base
      def initialize(connection)
        @connection = connection
        @parser = SQLParser::Parser.new
      end

      # Transform method that must be implemented in subclasses
      def transform
        raise 'Implement this method in subclass'
      end

      # Gets the full list of tables from a query. It assumes that there are
      # no sub-queries involved
      # @return [Array<String>] the list of tables
      def tables(query)
        SqlAssess::Parsers::Tables.new(query).tables.map do |table|
          if table.key?(:join_type)
            table[:table][:table].remove('`')
          else
            table[:table].remove('`')
          end
        end
      end

      private

      def traverse_from(node)
        if node.is_a?(SQLParser::Statement::QualifiedJoin)
          node.class.new(
            traverse_from(node.left),
            traverse_from(node.right),
            SQLParser::Statement::On.new(
              transform_tree(node.search_condition.search_condition)
            )
          )
        elsif node.is_a?(SQLParser::Statement::JoinedTable)
          node.class.new(
            traverse_from(node.left),
            traverse_from(node.right)
          )
        else
          node
        end
      end
    end
  end
end

require_relative 'all_columns'
require_relative 'from_subquery'

require_relative 'not/base'
require_relative 'ambigous_columns/base'
require_relative 'between_predicate/base'
require_relative 'comparison_predicate/base'
require_relative 'equivalent_columns/base'
```

lib/sql_assess/transformers/between_predicate/base.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    # Transformers for the between predicate
    module BetweenPredicate
      # @author Vlad Stoica
      # Base class for transformers for between predicate to two >= and <=
      class Base < SqlAssess::Transformers::Base
        # The list of between predicate transformers
        def self.transformers
          [From, Where, Having]
        end
        private

        def transform_between(between)
          SQLParser::Statement::And.new(
            SQLParser::Statement::GreaterOrEquals.new(between.left, between.min),
            SQLParser::Statement::LessOrEquals.new(between.left, between.max)
          )
        end

        def transform_tree(node)
          if node.is_a?(SQLParser::Statement::SearchCondition)
            node.class.new(
              transform_tree(node.left),
              transform_tree(node.right)
            )
          elsif node.is_a?(SQLParser::Statement::Between)
            transform_between(node)
          else
            node
          end
        end
      end
    end
  end
end

require_relative 'from'
require_relative 'where'
require_relative 'having'
```

lib/sql_assess/transformers/between_predicate/from.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module BetweenPredicate
      # Between transformer for FROM clause
      # @author Vlad Stoica
      class From < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        #
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT * FROM `t1` LEFT JOIN `t2` ON id1 BETWEEN id2 and 3
        #   is transformed
        #   SELECT * FROM `t1` LEFT JOIN `t2` ON id1 >= id2 AND id1 <= 3
        def transform(query)
          parsed_query = @parser.scan_str(query)

          join_clause = parsed_query.query_expression&.table_expression&.from_clause

          return query if join_clause.nil?

          new_tables = join_clause.tables.map do |table|
            traverse_from(table)
          end

          parsed_query.query_expression.table_expression.from_clause.instance_variable_set(
            '@tables', new_tables
          )
          parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/between_predicate/having.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module BetweenPredicate
      # Between transformer for Having clause
      # @author Vlad Stoica
      class Having < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT `id1`, SUM(`id3`) FROM `t1`, `t2` HAVING SUM(`id3`) BETWEEN 1 AND 3 GROUP BY 1
        #   is transformed to
        #   SELECT `id1`, SUM(`id3`) FROM `t1`, `t2` HAVING SUM(`id3`) >=1 AND HAVING SUM(`id3`) <= 3 GROUP BY
        ↪  1
        def transform(query)
          parsed_query = @parser.scan_str(query)

          having_clause = parsed_query.query_expression.table_expression.having_clause

          return query if having_clause.nil?

          transformed_having_clause = transform_tree(having_clause.search_condition)

          parsed_query.query_expression.table_expression.having_clause.instance_variable_set(
            '@search_condition', transformed_having_clause
          )

          parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/between_predicate/where.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module BetweenPredicate
      # Between transformer for WHERE clause
      # @author Vlad Stoica
      class Where < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        #
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT `id1` FROM `t1`, `t2` WHERE `id3` BETWEEN 1 AND 3
        #   is transformed to
        #   SELECT `id1` FROM `t1`, `t2` WHERE `id3` >=1 AND `id3` <= 3
        def transform(query)
          parsed_query = @parser.scan_str(query)

          where_clause = parsed_query.query_expression.table_expression.where_clause

          return query if where_clause.nil?

          transformed_where_clause = transform_tree(where_clause.search_condition)

          parsed_query.query_expression.table_expression.where_clause.instance_variable_set(
            '@search_condition', transformed_where_clause
          )

          parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/comparison_predicate/base.rb

```
# frozen_string_literal: true

module SqlAssess
  module Transformers
    # Transformers for comparison predicate
    module ComparisonPredicate
      # @author Vlad Stoica
      # Base class for transformers for comparison predicate
      class Base < SqlAssess::Transformers::Base
        # The list of comparison predicate transformers
        def self.transformers
          [From, Where, Having]
        end

        private

        def transform_comparison_predicate(predicate)
          if predicate.is_a?(SQLParser::Statement::Greater)
            SQLParser::Statement::Less.new(
              predicate.right,
              predicate.left
            )
          elsif predicate.is_a?(SQLParser::Statement::GreaterOrEquals)
            SQLParser::Statement::LessOrEquals.new(
              predicate.right,
              predicate.left
            )
          else
            predicate
          end
        end

        def transform_tree(node)
          if node.is_a?(SQLParser::Statement::SearchCondition)
            node.class.new(
              transform_tree(node.left),
              transform_tree(node.right)
            )
          else
```

```ruby
            transform_comparison_predicate(node)
          end
        end
      end
    end
  end
end

require_relative 'from'
require_relative 'where'
require_relative 'having'
```

lib/sql_assess/transformers/comparison_predicate/from.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module ComparisonPredicate
      # Comparison predicate transformer for FROM clause
      # @author Vlad Stoica
      class From < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        #
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT * FROM `t1` LEFT JOIN `t2` ON id1 > id2
        #   is transformed
        #   SELECT * FROM `t1` LEFT JOIN `t2` ON id2 < id1
        def transform(query)
          parsed_query = @parser.scan_str(query)

          join_clause = parsed_query.query_expression&.table_expression&.from_clause

          return query if join_clause.nil?

          new_tables = join_clause.tables.map do |table|
            traverse_from(table)
          end

          parsed_query.query_expression.table_expression.from_clause.instance_variable_set(
            '@tables', new_tables
          )

          parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/comparison_predicate/having.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module ComparisonPredicate
      # Comparison predicate transformer for HAVING clause
      # @author Vlad Stoica
      class Having < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        #
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT `id1`, SUM(`id3`) FROM `t1`, `t2` HAVING SUM(`id3`) > 1 GROUP BY 1
        #   is transformed to
        #   SELECT `id1`, SUM(`id3`) FROM `t1`, `t2` 1 < HAVING SUM(`id3`) GROUP BY 1
        def transform(query)
          parsed_query = @parser.scan_str(query)

          having_clause = parsed_query.query_expression.table_expression.having_clause

          return query if having_clause.nil?

          transformed_having_clause = transform_tree(having_clause.search_condition)

          parsed_query.query_expression.table_expression.having_clause.instance_variable_set(
            '@search_condition', transformed_having_clause
          )

          parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/comparison_predicate/where.rb

```
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module ComparisonPredicate
      # Comparison predicate transformer for WHERE clause
      # @author Vlad Stoica
      class Where < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        #
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT `id1` FROM `t1`, `t2` WHERE `id3` > 1
        #   is transformed to
        #   SELECT `id1` FROM `t1`, `t2` WHERE 1 < `id3`
        def transform(query)
          parsed_query = @parser.scan_str(query)

          where_clause = parsed_query.query_expression.table_expression.where_clause

          return query if where_clause.nil?

          transformed_where_clause = transform_tree(where_clause.search_condition)

          parsed_query.query_expression.table_expression.where_clause.instance_variable_set(
            '@search_condition', transformed_where_clause
          )

          parsed_query.to_sql
        end
```

```ruby
      end
    end
  end
end
```

lib/sql_assess/transformers/equivalent_columns/base.rb

```ruby
# frozen_string_literal: true

require 'rgl/adjacency'
require 'rgl/condensation.rb'

module SqlAssess
  module Transformers
    # Transformers for equivalent columns
    module EquivalentColumns
      # @author Vlad Stoica
      # Base class for transformers for equivalent column
      class Base < SqlAssess::Transformers::Base
        # The list of equivalent columns transformers
        def self.transformers
          [Select, Where, Group, OrderBy, Having]
        end

        private

        def transform_column(column)
          if column.is_a?(SQLParser::Statement::QualifiedColumn)
            equivalence = equivalences_list.detect do |equivalences|
              equivalences.include?(column.to_sql)
            end

            if equivalence.present?
              table_name, column_name = equivalence.sort.first.remove('`').split('.')

              SQLParser::Statement::QualifiedColumn.new(
                SQLParser::Statement::Table.new(table_name),
                SQLParser::Statement::Column.new(column_name)
              )
            else
              column
            end
          elsif column.is_a?(SQLParser::Statement::Aggregate)
            column.class.new(transform_column(column.column))
          elsif column.is_a?(SQLParser::Statement::Arithmetic) ||
           ↪  column.is_a?(SQLParser::Statement::ComparisonPredicate)
            column.class.new(
              transform_column(column.left),
              transform_column(column.right)
            )
          else
            column
          end
        end

        def transform_tree(node)
          if node.is_a?(SQLParser::Statement::SearchCondition)
            node.class.new(
              transform_tree(node.left),
              transform_tree(node.right)
            )
          else
            transform_column(node)
          end
        end

        def equivalences_list
          @equivalences_list ||= build_equivalence_graph.map(&:to_a)
        end

        def build_equivalence_graph
          graph = RGL::DirectedAdjacencyGraph.new

          join_conditions = @parsed_query.query_expression.table_expression.from_clause.tables.first

          equivalences = find_equivalences(join_conditions)

          equivalences.each do |equivalence|
            graph.add_edge(equivalence[:equivalence_left].to_sql, equivalence[:equivalence_right].to_sql)
            graph.add_edge(equivalence[:equivalence_right].to_sql, equivalence[:equivalence_left].to_sql)
          end

          graph.condensation_graph.vertices
        end

        def find_equivalences(clause)
          if clause.is_a?(SQLParser::Statement::QualifiedJoin)
            [
              find_equivalences_search_condition(
                clause.search_condition.search_condition
              ),
              find_equivalences(clause.left),
              find_equivalences(clause.right),
            ].flatten
          elsif clause.is_a?(SQLParser::Statement::JoinedTable)
            [
              find_equivalences(clause.left),
              find_equivalences(clause.right),
            ].flatten
          else
            []
          end
        end

        def find_equivalences_search_condition(search_condition)
          if search_condition.is_a?(SQLParser::Statement::And)
            [
              find_equivalences_search_condition(search_condition.left),
              find_equivalences_search_condition(search_condition.right),
            ].flatten
          elsif search_condition.is_a?(SQLParser::Statement::Equals)
            [
              {
                equivalence_left: search_condition.left,
                equivalence_right: search_condition.right,
              },
            ]
          else
            []
          end
        end
      end
    end
  end
end

require_relative 'group'
require_relative 'order_by'
require_relative 'select'
require_relative 'where'
require_relative 'having'
```

lib/sql_assess/transformers/equivalent_columns/group.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module EquivalentColumns
      # @author Vlad Stoica
      # Equivalent columns transformer for GROUP clause
      class Group < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        # @example
        #   SELECT *
        #   FROM `b` LEFT JOIN `a` ON `a`.`id` = `b`.`id`
        #   GROUP BY `b`.`id`
        #
        #   is transformed to
        #
        #   SELECT *
        #   FROM `b` LEFT JOIN `a` ON `a`.`id` = `b`.`id`
        #   GROUP BY `a`.`id`
        def transform(query)
          @query = query

          @parsed_query = @parser.scan_str(query)

          if @parsed_query.query_expression.table_expression.group_by_clause.nil?
            return @parsed_query.to_sql
          end

          columns = @parsed_query.query_expression.table_expression.group_by_clause.columns.map do |column|
            transform_column(column)
          end

          @parsed_query.query_expression.table_expression.group_by_clause.instance_variable_set(
            '@columns',
            columns
          )

          @parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/equivalent_columns/having.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module EquivalentColumns
      # @author Vlad Stoica
      # Equivalent columns transformer for HAVING clause
      class Having < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        # @example
        #   SELECT *
        #   FROM `b` LEFT JOIN `a` ON `a`.`id` = `b`.`id`
        #   HAVING SUM(`b`.`id`) > 3
        #
        #   is transformed to
        #
        #   SELECT *
        #   FROM `b` LEFT JOIN `a` ON `a`.`id` = `b`.`id`
        #   HAVING SUM(`a`.`id`) > 3
        def transform(query)
          @parsed_query = @parser.scan_str(query)

          having_clause = @parsed_query.query_expression.table_expression.having_clause

          return query if having_clause.nil?

          transformed_having_clause = transform_tree(having_clause.search_condition)

          @parsed_query.query_expression.table_expression.having_clause.instance_variable_set(
            '@search_condition', transformed_having_clause
          )

          @parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/equivalent_columns/order_by.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module EquivalentColumns
      # @author Vlad Stoica
      # Equivalent columns transformer for Order clause
      class OrderBy < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        # @example
        #   SELECT *
        #   FROM `b` LEFT JOIN `a` ON `a`.`id` = `b`.`id`
        #   ORDER BY `b`.`id`
        #
        #   is transformed to
        #
        #   SELECT *
        #   FROM `b` LEFT JOIN `a` ON `a`.`id` = `b`.`id`
        #   ORDER BY `a`.`id`
        def transform(query)
          @parsed_query = @parser.scan_str(query)

          return @parsed_query.to_sql if @parsed_query.order_by.nil?

          sort_specification = @parsed_query.order_by.sort_specification.map do |specification|
            specification.class.new(
              transform_column(specification.column)
            )
          end

          @parsed_query.order_by.instance_variable_set(
            '@sort_specification',
            sort_specification
          )

          @parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/equivalent_columns/select.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module EquivalentColumns
      # @author Vlad Stoica
      # Equivalent columns transformer for columns list
      class Select < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        # @example
        #   SELECT `b`.`id`
        #   FROM `b` LEFT JOIN `a` ON `a`.`id` = `b`.`id`
        #
        #   is transformed to
        #
        #   SELECT `a`.`id`
        #   FROM `b` LEFT JOIN `a` ON `a`.`id` = `b`.`id`
        def transform(query)
          @query = query
          @parsed_query = @parser.scan_str(query)

          columns = @parsed_query.query_expression.list.columns.map do |column|
            transform_column(column)
          end

          @parsed_query.query_expression.list.instance_variable_set(
            '@columns',
            columns
          )

          @parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/equivalent_columns/where.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module EquivalentColumns
```

```
      # @author Vlad Stoica
      # Equivalent columns transformer for WHERE clause
      class Where < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        # @example
        #   SELECT *
        #   FROM `b` LEFT JOIN `a` ON `a`.`id` = `b`.`id`
        #   WHERE `b`.`id` > 3
        #
        #   is transformed to
        #
        #   SELECT *
        #   FROM `b` LEFT JOIN `a` ON `a`.`id` = `b`.`id`
        #   WHERE `a`.`id` > 3
        def transform(query)
          @parsed_query = @parser.scan_str(query)

          where_clause = @parsed_query.query_expression.table_expression.where_clause

          return query if where_clause.nil?

          transformed_where_clause = transform_tree(where_clause.search_condition)

          @parsed_query.query_expression.table_expression.where_clause.instance_variable_set(
            '@search_condition', transformed_where_clause
          )

          @parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/from_subquery.rb

```
# frozen_string_literal: true

module SqlAssess
  module Transformers
    # @author Vlad Stoica
    # Equivalent columns transformer for subqueries in the FROM clause
    # @deprecated Do not use
    class FromSubquery < Base
      # Transforms the query
      #
      # @param [String] query the initial query
      # @return [String] the transformed query
      def transform(query)
        parsed_query = @parser.scan_str(query)

        join_clause = parsed_query.query_expression&.table_expression&.from_clause

        return query if join_clause.nil?

        new_tables = join_clause.tables.map do |table|
          transform_table(table)
        end

        parsed_query.query_expression.table_expression.from_clause.instance_variable_set(
          '@tables', new_tables
```

```ruby
        )

        parsed_query.to_sql
      end

      private

      def transform_table(table)
        if table.is_a?(SQLParser::Statement::QualifiedJoin)
          table.class.new(
            transform_table(table.left),
            transform_table(table.right),
            SQLParser::Statement::On.new(
              table.search_condition.search_condition
            )
          )
        elsif table.is_a?(SQLParser::Statement::Subquery)
          SQLParser::Statement::Subquery.new(
            @parser.scan_str(
              SqlAssess::QueryTransformer.new(@connection).transform(
                table.query_specification.to_sql
              )
            )
          )
        else
          table
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/not/base.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    # Namespace for NOT transformers
    module Not
      # @author Vlad Stoica
      # Base class for transformers for not
      class Base < SqlAssess::Transformers::Base
        # The list not columns transformers
        def self.transformers
          [From, Where, Having]
        end

        private

        def transform_not(not_statement)
          # Greater
          if not_statement.value.is_a?(SQLParser::Statement::Greater)
            SQLParser::Statement::LessOrEquals.new(not_statement.value.left, not_statement.value.right)
          elsif not_statement.value.is_a?(SQLParser::Statement::GreaterOrEquals)
            SQLParser::Statement::Less.new(not_statement.value.left, not_statement.value.right)
          # Less
          elsif not_statement.value.is_a?(SQLParser::Statement::Less)
            SQLParser::Statement::GreaterOrEquals.new(not_statement.value.left, not_statement.value.right)
          elsif not_statement.value.is_a?(SQLParser::Statement::LessOrEquals)
            SQLParser::Statement::Greater.new(not_statement.value.left, not_statement.value.right)
          else
            not_statement
          end
        end

        def transform_tree(node)
          if node.is_a?(SQLParser::Statement::SearchCondition)
            node.class.new(
              transform_tree(node.left),
              transform_tree(node.right)
            )
          elsif node.is_a?(SQLParser::Statement::Not)
            transform_not(node)
          else
            node
          end
        end
      end
    end
  end
end

require_relative 'from'
require_relative 'where'
require_relative 'having'
```

lib/sql_assess/transformers/not/from.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module Not
      # @author Vlad Stoica
      # NOT transformer for the FROM clause
      class From < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        #
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT * FROM `t1` LEFT JOIN `t2` ON NOT id1 > id2
        #   is transformed
        #   SELECT * FROM `t1` LEFT JOIN `t2` ON id1 <= id2
        def transform(query)
          parsed_query = @parser.scan_str(query)

          join_clause = parsed_query.query_expression&.table_expression&.from_clause

          return query if join_clause.nil?

          new_tables = join_clause.tables.map do |table|
            traverse_from(table)
          end

          parsed_query.query_expression.table_expression.from_clause.instance_variable_set(
            '@tables', new_tables
          )

          parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/not/having.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module Not
      # NOT transformer for the HAVING clause
      class Having < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT `id1`, SUM(`id3`) FROM `t1`, `t2` HAVING NOT SUM(`id3`) > 3 GROUP BY 1
        #   is transformed to
        #   SELECT `id1`, SUM(`id3`) FROM `t1`, `t2` HAVING SUM(`t2`.`id3`) <= 3 GROUP BY 1
        def transform(query)
          parsed_query = @parser.scan_str(query)

          having_clause = parsed_query.query_expression.table_expression.having_clause

          return query if having_clause.nil?

          transformed_having_clause = transform_tree(having_clause.search_condition)

          parsed_query.query_expression.table_expression.having_clause.instance_variable_set(
            '@search_condition', transformed_having_clause
          )

          parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/transformers/not/where.rb

```ruby
# frozen_string_literal: true

module SqlAssess
  module Transformers
    module Not
      # NOT transformer for the WHERE clause
      class Where < Base
        # Transforms the query
        #
        # @param [String] query the initial query
        # @return [String] the transformed query
        #
        # @example
        #   With tables: t1(id1), t2(id3);
        #   SELECT * FROM `t1`, `t2` WHERE NOT id1 > id2
        #   is transformed
        #   SELECT * FROM `t1`, `t2` WHERE id1 <= id2
        def transform(query)
          parsed_query = @parser.scan_str(query)

          where_clause = parsed_query.query_expression.table_expression.where_clause

          return query if where_clause.nil?
          transformed_where_clause = transform_tree(where_clause.search_condition)

          parsed_query.query_expression.table_expression.where_clause.instance_variable_set(
            '@search_condition', transformed_where_clause
          )

          parsed_query.to_sql
        end
      end
    end
  end
end
```

lib/sql_assess/version.rb

```
# frozen_string_literal: true

module SqlAssess
  # Version of the gem
  VERSION = '0.1.0'
end
```

spec/fixtures/assesor_integration_tests.yml

```
-
  schema: CREATE TABLE t1(id integer);
  seed: INSERT INTO t1(id) VALUES (122);
  instructor_query: SELECT * from t1;
  student_query: SELECT 2 from t1;
  message: Your query is not correct. Check what columns you are selecting.
-
  schema: CREATE TABLE t1(id integer);
  seed: INSERT INTO t1(id) VALUES (122);
  instructor_query: SELECT * from t1;
  student_query: SELECT * from t1;
  message: Congratulations! Your solution is correct
-
  schema: CREATE TABLE t1(id integer);
  seed: INSERT INTO t1(id) VALUES (122);
  instructor_query: SELECT * from t1;
  student_query: SELECT * FROM t1 ORDER BY id ASC;
  message: Your query is not correct. Are you ordering the rows correctly?
-
  schema: CREATE TABLE t1(id integer);
  seed: INSERT INTO t1(id) VALUES (122);
  instructor_query: SELECT * from t1;
  student_query: SELECT * FROM t1 LIMIT 1;
  message: Your query is not correct. Are you selecting the correct number of rows?
-
  schema: CREATE TABLE t1(id integer);
  seed: INSERT INTO t1(id) VALUES (122);
  instructor_query: SELECT * from t1;
  student_query: SELECT * FROM t1 WHERE id = 1;
  message: Your query is not correct. Looks like you are selecting the right columns, but you are not
  ↪  selecting only the correct rows.
```

spec/fixtures/transformer_hacker_rank_integration_tests.yml

```
# Basic Select
-
  name: Revising the Select Query I
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
    ↪  population integer)
  query: SELECT * from CITY WHERE `POPULATION` > 100000 and `COUNTRYCODE` = "USA"
  expected_result: SELECT `CITY`.`id`, `CITY`.`name`, `CITY`.`countrycode`, `CITY`.`district`,
   ↪  `CITY`.`population` FROM `CITY` WHERE (100000 < `CITY`.`POPULATION` AND `CITY`.`COUNTRYCODE` = 'USA')
-
  name: Revising the Select Query II
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
 ↪  population integer)
  query: select name from CITY where POPULATION > 120000 and `COUNTRYCODE` = 'USA'
  expected_result: SELECT `CITY`.`name` FROM `CITY` WHERE (120000 < `CITY`.`POPULATION` AND
 ↪  `CITY`.`COUNTRYCODE` = 'USA')
-
  name: Select All
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
 ↪  population integer)
  query: SELECT * from CITY
  expected_result: SELECT `CITY`.`id`, `CITY`.`name`, `CITY`.`countrycode`, `CITY`.`district`,
 ↪  `CITY`.`population` FROM `CITY`
-
  name: Select By Id
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
 ↪  population integer)
  query: select * from CITY WHERE ID = 1661
  expected_result: SELECT `CITY`.`id`, `CITY`.`name`, `CITY`.`countrycode`, `CITY`.`district`,
 ↪  `CITY`.`population` FROM `CITY` WHERE `CITY`.`ID` = 1661
-
  name: Japanese Cities' Attributes
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
 ↪  population integer)
  query: select * from CITY WHERE `COUNTRYCODE` = 'JPN'
  expected_result: SELECT `CITY`.`id`, `CITY`.`name`, `CITY`.`countrycode`, `CITY`.`district`,
 ↪  `CITY`.`population` FROM `CITY` WHERE `CITY`.`COUNTRYCODE` = 'JPN'
-
  name: Japanese Cities' Names
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
 ↪  population integer)
  query: select name from CITY WHERE `COUNTRYCODE` = 'JPN'
  expected_result: SELECT `CITY`.`name` FROM `CITY` WHERE `CITY`.`COUNTRYCODE` = 'JPN'
-
  name: Weather Observation Station 1
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: SELECT city, state from STATION
  expected_result: SELECT `STATION`.`city`, `STATION`.`state` FROM `STATION`
-
  name: Weather Observation Station 3
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: select DISTINCT(CITY) from STATION where ID % 2 = 0 ORDER by CITY DESC
  expected_result: SELECT DISTINCT `STATION`.`CITY` FROM `STATION` WHERE (`STATION`.`ID` % 2) = 0 ORDER BY
 ↪  `STATION`.`CITY` DESC
-
  name: Weather Observation Station 4
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: select count(CITY) - count(DISTINCT CITY) FROM STATION;
  support: false
-
  name: Weather Observation Station 5 - part 1
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: select city, length(city) from STATION order by length(city) ASC, city ASC LIMIT 2
  support: false
-
  name: Weather Observation Station 5 - part 2
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: select city, length(city) from STATION order by length(city) desc, city ASC limit 1
  support: false
-
  name: Weather Observation Station 6
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: select DISTINCT CITY from STATION WHERE CITY LIKE 'A%' OR CITY LIKE 'E%' OR CITY LIKE 'I%' OR CITY
↪  LIKE 'O%' OR CITY LIKE 'U%'
  expected_result: SELECT DISTINCT `STATION`.`CITY` FROM `STATION` WHERE ((((`STATION`.`CITY` LIKE 'A%' OR
↪  `STATION`.`CITY` LIKE 'E%') OR `STATION`.`CITY` LIKE 'I%') OR `STATION`.`CITY` LIKE 'O%') OR
↪  `STATION`.`CITY` LIKE 'U%')
-
  name: Weather Observation Station 7
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: select DISTINCT CITY from STATION WHERE CITY LIKE '%A' OR CITY LIKE '%E' OR CITY LIKE '%I' OR CITY
↪  LIKE '%O' OR CITY LIKE '%U'
  expected_result: SELECT DISTINCT `STATION`.`CITY` FROM `STATION` WHERE ((((`STATION`.`CITY` LIKE '%A' OR
↪  `STATION`.`CITY` LIKE '%E') OR `STATION`.`CITY` LIKE '%I') OR `STATION`.`CITY` LIKE '%O') OR
↪  `STATION`.`CITY` LIKE '%U')
-
  name: Weather Observation Station 8
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: select DISTINCT CITY from STATION WHERE CITY LIKE '[AEIOU]%[AEIOU]'
  expected_result: SELECT DISTINCT `STATION`.`CITY` FROM `STATION` WHERE `STATION`.`CITY` LIKE
↪  '[AEIOU]%[AEIOU]'
-
  name: Weather Observation Station 9
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: select DISTINCT CITY from STATION WHERE CITY NOT LIKE '[AEIOUaeiou]%'
  expected_result: SELECT DISTINCT `STATION`.`CITY` FROM `STATION` WHERE `CITY` NOT LIKE '[AEIOUaeiou]%'
-
  name: Weather Observation Station 10
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: select DISTINCT CITY from STATION WHERE CITY LIKE '%[^AEIOU]'
  expected_result: SELECT DISTINCT `STATION`.`CITY` FROM `STATION` WHERE `STATION`.`CITY` LIKE '%[^AEIOU]'
-
  name: Weather Observation Station 11
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: select DISTINCT CITY from STATION WHERE CITY LIKE '[^AEIOU]%' OR CITY LIKE '%[^AEIOU]'
  expected_result: SELECT DISTINCT `STATION`.`CITY` FROM `STATION` WHERE (`STATION`.`CITY` LIKE '[^AEIOU]%' OR
↪  `STATION`.`CITY` LIKE '%[^AEIOU]')
-
  name: Weather Observation Station 12
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: select DISTINCT CITY from STATION WHERE CITY LIKE '[^AEIOU]%[^AEIOU]'
  expected_result: SELECT DISTINCT `STATION`.`CITY` FROM `STATION` WHERE `STATION`.`CITY` LIKE
↪  '[^AEIOU]%[^AEIOU]'
-
  name: Higher Than 75 Marks
  schema: |
    CREATE TABLE STUDENTS(id integer, name varchar(255), marks integer)
  query: select Name from STUDENTS where Marks > 75 order by substr(Name, -3) ASC, ID ASC
  support: false
-
  name: Employee names
  schema: |
    CREATE TABLE EMPLOYEE(employee_id integer, name varchar(255), months integer, salary integer)
  query: SELECT name FROM `EMPLOYEE` ORDER BY name ASC
  expected_result: SELECT `EMPLOYEE`.`name` FROM `EMPLOYEE` ORDER BY `EMPLOYEE`.`name` ASC
-
  name: Employee salaries
  schema: |
    CREATE TABLE EMPLOYEE(employee_id integer, name varchar(255), months integer, salary integer)
  query: SELECT name FROM `EMPLOYEE` WHERE salary > 2000 and months < 10
  expected_result: SELECT `EMPLOYEE`.`name` FROM `EMPLOYEE` WHERE (2000 < `EMPLOYEE`.`salary` AND
↪  `EMPLOYEE`.`months` < 10)
# Basic join
-
  name: Asian Population
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
↪  population integer);
    CREATE TABLE COUNTRY(
      code varchar(255), name varchar(255), continent varchar(255),
      region varchar(255), surfacearea integer, indepyear varchar(255),
      population integer, lifeexpectancy varchar(255), gnp integer,
      gnpold varchar(255), localname varchar(255), governmentform varchar(255),
      headofstate varchar(255), capital varchar(255), code2 varchar(255)
    );
  query: SELECT SUM(CITY.POPULATION) FROM CITY LEFT JOIN `COUNTRY` ON `COUNTRY`.CODE = CITY.`COUNTRYCODE`
↪  WHERE `COUNTRY`.CONTINENT = "ASIA"
  expected_result: SELECT SUM(`CITY`.`POPULATION`) FROM `CITY` LEFT JOIN `COUNTRY` ON `COUNTRY`.`CODE` =
↪  `CITY`.`COUNTRYCODE` WHERE `COUNTRY`.`CONTINENT` = 'ASIA'
-
  name: African cities
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
↪  population integer);
    CREATE TABLE COUNTRY(
      code varchar(255), name varchar(255), continent varchar(255),
      region varchar(255), surfacearea integer, indepyear varchar(255),
      population integer, lifeexpectancy varchar(255), gnp integer,
      gnpold varchar(255), localname varchar(255), governmentform varchar(255),
      headofstate varchar(255), capital varchar(255), code2 varchar(255)
    );
  query: SELECT CITY.NAME FROM CITY LEFT JOIN `COUNTRY` ON `COUNTRY`.CODE = CITY.`COUNTRYCODE` WHERE
↪  `COUNTRY`.CONTINENT = 'AFRICA'
  expected_result: SELECT `CITY`.`NAME` FROM `CITY` LEFT JOIN `COUNTRY` ON `COUNTRY`.`CODE` =
↪  `CITY`.`COUNTRYCODE` WHERE `COUNTRY`.`CONTINENT` = 'AFRICA'
-
  name: Average Population of Each Continent
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
↪  population integer);
    CREATE TABLE COUNTRY(
      code varchar(255), name varchar(255), continent varchar(255),
      region varchar(255), surfacearea integer, indepyear varchar(255),
      population integer, lifeexpectancy varchar(255), gnp integer,
      gnpold varchar(255), localname varchar(255), governmentform varchar(255),
      headofstate varchar(255), capital varchar(255), code2 varchar(255)
    );
  query: SELECT `COUNTRY`.CONTINENT, AVG(CITY.POPULATION) FROM CITY LEFT JOIN `COUNTRY` ON `COUNTRY`.CODE =
↪  CITY.`COUNTRYCODE` GROUP BY `COUNTRY`.CONTINENT
```

```
  expected_result: SELECT `COUNTRY`.`CONTINENT`, AVG(`CITY`.`POPULATION`) FROM `CITY` LEFT JOIN `COUNTRY` ON
↪  `COUNTRY`.`CODE` = `CITY`.`COUNTRYCODE` GROUP BY `COUNTRY`.`CONTINENT`
-
  name: The report
  schema: |
    CREATE TABLE Students(id integer, name varchar(255), marks integer);
    CREATE TABLE Grades(grade integer, min_mark integer, max_mark integer);
  query: SELECT CASE WHEN `Grades`.`grade` < 8 THEN NULL ELSE Students.Name END, Grades.grade, Students.Marks
↪  FROM Students LEFT JOIN Grades ON Students.Marks >= Grades.`Min_Mark` AND Students.Marks <=
↪  Grades.`Max_Mark` ORDER BY Grades.grade DESC, Students.Name ASC
  support: false
-
  name: Top competitors
  schema: |
    CREATE TABLE hackers(hacker_id integer, name varchar(255));
    CREATE TABLE difficulty(difficulty_level integer, score integer);
    CREATE TABLE challenges(difficulty_level integer, hacker_id integer, challenge_id integer);
    CREATE TABLE submissions(submission_id integer, hacker_id integer, challenge_id integer, score integer);
  query: |
    select hackers.hacker_id, hackers.name
    from
      submissions
      inner join challenges on submissions.challenge_id = challenges.challenge_id
      inner join difficulty on challenges.difficulty_level = difficulty.difficulty_level
      inner join hackers on submissions.hacker_id = hackers.hacker_id
    where submissions.score = difficulty.score and challenges.difficulty_level = difficulty.difficulty_level
    group by hackers.hacker_id, hackers.name
    having count(submissions.hacker_id) > 1
    order by count(submissions.hacker_id) desc, submissions.hacker_id asc
  expected_result: |
    SELECT `hackers`.`hacker_id`, `hackers`.`name`
    FROM
      `submissions`
      INNER JOIN `challenges` ON `submissions`.`challenge_id` = `challenges`.`challenge_id`
      INNER JOIN `difficulty` ON `challenges`.`difficulty_level` = `difficulty`.`difficulty_level`
      INNER JOIN `hackers` ON `submissions`.`hacker_id` = `hackers`.`hacker_id`
    WHERE (`submissions`.`score` = `difficulty`.`score` AND `challenges`.`difficulty_level` =
↪  `challenges`.`difficulty_level`)
    GROUP BY `hackers`.`hacker_id`, `hackers`.`name`
    HAVING 1 < COUNT(`hackers`.`hacker_id`)
    ORDER BY COUNT(`hackers`.`hacker_id`) DESC, `hackers`.`hacker_id` ASC
-
  name: Challenges
  schema: |
    CREATE TABLE hackers(hacker_id integer, name varchar(255));
    CREATE TABLE difficulty(difficulty_level integer, score integer);
    CREATE TABLE challenges(difficulty_level integer, hacker_id integer, challenge_id integer);
    CREATE TABLE submissions(submission_id integer, hacker_id integer, challenge_id integer, score integer);
  query: |
    select * from hackers
  support: false
-
  name: Contest leaderboard
  schema: |
    CREATE TABLE hackers(hacker_id integer, name varchar(255));
    CREATE TABLE difficulty(difficulty_level integer, score integer);
    CREATE TABLE challenges(difficulty_level integer, hacker_id integer, challenge_id integer);
    CREATE TABLE submissions(submission_id integer, hacker_id integer, challenge_id integer, score integer);
  query: |
    select * from hackers
  support: false
# Advance select
-
```

```
  name: Type of triangle
  support: false
-
  name: The pads
  support: false
-
  name: Occupations
  support: false
-
  name: Binary Tree nodes
  support: false
-
  name: New companies
  schema: |
    CREATE TABLE Company(company_code varchar(255), founder varchar(255));
    CREATE TABLE Lead_Manager(company_code varchar(255), lead_manager_code varchar(255));
    CREATE TABLE Senior_Manager(company_code varchar(255), lead_manager_code varchar(255), senior_manager_code
↪   varchar(255));
    CREATE TABLE Manager(company_code varchar(255), lead_manager_code varchar(255), senior_manager_code
↪   varchar(255), manager_code varchar(255));
    CREATE TABLE Employee(company_code varchar(255), lead_manager_code varchar(255), senior_manager_code
↪   varchar(255), manager_code varchar(255), employee_code varchar(255));
  query: |
    select Company.company_code, Company.founder,
      count(Lead_Manager.lead_manager_code), count(Senior_Manager.senior_manager_code),
      count(Manager.manager_code), count(`Employee`.`employee_code`)
    from Company, Lead_Manager, Senior_Manager, Manager, `Employee`
    where Company.company_code = Lead_Manager.company_code
      and Lead_Manager.lead_manager_code = Senior_Manager.lead_manager_code
      and Senior_Manager.senior_manager_code = Manager.senior_manager_code
      and Manager.manager_code = `Employee`.manager_code
    group by Company.company_code, Company.founder
    order by Company.company_code
  expected_result: |
    SELECT
      `Company`.`company_code`,
      `Company`.`founder`,
      COUNT(`Lead_Manager`.`lead_manager_code`),
      COUNT(`Senior_Manager`.`senior_manager_code`),
      COUNT(`Manager`.`manager_code`),
      COUNT(`Employee`.`employee_code`)
    FROM
      `Company`
      CROSS JOIN `Lead_Manager`
      CROSS JOIN `Senior_Manager`
      CROSS JOIN `Manager`
      CROSS JOIN `Employee`
    WHERE
      (((`Company`.`company_code` = `Lead_Manager`.`company_code` AND `Lead_Manager`.`lead_manager_code` =
↪   `Senior_Manager`.`lead_manager_code`) AND `Senior_Manager`.`senior_manager_code` =
↪   `Manager`.`senior_manager_code`) AND `Manager`.`manager_code` = `Employee`.`manager_code`)
    GROUP BY
      `Company`.`company_code`,
      `Company`.`founder`
    ORDER BY `Company`.`company_code` ASC
# Aggregate
-
  name: Revising Aggregations - The Count Function
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
↪   population integer)
  query: SELECT COUNT(id) FROM CITY WHERE population > 100000
  expected_result: SELECT COUNT(`CITY`.`id`) FROM `CITY` WHERE 100000 < `CITY`.`population`
```

```
-
  name: Revising Aggregations - The Sum Function
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
↪  population integer)
  query: SELECT SUM(POPULATION) FROM CITY WHERE DISTRICT="CALIFORNIA" GROUP BY DISTRICT
  expected_result: SELECT SUM(`CITY`.`POPULATION`) FROM `CITY` WHERE `CITY`.`DISTRICT` = 'CALIFORNIA' GROUP BY
↪  `CITY`.`DISTRICT`
-
  name: Revising Aggregations - Averages
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
↪  population integer)
  query: SELECT AVG(population) FROM CITY WHERE district = 'California'
  expected_result: SELECT AVG(`CITY`.`population`) FROM `CITY` WHERE `CITY`.`district` = 'California'
-
  name: Average Population
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
↪  population integer)
  query: SELECT AVG(POPULATION) FROM CITY
  expected_result: SELECT AVG(`CITY`.`POPULATION`) FROM `CITY`
-
  name: Japan Population
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
↪  population integer)
  query: SELECT SUM(POPULATION) FROM CITY  WHERE `COUNTRYCODE` ='JPN'
  expected_result: SELECT SUM(`CITY`.`POPULATION`) FROM `CITY` WHERE `CITY`.`COUNTRYCODE` = 'JPN'
-
  name: Population Density Difference
  schema: |
    CREATE TABLE CITY(id integer, name varchar(255), countrycode varchar(255), district varchar(255),
↪  population integer)
  query: SELECT MAX(Population) - MIN(Population) FROM CITY
  expected_result: SELECT (MAX(`CITY`.`Population`) - MIN(`CITY`.`Population`)) FROM `CITY`
-
  name: The Blunder
  schema: |
    CREATE TABLE employees(id integer, name varchar(255), salary integer)
  query: SELECT AVG(salary - REPLACE(salary, '0', '')) FROM employees;
  support: false
-
  name: Top earners
  schema: |
    CREATE TABLE EMPLOYEE(employee_id integer, name varchar(255), months integer, salary integer)
  query: select salary * months FROM `EMPLOYEE` group by 1 order by 1 desc
  expected_result: SELECT (`EMPLOYEE`.`salary` * `EMPLOYEE`.`months`) FROM `EMPLOYEE` GROUP BY
↪  (`EMPLOYEE`.`salary` * `EMPLOYEE`.`months`) ORDER BY (`EMPLOYEE`.`salary` * `EMPLOYEE`.`months`) DESC
-
  name: Weather Observation Station 2
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: SELECT ROUND(SUM(LAT_N), 2), ROUND(SUM(LONG_W), 2) FROM STATION;
  support: false
-
  name: Weather Observation Station 13
  schema: |
    CREATE TABLE station(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: select sum(lat_n) from station where lat_n>38.7880 and lat_n<137.2345
  expected_result: SELECT SUM(`station`.`lat_n`) FROM `station` WHERE (38.788 < `station`.`lat_n` AND
↪  `station`.`lat_n` < 137.2345)
-
```

```
  name: Weather Observation Station 14
  schema: |
    CREATE TABLE station(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  support: false
-
  name: Weather Observation Station 16
  schema: |
    CREATE TABLE station(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  support: false
-
  name: Weather Observation Station 17
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  query: select LONG_W from STATION where LAT_N>38.7780 order by LAT_N
  expected_result: SELECT `STATION`.`LONG_W` FROM `STATION` WHERE 38.778 < `STATION`.`LAT_N` ORDER BY
↪  `STATION`.`LAT_N` ASC
-
  name: Weather Observation Station 18
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  support: false
-
  name: Weather Observation Station 19
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  support: false
-
  name: Weather Observation Station 20
  schema: |
    CREATE TABLE STATION(id integer, CITY varchar(255), STATE varchar(255), LAT_N DOUBLE, LONG_W DOUBLE)
  support: false
```

spec/fixtures/transformer_integration_tests.yml

```
-
  schema: |
    CREATE TABLE table1 (id1 integer, id2 integer)
  query: SELECT * from table1
  expected_result: SELECT `table1`.`id1`, `table1`.`id2` FROM `table1`
-
  schema: |
    CREATE TABLE table1 (name integer, id2 integer)
  query: SELECT table1.name from table1
  expected_result: SELECT `table1`.`name` FROM `table1`
-
  schema: |
    CREATE TABLE table1 (id1 integer, id2 integer)
  query: SELECT * from table1 ORDER BY id1 DESC
  expected_result: SELECT `table1`.`id1`, `table1`.`id2` FROM `table1` ORDER BY `table1`.`id1` DESC
-
  schema: |
    CREATE TABLE table1 (id1 integer, id2 integer);
    CREATE TABLE table2 (id3 integer, id4 integer)
  query: SELECT * from table1, table2
  expected_result: SELECT `table1`.`id1`, `table1`.`id2`, `table2`.`id3`, `table2`.`id4` FROM `table1` CROSS
↪  JOIN `table2`
-
  schema: |
    CREATE TABLE table1 (id1 integer, id2 integer);
    CREATE TABLE table2 (id3 integer, id4 integer)
  query: SELECT * from table1 LEFT JOIN table2 on table1.id1 = table2.id3
  expected_result: SELECT `table1`.`id1`, `table1`.`id2`, `table1`.`id1`, `table2`.`id4` FROM `table1` LEFT
↪  JOIN `table2` ON `table1`.`id1` = `table2`.`id3`
-
```

```
  schema: |
    CREATE TABLE table1 (id1 integer, id2 integer);
    CREATE TABLE table2 (id3 integer, id4 integer)
  query: SELECT * from table1, table2 WHERE id1 > 3
  expected_result: SELECT `table1`.`id1`, `table1`.`id2`, `table2`.`id3`, `table2`.`id4` FROM `table1` CROSS
 ↪  JOIN `table2` WHERE 3 < `table1`.`id1`
-
  schema: |
    CREATE TABLE table1 (id1 integer, id2 integer);
    CREATE TABLE table2 (id3 integer, id4 integer)
  query: SELECT * from table1, table2 WHERE id1 BETWEEN 1 and 3
  expected_result: SELECT `table1`.`id1`, `table1`.`id2`, `table2`.`id3`, `table2`.`id4` FROM `table1` CROSS
 ↪  JOIN `table2` WHERE (1 <= `table1`.`id1` AND `table1`.`id1` <= 3)
-
  schema: |
    CREATE TABLE table1 (id1 integer, id2 integer);
    CREATE TABLE table2 (id3 integer, id4 integer)
  query: SELECT * from table1, table2 WHERE id1 BETWEEN 1 and 3 AND id2 > 3 ORDER BY 1
  expected_result: SELECT `table1`.`id1`, `table1`.`id2`, `table2`.`id3`, `table2`.`id4` FROM `table1` CROSS
 ↪  JOIN `table2` WHERE ((1 <= `table1`.`id1` AND `table1`.`id1` <= 3) AND 3 < `table1`.`id2`) ORDER BY
 ↪  `table1`.`id1` ASC
```

spec/spec_helper.rb

```ruby
require "bundler/setup"

unless ENV["CODECOV_TOKEN"].nil?
  require 'simplecov'
  SimpleCov.start

  require 'codecov'
  SimpleCov.formatter = SimpleCov::Formatter::Codecov
end

require "sql_assess"
require "timecop"
require "pry"

module SharedConnection
  def connection
    @shared_connection
  end
end

RSpec.configure do |config|
  # Enable flags like --only-failures and --next-failure
  config.example_status_persistence_file_path = ".rspec_status"

  # Disable RSpec exposing methods globally on `Module` and `main`
  config.disable_monkey_patching!

  config.expect_with :rspec do |c|
    c.syntax = :expect
  end

  config.include SharedConnection

  config.before(:suite) do
    SqlAssess::DatabaseConnection.new(database: "local_db")
  end

  config.before(:all) do
    @shared_connection = SqlAssess::DatabaseConnection.new(database: "local_db")
  end
  config.before(:each) do
    @shared_connection.delete_database
  end
end
```

spec/sql_assess/assesor_spec.rb

```ruby
require "spec_helper"
require "yaml"

RSpec.describe SqlAssess::Assesor do
  before do
    allow(SqlAssess::DatabaseConnection).to receive(:new).and_return(@shared_connection)
  end

  context "#compile" do
    context "without any errors" do
      it "returns the result from data extractor" do
        result = subject.compile(
          create_schema_sql_query: "CREATE TABLE table1 (id integer)",
          instructor_sql_query: "SELECT * from table1",
          seed_sql_query: "INSERT INTO table1 (id) VALUES (1)"
        )

        expect(result).to eq([{
          name: "table1",
          columns: [
            {
              name: "id",
              type: "int(11)"
            },
          ],
          data: [
            "id" => 1
          ],
        }])
      end
    end
  end

  context "#assess" do
    let(:schema_sql_query) { "CREATE TABLE table1 (id integer)" }
    let(:instructor_sql_query) { "SELECT * from table1" }
    let(:seed_sql_query) { "INSERT INTO table1 (id) VALUES (1)" }

    context "with a wrong student query" do
      let(:student_sql_query) { "SELECT * from table2" }
      it "raises an error and clears the database" do
        expect { do_assess }.to raise_error(SqlAssess::DatabaseQueryExecutionFailed)

        tables = subject.connection.query("SHOW tables");
        expect(tables.size).to eq(0)
      end
    end

    context "with a correct student query" do
      let(:student_sql_query) { "SELECT * from table1" }
      it "returns a result" do
        expect(do_assess).to be_a(SqlAssess::QueryComparisonResult)
      end
    end
  end
  yaml = YAML.load_file("spec/fixtures/assesor_integration_tests.yml")

  yaml.each_with_index do |test, i|
    it "correctly asess integration test #{i}" do
      result = subject.assess(
        create_schema_sql_query: test["schema"],
        instructor_sql_query: test["instructor_query"],
        seed_sql_query: test["seed"],
        student_sql_query: test["student_query"]
      )
      expect(result.message).to eq(test["message"])
    end
  end

  private

  def do_assess
    subject.assess(
      create_schema_sql_query: schema_sql_query,
      instructor_sql_query: instructor_sql_query,
      seed_sql_query: seed_sql_query,
      student_sql_query: student_sql_query
    )
  end
end
```

spec/sql_assess/data_extractor_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::DataExtractor do
  subject { described_class.new(connection) }

  context "with a single table" do
    before do
      connection.query('CREATE TABLE table1 (id integer);')
      connection.query('CREATE TABLE table2 (id integer);')
      connection.query('INSERT INTO table1 (id) values(1);')
      connection.query('INSERT INTO table1 (id) values(2);')
    end

    it "returns the correct answer" do
      expect(subject.run).to eq([
        {
          name: "table1",
          columns: [{ name: "id", type: "int(11)" }],
          data: [{ "id" => 1 }, { "id" => 2 }]
        },
        {
          name: "table2",
          columns: [{ name: "id", type: "int(11)" }],
          data: []
        },
      ])
    end
  end
end
```

spec/sql_assess/database_connection_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::DatabaseConnection do
  let(:do_not_delete_database) { false }
  around do |example|
    Timecop.freeze(Time.local(1990)) do
      example.run
      subject.delete_database unless do_not_delete_database
    end
  end

  describe "#initialize" do
    context "when the user is invalid" do
      let(:do_not_delete_database) { true }
      it "throws an error" do
        expect { described_class.new(username: "test") }.to raise_error(SqlAssess::DatabaseConnectionError)
      end
    end

    context "when everything is valid" do
      it "doesn't throw an error" do
        expect { subject }.to_not raise_error
      end
    end
  end

  describe "#database_name" do
    context "with no existing database" do
      it "uses the correct name" do
        subject
        expect(subject.query("SELECT DATABASE();").first["DATABASE()"]).to eq("000000")
      end
    end

    context "with an existing database" do
      it "creates a database with attempt in it" do
        existing_connection = described_class.new
        expect(existing_connection.query("SELECT DATABASE();").first["DATABASE()"]).to eq("000000")

        subject

        expect(subject.query("SELECT DATABASE();").first["DATABASE()"]).to eq("000000_1")

        existing_connection.delete_database
      end
    end
  end

  describe "#query" do
    it "runs the query" do
      expect(subject.query("SHOW tables").count).to eq(0)
    end

    context "when trying to create another database" do
      it "throws an error" do
        expect { subject.query("CREATE DATABASE TEST") }.to raise_error(Mysql2::Error)
      end
    end
  end

  describe "#multi_query" do
    it "runs the query" do
      result = subject.multiple_query("SELECT 1; SELECT 2; SELECT 3")
      expect(result.count).to eq(3)
      expect(result.map(&:first)).to eq([{ "1" => 1 }, { "2" => 2 }, { "3" => 3 }])
    end

    context "when trying to create another database" do
      it "throws an error" do
        expect { subject.multiple_query("CREATE DATABASE TEST") }.to raise_error(Mysql2::Error)
      end
    end
  end

  describe "#delete_database" do
    context "when using default table name" do
      let(:do_not_delete_database) { true }

      it "deletes the database" do
        subject

        expect(subject.query("SELECT DATABASE();").first["DATABASE()"]).to eq("000000")

        subject.delete_database

        expect(subject.query("SHOW DATABASES;").map { |r| r["Database"] }).to_not include("000000")
      end
    end

    context "when passing default database" do
      subject { described_class.new(database: "local_db") }
      it "leaves FOREIGN_KEY_CHECKS set to ON" do
        subject.delete_database

        expect(subject.query("SHOW Variables WHERE Variable_name='foreign_key_checks';").first["Value"])
          .to eq("ON")
      end

      context "when there are no existing tables" do
        it "doesn't throw an error" do
          expect { subject.delete_database }.to_not raise_error

          tables = connection.query('SHOW tables;')

          expect(tables.count).to eq(0)
        end
      end

      context "when there are existing tables with data" do
        context "without foreign keys" do
          before do
            subject.query('CREATE TABLE table1 (id integer);')
            subject.query('CREATE TABLE table2 (id integer);')
            subject.query('INSERT INTO table1 (id) values(1);')
            subject.query('INSERT INTO table2 (id) values(1);')

            tables = subject.query("SHOW tables;")
          end

          it "drops all tables" do
            subject.delete_database

            tables = subject.query('SHOW tables;')

            expect(tables.count).to eq(0)
          end

          it "leaves FOREIGN_KEY_CHECKS set to ON" do
            subject.delete_database

            expect(subject.query("SHOW Variables WHERE Variable_name='foreign_key_checks';").first["Value"])
```

```ruby
              .to eq("ON")
          end
        end

        context "with foreign keys" do
          before do
            subject.query('
              CREATE TABLE table1 (
                id integer,
                PRIMARY KEY (id)
              );
            ')
            subject.query('
              CREATE TABLE table2 (
                table1_id integer,
                FOREIGN KEY(table1_id) REFERENCES table1(id)
              );
            ')

            subject.query('INSERT INTO table1 (id) values(1);')
            subject.query('INSERT INTO table2 (table1_id) values(1);')

            tables = subject.query("SHOW tables;")
          end

          it "drops all tables" do
            subject.delete_database

            tables = subject.query('SHOW tables;')

            expect(tables.count).to eq(0)
          end

          it "leaves FOREIGN_KEY_CHECKS set to ON" do
            subject.delete_database

            expect(subject.query("SHOW Variables WHERE Variable_name='foreign_key_checks';").first["Value"])
              .to eq("ON")
          end
        end
      end
    end
  end
end
```

spec/sql_assess/grader/base_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Grader::Base do
  subject { described_class.new(student_attributes: double, instructor_attributes: double) }
  context "#levenshtein_distance" do
    it "returns the correct distance for a and empty string" do
      expect(subject.levenshtein_distance("a", "")).to eq(1)
    end

    it "returns the correct distance for a and a" do
      expect(subject.levenshtein_distance("a", "a")).to eq(0)
    end

    it "returns the correct distance for a and b" do
      expect(subject.levenshtein_distance("a", "b")).to eq(1)
    end
    it "returns the correct distance for a and ab" do
      expect(subject.levenshtein_distance("a", "ab")).to eq(1)
    end

    it "returns the correct distance for ab and ab" do
      expect(subject.levenshtein_distance("ab", "ab")).to eq(0)
    end

    it "returns the correct distance for ab and empty string" do
      expect(subject.levenshtein_distance("ab", "")).to eq(2)
    end

    it "returns the correct distance for ab and b" do
      expect(subject.levenshtein_distance("ab", "b")).to eq(1)
    end
  end
end
```

spec/sql_assess/grader/columns_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Grader::Columns do
  subject do
    described_class.new(
      student_attributes: student_columns,
      instructor_attributes: instructor_columns
    )
  end

  context "example 1 - same columns" do
    let(:student_columns) { ["table1.column"] }
    let(:instructor_columns) { ["table1.column"] }

    it { expect(subject.rounded_grade).to eq(1) }
  end

  context "example 2 - two of same correct column for student" do
    let(:student_columns) { ["table1.column", "table1.column"] }
    let(:instructor_columns) { ["table1.column"] }

    it { expect(subject.rounded_grade).to eq(0.67) }
  end

  context "example 3 - one correct column and one incorrect for student" do
    let(:student_columns) { ["table1.column", "table1.column2"] }
    let(:instructor_columns) { ["table1.column"] }

    it { expect(subject.rounded_grade).to eq(0.67) }
  end

  context "example 4 - slightly different columns" do
    let(:student_columns) { ["table1.column"] }
    let(:instructor_columns) { ["table1.column_2"] }

    it { expect(subject.rounded_grade).to eq(0.17) }
  end

  context "example 5 - totally different columns" do
    let(:student_columns) { ["table1.column"] }
    let(:instructor_columns) { ["table2.column_2"] }

    it { expect(subject.rounded_grade).to eq(0) }
  end
```

```ruby
end
```

spec/sql_assess/grader/distinct_filter_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Grader::DistinctFilter do
  subject do
    described_class.new(
      student_attributes: student_distinct_filter,
      instructor_attributes: instructor_distinct_filter
    )
  end

  context "same filter" do
    let(:student_distinct_filter) { "ALL" }
    let(:instructor_distinct_filter) { "ALL" }

    it { expect(subject.rounded_grade).to eq(1) }
  end

  context "different filter" do
    let(:student_distinct_filter) { "ALL" }
    let(:instructor_distinct_filter) { "DISTINCT" }

    it { expect(subject.rounded_grade).to eq(0) }
  end

  context "different filter - but both including distinct" do
    let(:student_distinct_filter) { "DISTINCTROW" }
    let(:instructor_distinct_filter) { "DISTINCT" }

    it { expect(subject.rounded_grade).to eq(0.5) }
  end

  context "different filter - but both including distinct" do
    let(:student_distinct_filter) { "DISTINCT" }
    let(:instructor_distinct_filter) { "DISTINCTROW" }

    it { expect(subject.rounded_grade).to eq(0.5) }
  end
end
```

spec/sql_assess/grader/group_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Grader::Group do
  subject do
    described_class.new(
      student_attributes: student_group,
      instructor_attributes: instructor_group
    )
  end

  context "example 1 - same columns" do
    let(:student_group) { ["table1.column"] }
    let(:instructor_group) { ["table1.column"] }

    it { expect(subject.rounded_grade).to eq(1) }
  end

  context "example 2 - two of same correct column for student" do
    let(:student_group) { ["table1.column", "table1.column"] }
    let(:instructor_group) { ["table1.column"] }
    it { expect(subject.rounded_grade).to eq(0.67) }
  end

  context "example 3 - one correct column and one incorrect for student" do
    let(:student_group) { ["table1.column", "table1.column2"] }
    let(:instructor_group) { ["table1.column"] }

    it { expect(subject.rounded_grade).to eq(0.67) }
  end

  context "example 4 - slightly different columns" do
    let(:student_group) { ["table1.column"] }
    let(:instructor_group) { ["table1.column_2"] }

    it { expect(subject.rounded_grade).to eq(0.17) }
  end

  context "example 5 - totally different columns" do
    let(:student_group) { ["table1.column"] }
    let(:instructor_group) { ["table2.column_2"] }

    it { expect(subject.rounded_grade).to eq(0) }
  end
end
```

spec/sql_assess/grader/having_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Grader::Having do
  subject do
    described_class.new(
      student_attributes: attributes[:student][:having_tree],
      instructor_attributes: attributes[:instructor][:having_tree]
    )
  end

  let(:attributes) do
    SqlAssess::QueryAttributeExtractor.new.extract(
       instructor_query, student_query
    )
  end

  context "with no having statements" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
      SQL
    end

    it { expect(subject.rounded_grade).to eq(1) }
  end


  context "with no having for student, but having for teacher" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
        HAVING a > 1
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0) }
  end


  context "with having for student, but no having for teacher" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        HAVING a > 1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0) }
  end

  context "with equal having" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        HAVING a > 1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
        HAVING a > 1
      SQL
    end

    it { expect(subject.rounded_grade).to eq(1) }
  end

  context "with different having" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        HAVING a > 2
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
        HAVING a > 1
      SQL
```

```ruby
    end

    it { expect(subject.rounded_grade).to eq(0) }
  end

  context "with different but one matching clause" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        HAVING a > 2 AND a > 1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
        HAVING a > 1
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0.33) }
  end

  context "with matching clauses but different boolean operator" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        HAVING a > 2 AND a > 1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
        having a > 2 OR a > 1
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0.5) }
  end

  context "with not matching clauses and different boolean operator" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        HAVING a > 2 AND a > 1 OR a > 3
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
        HAVING a > 2 OR a > 3
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0.73) }
  end
end
```

spec/sql_assess/grader/limit_spec.rb

```ruby
require "spec_helper"
```

```ruby
RSpec.describe SqlAssess::Grader::Limit do
  subject do
    described_class.new(
      student_attributes: student_limit,
      instructor_attributes: instructor_limit
    )
  end

  context "same limit and offsert" do
    let(:student_limit) { { "limit": 1, "offset": 0 } }
    let(:instructor_limit) { { "limit": 1, "offset": 0 } }

    it { expect(subject.rounded_grade).to eq(1) }
  end

  context "same limit but different offset" do
    let(:student_limit) { { "limit": 1, "offset": 1 } }
    let(:instructor_limit) { { "limit": 1, "offset": 0 } }

    it { expect(subject.rounded_grade).to eq(0.5) }
  end

  context "different limit but same offset" do
    let(:student_limit) { { "limit": 2, "offset": 0 } }
    let(:instructor_limit) { { "limit": 1, "offset": 0 } }

    it { expect(subject.rounded_grade).to eq(0.5) }
  end

  context "different limit and offset" do
    let(:student_limit) { { "limit": 2, "offset": 0 } }
    let(:instructor_limit) { { "limit": 1, "offset": 2 } }

    it { expect(subject.rounded_grade).to eq(0) }
  end
end
```

spec/sql_assess/grader/order_by_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Grader::OrderBy do
  subject do
    described_class.new(
      student_attributes: attributes[:student][:order_by],
      instructor_attributes: attributes[:instructor][:order_by]
    )
  end

  before do
    connection.query("CREATE TABLE table1 (id1 integer, id2 integer)")
    connection.query("CREATE TABLE table2 (id3 integer, id4 integer)")
  end

  let(:attributes) do
    SqlAssess::QueryAttributeExtractor.new.extract(
       instructor_query, student_query
    )
  end

  context "with no order by" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table1
      SQL
    end

    it { expect(subject.rounded_grade).to eq(1) }
  end

  context "with one equal order by" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        ORDER BY a ASC
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table1
        ORDER BY a ASC
      SQL
    end

    it { expect(subject.rounded_grade).to eq(1) }
  end

  context "with two equal order by" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        ORDER BY a, b
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table1
        ORDER BY a, b
      SQL
    end

    it { expect(subject.rounded_grade).to eq(1) }
  end

  context "with one equal and one different order by" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        ORDER BY a, b
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table1
        ORDER BY a, c
      SQL
    end
```

```
    it { expect(subject.rounded_grade).to eq(0.5) }
  end

  context "with one equal and one different order by" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        ORDER BY a, b
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table1
        ORDER BY a
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0.67) }
  end

  context "with reversed two order by" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        ORDER BY a, b
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table1
        ORDER BY b, a
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0.25) }
  end

  context "with reversed two order by" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        ORDER BY a ASC, b
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table1
        ORDER BY b, a DESC
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0.19) }
  end
end
```

spec/sql_assess/grader/tables_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Grader::Tables do
  subject do
```

```
    described_class.new(
      student_attributes: attributes[:student][:tables],
      instructor_attributes: attributes[:instructor][:tables]
    )
  end

  before do
    connection.query("CREATE TABLE table1 (id1 integer, id2 integer)")
    connection.query("CREATE TABLE table2 (id3 integer, id4 integer)")
  end

  let(:attributes) do
    SqlAssess::QueryAttributeExtractor.new.extract(
       instructor_query, student_query
    )
  end

  context "with only base table but different" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0) }
  end

  context "with only base table equal" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table1
      SQL
    end

    it { expect(subject.rounded_grade).to eq(1) }
  end

  context "with only base table and join equal" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1 left join table2 on table2.id = table1.id
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table1 left join table2 on table2.id = table1.id
      SQL
    end

    it { expect(subject.rounded_grade).to eq(1) }
```

```ruby
end

context "with base equal but join condition totally different" do
  let(:student_query) do
    <<-SQL
      SELECT a from table1 left join table2 on table2.id = table1.id
    SQL
  end

  let(:instructor_query) do
    <<-SQL
      SELECT a from table1 left join table3 on table3.id = table1.id2
    SQL
  end

  it { expect(subject.rounded_grade).to eq(0.5) }
end

context "with base equal but join type different" do
  let(:student_query) do
    <<-SQL
      SELECT a from table1 left join table2 on table2.id = table1.id
    SQL
  end

  let(:instructor_query) do
    <<-SQL
      SELECT a from table1 right join table2 on table2.id = table1.id
    SQL
  end

  it { expect(subject.rounded_grade).to eq(BigDecimal(0.69, 2)) }
end

context "with base equal but join condition different" do
  let(:student_query) do
    <<-SQL
      SELECT a from table1 left join table2 on table2.id = table1.id
    SQL
  end

  let(:instructor_query) do
    <<-SQL
      SELECT a from table1 left join table2 on table2.id2 = table1.id
    SQL
  end

  it { expect(subject.rounded_grade).to eq(BigDecimal.new(0.69, 2)) }
end

context "with base equal but join condition and type different" do
  let(:student_query) do
    <<-SQL
      SELECT a from table1 left join table2 on table2.id = table1.id
    SQL
  end

  let(:instructor_query) do
    <<-SQL
      SELECT a from table1 right join table2 on table2.id2 = table1.id
    SQL
  end
```

```
    it { expect(subject.rounded_grade).to eq(0.63) }
  end

  context "with two equal subquery" do
    let(:student_query) do
      <<-SQL
        SELECT id1 from (SELECT id1 from table1)
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT id1 from (SELECT id1 from table1)
      SQL
    end

    it { expect(subject.rounded_grade).to eq(1) }
  end
end
```

spec/sql_assess/grader/where_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Grader::Where do
  subject do
    described_class.new(
      student_attributes: attributes[:student][:where_tree],
      instructor_attributes: attributes[:instructor][:where_tree]
    )
  end

  let(:attributes) do
    SqlAssess::QueryAttributeExtractor.new.extract(
       instructor_query, student_query
    )
  end

  context "with no where statements" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
      SQL
    end

    it { expect(subject.rounded_grade).to eq(1) }
  end


  context "with no where for student, but where for teacher" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
        WHERE a > 1
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0) }
  end


  context "with where for student, but no where for teacher" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        WHERE a > 1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0) }
  end

  context "with equal where" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        WHERE a > 1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
        WHERE a > 1
      SQL
    end

    it { expect(subject.rounded_grade).to eq(1) }
  end

  context "with different where" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        WHERE a > 2
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
        WHERE a > 1
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0) }
  end

  context "with different but one matching clause" do
```

```ruby
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        WHERE a > 2 AND a > 1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
        WHERE a > 1
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0.33) }
  end

  context "with matching clauses but different boolean operator" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        WHERE a > 2 AND a > 1
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
        WHERE a > 2 OR a > 1
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0.5) }
  end

  context "with not matching clauses and different boolean operator" do
    let(:student_query) do
      <<-SQL
        SELECT a from table1
        WHERE a > 2 AND a > 1 OR a > 3
      SQL
    end

    let(:instructor_query) do
      <<-SQL
        SELECT a from table2
        WHERE a > 2 OR a > 3
      SQL
    end

    it { expect(subject.rounded_grade).to eq(0.73) }
  end
end
```

spec/sql_assess/parsers/columns_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Parsers::Columns do
  subject { described_class.new(query) }

  context "with one column in select" do
    let(:query) { "SELECT id" }
    it "returns star" do
```

```
      expect(subject.columns).to eq(["`id`"])
    end
  end

  context "with two column in select" do
    let(:query) { "SELECT id, id2" }
    it "returns star" do
      expect(subject.columns).to eq(["`id`", "`id2`"])
    end
  end
end
```

spec/sql_assess/parsers/distinct_filter_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Parsers::DistinctFilter do
  subject { described_class.new(query) }

  context "with no filter in select" do
    let(:query) { "SELECT id, id2" }
    it "returns ALL" do
      expect(subject.distinct_filter).to eq("ALL")
    end
  end

  context "with ALL in select" do
    let(:query) { "SELECT ALL id, id2" }
    it "returns ALL" do
      expect(subject.distinct_filter).to eq("ALL")
    end
  end

  context "with DISTINCTROW in select" do
    let(:query) { "SELECT DISTINCTROW id, id3" }
    it "returns DISTINCTROW" do
      expect(subject.distinct_filter).to eq("DISTINCTROW")
    end
  end

  context "with DISTINCT in select" do
    let(:query) { "SELECT DISTINCT C1, c2, c3 FROM t1" }
    it "returns DISTINCT" do
      expect(subject.distinct_filter).to eq("DISTINCT")
    end
  end
end
```

spec/sql_assess/parsers/group_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Parsers::Group do
  subject { described_class.new(query) }

  context "with no group" do
    let(:query) { "SELECT id FROM t1 " }
    it "returns star" do
      expect(subject.group).to eq([])
    end
  end

  context "with one column in group" do
    let(:query) { "SELECT id, id2 FROM t1 GROUP BY id" }
    it "returns star" do
```

```ruby
      expect(subject.group).to eq(["`id`"])
    end
  end

  context "with two columns in group" do
    let(:query) { "SELECT id, id2 FROM t1 GROUP BY id, id2" }
    it "returns star" do
      expect(subject.group).to eq(["`id`", "`id2`"])
    end
  end
end
```

spec/sql_assess/parsers/having_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Parsers::Having do
  subject { described_class.new(query) }

  context "#having" do
    context "with no having clause" do
      let(:query) { "SELECT * from table1" }

      it "returns an empty hash" do
        expect(subject.having).to eq({})
      end
    end

    context "with a single having condition" do
      context "equal condition" do
        let(:query) { "SELECT * from table1 HAVING id = 1" }

        it "returns the correct result" do
          expect(subject.having).to eq({
            type: "EQUALS",
            left: "`id`",
            right: "1",
            sql: "`id` = 1",
          })
        end
      end

      context "less condition" do
        let(:query) { "SELECT * from table1 HAVING id < 1" }

        it "returns the correct result" do
          expect(subject.having).to eq({
            type: "LESS",
            left: "`id`",
            right: "1",
            sql: "`id` < 1",
          })
        end
      end
    end

    context "with an AND conidtion" do
      context "with two queries" do
        let(:query) { "SELECT * from table1 HAVING id = 1 AND id < 3" }

        it "returns the correct result" do
          expect(subject.having).to eq({
            type: "AND",
            clauses: [
          {
            type: "EQUALS",
            left: "`id`",
            right: "1",
            sql: "`id` = 1",
          },
          {
            type: "LESS",
            left: "`id`",
            right: "3",
            sql: "`id` < 3",
          }
        ]
      })
    end
  end

  context "with three queries" do
    let(:query) { "SELECT * from table1 HAVING id = 1 AND id < 3 AND id < 4" }

    it "returns the correct result" do
      expect(subject.having).to eq({
        type: "AND",
        clauses: [
          {
            type: "EQUALS",
            left: "`id`",
            right: "1",
            sql: "`id` = 1",
          },
          {
            type: "LESS",
            left: "`id`",
            right: "3",
            sql: "`id` < 3",
          },
          {
            type: "LESS",
            left: "`id`",
            right: "4",
            sql: "`id` < 4",
          }
        ]
      })
    end
  end
end

context "with an OR conidtion" do
  context "with two queries" do
    let(:query) { "SELECT * from table1 HAVING id = 1 OR id < 3" }

    it "returns the correct result" do
      expect(subject.having).to eq({
        type: "OR",
        clauses: [
          {
            type: "EQUALS",
            left: "`id`",
            right: "1",
            sql: "`id` = 1",
          },
          {
            type: "LESS",
            left: "`id`",
            right: "3",
            sql: "`id` < 3",
          }
        ]
      })
    end
  end

  context "with three queries" do
    let(:query) { "SELECT * from table1 HAVING id = 1 OR id < 3 OR id < 4" }

    it "returns the correct result" do
      expect(subject.having).to eq({
        type: "OR",
        clauses: [
          {
            type: "EQUALS",
            left: "`id`",
            right: "1",
            sql: "`id` = 1",
          },
          {
            type: "LESS",
            left: "`id`",
            right: "3",
            sql: "`id` < 3",
          },
          {
            type: "LESS",
            left: "`id`",
            right: "4",
            sql: "`id` < 4",
          }
        ]
      })
    end
  end
end

context "with an AND and OR conditions" do
  let(:query) { "SELECT * from table1 HAVING id = 1 AND id < 3 OR id < 4" }

  it "returs the correct hash" do
    expect(subject.having).to eq({
      type: "OR",
      clauses: [
        {
          type: "AND",
          clauses: [
            {
              type: "EQUALS",
              left: "`id`",
              right: "1",
              sql: "`id` = 1",
            },
            {
              type: "LESS",
              left: "`id`",
              right: "3",
              sql: "`id` < 3",
            },
            ]
          },
          {
            type: "LESS",
            left: "`id`",
            right: "4",
            sql: "`id` < 4",
          }
        ]
      })
    end
  end
end

context '#having_tree' do
  context 'with no clause' do
    let(:query) { 'SELECT * from table1' }

    it { expect(subject.having_tree).to eq({}) }
  end

  context 'with a having clause' do
    let(:query) { "SELECT * from table1 HAVING #{conditions}" }

    context 'with only one condition' do
      let(:conditions) { 'a > 1' }

      it 'returns the appropiate tree' do
        expect(subject.having_tree).to eq({
          is_inner: false,
          type: "GREATER",
          left: "`a`",
          right: "1",
          sql: "`a` > 1"
        })
      end
    end

    context 'with two condition a ^ b' do
      let(:conditions) { 'a > 1 AND b > 1' }

      it 'returns the appropiate tree' do
        expect(subject.having_tree).to eq({
          is_inner: true,
          type: "AND",
          left_clause: {
            is_inner: false,
            type: "GREATER",
            left: "`a`",
            right: "1",
            sql: "`a` > 1"
          },
          right_clause: {
            is_inner: false,
            type: "GREATER",
            left: "`b`",
            right: "1",
            sql: "`b` > 1"
          }
        })
      end
    end
```

```
context 'with three conditions a ^ b ^ c' do
  let(:conditions) { 'a > 1 AND b > 1 AND c > 1' }

  it 'returns the appropiate tree' do
    expect(subject.having_tree).to eq({
      is_inner: true,
      type: "AND",
      left_clause: {
        is_inner: true,
        type: "AND",
        left_clause: {
          is_inner: false,
          type: "GREATER",
          left: "`a`",
          right: "1",
          sql: "`a` > 1"
        },
        right_clause: {
          is_inner: false,
          type: "GREATER",
          left: "`b`",
          right: "1",
          sql: "`b` > 1"
        }
      },
      right_clause: {
        is_inner: false,
        type: "GREATER",
        left: "`c`",
        right: "1",
        sql: "`c` > 1"
      }
    })
  end
end

context 'with three conditions a ^ b V C' do
  let(:conditions) { 'a > 1 AND b > 1 OR c > 1' }

  it 'returns the appropiate tree' do
    expect(subject.having_tree).to eq({
      is_inner: true,
      type: "OR",
      left_clause: {
        is_inner: true,
        type: "AND",
        left_clause: {
          is_inner: false,
          type: "GREATER",
          left: "`a`",
          right: "1",
          sql: "`a` > 1"
        },
        right_clause: {
          is_inner: false,
          type: "GREATER",
          left: "`b`",
          right: "1",
          sql: "`b` > 1"
        }
      },
      right_clause: {
        is_inner: false,
        type: "GREATER",
        left: "`c`",
        right: "1",
        sql: "`c` > 1"
      }
    })
  end
end

context 'with three conditions a ^ (b V C)' do
  let(:conditions) { 'a > 1 AND (b > 1 OR c > 1)' }

  it 'returns the appropiate tree' do
    expect(subject.having_tree).to eq({
      is_inner: true,
      type: "AND",
      left_clause: {
        is_inner: false,
        type: "GREATER",
        left: "`a`",
        right: "1",
        sql: "`a` > 1"
      },
      right_clause: {
        is_inner: true,
        type: "OR",
        left_clause: {
          is_inner: false,
          type: "GREATER",
          left: "`b`",
          right: "1",
          sql: "`b` > 1"
        },
        right_clause: {
          is_inner: false,
          type: "GREATER",
          left: "`c`",
          right: "1",
          sql: "`c` > 1"
        }
      },
    })
  end
end

context 'with four conditions (a V c)^ (b V C)' do
  let(:conditions) { '(a > 1 OR c > 1) AND (b > 1 OR c > 1)' }

  it 'returns the appropiate tree' do
    expect(subject.having_tree).to eq({
      is_inner: true,
      type: "AND",
      left_clause: {
        is_inner: true,
        type: "OR",
        left_clause: {
          is_inner: false,
          type: "GREATER",
          left: "`a`",
          right: "1",
          sql: "`a` > 1"
        },
        right_clause: {
            is_inner: false,
            type: "GREATER",
            left: "`c`",
            right: "1",
            sql: "`c` > 1"
          }
        },
        right_clause: {
          is_inner: true,
          type: "OR",
          left_clause: {
            is_inner: false,
            type: "GREATER",
            left: "`b`",
            right: "1",
            sql: "`b` > 1"
          },
          right_clause: {
            is_inner: false,
            type: "GREATER",
            left: "`c`",
            right: "1",
            sql: "`c` > 1"
          }
        },
      })
    end
  end
end
end
end
```

spec/sql_assess/parsers/limit_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Parsers::Limit do
  subject { described_class.new(query) }

  context "with no limit" do
    let(:query) { "SELECT * from table1" }
    it "returns the correct limit" do
      expect(subject.limit).to eq({
        "limit": "inf",
        "offset": 0
      })
    end
  end

  context "with limit but no offset" do
    let(:query) { "SELECT * from table1 LIMIT 1" }
    it "returns the correct limit" do
      expect(subject.limit).to eq({
        "limit": 1,
        "offset": 0
      })
    end
  end

  context "with limit and offsert" do
    let(:query) { "SELECT * from table1 LIMIT 1 OFFSET 2" }

    it "returns the correct limit" do
      expect(subject.limit).to eq({
        "limit": 1,
        "offset": 2
      })
    end
  end
end
```

spec/sql_assess/parsers/order_by_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Parsers::OrderBy do
  subject { described_class.new(query) }

  context "with order by one column" do
    let(:query) { "SELECT * from table1 ORDER BY id" }
    it "returns the order by clause" do
      expect(subject.order).to eq([{
        column: "`id` ASC",
        position: 0,
      }])
    end
  end

  context "with order by multiple columns" do
    let(:query) { "SELECT * from table1 ORDER BY id, id2 DESC" }
    it "returns the order by clause" do
      expect(subject.order).to eq([
        {
          column: "`id` ASC",
          position: 0,
        }, {
          column: "`id2` DESC",
          position: 1,
        }
      ])
    end
  end

  context "with order by not present" do
    let(:query) { "SELECT * from table1" }

    it "returns empty array" do
      expect(subject.order).to eq([])
    end
  end
end
```

spec/sql_assess/parsers/tables_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Parsers::Tables do
  subject { described_class.new(query) }

  context "with no table" do
    let(:query) { "SELECT 1" }

    it "returns an empty array" do
      expect(subject.tables).to eq([])
    end
  end

  context "with one table" do
    let(:query) { "SELECT * from table1" }
    it "returns an array containing the tables" do
      expect(subject.tables).to eq([
        {
          type: "table",
          table: "`table1`",
          sql: "`table1`",
        }
      ])
    end
  end

  context "with a cross join" do
    let(:query) { "SELECT * from table1, table2" }

    it "returns an array containing the tables" do
      expect(subject.tables).to eq([
        {
          type: "table",
          table: "`table1`",
          sql: "`table1`",
        },
        {
          join_type: "CROSS JOIN",
          table: {
            type: "table",
            table: "`table2`",
            sql: "`table2`",
          },
          sql: "CROSS JOIN `table2`",
        }
      ])
    end
  end

  context "a table and a inner join" do
    let(:query) { "SELECT * FROM table1 INNER JOIN table2 ON table1.id = table2.id" }

    it "returns an array containing the tables" do
      expect(subject.tables).to eq([
        {
          type: "table",
          table: "`table1`",
          sql: "`table1`",
        },
        {
          join_type: "INNER JOIN",
          table: {
            type: "table",
            table: "`table2`",
            sql: "`table2`",
          },
          sql: "INNER JOIN `table2` ON `table1`.`id` = `table2`.`id`",
          condition: {
            type: "EQUALS",
            left: "`table1`.`id`",
            right: "`table2`.`id`",
            sql: "`table1`.`id` = `table2`.`id`"
          }
        }
      ])
    end
  end
```

```
context "a table and two left join" do
  let(:query) do
    <<-SQL.squish
      SELECT *
      FROM
        table1
        LEFT JOIN table2 ON table1.id = table2.id
        LEFT JOIN table3 ON table3.id = table2.id
    SQL
  end

  it "returns an array containing the tables" do
    expect(subject.tables).to eq([
      {
        type: "table",
        table: "`table1`",
        sql: "`table1`",
      },
      {
        join_type: "LEFT JOIN",
        table: {
          type: "table",
          table: "`table2`",
          sql: "`table2`",
        },
        condition: {
          type: "EQUALS",
          left: "`table1`.`id`",
          right: "`table2`.`id`",
          sql: "`table1`.`id` = `table2`.`id`"
        },
        sql: "LEFT JOIN `table2` ON `table1`.`id` = `table2`.`id`"
      },
      {
        join_type: "LEFT JOIN",
        table: {
          type: "table",
          table: "`table3`",
          sql: "`table3`",
        },
        condition: {
          type: "EQUALS",
          left: "`table3`.`id`",
          right: "`table2`.`id`",
          sql: "`table3`.`id` = `table2`.`id`"
        },
        sql: "LEFT JOIN `table3` ON `table3`.`id` = `table2`.`id`"
      }
    ])
  end
end

context "a subquery" do
  let(:query) do
    <<-SQL.squish
      SELECT *
      FROM (SELECT id FROM table1)
    SQL
  end

  it "returns an array containing the tables" do
    expect(subject.tables).to eq([
      {
          type: "Subquery",
          sql: "(SELECT `id` FROM `table1`)",
          attributes: {
            columns: ["`id`"],
            order_by: [],
            where: {},
            where_tree: {},
            tables: [{type: "table", table: "`table1`", sql: "`table1`"}],
            distinct_filter: "ALL",
            limit: {limit: "inf", offset: 0},
            group: [],
            having: {},
            having_tree: {},
          }
        }
      ])
    end
  end
end
```

spec/sql_assess/parsers/where_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Parsers::Where do
  subject { described_class.new(query) }

  context "#where" do
    context "with no where clause" do
      let(:query) { "SELECT * from table1" }

      it "returns an empty hash" do
        expect(subject.where).to eq({})
      end
    end

    context "with a single where condition" do
      context "equal condition" do
        let(:query) { "SELECT * from table1 WHERE id = 1" }

        it "returns the correct result" do
          expect(subject.where).to eq({
            type: "EQUALS",
            left: "`id`",
            right: "1",
            sql: "`id` = 1",
          })
        end
      end

      context "less condition" do
        let(:query) { "SELECT * from table1 WHERE id < 1" }

        it "returns the correct result" do
          expect(subject.where).to eq({
            type: "LESS",
            left: "`id`",
            right: "1",
            sql: "`id` < 1",
          })
        end
      end
    end
```

```
context "with an AND conidtion" do
  context "with two queries" do
    let(:query) { "SELECT * from table1 WHERE id = 1 AND id < 3" }

    it "returns the correct result" do
      expect(subject.where).to eq({
        type: "AND",
        clauses: [
          {
            type: "EQUALS",
            left: "`id`",
            right: "1",
            sql: "`id` = 1",
          },
          {
            type: "LESS",
            left: "`id`",
            right: "3",
            sql: "`id` < 3",
          }
        ]
      })
    end
  end

  context "with three queries" do
    let(:query) { "SELECT * from table1 WHERE id = 1 AND id < 3 AND id < 4" }

    it "returns the correct result" do
      expect(subject.where).to eq({
        type: "AND",
        clauses: [
          {
            type: "EQUALS",
            left: "`id`",
            right: "1",
            sql: "`id` = 1",
          },
          {
            type: "LESS",
            left: "`id`",
            right: "3",
            sql: "`id` < 3",
          },
          {
            type: "LESS",
            left: "`id`",
            right: "4",
            sql: "`id` < 4",
          }
        ]
      })
    end
  end
end

context "with an OR conidtion" do
  context "with two queries" do
    let(:query) { "SELECT * from table1 WHERE id = 1 OR id < 3" }

    it "returns the correct result" do
      expect(subject.where).to eq({
        type: "OR",
        clauses: [
          {
            type: "EQUALS",
            left: "`id`",
            right: "1",
            sql: "`id` = 1",
          },
          {
            type: "LESS",
            left: "`id`",
            right: "3",
            sql: "`id` < 3",
          }
        ]
      })
    end
  end

  context "with three queries" do
    let(:query) { "SELECT * from table1 WHERE id = 1 OR id < 3 OR id < 4" }

    it "returns the correct result" do
      expect(subject.where).to eq({
        type: "OR",
        clauses: [
          {
            type: "EQUALS",
            left: "`id`",
            right: "1",
            sql: "`id` = 1",
          },
          {
            type: "LESS",
            left: "`id`",
            right: "3",
            sql: "`id` < 3",
          },
          {
            type: "LESS",
            left: "`id`",
            right: "4",
            sql: "`id` < 4",
          }
        ]
      })
    end
  end
end

context "with an AND and OR conditions" do
  let(:query) { "SELECT * from table1 WHERE id = 1 AND id < 3 OR id < 4" }

  it "returs the correct hash" do
    expect(subject.where).to eq({
      type: "OR",
      clauses: [
        {
          type: "AND",
          clauses: [
            {
              type: "EQUALS",
              left: "`id`",
              right: "1",
              sql: "`id` = 1",
            },
            {
              type: "LESS",
              left: "`id`",
              right: "3",
              sql: "`id` < 3",
            },
          ]
        },
        {
          type: "LESS",
          left: "`id`",
          right: "4",
          sql: "`id` < 4",
        }
      ]
    })
  end
end
end

context '#where_tree' do
  context 'with no clause' do
    let(:query) { 'SELECT * from table1' }

    it { expect(subject.where_tree).to eq({}) }
  end

  context 'with a where clause' do
    let(:query) { "SELECT * from table1 WHERE #{conditions}" }

    context 'with only one condition' do
      let(:conditions) { 'a > 1' }

      it 'returns the appropiate tree' do
        expect(subject.where_tree).to eq({
          is_inner: false,
          type: "GREATER",
          left: "`a`",
          right: "1",
          sql: "`a` > 1"
        })
      end
    end

    context 'with two condition a ^ b' do
      let(:conditions) { 'a > 1 AND b > 1' }

      it 'returns the appropiate tree' do
        expect(subject.where_tree).to eq({
          is_inner: true,
          type: "AND",
          left_clause: {
            is_inner: false,
            type: "GREATER",
            left: "`a`",
            right: "1",
            sql: "`a` > 1"
          },
          right_clause: {
            is_inner: false,
            type: "GREATER",
            left: "`b`",
            right: "1",
            sql: "`b` > 1"
          }
        })
      end
    end

    context 'with three conditions a ^ b ^ c' do
      let(:conditions) { 'a > 1 AND b > 1 AND c > 1' }

      it 'returns the appropiate tree' do
        expect(subject.where_tree).to eq({
          is_inner: true,
          type: "AND",
          left_clause: {
            is_inner: true,
            type: "AND",
            left_clause: {
              is_inner: false,
              type: "GREATER",
              left: "`a`",
              right: "1",
              sql: "`a` > 1"
            },
            right_clause: {
              is_inner: false,
              type: "GREATER",
              left: "`b`",
              right: "1",
              sql: "`b` > 1"
            }
          },
          right_clause: {
            is_inner: false,
            type: "GREATER",
            left: "`c`",
            right: "1",
            sql: "`c` > 1"
          }
        })
      end
    end

    context 'with three conditions a ^ b V C' do
      let(:conditions) { 'a > 1 AND b > 1 OR c > 1' }

      it 'returns the appropiate tree' do
        expect(subject.where_tree).to eq({
          is_inner: true,
          type: "OR",
          left_clause: {
            is_inner: true,
            type: "AND",
            left_clause: {
              is_inner: false,
              type: "GREATER",
              left: "`a`",
              right: "1",
              sql: "`a` > 1"
            },
            right_clause: {
              is_inner: false,
              type: "GREATER",
              left: "`b`",
              right: "1",
              sql: "`b` > 1"
            }
          },
          right_clause: {
            is_inner: false,
            type: "GREATER",
            left: "`c`",
            right: "1",
            sql: "`c` > 1"
          }
        })
      end
    end

    context 'with three conditions a ^ (b V C)' do
      let(:conditions) { 'a > 1 AND (b > 1 OR c > 1)' }

      it 'returns the appropiate tree' do
        expect(subject.where_tree).to eq({
          is_inner: true,
          type: "AND",
          left_clause: {
            is_inner: false,
            type: "GREATER",
            left: "`a`",
            right: "1",
            sql: "`a` > 1"
          },
          right_clause: {
            is_inner: true,
            type: "OR",
            left_clause: {
              is_inner: false,
              type: "GREATER",
              left: "`b`",
              right: "1",
              sql: "`b` > 1"
            },
            right_clause: {
              is_inner: false,
              type: "GREATER",
              left: "`c`",
              right: "1",
              sql: "`c` > 1"
            }
          },
        })
      end
    end

    context 'with four conditions (a V c)^ (b V C)' do
      let(:conditions) { '(a > 1 OR c > 1) AND (b > 1 OR c > 1)' }

      it 'returns the appropiate tree' do
        expect(subject.where_tree).to eq({
          is_inner: true,
          type: "AND",
          left_clause: {
            is_inner: true,
            type: "OR",
              left_clause: {
                is_inner: false,
                type: "GREATER",
                left: "`a`",
                right: "1",
                sql: "`a` > 1"
              },
              right_clause: {
                is_inner: false,
                type: "GREATER",
                left: "`c`",
                right: "1",
                sql: "`c` > 1"
              }
            },
            right_clause: {
              is_inner: true,
              type: "OR",
              left_clause: {
                is_inner: false,
                type: "GREATER",
                left: "`b`",
                right: "1",
                sql: "`b` > 1"
              },
              right_clause: {
                is_inner: false,
                type: "GREATER",
                left: "`c`",
                right: "1",
                sql: "`c` > 1"
              }
            },
          })
        end
      end
    end
  end
end
```

spec/sql_assess/query_attribute_extractor_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::QueryAttributeExtractor do
  subject { described_class.new }

  context "columns" do
    before do
      connection.query('CREATE TABLE table1 (id integer, second integer);')
      connection.query('INSERT INTO table1 (id, second) values(1, 3);')
      connection.query('INSERT INTO table1 (id, second) values(2, 4);')
    end

    let(:instructor_query) { "SELECT id from table1" }
    let(:student_query) { "SELECT second from table1" }

    it "returns the correct format" do
      result = subject.extract(instructor_query, student_query)
      expect(result).to match({
        student: {
          columns: an_instance_of(Array),
          order_by: an_instance_of(Array),
          where: an_instance_of(Hash),
```

```ruby
          where_tree: an_instance_of(Hash),
          tables: an_instance_of(Array),
          distinct_filter: an_instance_of(String),
          limit: an_instance_of(Hash),
          group: an_instance_of(Array),
          having: an_instance_of(Hash),
          having_tree: an_instance_of(Hash),
        },
        instructor: {
          columns: an_instance_of(Array),
          order_by: an_instance_of(Array),
          where: an_instance_of(Hash),
          where_tree: an_instance_of(Hash),
          tables: an_instance_of(Array),
          distinct_filter: an_instance_of(String),
          limit: an_instance_of(Hash),
          group: an_instance_of(Array),
          having: an_instance_of(Hash),
          having_tree: an_instance_of(Hash),
        },
      })
    end
  end
end
```

spec/sql_assess/query_comparator_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::QueryComparator do
  subject { described_class.new(connection) }

  context "success" do
    before do
      connection.query('CREATE TABLE table1 (id integer);')
      connection.query('INSERT INTO table1 (id) values(1);')
      connection.query('INSERT INTO table1 (id) values(2);')
    end

    context "when the results are the same" do
      it "returns the right result" do
        query = "SELECT * from table1 WHERE id = 1";

        expect(subject.compare(query, query)).to eq(true)
      end
    end

    context "when the results are different" do
      context "when the count is different" do
        it "returns the right result" do
          query = "SELECT * from table1 WHERE id = 1";
          wrong_query = "SELECT * from table1 WHERE id = 3";

          expect(subject.compare(query, wrong_query)).to eq(false)
        end
      end

      context "when the count is the same" do
        it "returns the right result" do
          query = "SELECT * from table1 WHERE id = 1";
          wrong_query = "SELECT * from table1 WHERE id = 2";

          expect(subject.compare(query, wrong_query)).to eq(false)
        end
      end
    end
  end
end
```

spec/sql_assess/query_comparison_result_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::QueryComparisonResult do
  subject { described_class.new(success: true, attributes: attributes) }

  let(:attributes) do
    SqlAssess::QueryAttributeExtractor.new.extract(
      (
        <<-SQL.squish
          SELECT table1.a
          FROM table1
        SQL
      ), (
        <<-SQL.squish
          SELECT table1.a
          FROM table1
        SQL
      )
    )
  end
  context "#attributes_grade" do
    it "returns a hash" do
      expect(subject.attributes_grade).to match({
        columns: an_instance_of(BigDecimal),
        order_by: an_instance_of(BigDecimal),
        where: an_instance_of(BigDecimal),
        distinct_filter: an_instance_of(BigDecimal),
        limit: an_instance_of(BigDecimal),
        tables: an_instance_of(BigDecimal),
        group: an_instance_of(BigDecimal),
        having: an_instance_of(BigDecimal),
      })
    end
  end

  context "#message" do
    context "with grade = 100" do
      before do
        allow_any_instance_of(described_class).to receive(:calculate_grade).and_return(1)
      end

      it { expect(subject.message).to eq("Congratulations! Your solution is correct") }
    end

    context "with grade < 100" do
      before do
        allow_any_instance_of(described_class).to receive(:calculate_grade).and_return(0.9)
        allow_any_instance_of(described_class).to receive(:first_wrong_component).and_return(component)
      end

      context "with columns first_wrong_attribute" do
        let(:component) { :columns }

        it { expect(subject.message).to eq("Your query is not correct. Check what columns you are selecting.")
         ↪  }
      end
      context "with tables first_wrong_attribute" do
        let(:component) { :tables }

        it { expect(subject.message).to eq("Your query is not correct. Are you sure you are selecting the
        ↪  right tables?") }
      end

      context "with order_by first_wrong_attribute" do
        let(:component) { :order_by }

        it { expect(subject.message).to eq("Your query is not correct. Are you ordering the rows correctly?")
        ↪  }
      end

      context "with where first_wrong_attribute" do
        let(:component) { :where }

        it { expect(subject.message).to eq("Your query is not correct. Looks like you are selecting the right
        ↪  columns, but you are not selecting only the correct rows.") }
      end

      context "with distinct_filter first_wrong_attribute" do
        let(:component) { :distinct_filter }

        it { expect(subject.message).to eq("Your query is not correct. What about duplicates? What does the
        ↪  exercise say?") }
      end

      context "with limit first_wrong_attribute" do
        let(:component) { :limit }

        it { expect(subject.message).to eq("Your query is not correct. Are you selecting the correct number of
        ↪  rows?") }
      end

      context "with group first_wrong_attribute" do
        let(:component) { :group }

        it { expect(subject.message).to eq("Your query is not correct. Are you grouping by the correct
        ↪  columns?") }
      end
    end
  end
end
```

spec/sql_assess/query_transformer_spec.rb

```
require "spec_helper"
require 'yaml'

RSpec.describe SqlAssess::QueryTransformer do
  subject { described_class.new(connection) }

  context "when encountering an error" do
    it "raises a CanonicalizationError" do
      expect { subject.transform("adad * from a") }
        .to raise_error(SqlAssess::CanonicalizationError)
    end
  end

  yaml = YAML.load_file("spec/fixtures/transformer_integration_tests.yml")

  yaml.each do |test|
    it "transform #{test['query']} to #{test['expected_result']}" do
```

```ruby
      # Seed data
      connection.multiple_query(test["schema"])
      # Check if queries from file are correct
      connection.query(test["query"])
      connection.query(test["expected_result"])
      # Check transformation
      expect(subject.transform(test["query"])).to eq(test["expected_result"])
    end
  end

  yaml2 = YAML.load_file("spec/fixtures/transformer_hacker_rank_integration_tests.yml")

  yaml2.each do |test|
    if test["support"] == false
      xit "#{test['name']}" do
        execute_query(test)
      end
    else
      it "#{test['name']}: transform #{test['query'].squish} to #{test['expected_result'].squish}" do
        execute_query(test)
      end
    end
  end

  def execute_query(test)
    # Seed data
    connection.multiple_query(test["schema"])
    # Check if queries from file are correct
    connection.query(test["query"])
    connection.query(test["expected_result"])
    # Check transformation
    expect(subject.transform(test["query"])).to eq(test["expected_result"].squish)
  end
end
```

spec/sql_assess/runner_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Runner do
  subject { described_class.new(connection) }

  describe "#create_schema" do
    context "with a correct command" do
      context "with a single command" do
        it "runs the command" do
          subject.create_schema('CREATE TABLE table1 (id integer);')

          tables = connection.query("SHOW tables");
          expect(tables.first["Tables_in_local_db"]).to eq("table1")
        end
      end

      context "with multiple commands" do
        it "runs all commands" do
          subject.create_schema('CREATE TABLE table1 (id integer); CREATE TABLE table2 (id integer);')

          tables = connection.query("SHOW tables");
          expect(tables.size).to eq(2)
          expect(tables.map{ |line| line["Tables_in_local_db"] }).to eq(["table1", "table2"])
        end
      end
    end
    context "with an incorrect command" do
      it "raises an exception" do
        expect do
          subject.create_schema('CREATE TABLES table1 (id integer);')
        end.to raise_error(
          SqlAssess::DatabaseSchemaError,
          /near .+ at line 1/
        )
      end
    end
  end

  describe "#seed_initial_data" do
    before do
      subject.create_schema('CREATE TABLE table1 (id integer);')
    end

    context "with a correct command" do
      context "with a single command" do
        it "runs the command" do
          subject.seed_initial_data('INSERT INTO table1 (id) values(1);')

          rows = connection.query('SELECT * FROM table1');
          expect(rows.count).to eq(1)
          expect(rows.first["id"]).to eq(1)
        end
      end

      context "with multiple commands" do
        it "runs all commands" do
          subject.seed_initial_data('INSERT INTO table1 (id) values(1); INSERT INTO table1 (id) values(2);')

          rows = connection.query('SELECT * FROM table1');
          expect(rows.size).to eq(2)
          expect(rows.map{ |line| line["id"] }).to eq([1, 2])
        end
      end
    end

    context "with an incorrect command" do
      it "raises an exception" do
        expect do
          subject.seed_initial_data('INSERT INTO table1 (id2) values("ab");')
        end.to raise_error(
          SqlAssess::DatabaseSeedError,
          "Unknown column 'id2' in 'field list'"
        )
      end
    end
  end

  context "#execute_query" do
    before do
      connection.query('CREATE TABLE table1 (id integer);')
      connection.query('INSERT INTO table1 (id) values(1);')
      connection.query('INSERT INTO table1 (id) values(2);')
    end

    context "with a wrong query" do
      let(:query) { "SELECT id2 from table1;" }

      it "raises an exception" do
        expect { subject.execute_query(query) }.to raise_error(
          SqlAssess::DatabaseQueryExecutionFailed
        )
      end
    end

    context "with a correct query" do
      let(:query) { "SELECT id2 from table1;" }

      it "raises an exception" do
        expect { subject.execute_query(query) }.to raise_error(
          SqlAssess::DatabaseQueryExecutionFailed,
          "Unknown column 'id2' in 'field list'"
        )
      end
    end
  end
end
```

spec/sql_assess/transformers/all_columns_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::AllColumns do
  subject { described_class.new(connection) }

  before do
    connection.query("CREATE TABLE table1 (id1 integer, id2 integer)")
    connection.query("CREATE TABLE table2 (id3 integer, id4 integer)")
  end

  context "for a non-join query" do
    context "when there is no *" do
      it "returns the same query" do
        expect(subject.transform("SELECT id1 FROM table1")).to eq("SELECT `id1` FROM `table1`")
      end

      it "returns the same query" do
        expect(subject.transform("SELECT id2 FROM table1")).to eq("SELECT `id2` FROM `table1`")
      end

      it "returns the same query" do
        expect(subject.transform("SELECT id1, id2 FROM table1")).to eq("SELECT `id1`, `id2` FROM `table1`")
      end
    end

    context "when there is *" do
      it "returns the query containing all columns in select" do
        expect(subject.transform("SELECT * FROM table1")).to eq("SELECT `table1`.`id1`, `table1`.`id2` FROM
         ↪  `table1`")
      end
    end
  end

  context "for a join query" do
    context "when there is no *" do
      it "returns the same query" do
        expect(subject.transform("SELECT id1 FROM table1, table2")).to eq("SELECT `id1` FROM `table1` CROSS
         ↪  JOIN `table2`")
      end

      it "returns the same query" do
        expect(subject.transform("SELECT id4 FROM table1, table2")).to eq("SELECT `id4` FROM `table1` CROSS
         ↪  JOIN `table2`")
      end
```

```ruby
      it "returns the same query" do
        expect(subject.transform("SELECT id1, id2, id3 FROM table1, table2")).to eq("SELECT `id1`, `id2`,
        ↪  `id3` FROM `table1` CROSS JOIN `table2`")
      end
    end

    context "when there is *" do
      it "returns the query containing all columns in select" do
        expect(subject.transform("SELECT * FROM table1, table2"))
          .to eq("SELECT `table1`.`id1`, `table1`.`id2`, `table2`.`id3`, `table2`.`id4` FROM `table1` CROSS
          ↪  JOIN `table2`")
      end
    end
  end
end
```

spec/sql_assess/transformers/ambigous_columns/from_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Transformers::AmbigousColumns::From do
  subject { described_class.new(connection) }

  before do
    connection.query("CREATE TABLE table1 (id1 integer, id2 integer)")
    connection.query("CREATE TABLE table2 (id3 integer, id4 integer)")
  end

  context "with no join" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
      SQL
    end

    it "doesn't change the query" do
      expect(subject.transform(query)).to eq(query)
    end
  end

  context "with join clause but no ambigous column" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1` LEFT JOIN `table2` ON `table2`.`id3` = `table1`.`id1`
      SQL
    end

    it "doesn't change the query" do
      expect(subject.transform(query)).to eq(query)
    end
  end

  context "with join clause but with ambigous column" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1` LEFT JOIN `table2` ON `id3` = `id1`
        GROUP BY `id1`
      SQL
    end
    it "changes the query" do
      expect(subject.transform(query)).to eq(
        <<-SQL.squish
          SELECT `table1`.`id`, `table1`.`id2`
          FROM `table1` LEFT JOIN `table2` ON `table2`.`id3` = `table1`.`id1`
          GROUP BY `id1`
        SQL
      )
    end
  end

  context "with join clause but with ambigous column" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1` LEFT JOIN `table2` ON `id3` = `id1` AND `id4` = `id2`
        GROUP BY `id1`
      SQL
    end

    it "changes the query" do
      expect(subject.transform(query)).to eq(
        <<-SQL.squish
          SELECT `table1`.`id`, `table1`.`id2`
          FROM `table1` LEFT JOIN `table2` ON (`table2`.`id3` = `table1`.`id1` AND `table2`.`id4` =
↪  `table1`.`id2`)
          GROUP BY `id1`
        SQL
      )
    end
  end
end
```

spec/sql_assess/transformers/ambigous_columns/group_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::AmbigousColumns::Group do
  subject { described_class.new(connection) }

  before do
    connection.query("CREATE TABLE table1 (id1 integer, id2 integer)")
    connection.query("CREATE TABLE table2 (id3 integer, id4 integer)")
  end

  context "with no group clause" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
      SQL
    end

    it "doesn't change the query" do
      expect(subject.transform(query)).to eq(query)
    end
  end

  context "with group clause but no ambigous column" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
        GROUP BY `table1`.`id1`
      SQL
    end

    it "doesn't change the query" do
      expect(subject.transform(query)).to eq(query)
    end
  end

  context "with group clause but with an ambigous column" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
        GROUP BY `id1`
      SQL
    end

    it "changes the query" do
      expect(subject.transform(query)).to eq(
        <<-SQL.squish
          SELECT `table1`.`id`, `table1`.`id2`
          FROM `table1`
          GROUP BY `table1`.`id1`
        SQL
      )
    end
  end

  context "with group clause but with a column number" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
        GROUP BY 1
      SQL
    end

    it "changes the query" do
      expect(subject.transform(query)).to eq(
        <<-SQL.squish
          SELECT `table1`.`id`, `table1`.`id2`
          FROM `table1`
          GROUP BY `table1`.`id`
        SQL
      )
    end
  end
end
```

spec/sql_assess/transformers/ambigous_columns/having_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::AmbigousColumns::Having do
  subject { described_class.new(connection) }

  before do
    connection.query("CREATE TABLE table1 (id1 integer, id2 integer)")
    connection.query("CREATE TABLE table2 (id3 integer, id4 integer)")
  end

  context "with no HAVING clause" do
    let(:query) do
      <<-SQL.squish
      SELECT `table1`.`id`, `table1`.`id2`
      FROM `table1`
    SQL
  end

  it "doesn't change the query" do
    expect(subject.transform(query)).to eq(query)
  end
end

context "with HAVING clause but no ambigous column" do
  let(:query) do
    <<-SQL.squish
      SELECT `table1`.`id`, `table1`.`id2`
      FROM `table1`
      HAVING `table1`.`id1` > 1
    SQL
  end

  it "doesn't change the query" do
    expect(subject.transform(query)).to eq(query)
  end
end

context "with HAVING clause but with an ambigous column" do
  let(:query) do
    <<-SQL.squish
      SELECT `table1`.`id`, `table1`.`id2`
      FROM `table1`
      HAVING `id1` > 1
    SQL
  end

  it "changes the query" do
    expect(subject.transform(query)).to eq(
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
        HAVING `table1`.`id1` > 1
      SQL
    )
  end
end

context "with HAVING clause but with an ambigous column" do
  let(:query) do
    <<-SQL.squish
      SELECT `table1`.`id`, `table1`.`id2`
      FROM `table1`
      HAVING `id1` > 1 AND `id2` > 1
    SQL
  end

  it "changes the query" do
    expect(subject.transform(query)).to eq(
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
        HAVING (`table1`.`id1` > 1 AND `table1`.`id2` > 1)
      SQL
    )
  end
end
```

```
end
```

spec/sql_assess/transformers/ambigous_columns/order_by_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Transformers::AmbigousColumns::OrderBy do
  subject { described_class.new(connection) }

  before do
    connection.query("CREATE TABLE table1 (id1 integer, id2 integer)")
    connection.query("CREATE TABLE table2 (id3 integer, id4 integer)")
  end

  context "with no order clause" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
      SQL
    end

    it "doesn't change the query" do
      expect(subject.transform(query)).to eq(query)
    end
  end

  context "with order clause but no ambigous column" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
        ORDER BY `table1`.`id1` ASC
      SQL
    end

    it "doesn't change the query" do
      expect(subject.transform(query)).to eq(query)
    end
  end

  context "with order clause but with an ambigous column" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
        ORDER BY `id1` ASC
      SQL
    end

    it "changes the query" do
      expect(subject.transform(query)).to eq(
        <<-SQL.squish
          SELECT `table1`.`id`, `table1`.`id2`
          FROM `table1`
          ORDER BY `table1`.`id1` ASC
        SQL
      )
    end
  end

  context "with order clause but with a column number" do
    let(:query) do
```

```
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
        ORDER BY 1 ASC
      SQL
    end

    it "changes the query" do
      expect(subject.transform(query)).to eq(
        <<-SQL.squish
          SELECT `table1`.`id`, `table1`.`id2`
          FROM `table1`
          ORDER BY `table1`.`id` ASC
        SQL
      )
    end
  end
end
```

spec/sql_assess/transformers/ambigous_columns/select_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::AmbigousColumns::Select do
  subject { described_class.new(connection) }

  before do
    connection.query("CREATE TABLE table1 (id1 integer, id2 integer)")
    connection.query("CREATE TABLE table2 (id3 integer, id4 integer)")
  end

  it "adds the table name in front of the column" do
    expect(subject.transform("SELECT id1 FROM table1"))
      .to eq("SELECT `table1`.`id1` FROM `table1`")
  end

  it "leaves existing qualified columns unchanged" do
    expect(subject.transform("SELECT table1.id1 FROM table1"))
      .to eq("SELECT `table1`.`id1` FROM `table1`")
  end

  it "adds the table name in front of the column" do
    expect(subject.transform("SELECT id1, id3 FROM table1, table2"))
      .to eq("SELECT `table1`.`id1`, `table2`.`id3` FROM `table1` CROSS JOIN `table2`")
  end

  it "transforms the query" do
    expect(subject.transform("SELECT SUM(id1) FROM table1")).to eq("SELECT SUM(`table1`.`id1`) FROM `table1`")
  end
end
```

spec/sql_assess/transformers/ambigous_columns/where_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::AmbigousColumns::Where do
  subject { described_class.new(connection) }

  before do
    connection.query("CREATE TABLE table1 (id1 integer, id2 integer)")
    connection.query("CREATE TABLE table2 (id3 integer, id4 integer)")
  end

  context "with no where clause" do
    let(:query) do
```

```
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
      SQL
    end

    it "doesn't change the query" do
      expect(subject.transform(query)).to eq(query)
    end
  end

  context "with WHERE clause but no ambigous column" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
        WHERE `table1`.`id1` > 1
      SQL
    end

    it "doesn't change the query" do
      expect(subject.transform(query)).to eq(query)
    end
  end

  context "with where clause but with an ambigous column" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
        WHERE `id1` > 1
      SQL
    end

    it "changes the query" do
      expect(subject.transform(query)).to eq(
        <<-SQL.squish
          SELECT `table1`.`id`, `table1`.`id2`
          FROM `table1`
          WHERE `table1`.`id1` > 1
        SQL
      )
    end
  end

  context "with where clause but with an ambigous column" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
        WHERE `id1` > 1 AND `id2` > 1
      SQL
    end

    it "changes the query" do
      expect(subject.transform(query)).to eq(
        <<-SQL.squish
          SELECT `table1`.`id`, `table1`.`id2`
          FROM `table1`
          WHERE (`table1`.`id1` > 1 AND `table1`.`id2` > 1)
        SQL
      )
    end
  end
end
```

spec/sql_assess/transformers/base_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::Base do
  subject { described_class.new(@connection) }

  context "#transform" do
    it "throws an error" do
      expect { subject.transform }.to raise_error('Implement this method in subclass')
    end
  end

  context "#tables" do
    context "one table" do
      let(:query) { "SELECT * from t1" }

      it "returns the table" do
        expect(subject.tables(query)).to eq(["t1"])
      end
    end

    context "two tables" do
      let(:query) { "SELECT * from t1, t2" }

      it "returns the table" do
        expect(subject.tables(query)).to eq(["t1", "t2"])
      end
    end

    context "three tables" do
      let(:query) { "SELECT * from t1, t2 LEFT JOIN t3 on t1.id = t3.id" }

      it "returns the table" do
        expect(subject.tables(query)).to eq(["t1", "t2", "t3"])
      end
    end
  end
end
```

spec/sql_assess/transformers/between_prediate/from_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::BetweenPredicate::From do
  subject { described_class.new(connection) }

  context "when there is no having clause" do
    it "returns the same query" do
      expect(subject.transform("SELECT * FROM table")).to eq("SELECT * FROM `table`")
    end
  end

  context "when there is a having clause" do
    context "with no between query" do
      it "returns the same query" do
        expect(subject.transform("SELECT * FROM table, t3 LEFT JOIN t2 ON a = 1"))
          .to eq("SELECT * FROM `table` CROSS JOIN `t3` LEFT JOIN `t2` ON `a` = 1")
      end
    end

    context "with only a between query" do
```

```
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table, t3 LEFT JOIN t2 ON a BETWEEN 1 and 3"))
          .to eq("SELECT * FROM `table` CROSS JOIN `t3` LEFT JOIN `t2` ON (`a` >= 1 AND `a` <= 3)")
      end
    end

    context "with a between query and another type of query" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table LEFT JOIN t2 ON (a BETWEEN 1 and 3) AND b = 2"))
          .to eq("SELECT * FROM `table` LEFT JOIN `t2` ON ((`a` >= 1 AND `a` <= 3) AND `b` = 2)")
      end
    end

    context "with a between query and two other type of query" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table LEFT JOIN t2 ON a BETWEEN 1 and 3 AND b = 2 AND c = 3"))
          .to eq("SELECT * FROM `table` LEFT JOIN `t2` ON (((`a` >= 1 AND `a` <= 3) AND `b` = 2) AND `c` =
          ↪  3)")
      end
    end
  end
end
```

spec/sql_assess/transformers/between_prediate/having_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::BetweenPredicate::Having do
  subject { described_class.new(connection) }

  context "when there is no having clause" do
    it "returns the same query" do
      expect(subject.transform("SELECT * FROM table")).to eq("SELECT * FROM table")
    end
  end

  context "when there is a having clause" do
    context "with no between query" do
      it "returns the same query" do
        expect(subject.transform("SELECT * FROM table HAVING a = 1"))
          .to eq("SELECT * FROM `table` HAVING `a` = 1")
      end
    end

    context "with only a between query" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table HAVING a BETWEEN 1 and 3"))
          .to eq("SELECT * FROM `table` HAVING (`a` >= 1 AND `a` <= 3)")
      end
    end

    context "with a between query and another type of query" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table HAVING (a BETWEEN 1 and 3) AND b = 2"))
          .to eq("SELECT * FROM `table` HAVING ((`a` >= 1 AND `a` <= 3) AND `b` = 2)")
      end
    end

    context "with a between query and two other type of query" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table HAVING a BETWEEN 1 and 3 AND b = 2 AND c = 3"))
          .to eq("SELECT * FROM `table` HAVING (((`a` >= 1 AND `a` <= 3) AND `b` = 2) AND `c` = 3)")
      end
    end
  end
end
```

spec/sql_assess/transformers/between_prediate/where_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Transformers::BetweenPredicate::Where do
  subject { described_class.new(connection) }

  context "when there is no where clause" do
    it "returns the same query" do
      expect(subject.transform("SELECT * FROM table")).to eq("SELECT * FROM table")
    end
  end

  context "when there is a where clause" do
    context "with no between query" do
      it "returns the same query" do
        expect(subject.transform("SELECT * FROM table WHERE a = 1"))
          .to eq("SELECT * FROM `table` WHERE `a` = 1")
      end
    end

    context "with only a between query" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table WHERE a BETWEEN 1 and 3"))
          .to eq("SELECT * FROM `table` WHERE (`a` >= 1 AND `a` <= 3)")
      end
    end

    context "with a between query and another type of query" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table WHERE (a BETWEEN 1 and 3) AND b = 2"))
          .to eq("SELECT * FROM `table` WHERE ((`a` >= 1 AND `a` <= 3) AND `b` = 2)")
      end
    end

    context "with a between query and two other type of query" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table WHERE a BETWEEN 1 and 3 AND b = 2 AND c = 3"))
          .to eq("SELECT * FROM `table` WHERE (((`a` >= 1 AND `a` <= 3) AND `b` = 2) AND `c` = 3)")
      end
    end
  end
end
```

spec/sql_assess/transformers/comparison_predicate/from_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Transformers::ComparisonPredicate::From do
  subject { described_class.new(connection) }

  context "when there is no join clause" do
    it "returns the same query" do
      expect(subject.transform("SELECT * FROM table")).to eq("SELECT * FROM `table`")
    end
  end

  context "when there is a join clause" do
    context "with no comparison predicate query" do
      it "returns the same query" do
        expect(subject.transform("SELECT * FROM table LEFT JOIN t1, t3 ON a BETWEEN 1 AND 3"))
          .to eq("SELECT * FROM `table` LEFT JOIN `t1` CROSS JOIN `t3` ON `a` BETWEEN 1 AND 3")
      end
    end

    context "with a >" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table, t3 LEFT JOIN t1 ON a > 1"))
          .to eq("SELECT * FROM `table` CROSS JOIN `t3` LEFT JOIN `t1` ON 1 < `a`")
      end
    end

    context "with a > and a <" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table LEFT JOIN t1 ON a > 1 AND a < 2"))
          .to eq("SELECT * FROM `table` LEFT JOIN `t1` ON (1 < `a` AND `a` < 2)")
      end
    end

    context "with a >=" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table LEFT JOIN t1 ON a >= 1"))
          .to eq("SELECT * FROM `table` LEFT JOIN `t1` ON 1 <= `a`")
      end
    end

    context "with a <" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table LEFT JOIN t1 ON a < 1"))
          .to eq("SELECT * FROM `table` LEFT JOIN `t1` ON `a` < 1")
      end
    end

    context "with a <=" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table LEFT JOIN t1 ON a <= 1"))
          .to eq("SELECT * FROM `table` LEFT JOIN `t1` ON `a` <= 1")
      end
    end
  end
end
```

spec/sql_assess/transformers/comparison_predicate/having_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::ComparisonPredicate::Having do
  subject { described_class.new(connection) }

  context "when there is no having clause" do
    it "returns the same query" do
      expect(subject.transform("SELECT * FROM table")).to eq("SELECT * FROM table")
    end
  end

  context "when there is a having clause" do
    context "with no comparison predicate query" do
      it "returns the same query" do
        expect(subject.transform("SELECT * FROM table HAVING a BETWEEN 1 AND 3"))
          .to eq("SELECT * FROM `table` HAVING `a` BETWEEN 1 AND 3")
      end
    end

    context "with a >" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table HAVING a > 1"))
```

```ruby
          .to eq("SELECT * FROM `table` HAVING 1 < `a`")
      end
    end

    context "with a >=" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table HAVING a >= 1"))
          .to eq("SELECT * FROM `table` HAVING 1 <= `a`")
      end
    end

    context "with a <" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table HAVING a < 1"))
          .to eq("SELECT * FROM `table` HAVING `a` < 1")
      end
    end

    context "with a <=" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table HAVING a <= 1"))
          .to eq("SELECT * FROM `table` HAVING `a` <= 1")
      end
    end

    context "with a <= and a >" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table HAVING a <= 1 AND a > 1"))
          .to eq("SELECT * FROM `table` HAVING (`a` <= 1 AND 1 < `a`)")
      end
    end
  end
end
```

spec/sql_assess/transformers/comparison_predicate/where_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Transformers::ComparisonPredicate::Where do
  subject { described_class.new(connection) }

  context "when there is no where clause" do
    it "returns the same query" do
      expect(subject.transform("SELECT * FROM table")).to eq("SELECT * FROM table")
    end
  end

  context "when there is a where clause" do
    context "with no comparison predicate query" do
      it "returns the same query" do
        expect(subject.transform("SELECT * FROM table WHERE a BETWEEN 1 AND 3"))
          .to eq("SELECT * FROM `table` WHERE `a` BETWEEN 1 AND 3")
      end
    end

    context "with a >" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table WHERE a > 1"))
          .to eq("SELECT * FROM `table` WHERE 1 < `a`")
      end
    end

    context "with a >=" do
      it "returns the updated query" do
```

```
        expect(subject.transform("SELECT * FROM table WHERE a >= 1"))
          .to eq("SELECT * FROM `table` WHERE 1 <= `a`")
      end
    end

    context "with a <" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table WHERE a < 1"))
          .to eq("SELECT * FROM `table` WHERE `a` < 1")
      end
    end

    context "with a <=" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table WHERE a <= 1"))
          .to eq("SELECT * FROM `table` WHERE `a` <= 1")
      end
    end

    context "with a <= and a >" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM table WHERE a <= 1 AND a > 1"))
          .to eq("SELECT * FROM `table` WHERE (`a` <= 1 AND 1 < `a`)")
      end
    end
  end
end
```

spec/sql_assess/transformers/equivalent_columns/group_by_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::EquivalentColumns::Group do
  subject { described_class.new(connection).transform(sql) }

  context "no equivalence" do
    context "with no join clause" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `table1`
          GROUP BY `table1`.`id`
        SQL
      end

      it "returns the same query" do
        expect(subject).to eq(sql)
      end
    end

    context "with a join clause but no equivalence" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `table1`
            LEFT JOIN `table2` ON `table2`.`id` = `table1`.`id2`
            LEFT JOIN `table3` ON `table3`.`id` = `table1`.`id3`
          GROUP BY `table1`.`id`
        SQL
      end

      it "returns the same query" do
        expect(subject).to eq(sql)
      end
    end
  end

  context "with an equivalence" do
    context "with a left join" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `b`
            LEFT JOIN `a` ON `a`.`id` = `b`.`id`
          GROUP BY `b`.`id`
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT *
            FROM
              `b`
              LEFT JOIN `a` ON `a`.`id` = `b`.`id`
            GROUP BY `a`.`id`
          SQL
        )
      end
    end

    context "with two left joins" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `c`
            LEFT JOIN `a` ON `a`.`id` = `c`.`id`
            LEFT JOIN `b` ON `b`.`id` = `c`.`id`
          GROUP BY `c`.`id`
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT *
            FROM
              `c`
              LEFT JOIN `a` ON `a`.`id` = `c`.`id`
              LEFT JOIN `b` ON `b`.`id` = `c`.`id`
            GROUP BY `a`.`id`
          SQL
        )
      end
    end

    context "with two joins" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `c`
            LEFT JOIN `a` ON `a`.`id` = `c`.`id`
```

```
            RIGHT JOIN `b` ON `b`.`id` = `c`.`id`
          GROUP BY `c`.`id`
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT *
            FROM
              `c`
              LEFT JOIN `a` ON `a`.`id` = `c`.`id`
              RIGHT JOIN `b` ON `b`.`id` = `c`.`id`
            GROUP BY `a`.`id`
          SQL
        )
      end
    end
  end
end
```

spec/sql_assess/transformers/equivalent_columns/having_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::EquivalentColumns::Having do
  subject { described_class.new(connection).transform(sql) }

  context "no equivalence" do
    context "with no join clause" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `table1`
          HAVING `table1`.`id` = 1
        SQL
      end

      it "returns the same query" do
        expect(subject).to eq(sql)
      end
    end

    context "with a join clause but no equivalence" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `table1`
            LEFT JOIN `table2` ON `table2`.`id` = `table1`.`id2`
            LEFT JOIN `table3` ON `table3`.`id` = `table1`.`id3`
          HAVING `table1`.`id` = 1
        SQL
      end

      it "returns the same query" do
        expect(subject).to eq(sql)
      end
    end
  end

  context "with an equivalence" do
    context "with a left join" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `b`
            LEFT JOIN `a` ON `a`.`id` = `b`.`id`
          HAVING `b`.`id` = 1
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT *
            FROM
              `b`
              LEFT JOIN `a` ON `a`.`id` = `b`.`id`
            HAVING `a`.`id` = 1
          SQL
        )
      end
    end

    context "with two left joins" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `c`
            LEFT JOIN `a` ON `a`.`id` = `c`.`id`
            LEFT JOIN `b` ON `b`.`id` = `c`.`id`
          HAVING `c`.`id` = 1
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT *
            FROM
              `c`
              LEFT JOIN `a` ON `a`.`id` = `c`.`id`
              LEFT JOIN `b` ON `b`.`id` = `c`.`id`
            HAVING `a`.`id` = 1
          SQL
        )
      end
    end

    context "with two joins" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `c`
            LEFT JOIN `a` ON `a`.`id` = `c`.`id`
            RIGHT JOIN `b` ON `b`.`id` = `c`.`id`
          HAVING `c`.`id` = 1
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT *
            FROM
              `c`
              LEFT JOIN `a` ON `a`.`id` = `c`.`id`
              RIGHT JOIN `b` ON `b`.`id` = `c`.`id`
            HAVING `a`.`id` = 1
          SQL
        )
      end
    end
  end
end
```

spec/sql_assess/transformers/equivalent_columns/order_by_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::EquivalentColumns::OrderBy do
  subject { described_class.new(connection).transform(sql) }

  context "no equivalence" do
    context "with no join clause" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `table1`
          ORDER BY `table1`.`id` ASC
        SQL
      end

      it "returns the same query" do
        expect(subject).to eq(sql)
      end
    end

    context "with a join clause but no equivalence" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `table1`
            LEFT JOIN `table2` ON `table2`.`id` = `table1`.`id2`
            LEFT JOIN `table3` ON `table3`.`id` = `table1`.`id3`
          ORDER BY `table1`.`id` ASC
        SQL
      end

      it "returns the same query" do
        expect(subject).to eq(sql)
      end
    end
  end

  context "with an equivalence" do
    context "with a left join" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `b`
            LEFT JOIN `a` ON `a`.`id` = `b`.`id`
          ORDER BY `b`.`id` ASC
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT *
            FROM
              `b`
              LEFT JOIN `a` ON `a`.`id` = `b`.`id`
            ORDER BY `a`.`id` ASC
          SQL
        )
      end
    end

    context "with two left joins" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `c`
            LEFT JOIN `a` ON `a`.`id` = `c`.`id`
            LEFT JOIN `b` ON `b`.`id` = `c`.`id`
          ORDER BY `c`.`id` ASC
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT *
            FROM
              `c`
              LEFT JOIN `a` ON `a`.`id` = `c`.`id`
              LEFT JOIN `b` ON `b`.`id` = `c`.`id`
            ORDER BY `a`.`id` ASC
          SQL
        )
      end
    end

    context "with two joins" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `c`
            LEFT JOIN `a` ON `a`.`id` = `c`.`id`
            RIGHT JOIN `b` ON `b`.`id` = `c`.`id`
          ORDER BY `c`.`id` ASC
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT *
            FROM
              `c`
              LEFT JOIN `a` ON `a`.`id` = `c`.`id`
              RIGHT JOIN `b` ON `b`.`id` = `c`.`id`
            ORDER BY `a`.`id` ASC
```

```
          SQL
        )
      end
    end
  end
end
```

spec/sql_assess/transformers/equivalent_columns/select_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::EquivalentColumns::Select do
  subject { described_class.new(connection).transform(sql) }

  context "no equivalence" do
    context "with no join clause" do
      let(:sql) do
        <<-SQL.squish
          SELECT `table1`.`id`
          FROM
            `table1`
        SQL
      end

      it "returns the same query" do
        expect(subject).to eq(sql)
      end
    end

    context "with a join clause but no equivalence" do
      let(:sql) do
        <<-SQL.squish
          SELECT `table1`.`id`
          FROM
            `table1`
            LEFT JOIN `table2` ON `table2`.`id` = `table1`.`id2`
            LEFT JOIN `table3` ON `table3`.`id` = `table1`.`id3`
        SQL
      end

      it "returns the same query" do
        expect(subject).to eq(sql)
      end
    end

    context "with a join clause but no equivalence" do
      let(:sql) do
        <<-SQL.squish
          SELECT `a`.`id`
          FROM
            `b`
            LEFT JOIN `a` ON (`a`.`id` = `b`.`id` OR `a`.`id` = `b`.`id2`)
        SQL
      end

      it "returns the same query" do
        expect(subject).to eq(sql)
      end
    end
  end

  context "with an equivalence" do
    context "with a left join" do
      let(:sql) do
        <<-SQL.squish
          SELECT `b`.`id`
          FROM
            `b`
            LEFT JOIN `a` ON (`a`.`id` = `b`.`id` AND `a`.`id2` = `b`.`id2`)
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT `a`.`id`
            FROM
              `b`
              LEFT JOIN `a` ON (`a`.`id` = `b`.`id` AND `a`.`id2` = `b`.`id2`)
          SQL
        )
      end
    end

    context "with two left joins" do
      let(:sql) do
        <<-SQL.squish
          SELECT `c`.`id`
          FROM
            `c`
            LEFT JOIN `a` ON `a`.`id` = `c`.`id`
            LEFT JOIN `b` ON `b`.`id` = `c`.`id`
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT `a`.`id`
            FROM
              `c`
              LEFT JOIN `a` ON `a`.`id` = `c`.`id`
              LEFT JOIN `b` ON `b`.`id` = `c`.`id`
          SQL
        )
      end
    end

    context "with two left joins" do
      let(:sql) do
        <<-SQL.squish
          SELECT `c`.`id`
          FROM
            `c`
            LEFT JOIN `a` ON `a`.`id` = `c`.`id`
            RIGHT JOIN `b` ON `b`.`id` = `c`.`id`
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT `a`.`id`
            FROM
              `c`
              LEFT JOIN `a` ON `a`.`id` = `c`.`id`
              RIGHT JOIN `b` ON `b`.`id` = `c`.`id`
        SQL
      )
    end
  end
 end
end
```

spec/sql_assess/transformers/equivalent_columns/where_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::EquivalentColumns::Where do
  subject { described_class.new(connection).transform(sql) }

  context "no equivalence" do
    context "with no join clause" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `table1`
          WHERE `table1`.`id` = 1
        SQL
      end

      it "returns the same query" do
        expect(subject).to eq(sql)
      end
    end

    context "with a join clause but no equivalence" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `table1`
            LEFT JOIN `table2` ON `table2`.`id` = `table1`.`id2`
            LEFT JOIN `table3` ON `table3`.`id` = `table1`.`id3`
          WHERE `table1`.`id` = 1
        SQL
      end

      it "returns the same query" do
        expect(subject).to eq(sql)
      end
    end
  end

  context "with an equivalence" do
    context "with a left join" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `b`
            LEFT JOIN `a` ON `a`.`id` = `b`.`id`
          WHERE `b`.`id` = 1
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT *
            FROM
              `b`
              LEFT JOIN `a` ON `a`.`id` = `b`.`id`
            WHERE `a`.`id` = 1
          SQL
        )
      end
    end

    context "with two left joins" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `c`
            LEFT JOIN `a` ON `a`.`id` = `c`.`id`
            LEFT JOIN `b` ON `b`.`id` = `c`.`id`
          WHERE `c`.`id` = 1
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT *
            FROM
              `c`
              LEFT JOIN `a` ON `a`.`id` = `c`.`id`
              LEFT JOIN `b` ON `b`.`id` = `c`.`id`
            WHERE `a`.`id` = 1
          SQL
        )
      end
    end

    context "with two joins" do
      let(:sql) do
        <<-SQL.squish
          SELECT *
          FROM
            `c`
            LEFT JOIN `a` ON `a`.`id` = `c`.`id`
            RIGHT JOIN `b` ON `b`.`id` = `c`.`id`
          WHERE `c`.`id` = 1
        SQL
      end

      it "changes to the lowest string" do
        expect(subject).to eq(
          <<-SQL.squish
            SELECT *
            FROM
              `c`
              LEFT JOIN `a` ON `a`.`id` = `c`.`id`
              RIGHT JOIN `b` ON `b`.`id` = `c`.`id`
            WHERE `a`.`id` = 1
          SQL
        )
      end
    end
  end
 end
```

spec/sql_assess/transformers/from_subquery_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Transformers::FromSubquery do
  subject { described_class.new(connection).transform(sql) }

  before do
    connection.query("CREATE TABLE table1 (id1 integer, id2 integer)")
    connection.query("CREATE TABLE table2 (id3 integer, id4 integer)")
  end

  context "no subquery" do
    let(:sql) do
      <<-SQL.squish
        SELECT `table1`.`id`
        FROM
          `table1`
          LEFT JOIN `table2` ON `table1`.`id1` = `table2`.`id3`
      SQL
    end

    it "returns the same query" do
      expect(subject).to eq(sql)
    end
  end


  context "with subquery" do
    let(:sql) do
      <<-SQL.squish
        SELECT `table1`.`id`
        FROM (SELECT * from table1)
      SQL
    end

    it "returns the same query" do
      expect(subject).to eq("SELECT `table1`.`id` FROM (SELECT `table1`.`id1`, `table1`.`id2` FROM `table1`)")
    end
  end

  context "with subquery left join" do
    let(:sql) do
      <<-SQL.squish
        SELECT `table1`.`id`

        FROM table2 LEFT JOIN (SELECT * from table1) ON table1.id1 = table2.id3
      SQL
    end

    it "returns the same query" do
      expect(subject).to eq("SELECT `table1`.`id` FROM `table2` LEFT JOIN (SELECT `table1`.`id1`,
       ↪  `table1`.`id2` FROM `table1`) ON `table1`.`id1` = `table2`.`id3`")
    end
  end
end
```

spec/sql_assess/transformers/not/from_spec.rb

```ruby
require "spec_helper"

RSpec.describe SqlAssess::Transformers::Not::From do
  subject { described_class.new(connection) }

  context "with no join" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1`
      SQL
    end

    it "doesn't change the query" do
      expect(subject.transform(query)).to eq(query)
    end
  end

  context "with join clause but no not" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1` LEFT JOIN `table2` ON `table2`.`id3` = `table1`.`id1`
      SQL
    end

    it "doesn't change the query" do
      expect(subject.transform(query)).to eq(query)
    end
  end

  context "with join clause with not" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1` LEFT JOIN `table2` ON NOT `id3` > `id1`
        GROUP BY `id1`
      SQL
    end

    it "changes the query" do
      expect(subject.transform(query)).to eq(
        <<-SQL.squish
          SELECT `table1`.`id`, `table1`.`id2`
          FROM `table1` LEFT JOIN `table2` ON `id3` <= `id1`
          GROUP BY `id1`
        SQL
      )
    end
  end

  context "with join clause with not" do
    let(:query) do
      <<-SQL.squish
        SELECT `table1`.`id`, `table1`.`id2`
        FROM `table1` LEFT JOIN `table2` ON NOT `id3` > `id1` AND NOT `id3` >= `id1`
        GROUP BY `id1`
      SQL
    end

    it "changes the query" do
      expect(subject.transform(query)).to eq(
        <<-SQL.squish
          SELECT `table1`.`id`, `table1`.`id2`
          FROM `table1` LEFT JOIN `table2` ON (`id3` <= `id1` AND `id3` < `id1`)
          GROUP BY `id1`
        SQL
      )
    end
  end
```

```
end
```

spec/sql_assess/transformers/not/having_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::Not::Having do
  subject { described_class.new(connection) }

  context "when there is no HAVING clause" do
    it "returns the same query" do
      expect(subject.transform("SELECT * FROM t1")).to eq("SELECT * FROM t1")
    end
  end

  context "when there is a HAVING clause" do
    context "with no NOT query" do
      it "returns the same query" do
        expect(subject.transform("SELECT * FROM t1 HAVING a = 1"))
          .to eq("SELECT * FROM `t1` HAVING `a` = 1")
      end
    end

    context "with only a between query" do
      context "with a > clause" do
        it "returns the updated query" do
          expect(subject.transform("SELECT * FROM t1 HAVING NOT a > 1"))
            .to eq("SELECT * FROM `t1` HAVING `a` <= 1")
        end
      end

      context "with a < clause" do
        it "returns the updated query" do
          expect(subject.transform("SELECT * FROM t1 HAVING NOT a < 1"))
            .to eq("SELECT * FROM `t1` HAVING `a` >= 1")
        end
      end

      context "with a <= clause" do
        it "returns the updated query" do
          expect(subject.transform("SELECT * FROM t1 HAVING NOT a <= 1"))
            .to eq("SELECT * FROM `t1` HAVING `a` > 1")
        end
      end

      context "with a >= clause" do
        it "returns the updated query" do
          expect(subject.transform("SELECT * FROM t1 HAVING NOT a >= 1"))
            .to eq("SELECT * FROM `t1` HAVING `a` < 1")
        end
      end
    end

    context "with a not query and another type of query" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM t1 HAVING NOT a > 1 AND b = 2"))
          .to eq("SELECT * FROM `t1` HAVING (`a` <= 1 AND `b` = 2)")
      end
    end

    context "with a not query and two other type of query" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM t1 HAVING NOT a > 1 AND b = 2 AND c = 3"))
          .to eq("SELECT * FROM `t1` HAVING ((`a` <= 1 AND `b` = 2) AND `c` = 3)")
      end
    end

    context "with a not which is not transformable" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM t1 HAVING NOT a LIKE 'a'"))
          .to eq("SELECT * FROM `t1` HAVING `a` NOT LIKE 'a'")
      end
    end
  end
end
```

spec/sql_assess/transformers/not/where_spec.rb

```
require "spec_helper"

RSpec.describe SqlAssess::Transformers::Not::Where do
  subject { described_class.new(connection) }

  context "when there is no where clause" do
    it "returns the same query" do
      expect(subject.transform("SELECT * FROM t1")).to eq("SELECT * FROM t1")
    end
  end

  context "when there is a where clause" do
    context "with no NOT query" do
      it "returns the same query" do
        expect(subject.transform("SELECT * FROM t1 WHERE a = 1"))
          .to eq("SELECT * FROM `t1` WHERE `a` = 1")
      end
    end

    context "with only a between query" do
      context "with a > clause" do
        it "returns the updated query" do
          expect(subject.transform("SELECT * FROM t1 WHERE NOT a > 1"))
            .to eq("SELECT * FROM `t1` WHERE `a` <= 1")
        end
      end

      context "with a < clause" do
        it "returns the updated query" do
          expect(subject.transform("SELECT * FROM t1 WHERE NOT a < 1"))
            .to eq("SELECT * FROM `t1` WHERE `a` >= 1")
        end
      end

      context "with a <= clause" do
        it "returns the updated query" do
          expect(subject.transform("SELECT * FROM t1 WHERE NOT a <= 1"))
            .to eq("SELECT * FROM `t1` WHERE `a` > 1")
        end
      end

      context "with a >= clause" do
        it "returns the updated query" do
          expect(subject.transform("SELECT * FROM t1 WHERE NOT a >= 1"))
            .to eq("SELECT * FROM `t1` WHERE `a` < 1")
        end
      end
    end

    context "with a not query and another type of query" do
```

```
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM t1 WHERE NOT a > 1 AND b = 2"))
          .to eq("SELECT * FROM `t1` WHERE (`a` <= 1 AND `b` = 2)")
      end
    end

    context "with a not query and two other type of query" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM t1 WHERE NOT a > 1 AND b = 2 AND c = 3"))
          .to eq("SELECT * FROM `t1` WHERE ((`a` <= 1 AND `b` = 2) AND `c` = 3)")
      end
    end

    context "with a not which is not transformable" do
      it "returns the updated query" do
        expect(subject.transform("SELECT * FROM t1 WHERE NOT a LIKE 'a'"))
          .to eq("SELECT * FROM `t1` WHERE `a` NOT LIKE 'a'")
      end
    end
  end
end
```

spec/sql_assess_spec.rb

```
RSpec.describe SqlAssess do
  it "has a version number" do
    expect(SqlAssess::VERSION).not_to be nil
  end
end
```

sql_assess.gemspec

```
lib = File.expand_path("../lib", __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
require "sql_assess/version"

Gem::Specification.new do |spec|
  spec.name          = "sql_assess"
  spec.version       = SqlAssess::VERSION
  spec.authors       = ["Vlad Stoica"]
  spec.email         = ["vlad96stoica@gmail.com"]

  spec.summary       = "Ruby gem for assesing SQL"
  spec.homepage      = "https://vladstoica.com"
  spec.license       = "MIT"

  # Prevent pushing this gem to RubyGems.org. To allow pushes either set the 'allowed_push_host'
  # to allow pushing to a single host or delete this section to allow pushing to any host.
  if spec.respond_to?(:metadata)
    spec.metadata["allowed_push_host"] = "TODO: Set to 'http://mygemserver.com'"
  else
    raise "RubyGems 2.0 or newer is required to protect against " \
      "public gem pushes."
  end

  spec.files         = `git ls-files -z`.split("\x0").reject do |f|
    f.match(%r{^(test|spec|features)/})
  end
  spec.bindir        = "exe"
  spec.executables   = spec.files.grep(%r{^exe/}) { |f| File.basename(f) }
  spec.require_paths = ["lib"]

  spec.add_dependency "activesupport"
  spec.add_dependency "mysql2"
  spec.add_dependency 'sql-parser-vlad', '~> 0.0.15'
  spec.add_dependency "rgl"

  spec.add_development_dependency "bundler", "~> 1.16"
  spec.add_development_dependency "pry"
  spec.add_development_dependency "rake", "~> 10.0"
  spec.add_development_dependency "rspec", "~> 3.0"
  spec.add_development_dependency "timecop"
  spec.add_development_dependency "rubocop", '~> 0.54.0'
  spec.add_development_dependency "codecov"
end
```