Master's thesis

NTNU
Norwegian University of Science and Technology

Jostein Kjerstad

Automatic Evaluation and Grading of SQL Queries using Relational Algebra Trees

Master's thesis in Computer Science
Supervisor: Rune Hjelsvold
June 2020

NTNU
Norwegian University of
Science and Technology

Jostein Kjerstad

Automatic Evaluation and Grading of SQL Queries using Relational Algebra Trees

Master's thesis in Computer Science
Supervisor: Rune Hjelsvold
June 2020

Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

Acknowledgement

I would like to thank my supervisor Rune Hjelsvold for his help and support through both my master thesis and my literature study. Also I would like to thank my wife for all the moral support she has given me during this semester.

Abstract

The following paper conducts my master thesis from Computer Science at NTNU, Trondheim during the spring semester 2020. It describes a system that is developed to use Relational Algebra Trees (RAT) from a SQL-query in order to evaluate, compare and grade comparing it to a solution. The system also provides different sorts of output, allowing the user to get constructive feedback explaining the score that was given.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Keywords SQL, Python, Intelligent System, grading, evaluation, Relational Algebra, Relational Algebra Tree, database

Manually grading and evaluating SQL queries takes a lot of time and effort and hence creating a system that automatically grades queries would be of great advantage. Not only would it reduce workload but it could also be used as a pedagogical tool in terms of giving students a way to test and understand the query they have created.

This thesis describes such a system that automatically grades SQL-queries by using Relational Algebra Trees. The methods that are developed in order to create this tool are explained in detail step-by-step as well as with examples and figures. The tool is then tested on a set of real data, in order to check it's accuracy and usability.

Some knowledge of programming might be an advantage in order to fully understand how this system works but is not a requirement.

1.1 Research Questions

Q1: Can Relational Algebra Trees be used to grade SQL-queries?

Q2: How accurate will the grading be compared to manual evaluation of the same query?

Q3: Can the system use the tree to provide some constructive feedback that can be used for pedagogical purposes?

2 Background

An issue that rises when correcting SQL-queries is that more than one solution may exist to one single problem. Unfortunately, the number of solutions scales with the complexity of the exercise or task at hand. In order to make a fair grading system you would need a safe method to compare a solution against an answer, taking into consideration all of the possibilities that could lead to the same solution.

Testing the SQL-queries in a database-environment could be one way of creating such a system, but it has it weaknesses:

(1) There are very small margins whether a query actually executes or not. One single typo or mistake means the query is invalid and no data will be returned. In other words, Giving a partial score based on this can be difficult.

(2) Some SQL-queries may lead to the same result even when the queries are not correct when compared to the solution. This is because some databases contains relatively small amounts of data that two different queries might end up representing the same data, because the data that would make the difference is "missing".

Taking both those weaknesses into consideration this method makes for a non-optimized way to auto grade queries, and hence we must look for something better. This paper focuses on doing this by using a more generalized and unified method - By translating the SQL queries into relational algebra trees and then comparing the trees against each other. In addition, showing students how their SQL query looks in terms of an algebra tree may help them understand more of SQL and how the query is executed in a Database Management System.

This tool is mainly for educational purposes but there is no reason for it not to be used in other environments where it could be useful.

3 Related Work

Over the years several different SQL learning tools has been presented in order to assist teaching of SQL. Some of them focuses on alternative visualising of queries while others focuses on how to better teach SQL in terms of creating intelligent tutors. Example of this is:

- XData (Chandra, Banerjee, Hazra, Joseph, and Sudarshan, 2019).
- Intelligent SQL-Tutor (Mitrovic, 2003)
- QueryViz (Danaparamita and Gatterbauer, 2011) (Alternative visualisation)
- LEARN-SQL(Abelló, Rodríguez, Urpí, Burgués, Casany, Martín, and Quer, 2008)

For more details regarding related work on SQL learning tools, see A study on Intelligent Tutor Systems and exercise generation in SQL (Kjerstad, 2019). Research done on the topic has revealed that this is the first system that uses Relational Algebra Tree in order to correct and grade queries.

4 Work

This sections contains all the work and implementations that has been done in order to test how algebra trees can be used to evaluate queries. This includes both methods used and the idea behind them.

The source code of the system can be found at:
`https://github.com/joskje29/RATgrader`

4.1 Technology

In order to test how precise the method mentioned in this paper is an algorithm would need to be programmed and run on a certain set of data. The programming language chosen for this is Python 3.8 but in theory almost any programming language could have been used. Most students and teachers with relation to the database courses typically has some knowledge with Python. Also Python is one of the fastest growing languages and one of the most used languages for educational purposes (Srinath, 2017) and hence it was chosen as the preferred language for this thesis.

4.2 Pre-process

4.2.1 Importing data

All data to be imported should be having the UTF-8 standards (8-bit Unicode Transformation Format) as defined in RFC 3629 and ISO/IEC 1064 (Yergeau, 2003). This allows for special characters like 'æ', 'ø', 'å' as well as handling math symbols, apostrophes and quotation marks properly. UTF-8 works for every every language and uses only one byte per character when using the English alphabet.

The data is imported based on a line-by-line read from a text file such as .txt or .doc. Every line relates to one query or answer which is then to be processed.

This may raise multiple issues:

1. Students tend to sometimes add comments to their answer, such as explanations or assumptions which is not to be a part of the query but can sometimes be hard to filter out
2. Faulty newline or line ending formatting due to improper file writing
3. Blank answers or only comments
4. Multiple answers in one line from one student or more then one query separated by ";" or other symbols

5. Students adding malicious parts to the line that may impact on the outcome of the program

(1) In the case of relatively small data sets the one option could be to manually look through the answers to remove such comments, but since it's supposed to be fully autonomous this goes against its purpose and can be very time consuming as well. The program handles this by removing anything behind the last ";" or comment symbols such as "#" or "//" in the line. If this fails due to the lack of proper notation the last part of the query will end up including those extra those comments, only having an impact on one single node in the Relational Algebra Tree

In some scenarios the comments may actually be relevant in the matter where it could determine reasoning behind the answer, for example if the task given could be misinterpreted. The system described in this thesis ignores all comments and pretends as they are not there but compared to a "manual" correction done by the teacher this could have an impact on the scores given. In order to create a "bullet proof" method to handle this an additional comment field could be added allowing students to put their comments separated from the actual query. The test data used in this thesis did not have this setup and the methods above has been implied.

(2) Newline, sometimes called line breaking or EoL(End of Line) is used to specify the end of a line and start of the new line in a text. This is usually managed by using a specific character or a sequence of characters encoded in a specific system such as ASCII. This may rise an issue as some editors automatically adds this "character" when someone uses the "Enter key" which can result in a line break within one answer which is not supposed to be there adding more lines then actual answers. Usually this just results in an empty line in the data and the line is then ignored. As long as all parts of the system uses the same encoding system line breaking should not case any issue except some empty lines here and there which can be ignored.

(3) Answers that contains only comments and no query are treated as a blank answer in this system and hence is given a score equal to 0.0

(4) In this system one answer equals to one query. Multiple queries is not yet possible and neither does it make any sense. Hence everything behind the ";" symbol will be treated as a comment. Sometimes a student typically tries to add more then one answer to cover his ground, but in this option only the first answer will be treated as an answer and everything else is ignored.

(5) Knowing that its an algorithm that decides your score and not a human, inserting malicious code into your submission could potentially do harm. In the scenario where the data is tested against a database to compare the actual result instead of comparing queries, the user can use SQL-injections to delete data or

to get a better score. Also python snippets used in answer could potentially worst case lead to cheating by falsifying your own score to 100. This can easily be prevented by sanitizing input and by using proper escaping (Anley, 2002).

The system described in this paper assumes that all input is "healthy" to reduce the amount of work load this would have taken to implement. Also, at this point the system does not actually run queries with a real database, eliminating the risk of actual SQL-injections to happen.

4.2.2 Formatting

In order to ensure that every answer is processed on the same rules of formal grammar or "language basis" every query goes through a formatting check. The purpose with doing this is to make it easier for the system to filter out special parts of the query such as words inside quotation marks, parentheses or symbols such as % which is all parts of the SQL syntax. Below is a list of all the potentially errors that the formatting function finds and corrects.

- Adding space after each ","
- Adding space around quoted words
- Adding space around parentheses
- Adding space around math operators such as plus, minus, greater than, lesser than, etc.
- Removing unnecessary spaces
- Removing space after dots which in SQL language is used for renaming purposes and should not have a space
- Removes mixed usage of quotation marks and apostrophes and replaces it with only quotation marks
- Removing ";" at end of query

The system does this by going through the query char by char looking for inconsistencies. When it finds a inconsistency it simply fixes it or adds the extra chars necessary to fix it and moves to the next char. The reasoning behind automatically fixing these inconsistencies is based on that these linguistic errors have nothing to do with whether the query is "correct" or not. This is simply done to make the further processing of the query easier.

One could argue with the fact that some of the answers would not run if tested in a real database environment due to the faulty formatting, but this would never be the case of a "manual" evaluation of the answers. The main purpose of any tasks related to SQL is to test whether a person understands how SQL

works or not. Some text editors would also automatically fix most of the errors mentioned above such as proper line breaker. The explanation behind some of the faulty formatting could also be a case of a typographical error, which will be covered in chapter 4.11

4.2.3 Word parsing

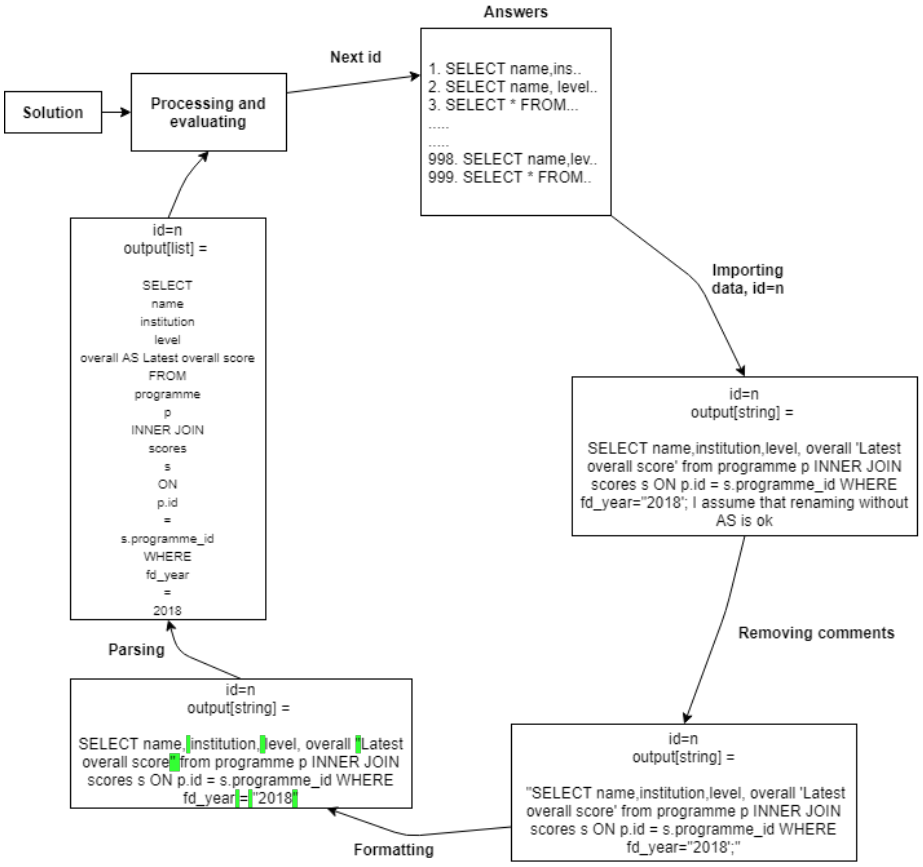
Next the system tries to split the query into words using a built-in function named Shlex — Simple lexical analysis (Van Rossum and Drake, 1995) (Python, 2001). This is a function that makes splitting queries easier because it takes into consideration if words are in quoted strings or parentheses and does not split up based only on spaces or commas. The output of the function is a list of words which the system then systematically parses by going through every single word. That means finding out what role every word has in the query.

Also the parser checks for specific syntax on some types of words such as operators (SELECT, FROM, WHERE, etc..) or "secondary" operator words such as "BY" and checks if they are in uppercase letters as they are supposed to be. One of the optional filters that is mentioned in chapter 4.11 takes into consideration whether the syntax is correct or not when estimating the total score. The output of the function is a list of all the parsed words of the query that sometimes means words has been merged together in order to describe their syntactic role of the query. Below is a list of everything that is handled by the parser:

- Removes "," from start of words as this is not a part of the Shlex functionality
- Checks if SQL Operators are written with capital letters(uppercase)
- Checks for keywords such as "AS" indicating a rename of a table
- Makes every other word lowercase in order for easier comparison, unless the "Case sensitive" filter is set to true.
- Merge words that has a relation and does not make any sense alone. An example of this is the words "INNER" and "JOIN" which in this case will be merged to "INNER JOIN" or "BY" and "GROUP" which is merged into "BY GROUP".
- Handles renames that are done by "AS" by merging them into one "word". An example of this is the query "SELECT A, B AS Beta, C FROM Letters". This query would first be parsed into the following words: [SELECT, A, B, AS, Beta, C, FROM, Letters] but due to the merging the correct output would be: [SELECT, A, B AS Beta, C, FROM, Letters].
- Merges every word in between start and end of a parentheses into one word which in SQL language indicating that they are representing a subquery (Beaulieu, 2009)

4.2.4 The pre-process cycle

Not every method mentioned in section 4.2.1 - 4.2.3 is necessary in order to process the answer but the idea behind it is to create a standard which makes it easier to process it further and make it more unified. Figure 1 shows an example of how the data is pre-processed.



The pre-processing cycle

Figure 1: Flow diagram showing an example of how the data is pre-processed using the methods described in Section 4.2.1 - 4.2.3

1. In the importing part of the cycle one answer is fetched from the data/answers.

2. Further on the comment made by the student *I assume that renaming without AS is ok* is removed. The program assumes that this is a comment based on that its written after the ";" symbol.

3. All the missing spaces behind commas and around math symbols are added. Also every apostrophe is replaced with a quoting symbol("). All the changes are marked by green on the figure.

4. During the parsing phase the parser filters out which word belongs together. The order of the words is still kept intact. the two words "INNER" and "JOIN" is merged. Also "from" has been changed to "FROM" due to the fact that this is an operator and should be in uppercase.

5. The result is now a list that is ready to be further processed and then evaluated.

4.3 Clause splitting

INN MED FIGUR!

Based on the list of standardized words presented in the previous chapter the system is now ready to process the answer. The system iterates through the list of words and every time a word matches an "Operator word" that is defined by the system or by filters, a so-called *clause* is created. A clause in the SQL language is known as a representation of a single part of the query. Below is a complete list of the most common SQL operators that is usually known and taught at basic courses in the universities: (Hursch, Hursch, and Hursch, 1988)

- SELECT
- FROM
- WHERE
- INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN
- HAVING
- GROUP BY, ORDER BY
- ASC, DESC
- EXISTS, NOT EXISTS
- AVG, MIN, MAX, SUM, COUNT

Then every word that comes next is added to the clause data unless the next word is a operator. If the next word is an operator the system then stores the clause and starts a new sub-query. Every clause has the following data information stored:

- Operator type
- A list of all the words related to the operator (I.e. table names, table values or math symbols)
- String representation of the clause (E.g. "SELECT name, level, institution")

After the iteration of the words is done the system starts to process each clause further. The methods that are used for this are different from every operator type because the related words are handled in a different way based on the type. An exception is the first part which is to find all the so-called "aliased" tables. Aliased values are normally represented on the format "alias.value", I.e. separated by a dot.

4.4 Anti-aliasing and renaming

A common method to structure queries is to use aliases. That is to rename table names inside a query with the main purpose of making the SQL query shorter and more readable. The longer and more complex the queries get the bigger are the chances that aliasing has been used. Another reason for renaming tables would be if you have sub-queries inside the queries that uses the same table as the main query. By using aliases you reduce the complexity and chance of any misuse. Aliasing is not to be confused with "renaming" which usually refers to changing the name of a column and not a table.

Aliasing only affects the structure and how readable the query is for the human eye while renaming shows on the actual output. Also, both Aliasing and renaming can be done in mainly two ways, depending on what the specific SQL language allows for. Some languages allows for both while some only allows for one. The two methods are:

1. Using the AS operator, on the format "tablename AS t" or "column AS c"
2. By using only a single space seperating the table name and the new alias. This will be on the format "tablename t" or "column c"

Since this system is made to be as flexible as possible both methods are allowed. This is also something that can be set as an optional filter. Aliasing and renaming is best shown with an example, see Listing 1

```
SELECT p.name, p.overall AS Latest overall score
FROM programme p
INNER JOIN scores AS s ON p.id = s.programme_id
WHERE fd_year = '2018';
```

Listing 1: An example of a SQL query that uses both methods in the same query

The first line shows an example of a column renaming, known as just renaming. The second line shows aliasing of the table programme using the (2) method described above, resulting the alias "p". The third line shows aliasing of the table scores using the AS operator aliasing it into "s". The example also shows how "p" and "s" is used to represent the tables they alias.

4.4.1 Fixing aliases

Now, in order to best compare the students answer against a solution in the terms of checking values the system removes all aliasing and replaces them with the full table name in both the answer and the solution. This makes it easier to compare because if the correct table has been aliased the values should now be identical. The way the system does this is by looking through each clause looking for aliases. If an alias is found both the alias and the full name is saved to a mutual dictionary.

After every clause has been checked the system goes over all the clauses one more time but this time it has a record of all the aliases and it then renames all columns that has been aliased to the full table name. Lets look at how the system handles aliases from the example shown in Listing 1

In the first iteration the system creates the following keyword dictionary: {'p' : programme, 's': scores'}. During the second iteration the system does a look-up in this dictionary for every value that has an alias. The system determines that a value is aliased whenever there is a dot(".") in the value. In the example, p.name and "p.level" are examples of aliases the system will find and change. The result of the anti-aliasing operation can be seen in Listing 2

```
SELECT programme.name, programme.level,
programme.institution
FROM programme p
INNER JOIN scores AS s
ON programme.id = scores.programme_id
WHERE programme.fd_year = 2018;
```

Listing 2: The result after aliases has been removed

Also, the system removes the renaming part from the word list belonging to the operator where the aliasing happened as this is no longer needed in order to compare and evaluate. Hence, the final output after the second iteration can be seen in Listing 3

```
Operator: SELECT,
Values: [programme.name, programme.level,
            programme.institution]
```

```
Operator: FROM,
Values: [programme]

Operator: INNER JOIN
Values: [scores, ON, programme.id, =,
            scores.programme_id]

Operator: WHERE,
Values: [programme.fd_year, =, 2018]
```

Listing 3: A visual representation of the systems output after removing aliases

4.4.2 Fixing renaming

During the first iteration described above the system also checks for so-called renames of columns. As mentioned this is sometimes done by using the "AS" operator as sometimes its only used by writing the column and then the renamed column only separated by a space. In order to unify this the system changes all renames onto the format where the AS operator is used. The reason behind this is that the AS operator is recognized and allowed in every major SQL language, while the other method mentioned is not allowed in all SQL languages, E.g. mySQL.

4.5 JOIN-clause processing

Now that the alias and renaming has been unified the system can now further process the clauses. By clause processing we here refer to resolve and re-organize the values into their respective operator type. This process is only necessary to some of the operator types and again this is to standardize them in order to make comparison and evaluation easier. This applies to whenever a join is happening, either through a regular JOIN operation or by having multiple tables in the FROM clause. Both scenarios requires may give the same results but in order to compare and evaluate they need to be standardized

4.5.1 Regular JOIN clause

By regular JOIN clause we mean a join that is happening because the SQL-query has one of the following four allowed join operators:

- INNER JOIN, sometimes referred to as only JOIN
- LEFT JOIN, sometimes referred to as LEFT OUTER JOIN

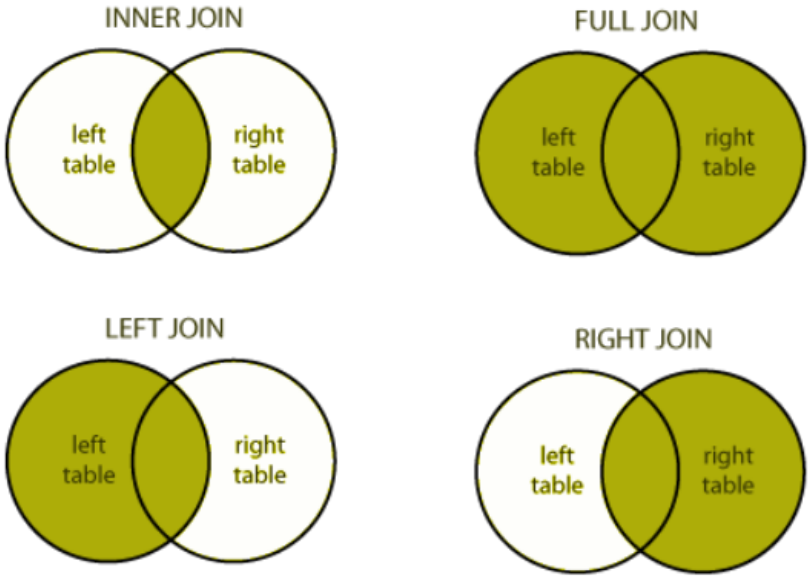


Figure 2: A Venn diagram showing the four different types of JOIN that is used in SQL

- RIGHT JOIN, sometimes referred to as RIGHT OUTER JOIN
- FULL OUTER JOIN

The difference between the joins is shown using a Venn diagram, see Figure 2 (Big Data Zone, 2020). Sometimes in small data sets or in special cases different JOIN-types can give the same result when fetching data from a database which is one of the weaknesses by evaluating answers only by testing them in a database environment.

In order to find the full information about the join the system looks at both the FROM and the JOIN-clause and merges them together into one new clause, consisting of four new values: Join type, left side, right side and conditions which is then stored in a dictionary. This is done by the following steps:

1. The left table of the join is picked from the FROM-clause. If the FROM clause is missing the left side will be left empty.
2. The right table of the join is picked from the JOIN-clause
3. The type of JOIN is picked from the JOIN-clause
4. The joining conditions are picked from the JOIN-clause

Both the old clauses will be deleted and new clause will be added in the following format:

Clause Type: JOIN type
Values: {'left': left_table, 'right': right_table, 'on': condition}

If we keep using the example from the last chapter, see Listing 4 we can now take a look at how this part of the system affects our result. The new changes can be seen in the Listing

```
Operator: SELECT,
Values: [programme.name, programme.level,
            programme.institution]

Operator: FROM,
Values: [programme]

Operator: INNER JOIN,
Values: [scores, ON, programme.id, =,
            scores.programme_id]

Operator: WHERE,
Values: [programme.fd_year, =, 2018]
```

Listing 4: A visual representation of the systems ready for the JOIN-clause modifications

We now create a new operator based on data from the FROM and INNER JOIN-clause which gives us the following result:

Operator: INNER JOIN,
Values: {'left': programme, 'right': scores, 'on': [programme.id, =, 2018]}

The system then deletes the old FROM and INNER JOIN-clause, replacing it with the new one. The result can be seen in Listing 5

```
Operator: SELECT,
Values: [programme.name, programme.level,
            programme.institution]

Operator: INNER JOIN,
Values: {
    'left': programme,
```

```
    'right': scores,
    'on': [programme.id, =, scores.programme_id]}

Operator: WHERE,
Values: [programme.fd_year, =, 2018]
```

Listing 5: The new INNER JOIN clause has been added and the FROM-clause has been removed

4.5.2 FROM-clause JOIN

The second type of join that is allowed in SQL is made out of the FROM-CLAUSE by having more then one table which is separated by commas. An example can be seen in Listing 6.

```
SELECT name
FROM programme, scores s
WHERE id = s.programme_id AND institution = 'idi'
```

Listing 6: An example of a SQL query that joins using two tables in the FROM-clause

This differs from the regular JOIN because:

1. It does not take into consideration which type of JOIN it is. Most databases treats this as a Cartesian product of all the tables, I.e FULL OUTER JOIN
2. The join condition is no longer found in the JOIN-clause but in the WHERE clause.

The benefits of a Cartesian product is that it allows for every possible combination if that is what you are looking for. On the other hand, this might also not be optimal when it comes to very large tables where the possible number of combinations to check will increase very fast. A typical scenario of this type of JOIN happens when someone is lacking knowledge of which join type to use, as this most likely will give them the result they want. A specified join type differs from this because it is both a Cartesian product and a selection which reduces the number of tables and values to join.

The system handles this type of a join by creating one or more INNER JOIN operator in almost the same way as in JOIN-clause processing by creating a dictionary including left, right and on. The only difference is that instead of taking the condition from the JOIN-clause the system now has to look for the on condition in the WHERE-clause. The values that are taken from the WHERE-clause is then removed. Also, the amount of INNER JOIN needed depends on how many tables the FROM-clause has. In general, the following rule applies:

For N tables in the FROM-clause, N-1 Operators must be created

The result by doing this process on our example, Listing 6, can be seen in Listing 7

```
Operator: SELECT,
Values: [name]

Operator: INNER JOIN,
Values: {
    'left': programme,
    'right': scores,
    'on': [programme.id, =, scores.programme_id]}

Operator: WHERE,
Values: [institution, =, idi]
```

Listing 7: The result of how the system handles a FROM-join

4.5.3 INNER JOIN vs WHERE syntax

The standard defined by ANSI (David, 1999) states that using the INNER JOIN operator should be used as default. The main reason behind this is that it is more readable for the human eye and the more joins you do the more unreadable it gets. This is because you would have to add a lot of conditions in the WHERE-clause. That said, for simple queries most SQL languages will handle these two queries in the same way. Hence, punishing answers that uses the other type of a JOIN than the solution can be seen as unfair and hence this is added as an optional filter, see more in Section 4.11

The main difference between the two syntax's is how they are handled within the SQL database in matter of column matching and performance. The outcome will always be the same in both a INNER JOIN and a FROM join.

4.6 Subqueries

Subqueries, sometimes refered to as inner queries, are queries that are nested inside a clause. This can happen inside different types of operators such as the SELECT or WHERE clause. Sometimes a subquery can be replaced by using a join while other times the only way to get the desired result is by using a subquery. In most cases where the subquery can be translated into a join there is no performance difference except in some special cases where a join would perform better.

An example follows showing two queries that will give the same result. Listing 8 shows a query that uses a subquery while Listing 29 shows the equivalent query using the INNER JOIN.

```
SELECT name
FROM programme
WHERE level = (SELECT level
    FROM programme
    WHERE institution = idi)
```

Listing 8: An example of a sub-query

```
SELECT p1.name
FROM programme AS p1
INNER JOIN programme AS p2 ON p1.level = p2.level
WHERE p2.instituion = idi
```

Listing 9: The equivalent query that uses a join instead of a sub-query

A subquery does not allow every operator to be used. Operators like ASC and DESC is only used to sort the final output. A full list of the allowed operators can be seen in the list below:

- SELECT clause
- FROM clause with one or more tables
- WHERE clause (optional)
- GROUP BY clause (optional)
- HAVING clause (optional)

Now how does the system handle subqueries? And why is not this handled in the early processing? First, lets look at an example that has a subquery to easier understand what will happen.

```
SELECT p.name, s.inspiration
FROM programme p
INNER JOIN scores s ON p.id = s.programme_id
WHERE s.inspiration >
    (SELECT AVG(inspiration) FROM scores)
```

Listing 10: Subquery in the WHERE clause requires the system to take action

Here the subquery is located inside the WHERE clause. In chapter 4.2.3 regarding word parsing we learned that anything inside parentheses will be treated as a single word. This is also the case for the subquery meaning that it will be stored as "one word" as one of the values in the WHERE-clause operator. If we let the system process it using the methods mentioned so far this is what the output would be looking like:

```
Operator: SELECT,
Values: [programme.name, scores.inspiration]

Operator: INNER JOIN,
Values: {
    'left': programme,
    'right': scores,
    'on': [programme.id, =, scores.programme_id}

Operator: WHERE:
Values: [scores.inspiration, >,
        (SELECT AVG(inspiration) FROM scores)]
```

Listing 11: The example broken down into clauses

Now, what the system does when it finds a subquery inside a clause like above is to isolate it from the subquery and process it as if it was a new individual query. That means that the subquery will go through all the same processes as the main query, including everything from chapter 4.2.1 to 4.6. This mean that the subquery will now get its own two clauses, one for the SELECT and one for the FROM.

The result after the subquery is processed will then be replacing the old subquery. The subquery is still represented inside the old clause, but now by a list of the subqueries clauses. This now means that the entire output is on the same format which makes it ready for next step. The new result can be seen in Listing 12.

```
Operator: SELECT,
Values: [programme.name, scores.inspiration]

Operator: INNER JOIN,
Values: {
    'left': programme,
    'right': scores,
    'on': [programme.id, =, scores.programme_id}

Operator: WHERE:
Values: [scores.inspiration, >,
            [Operator: SELECT,
            Values: [AVG(inspiration)

            Operator: FROM,
            Values: [scores]]
```

Listing 12: The subquery is now broken down into clauses like the rest of the query

4.7 Normalizing

In order to translate the SQL query into Relational Algebra the query may need to be normalized. On a general basis normalizing is necessary when one of the following criteria are met:

1. More then one table or relation in the FROM clause. This usually results in a Cartesian product of the relations.
2. The where clause consists of a subquery and one of the following non-normalized operators: $<$, $>$, $\leq$, $\geq$, $\neq$, IN or NOT IN. This is normalized by replacing the operators into one of the two quantifiers EXISTS and NOT EXISTS and moving the operator into the subquery.

Now, thankfully (1) has already been sorted out by the system, see Chapter 4.5.2. On the other hand, the second scenario (2) still needs to be handled by the system. This chapter will explain how the system does exactly that.

4.7.1 Quantifying

In order to normalize the query the subquery sometimes needs to be modified in order to be able to translate it into relational algebra which is covered in chapter 4.9. This happens when the FROM clause includes both a subquery and a operator that is not EXISTS or NOT EXISTS. In order to fix this a specific transformation must be done of both the operator and the subquery. To explain this further we will keep using the same example as in the last chapter:

```
Operator: SELECT,
Values: [programme.name, scores.inspiration]

Operator: INNER JOIN,
Values: {
    'left': programme,
    'right': scores,
    'on': [programme.id, =, scores.programme_id}

Operator: WHERE:
Values: [scores.inspiration, >,
            [Operator: SELECT,
            Values: [AVG(inspiration)

            Operator: FROM,
            Values: [scores]]
```

Listing 13: The example from the last chapter

The problem with the example in Listing 13 is that we have a greater than (>) symbol and then a sub query. Greater than (>) is not a normalized operator and hence the system takes action to normalize it. The goal is to end up with an EXISTS or NOT EXISTS as our math operator. In order to translate the operator different rules applies based on which constraint we have. Below follows a couple examples to show how some of the constraints are translated and unified, by the rules of relational algebra equivalence (Klug, 1982).

4.7.2 Example 1: IN

```
SELECT movieTitle
FROM StarsIn
WHERE starName IN (
    SELECT name
    FROM MovieStar
    WHERE birthdate=1960)
```

In order to translate we need to identify two parts:

1. Which attribute(s)/column(s) is "in front" of the IN operator - I.e. what are we looking for in the subquery
2. Which attribute(s)/column(s) is returned from the subquery

By looking at the example we can find the answes:
(1) We're looking for all results that are equal to the **StarName**
(2) All the result of the query contains the column **name**

Knowing those two columns we can now move on to the transforming. First, the column that is "in front" of the IN operator has to be moved inside the subquery and added to the WHERE clause by adding an "AND" operator.

Secondly, since "name" is what is returned from the query this is what we are checking against. Hence the new part of the WHERE clause will be "AND name = starName". The last part is to change the IN to an EXISTS. The new result is listed below:

```
SELECT movieTitle
FROM StarsIn
WHERE EXISTS (
    SELECT name
    FROM MovieStar
    WHERE birthdate=1960 AND name=starName)
```

4.7.3 Example 2: Greater than or equal to ($\geq$)

```
SELECT name FROM MovieExec
WHERE netWorth >= (
    SELECT E.netWorth
    FROM MovieExec E)
```

In this example the same two rules applies except that this time we need to convert $\geq$ into its normalized form. The rules for a greater equal than is that its replaced by a lesser than ($<$) inside the subquery, and the operator outside the subquery is changed to NOT EXISTS. This gives us the following result:

```
SELECT name FROM MovieExec
WHERE NOT EXISTS (
    SELECT E.netWorth
    FROM MovieExec E
    WHERE netWorth < E.netWorth)
```

4.7.4 Other operators

Below is a list of the remaining math operators and what they are translated into:

- $>$ is translated into EXISTS and a $<$ in the WHERE-clause in the subquery
- $<$ is translated into EXISTS and a $>$ in the WHERE-clause in the subquery
- $<=$ is translated into NOT EXISTS and a $>$ in the WHERE-clause in the subquery

4.7.5 Aggregated functions

A new problem rises whenever we have an aggregation inside the subquery. That is when a column is being aggregated by one of the following operators:

- SUM
- COUNT
- AVG
- MIN/MAX

Those are the main aggregation functions that are used within the SQL language and in the scenario where they are used in a subquery that needs normalizing an extra step must be added in order to convert it into Relational Algebra. Lets look at a general example with a subquery that uses an aggregating function:

```
SELECT C FROM S
WHERE C IN (
    SELECT SUM(B) FROM R
    GROUP BY A)
```

As IN is not a normalized operator this needs to be normalized just like we did in Example 1. The only problem here is that because we have an aggregation of B, SUM(B) we can no longer use the WHERE clause as an aggregation only works with the HAVING operator. IN is still changed to EXISTS, but "C" is new moved into the new HAVING-clause together with the SUM(B) column. This gives us the following normalization:

```
SELECT C FROM S
WHERE EXISTS (
    SELECT SUM(B) FROM R
    GROUP BY A
    HAVING SUM(B) = C)
```

4.7.6 Example 3

Using the example that was shown in the beginning of the chapter, see Listing 13 we will now look at what the output will be looking after the normalization process. Here we have a greater than (>) in the WHERE clause, as well as an aggregation function in the subquery (AVG(inspiration)) so both steps will be needed in order to normalize this.

First the WHERE-clause part is changed to EXISTS. Secondly we need to add a HAVING function inside the subquery because of the aggregation. Lastly we add the following statement to the HAVING-clause: "AVG(inspiration) < scores.inspiration)" The < is chosen by the rules in section 4.7.4.

Also since we here have an aggregation there should by default also be a GROUP BY operator but since most modern SQL languages does this automatically this is not added by the system. If the user of the system wants the system to add the GROUP BY operator this can be done by activating the filter mentioned in chapter 4.11 The result of the normalization can be seen in Listing 14

```
Operator: SELECT,
Values: [programme.name, scores.inspiration]

Operator: INNER JOIN,
Values: {
    'left': programme,
    'right': scores,
    'on': [programme.id, =, scores.programme_id}

Operator: WHERE EXISTS
Values: [Operator: SELECT,
         Values: [AVG(inspiration)]

         Operator: FROM,
         Values: [scores]

         Operator: HAVING,
         Values: [AVG(inspiration), <, scores.inspiration]
```

Listing 14: The result of the normalization process

4.8 Sorting

The final process the system needs to do before it is ready to convert the answer into a Relational Algebra Tree is to sort all the operators into the correct order based by how Relational algebra works (Ceri and Gottlob, 1985) This is the inverted order of a typical relational algebra representation of the query, but this is because the tree is constructed bottom-up while a relational algebra expression is usually made top down: the Operators will be sorted in the following order:

1. ASC, DESC
2. ORDER BY, GROUP BY,
3. SELECT
4. WHERE OR HAVING (Including EXISTS AND NOT EXISTS)
5. INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN
6. FROM

Again, using our main example lets look at how the new order of the operators will look like. Also the same sorting is done within the subquery part. The new result is shown in Listing 15

```
Operator: SELECT,
Values: [programme.name, scores.inspiration]

Operator: WHERE EXISTS
Values: [Operator: SELECT,
         Values: [AVG(inspiration)]

         Operator: HAVING,
         Values: [AVG(inspiration), <, scores.inspiration]

         Operator: FROM,
         Values: [scores]

Operator: INNER JOIN,
Values: {
    'left': programme,
    'right': scores,
    'on': [programme.id, =, scores.programme_id}
```

Listing 15: The sorted result

4.9 Converting into a Relational Algebra Tree

Now that the system has sorted the clauses into the correct order it can start to convert it into a Relational Algebra Tree (RAT). Relational Algebra tree is based upon the Relational Algebra "language" and shares parts of its syntax. In order to fully understand the syntax of RAT we will first look at the syntax of regular Relational Algebra.

4.9.1 Relational Algebra

Relational Algebra is a language to express SQL queries in a different and more "unified" method. There is mainly two reasons as to why you would want to look at the relational algebra equivalent of a query:

1. To learn of how the query works. The SQL language is defined as a "declarative" language meaning that it focuses on the "what". In difference, the relational algebra language focuses on the procedural, I.e. the "how". So rather on focusing on what needs to be in the query in order for it to run the focused is switched to the "meaning" of the query.
2. To understand how Database Management System's (DBMSs) executes query. The relational algebra is very equal to as how a database execute its query.

Here's an example of what a SQL-query typically looks like, written in the relational algebra language:

$$\pi_{\text{name}}\, \sigma_{\text{id=s.programme\_id} \wedge \text{institution='idi'}}\, (\text{Programme} \times \rho_{\text{s}}(\text{Scores}))$$

4.9.2 Relational Algebra Syntax

In Relational Algebra operator words or so-called statements are converted into a new syntax, containing of algebra symbol as well as Greek letters. Below is a list of the most used symbols in Relational Algebra:

- π (Pi) is used to represent the SELECT clause. This is known as the Projection operator in relational algebra
- σ (Sigma) is used to represent the WHERE or HAVING clause. This is known as the SELECT operator within relational algebra which can be confusing at times.

- ρ Is used to represent renaming or aliases.
- $\times$ Is used to represent a Cartesian product
- $\wedge$ Is used to represent the AND operator in SQL
- $\vee$ Is used to represent the OR operator in SQL
- $\bowtie$ Is used to represent the JOIN operator

4.9.3 Relational Algebra Tree

A relational Algebra Tree is a binary tree, which is "a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. The first node is referred to as the "root" and its children are referred to as "children" or "leaves" of the root. Every node can have both a left and a right children. Sub-tree's are a construction that is a part of the main tree and children within a sub-tree are called Siblings. The depth of the tree tells how many "levels" of children that exists within the tree. See Figure 3 for an example of a binary tree with depth = 3 may look like.

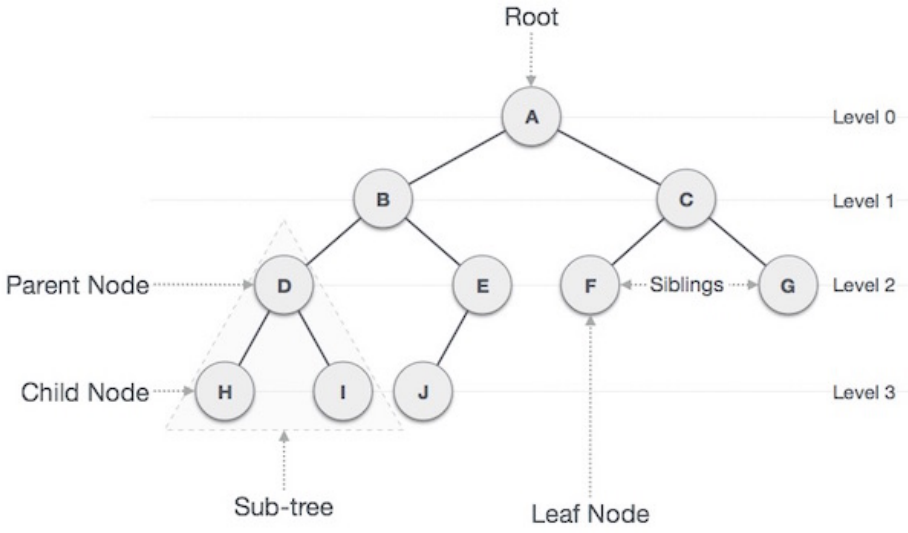


Figure 3: An example of a binary tree

The RAT is based on relational algebra and represents the same "execution" order and symbols. One of the main differences is the physical visualization and that it has another way of representing the SQL-query and execution. The second difference is that in by using binary tree's to compare two SQL queries to another a lot of possibilities opens up. In this system, every node in the tree contains the following data:

- Type
- Values
- Representation. This refers to its Relational Algebra Syntax, I.e. its "symbol" (pi, sigma, etc..)
- Left and right child

4.9.4 Tree Construction

Now how does the system construct the tree? First it creates an empty node which is referred to as the root. Because this a root its type is set to "root". Next, it iterates through all the sorted clauses and as it picks a new clause it also adds new nodes to the tree. Every node that is created is unique, based on the type of the clause and its values. If it finds a subquery inside one of the clauses it creates a sub-tree and inserts it to the main tree by repeating the same methods.

The system uses different methods to create the next node based on which type of a clause it is:

- **JOIN-clause**: The JOIN-clause values has three items: left, right and the on-condition. A new sub-tree is created in this case, and the value of the root will be equal to the on-condition value. Next, two children are created. The left child will have the data from the 'left' value of the clause while the right child gets the data from the 'right' value. The type of the parent node will be set to the same type as the JOIN-type specified in the clause and the children will both have type = FROM.
- **EXISTS / NOT EXISTS-clause**: This creates a "product" node without any data. Even though this node has no data it still represents a product of two children to come, where one of them is a subquery. The product can be seen on as a parent of a sub-tree in the new tree.
- **Every other clause**: Values are copied straight from the clause values into the node values and the type will be the same as the clause operator type.

In all the methods mentioned above the new node or sub-tree that is created will always be added as the left child of its parent. The right child is only used in the case of a JOIN-clause or a subquery. Now that we know how the system handles different types of clauses we can see how a relational algebra tree is constructed based on our main example shown in Listing 15 in chapter 4.8.

1. The root node is created

2. The SELECT-clause is the first clause to be converted. We set the type of the node to SELECT and the values to [programme.name, scores.inspiration"], see Figre 4

3. Next we have a WHERE EXISTS operator. This means that the system now creates a PRODUCT node with two children and since we have a subquery inside the clause we now move to the right child first to create the subquery. This is shown in Figure 5

4. The subquery contains three clauses. They all are normal clauses that will be added as three nodes in a row, as the left child of their parents. see Figure 6

5. Now that the sub-tree is constructed from the subquery we move back to our PRODUCT node and start creating the left child, this time using the INNER JOIN clause. The INNER JOIN-clause values is set to [programme.id, =, scores.progrmame_id] and type = INNER JOIN

6. Finally the two children nodes of the INNER JOIN is created from the 'left' and 'right' data from the INNER JOIN-clause. The type here will be set to FROM in both children. The tree is now finished

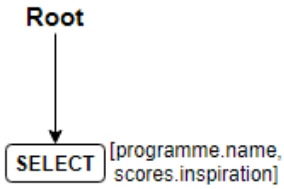


Figure 4: Constructing an algebra tree: Step 1 and 2

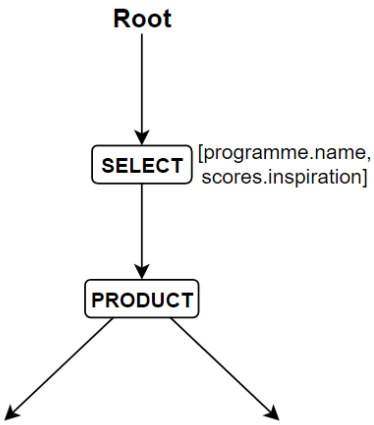


Figure 5: Constructing an algebra tree: Step 3

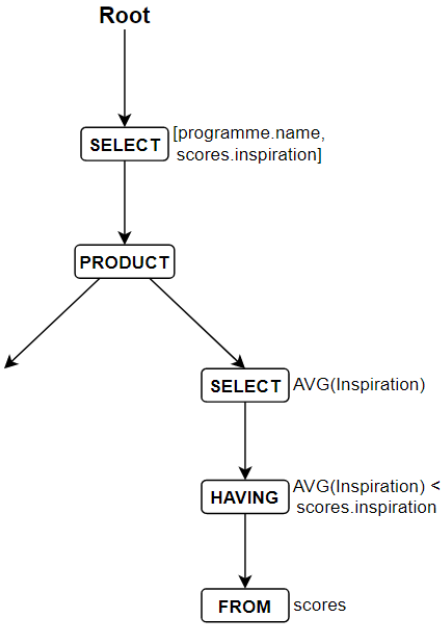


Figure 6: Constructing an algebra tree: Step 4

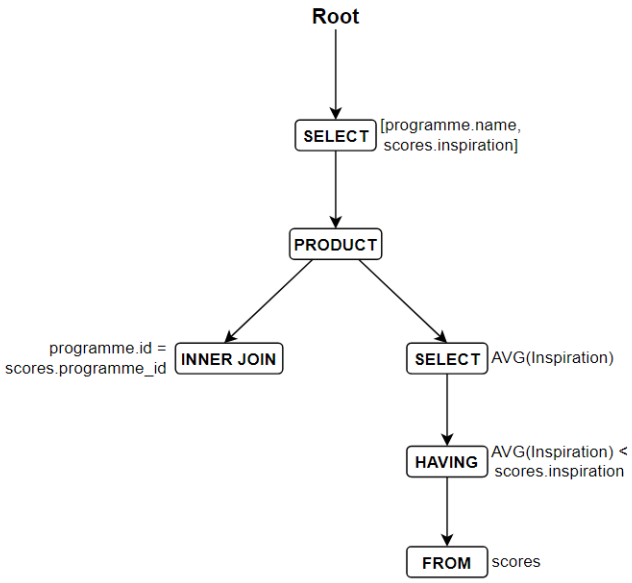


Figure 7: Constructing an algebra tree: Step 5

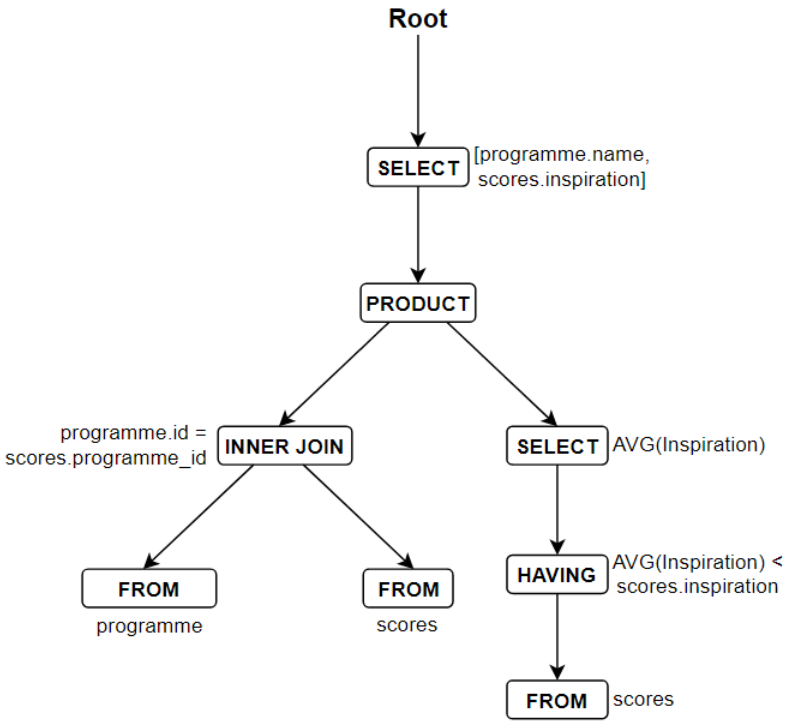


Figure 8: Constructing an algebra tree: Step 6

The results shown in Figure 8 is only a visual representation of how the data is handled by the system and to show how the tree is created. This is not comparable to how a "real" relational algebra tree would look like as that would require some other syntax. That said, the system is able to print a "real" relational algebra tree as one of its extra functionalities. The details around this will be discussed in chapter 4.12

4.10 Comparing and Evaluation

Now that we have a fully constructed Relational Algebra Tree the system is ready to evaluate and compare the answer against a solution. The solution is also added to the system as a SQL query before it is converted into a tree by the system. As the system iterates through all the answers it compares every answer tree against this solution tree.

The system compares the two trees by going through each node in both trees, comparing them to each other. This is done by using a Breadth-First search (BFS). Using a BFS in order to traverse the tree makes sense as the BFS starts with the root and traverses the tree in the same we as the system constructed it. Every node is compared to the other tree's node, checking both the type, data and the node's position. This results in one of the following scenarios:

1. The nodes are 100% equal. This means the type, its values and position are the same and hence a full score is given for this node.
2. The nodes have the same type and position, but the values are different. The system then calculates a score based on the amount of correct values and punishes any additional values if this filter is activated. Depending on which of the filters that are activated the system will also check the answer against typo's. Also, the system checks aliasing if this is used. If both the solution and the answer has "aliased" the same tables this is not necessary but in the case where only one uses aliasing this has to be checked in order to give a correct score. If tables are provided this is also checked against the table.
3. The nodes does not have the same JOIN-type, indicating that the wrong join operation has been used. The system then calculates a score based on the amount of correct values and punishes the usage of the wrong JOIN-type. The "weight" on this punishment is set by the user of the system. The default settings will give a maximum score of 50% if the wrong join type is used.
4. The nodes does not have the same type. The may be more reasons behind this so the system tries to find adjacent nodes that matches, indicating that the answer has more or less operators compared to the solution. If the system finds it further down in the tree that indicates the user has used too many operators and when it finds the node earlier the user has most likely missed one or more of the operators. It could also be that the user wrote them in the wrong order.

The way that it compares values are different from type to type. This is because the format of the values requires different methods in order to compare. Also, the order of the values might be of importance depending on the type. This

way of comparing can only give the answer a 100% score if both the trees are equal, meaning that every node has to be equal in both value, position and type. That is, unless the system finds that the values or type are actually equivalent in terms of result but not by "words" or order. This will be covered in the following section, 4.10.1.

After the system has traversed the search it returns a full score list as well as some additional data for every node it traversed. This includes the following data per node:

- Type of the node
- A score from 0-100
- Missing data values
- Extra data values
- A "comment" or explanation as to why a full score was not given, pointing out the error(s). In the case of typo's or aliasing this will be shown here, even when a full score is given.
- The answer data

An example of how the system scores and what is returned can be seen in section 4.10.2

4.10.1 Equivalent trees

As the SQL language sometimes allows for different methods to get the same result the system must always try to find those so-called equivalent methods. Some of the equivalences are already sorted out in the "normalizing"-process while others needs to be manually checked. Below is a list of the most common equivalences that the system checks for:

- Math operators. For example, A > B is the same as B < A. This usually happens in the WHERE or HAVING-clause and is easily checked by using algebra rules.
- Equivalent JOINS. In the case of some joins - for example a FULL OUTER JOIN - it does not matter which table is left and join when merging. In theory this can result in two different trees that are actually equivalent.

  This can also happen when a LEFT or RIGHT join is used, but the left and right tables are switched. E.g. A LEFT JOIN B is the same as B RIGHT JOIN A. This should give the same result when run in a database, and according to the mySQL documentation it always converts a RIGHT

JOIN into a left join before running the query in the database. (Widenius, Axmark, and Arno, 2002).

The system checks for this by "mirroring" the JOIN-nodes and tests for equivalence.

4.10.2 Example of scoring

In order to see how the system scores and to get a better understanding if its output we will here show this with an examples. This includes one solution shown in Listing 16 and two answers, shown in Listing 17 and Listing 18.

```
SELECT name, institution, level,
overall Latest overall score
FROM programme p
INNER JOIN scores s ON p.id = s.programme_id
WHERE fd_year = '2018';
```

Listing 16: The solution

```
SELECT p.name, p.institution, s.teaching, s.fd_year,
s.overall Latest overall score
FROM programme p
INNER JOIN scores s ON s.programme_id = p.id
WHERE s.fd_year = '2018';
```

Listing 17: Example 1

```
SELECT p.programee, p.institution, s.teaching,
        s.overall AS "Latest overall score"
FROM `Scores` s
INNER JOIN `Programee` p ON p.id = s.programme_id
```

Listing 18: Example 2

The following filters are active for both examples (For more info regarding the filters, see Section 4.11)
Punish extra data: True; Allow misspelled chars: True, n = 1 character

Example 1: This is almost identical as the solution except that it uses aliasing. After the system processes both queries and creates a score this example ends up with a 100% score. The complete "raw" output is shown in Listing 19

Example 2. This also uses aliasing, but unfortunately the table "Programee" is misspelled. Since the system allows typos with one char this will be fixed. Also, the values in the SELECT-clause are not an exact match and the WHERE-clause is missing as well. This example got a score of 66.67%, see Listing 20 for the full output.

```
{
  "statement": "SELECT",
  "percent_correct": 100.0,
  "missing": [],
  "extra": [],
  "comment": "Removed renaming",
  "data": ["name", "institution", "level",
    "overall AS latest overall score"]}
{
  "statement": "WHERE",
  "percent_correct": 100.0,
  "missing": [],
  "extra": [],
  "comment": "Removed renaming",
  "data": {
    "operator": "EQUAL",
    "left": "fd_year",
    "right": "2018"}}
{
  "statement": "INNER_JOIN",
  "percent_correct": 100.0,
  "missing": [],
  "extra": [],
  "comment": "Removed renaming",
  "data": ["id", "programme_id"]}
{
  "statement": "FROM",
  "percent_correct": 100.0,
  "missing": [],
  "extra": [],
  "comment": "Removed renaming",
  "data": "programme"}
{
  "statement": "FROM",
  "percent_correct": 100.0,
  "missing": [],
  "extra": [],
  "comment": "Removed renaming",
  "data": "scores"}

Score: 100.0
```

Listing 19: The output from evaluating Example 1 against the solution

```
{
  "statement": "SELECT",
  "percent_correct": 33.33,
  "missing": ["level","name"],
  "extra": ["programme.programee", "teaching"],
  "comment": "Value(s) does not match",
  "data": ["programme.programee", "institution",
       "teaching", "overall AS latest overall score"]}
{
  "statement": "WHERE",
  "percent_correct": 0.0,
  "missing": {
    "operator": "EQUAL",
    "left": "fd_year",
    "right": "2018"},
  "extra": [],
  "comment": "Missing Operator",
  "data": []}
{
  "statement": "INNER_JOIN",
  "percent_correct": 100.0,
  "missing": [],
  "extra": [],
  "comment": "Removed renaming",
  "data": ["id", "programme_id"]}
{
  "statement": "FROM",
  "percent_correct": 100.0,
  "missing": [],
  "extra": [],
  "comment": "Fixed misspelling",
  "data": "programme"}
{
  "statement": "FROM",
  "percent_correct": 100.0,
  "missing": [],
  "extra": [],
  "comment": "Removed renaming",
  "data": "scores"}

Score: 66.67
```

Listing 20: The output from evaluating Example 2 against the solution

4.11 Filters

In order for the system to be as flexible as possible the user is given some options or choices regarding how the system should evaluate and grade the answer. This is here referred to as *filters*. The main idea is to let the user or "grader" decide on how strict the system should be when it comes to correcting and handing out a score. This could be different from university to university or from teacher to teacher, and hence the flexibility is added to the system. All the filters can be seen in table 4.11. This also shows what the system uses as default.

Name	Details	Default
JOIN VS INNER-JOIN	If set to true, this allows the JOIN-operator as short for "INNER JOIN"-operator. This is accepted by most languages today, E.g mySQL allows for this. If this is not allowed the usage of only "JOIN" will result in an invalid operator.	Yes
FROM-JOIN	If set to true, this allows the "FROM"-clause join method to be 100% equal to a"INNER JOIN"-join. This is simply two different methods that always ends up giving the same result and for some people it is a personal preference which one to use.	Yes
TYPO	Allow typographical errors by n wrong characters per word. If allowed this is used in the comparison of the solution tree vs the answer tree if the two values are not equal.The comparison algorithm then checks if <= n characters changed will change the values into being equal. This does not apply to operators or math symbols, only table words/columns.	No, n=0
SPELL CHECKER	This is also some sort of typo correction and uses python library named pyspellchecker and determines whether a word is misspelled or not by using the Levenstein Distance Algorithm* to find words (permutations) within an edit distance of 2. The down-side of using this is that it makes the system very slow. Compared to the TYPO-filter this does not know what word it is looking for	No
TABLE COMPARISON	This filter allows for values and to be checked against tables to see if columns or table names actually exists in the tables. To use this filter a table list is required in addition to the solution. The advantage of using this filter is that it can help give a better feedback in terms of justifying the score. Also, this filter checks aliasing up against tables and columns..	No
PUNISH ADDITIONAL DATA	This filter decides whether or not if extra data should be punished when giving the score. Additional data means that the answer includes more than what is asked for. In almost every scenario this will lead to a different database result if the query is executed	Yes
CASE SENSITIVE	If set to True this requires every word to be correctly written in the matter of uppercase and lowercase. This also includes operators which should be written in UPPERCASE.	No

Table 1: Filters that are used within the system

4.12 Representation and feedback

The system allows for different types of representation both throughout the processing and of the final result. By representation we mean ways to show the user what is happening during the process by presenting the current output or transformation. There may be many reasons for the user wanting to get this kind of feedback from the system such as:

- The user wants to see how the system transforms certain parts in order to understand what the system does
- The user wants to compare how different queries are processed
- The user wants to use the different transformation as basis for reasoning when explaining why the specific score was given
- The user wants to use the different ways of representing the result for pedagogical purposes
- The developer can use this to find bugs or improvements to the system
- Different types of representations can be used for different platforms. A visual printing of the algebra tree may be used for visual representation to understand how queries are processed, while a full data representation of all the clauses could be used to construct an autonomous feedback system that analyses what the user has misunderstood.

4.12.1 JSON

Every output in the system is on the JSON (JavaScript Object Notation), which is one of the most used notations in term of sending/receieving data, also in SQL (Bourhis, Reutter, Suárez, and Vrgoč, 2017). By using the JSON format the data is easily convertible to other languages or platforms and makes it easy to integrate on web pages.

4.12.2 Feedback

The system gives some sort of feedback, seen in the example from Chapter 4.10.2. At this point this is relatively "simple" feedback and there is certainly room for some more. There should be possibilities to look into the how the system can generate tips based on analyzing the answer, but this is yet to be implemented. More about this can be found in section 8.

4.12.3 Tree print

The system also allows for a more "realistic" tree print of the answer, using the proper relational algebra syntax. An example of how this would look like can be seen in Figure 9. Python is not the best program to print pretty trees although it is possible with the correct libraries installed such as Pretty Print Binary Tree ((Jdmcpeek, 2017). A better solution would be to export the JSON representation of the tree onto a more suitable "platform" and draw the tree there. The reason behind allowing this representing is to allow the user to get a more visual aspect of what and how the SQL query work

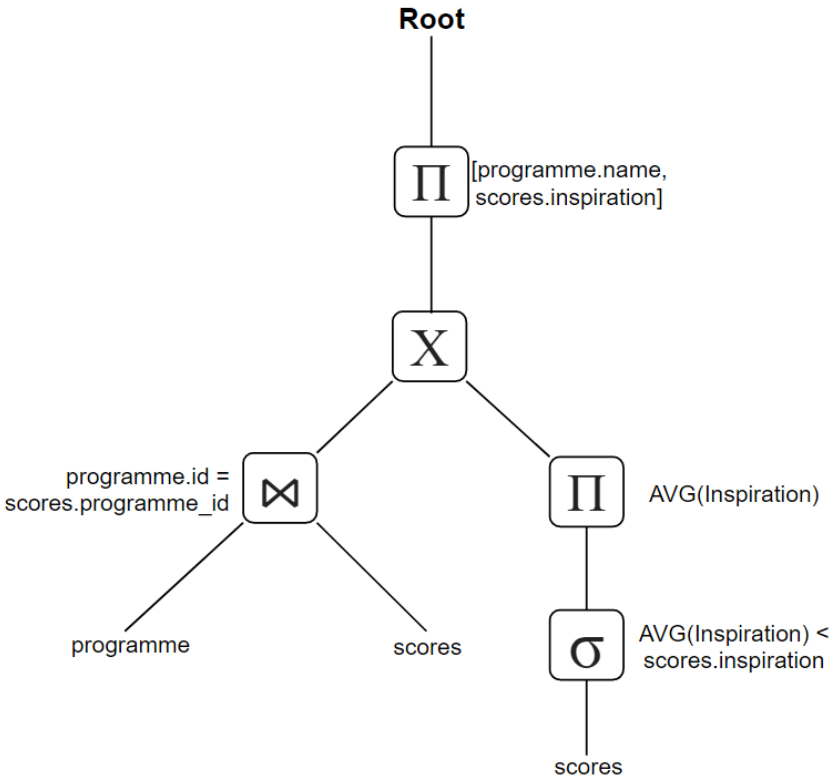


Figure 9: Relational Algebra Tree representation of Figure 8, using the relational algebra syntax

5 Results

The main goal of testing the system that is described in this paper is to evaluate whether Relational Algebra Tree's can or cannot be used in order to fully autonomously grade SQL-queries. In addition, finding out if the feedback given from the system can be used to pedagogical purposes is of big interest.

5.1 Testing Goals

In order to test the system and see if Relational Algebra Tree's can be used to analyze and correct SQL-queries certain sub-goals were made:

1. Find the percentage of queries that the system would be able to analyze or not, and try to understand why and find out how this can be fixed or improved.
2. Find the accuracy of the grading/scoring compared to manual correction.
3. Find out how well the system scales when more and more complex queries are used.
4. Find out how applying different filters to the system affects the scoring.
5. Find out how well the feedback given can be used for tutoring or pedagogical purposes.

5.2 Test Criteria

The system was implemented based on the "laws" of SQL and algebra, allowing it to be as flexible as possible and not "optimizing" one type of answers or queries. One could simply modify the system to be optimal for a specific group of people based on their SQL knowledge or previous answer history, but this goes against the purpose of making a fully autonomous grading system - That is without any bias or inclination influencing how the system grades.

To secure this way of implementing the "real" test data was never used during implementation or developing. This to make sure that when testing the real data the system would reveal as many "bugs" as possible (A bug is an error or failure in the system that gives unexpected results when testing the data). The result would then show a more realistic grading based on the fact that the system is not an AI - it does not learn or improve by itself. I.e. the system should be made to handle as many scenarios as possible and not by knowing the input.

The system was tested with several SQL-queries during developing, but this data was not used in the test and evaluation phase of the system. Also, in order to keep the system to a size that would fit the time frame, certain decisions were made in order to reduce the work load. Those decisions were:

- Ignoring all comments: In some scenarios the comments may actually be relevant in the matter where it could determine reasoning behind the answer, for example if the task given could be misinterpreted. The system described in this thesis ignores the comments it can find and pretends as they are not there.
- Disable the spell-checker. The spell-checker extension used approx. 20 seconds to check every query for misspelled words. Running the tests with the spell-checker would have taken more than six hours to finish, a lot of time considering how small the data set is and hence this filter were not used under testing.

5.3 Test data

The test data used to evaluate the system is real data from one part of an exam in a database course at NTNU. This includes five independent tasks, named Q1 to Q5, with a different level of difficulty. A total of 158 students took the exam meaning the total test data includes 790 SQL-queries to evaluate. Out of the 790 queries there is a total of 12 so-called "blank" answers, meaning that the student most likely handed in the blank answer knowing it would give a score of 0.0.

The given solution to the answers for all the five tasks are shown in Listing 21. As shown by the solutions the difficulty increases step-by-step and in task 5 a subquery is also needed in order to solve the requirements of the task. This gives the system a

```sql
/* Q1 solution */
SELECT name, institution, level
FROM programme WHERE name LIKE '%Computer%';

/* Q2 solution */
SELECT name, institution, level,
    overall 'Latest overall score'
FROM programme p
INNER JOIN scores s ON p.id = s.programme_id
WHERE fd_year = '2018';

/* Q3 solution */
SELECT p.name, p.institution, s.fd_year, s.teaching,
    s.feedback
FROM programme p
LEFT JOIN scores s ON p.id = s.programme_id;

/* Q4 solution */
SELECT p.name, p.institution, AVG(s.overall)
FROM programme p
INNER JOIN scores s ON p.id = s.programme_id
GROUP BY p.id
ORDER BY AVG(s.overall)
DESC;

/* Q5 solution */
SELECT p.name, p.institution, p.campus,
s.fd_year, s.inspiration
FROM programme p
INNER JOIN scores s ON p.id = s.programme_id
WHERE s.inspiration >
    (SELECT AVG(inspiration) FROM scores);
```

Listing 21: The given solution for task 1-5

5.4 Test Cases

For every task, Q1-Q5, the following cases were tested, sorted from strongest to weakest:

A: Strongest. Case sensitive = True, Punish Extra = True, Allow misspell = False (n=0)

B: Strong. Case sensitive = True, Punish Extra = True, Allow misspell = True (n=1)

C: Weak. Case sensitive = False, Punish Extra = True, Allow misspell = True (n=1)

D: Weakest. Case sensitive = False, Punish Extra = False, Allow misspell = True (n=1)

For all the test cases A-D comments made by students were removed manually from their answer. In addition to the test cases above the system also tested criteria D again, but this time the system were given the chance to remove the comments by itself:

E: Weak with comments. Equal to test case D, but comments were not removed manually

5.5 Test result

All the results from testing the data can be seen in Appendix A. This includes the following result data:

- List over the amount of queries that the system failed to analyse
- List over all the blank answers
- Every students score for every task, for every test case
- Average score on every task, for every test case
- Normal distribution for every test case, based on NTNU's recommended grading system (NTNU, 2020)

Some keywords from the result:

1. The best normal distribution comes from running test criteria C: Weak.
2. In the strongest case scenario the highest score was 63. Compared to manual evaluation of the answers this was not the real outcome, as more then one student got a full score.

3. Analyzing failure percentage on the different tests are:
   A: 2.3%
   B: 1.9%
   C: 1.9%
   D: 1.9%
   E: 2.9%

5.6 Performance

For every test the system graded 158 queries, a relative small number in terms of "data sets". The test was performed on a 2018 laptop with the following specs:

- CPU: Intel(R) Core(TM) i5-8265U CPU @1.60GHz 1.80Ghz
- RAM: 8.0GB DDR4
- OS: Windows 10 Home 64-bit

On average the system used 400ms to grade each task and a peak of 980ms from running test case scenario Q1.E. In total the system used approx. 10 seconds to perform all the tests, including both reading the answers form file and writing the results to a new file.

6 Discussion

6.1 Grading

The fact that many students got a 100.0 score on some of the more "complex" queries where different solutions may exists shows that using a relational algebra tree to compare allows for more than one solution.

This system is made to give a partial score for every query which means that a students final score in this system may not match reality. The reason for that is as follows: When someone manually evaluates an answer they might find that a 80 score is *good enough* and student is given a full score (100) on this task. Now, imagine if this happens on all five tasks. By manual evaluation the student ends up with a full score for every task, summing up to 500 out of 500, while in this system the student would end up with 5 times 80 = 400 out of 500, resulting in a worse grade.

Another issue that makes it hard to grade is to decide the weights of grading a query. using the correct operator type equal to getting all the values in terms of

6.2 Test cases and filters

By looking at the results we can see that the different test cases matches their strength compared to the other cases. This can be seen by looking on the average for every test case that it slightly gets higher and higher as we lower the test case filter requirements. This tells us the following:

1. Optional filters actually works as they can help fix minor mistakes that are made
2. Test case's are given the correct strength value (Weakest to Strongest)
3. Students makes typos and other small mistakes that are not relevant in terms of understanding the subject. More than 75% of the students made typo's that were one edit distance away from the solution.
4. The normal distribution from case A compared to case C shows how much fixing typos affects the distribution

In test case A, we did not allow for any typos as well as requiring an exact match of upper/lowercase on every word. By looking at the normal distribution for test case A(Strongest) we see that no one is given better than a 63 out of a 100 in terms of score. This does not reflect reality, which makes sense because (1) no auto correction is used on the answer and (2) humans tend to mistype relatively often when writing digitally, which means that it most likely happened on this

exam as well.

6.3 Analyzing Failure

The failure percentage is relatively low (less than 3%) in all the cases. Even in test case E, where we let the system try to remove comments by itself it stays below 3%. This despite the fact that more then 5% of the answers had comments. Unfortunately, the average scores also decreased on all the queries meaning that the system in some cases did not manage to remove the comment. Still it managed to move forward and grade it, but resulting in a lower score due to the comment being a part of the query.

The problem regarding the comments can easily be fixed by adding a separate comment field on the answer sheet/page allowing students to add comments separately.

6.4 Performance

Due to the fact that the system uses less than two seconds per test this is considered to be fast. The system is linear, built by simple iteration functions and in the case of a big data set, let's say 10,000 queries, the system should end up using approx 25 seconds. This is an acceptable time considering that this is not built to be a real-time application. Also, this system was run on an average computer and not a server built for speed and execution.

7 Conclusion

Below is a list of the six sub-goals from section 5

1. Find the percentage of queries that the system would be able to analyze or not, and try to understand why and find out how this can be fixed or improved.
2. Find the accuracy of the grading/scoring compared to manual correction.
3. Find out how well the system scales when more and more complex queries are used.
4. Find out how applying different filters to the system affects the scoring.
5. Find out how well the feedback given can be used for tutoring or pedagogical purposes.

Based on the results we can conclude with the following:

1. The system managed to have a worst case analyzing rate of 98.1% based on the results from test case B, C or D, leaving only 1.9% left for manual evaluation. Further on, manual analysis on the failed queries revealed that there were two reasons for failure:

   (a) The student had very little knowledge of how SQL works and the words/syntax used in the query has no relation to the language. No matter how good the system is this would still give problems in terms of grading. Manually evaluation of the query would most likely result in a score of 0.0.

   (b) The student used subqueries in a way that ended up making the query unparsable. This could for example be due to missing or mismatching parentheses. Automatic fixing such mistakes would be difficult if not impossible. In these scenarios a manual evaluation of the query would most likely have given a good score, dependent on the rest of the answer.

   In other words, the 1.9% of errors in this data is acceptable as those are errors that could be extremely difficult to handle.

2. Manual evaluation of some of the queries shows that the result is over-all accurate but that it has its weaknesses when it comes to two types of queries:

(a) Handling "poorly formulated" queries. That is queries made by students who are struggling to understand the SQL-language. They may end up creating their own operators or use a syntax that is not recognize by the system. A human being on the other hand, may understand what a student is trying to do and will give a partial score that makes sense. The system on the other hand will struggle to evaluate such a query.

(b) Queries that can make sense taking the comment into consideration. This usually results in the system giving a worse score than what is deserved.

(c) Wrong use of special symbols. By Using a ";" in the middle of a query the system sees everything behind the ; as a comment, which means that the student will be missing out on half of the score, even if the part coming after the ";" is 100% correct. Using a ";" in the middle of the query is wrong, but the way that the system punishes it does not reflect a manual evaluation scenario, ending up giving the student a potentially worse grade than deserved.

3. In terms of scaling the system seemed to scale well in both speed and execution times when comparing simple queries against more advanced queries. There were almost no time difference(less than 20%) on the data tested, even though the query length doubled.

   Also, longer or more complex queries will just result in a bigger tree. The system still parses small parts of the query at a time, solving part for part instead of looking at the query as "one problem". This allows for great scaling in terms of complexity added to the queries as well.

4. The results showed that filters clearly affects the scores. Based on manual evaluation of the queries as well as looking at the normal distribution before/after filters were activated we can see that they make a huge difference. If anything, it makes the system more similar to how a real person would evaluate the queries by letting the person choose the filters that they think would make the system fair compared to their own standards

5. The data given back to the user showing the outcome and reasoning is not enough by itself to be used for tutoring or pedagogical purposes but more as a guideline to understanding how the system works in terms of evaluating. On the other hand, using the tree representation could be of interesting to students as it shows them how a query is converted into relational algebra. Students could be given to parts of the software allowing

them to insert queries to learn more about relational algebra and also gain some insight in how a query is actually executed in a DBMS.

8 Future Work

Further development of the system could be of interest. Some of them are:

- Improve the feedback that is given by the system. Further analyzing into what the student has not understood could be useful, in order to help the student to learn SQL
- Take into consideration the comments that are made by the user by for example creating a module that allows
- Add more filters to the system that allows for more flexibility and user control. E.g. Creating a weight-module for the user to specify how different part of the SQL should be weighed could be of interest.
- Test the system on more complex and larger data sets
- Compare the system against other SQL Grading systems, such as XData.

A Test Results

Could not analyze		
A	Q1	0
	Q2	1
	Q3	2
	Q4	6
	Q5	9
B	Q1	0
	Q2	0
	Q3	0
	Q4	6
	Q5	9
C	Q1	0
	Q2	0
	Q3	0
	Q4	6
	Q5	9
D	Q1	0
	Q2	0
	Q3	0
	Q4	6
	Q5	9
E	Q1	0
	Q2	2
	Q3	1
	Q4	7
	Q5	13

Blank answers	
Q1	2
Q2	2
Q3	3
Q4	3
Q5	2

Figure 10: Table showing the amount of answers that the system were not able to test, as well as the amount of blank answers.

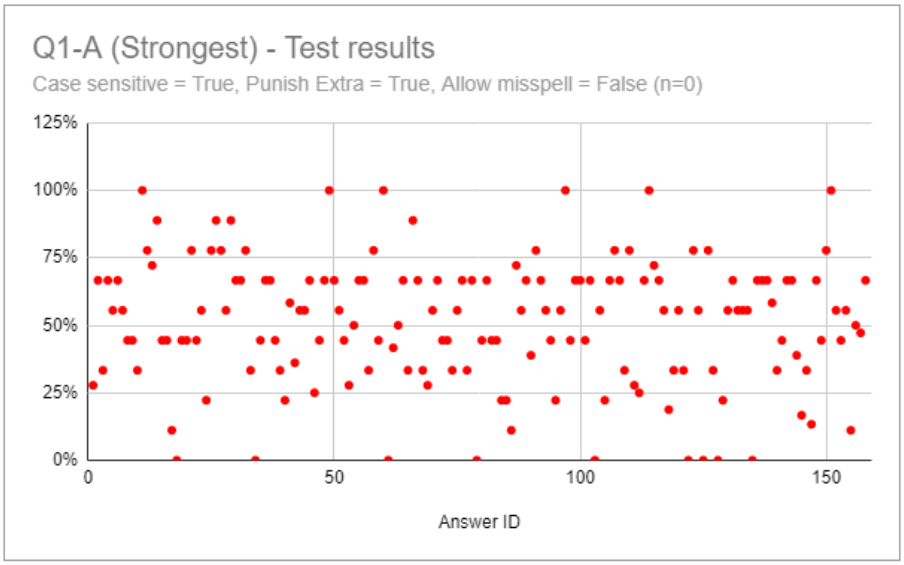


Figure 11: Q1.A Test results

Figure 12: Q1.B Test results

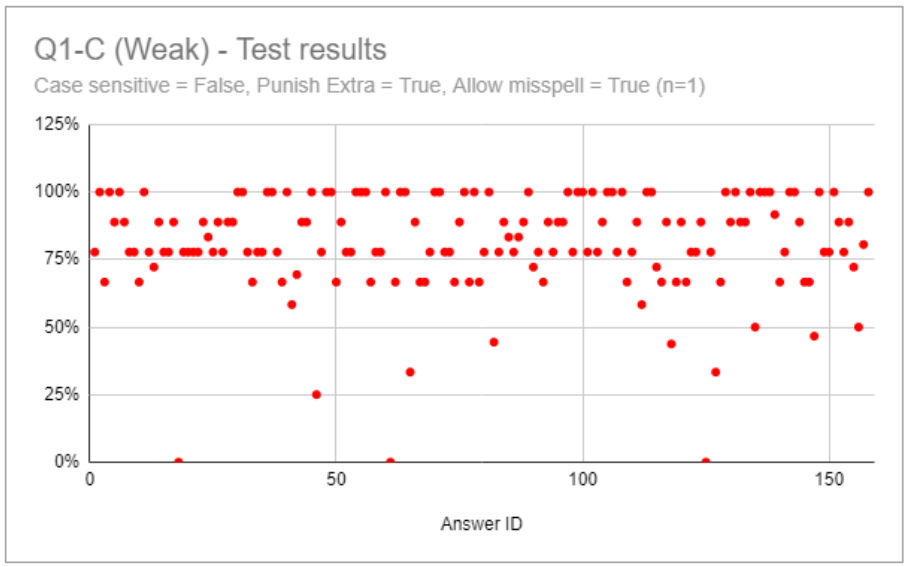


Figure 13: Q1.C Test results

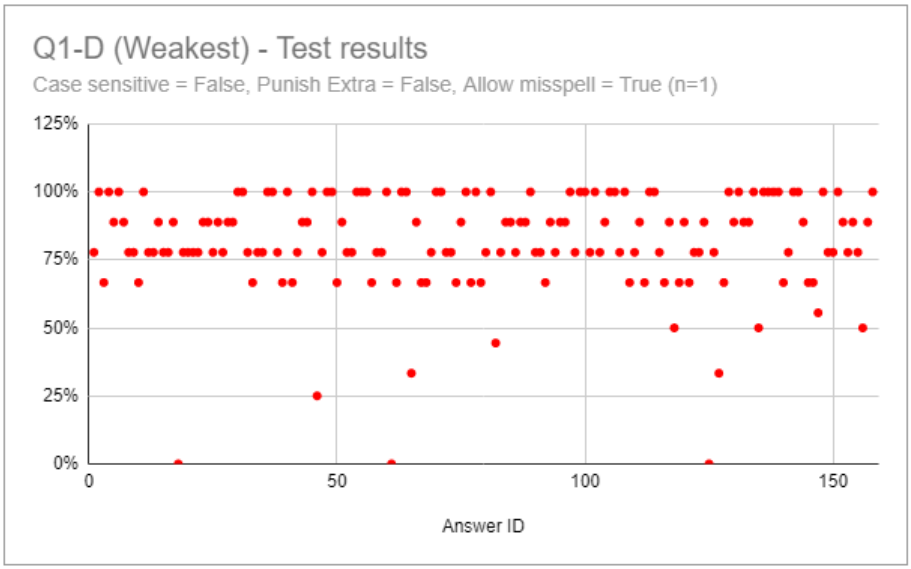


Figure 14: Q1.D Test results

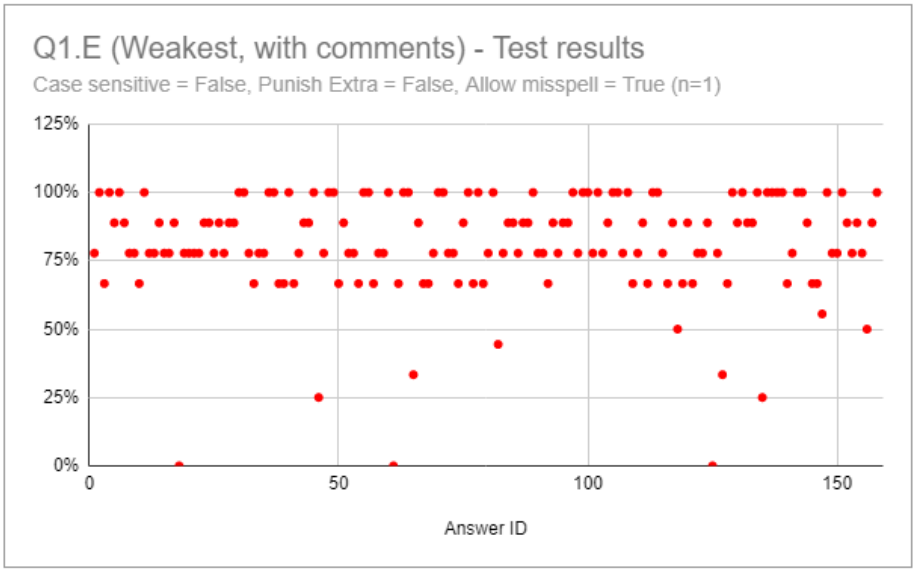


Figure 15: Q1.E Test results

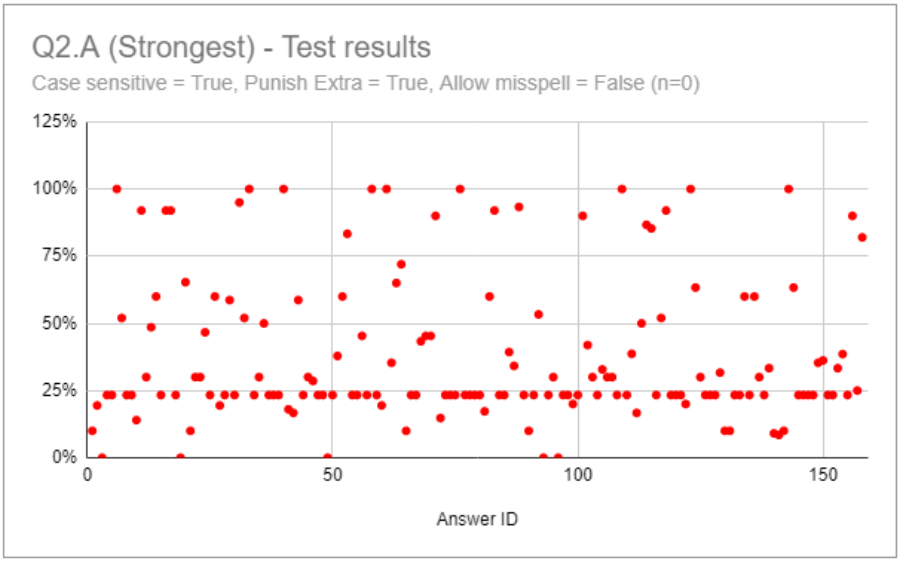


Figure 16: Q2.A Test results

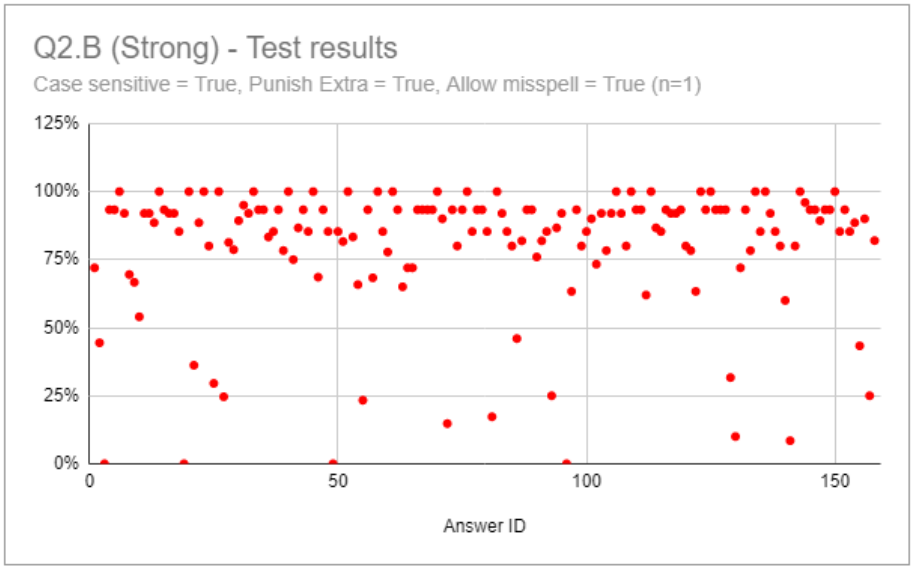


Figure 17: Q2.B Test results

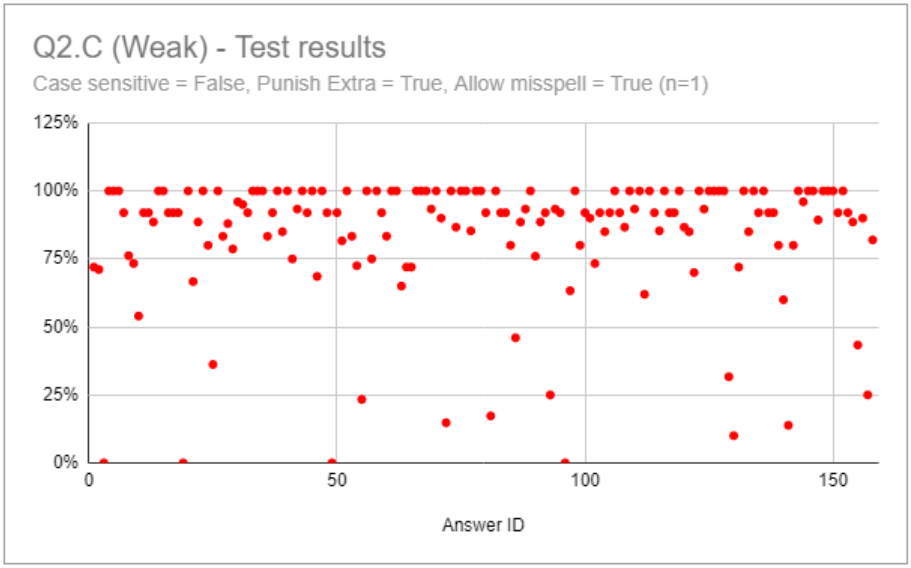


Figure 18: Q2.C Test results

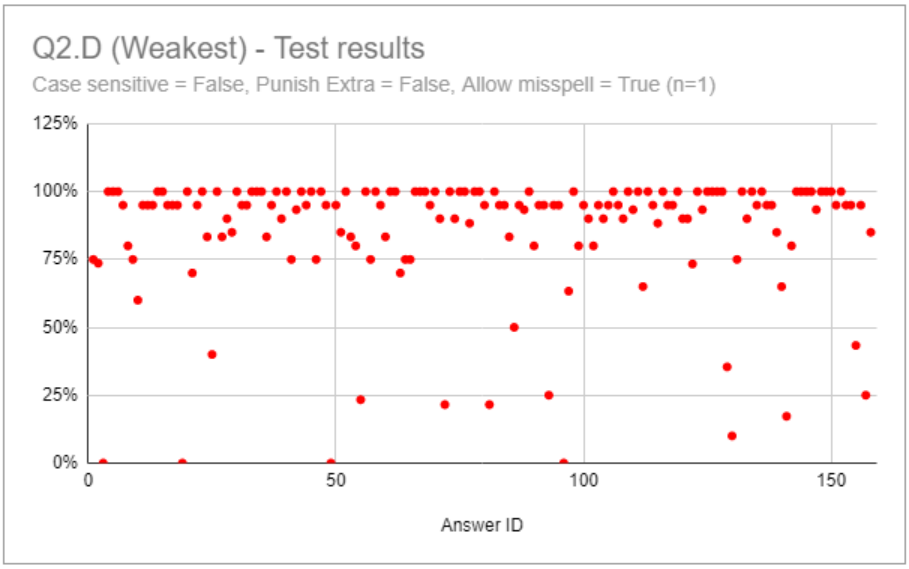


Figure 19: Q2.D Test results

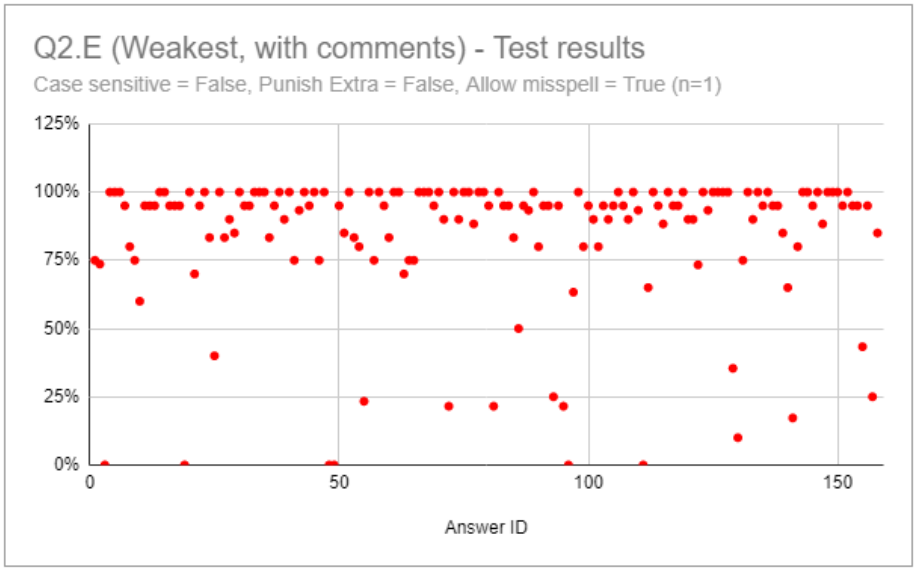


Figure 20: Q2.E Test results

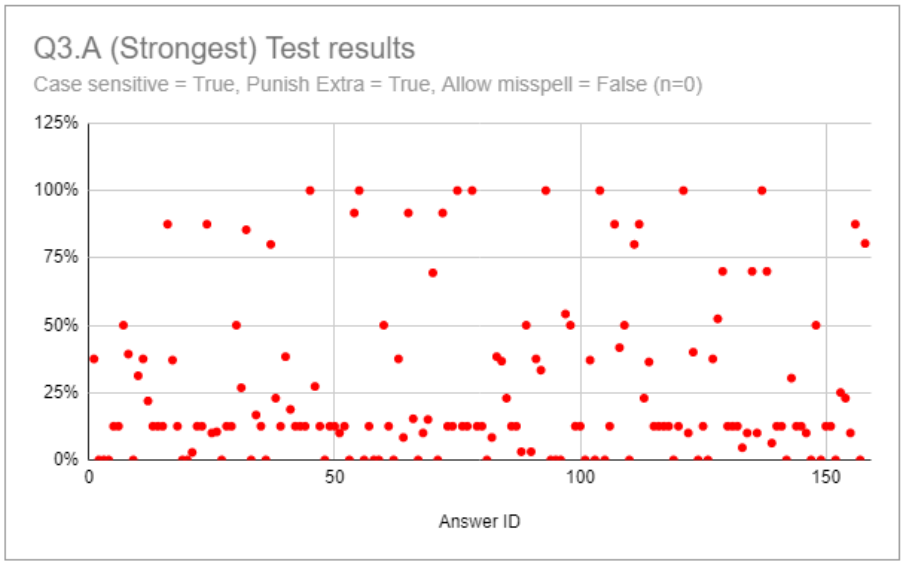


Figure 21: Q3.A Test results

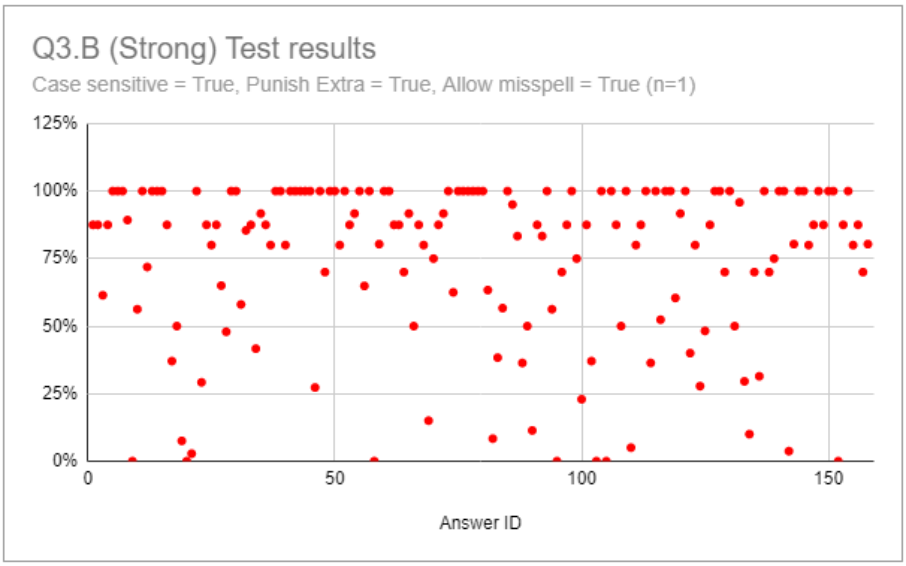


Figure 22: Q3.B Test results

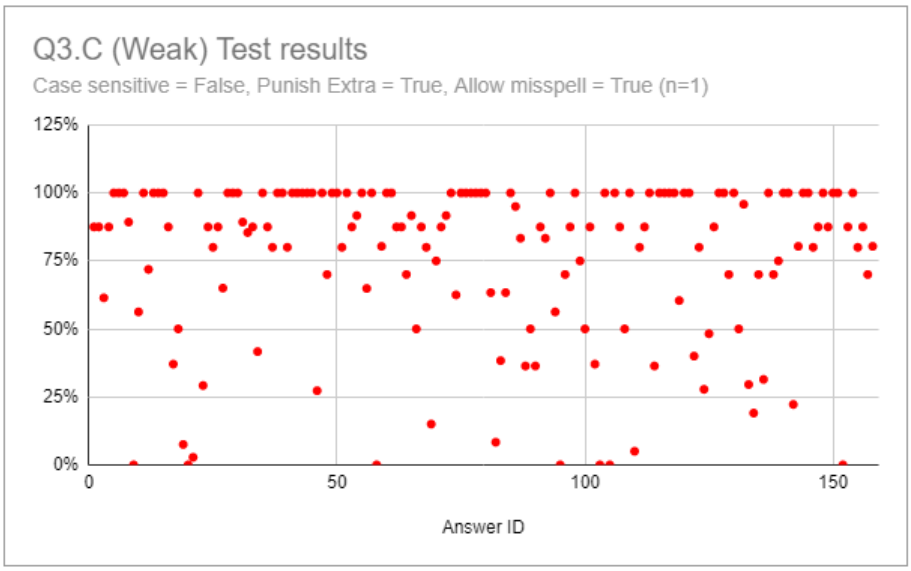


Figure 23: Q3.C Test results

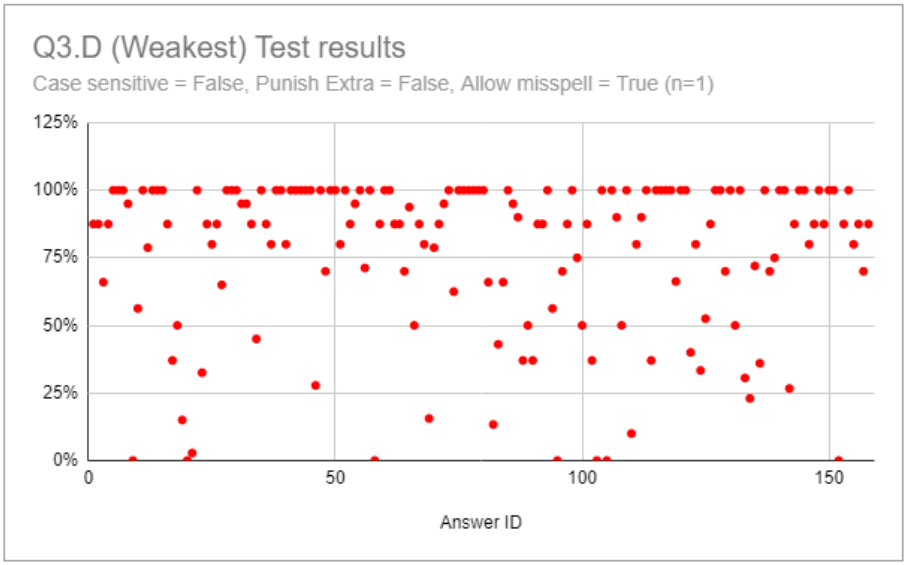


Figure 24: Q3.D Test results

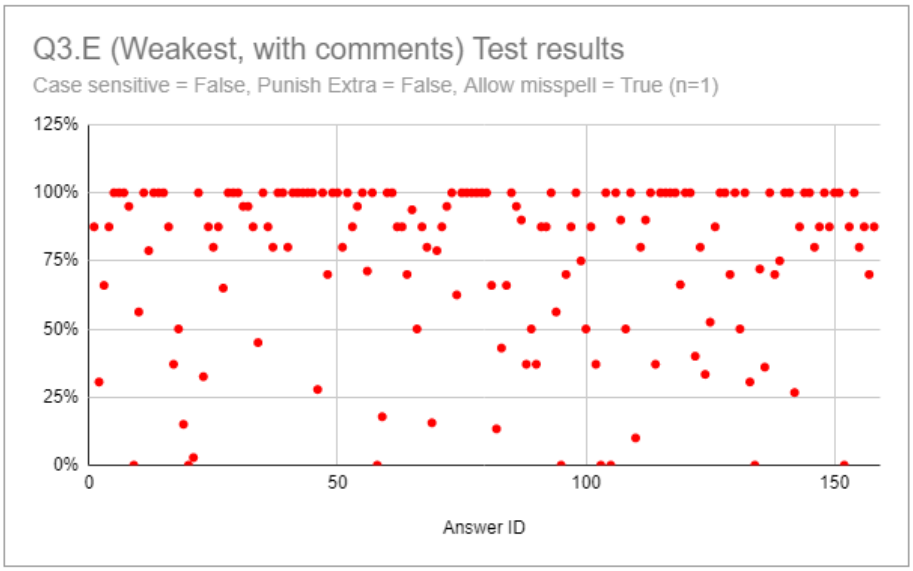


Figure 25: Q3.E Test results

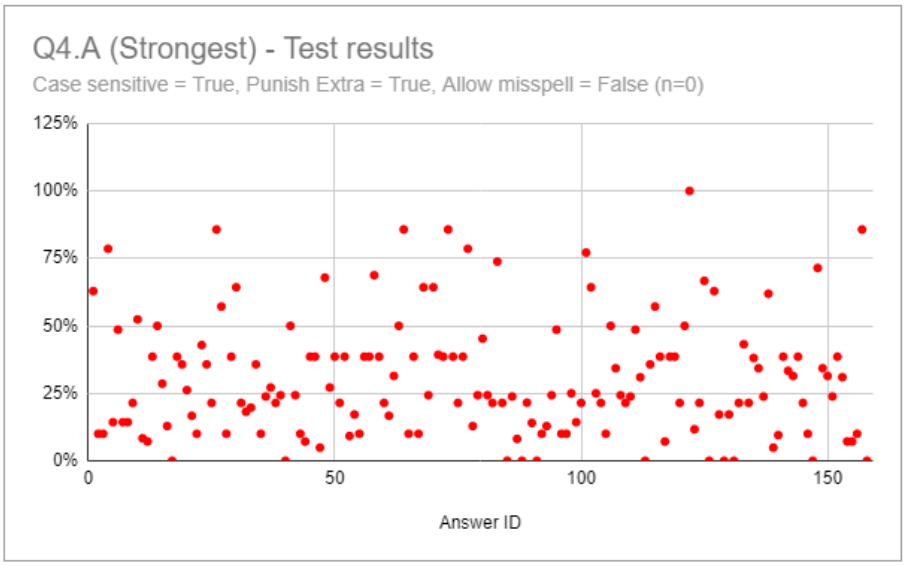


Figure 26: Q4.A Test results

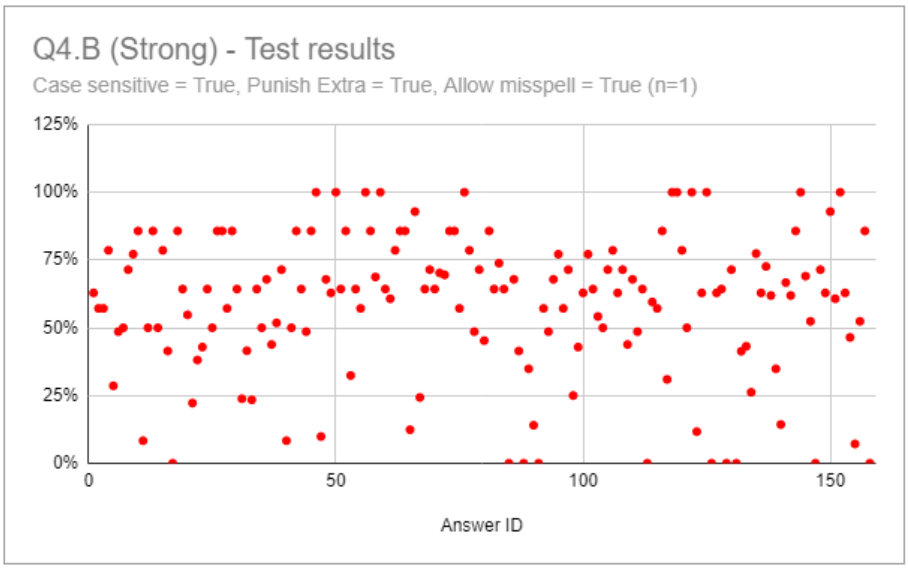


Figure 27: Q4.B Test results

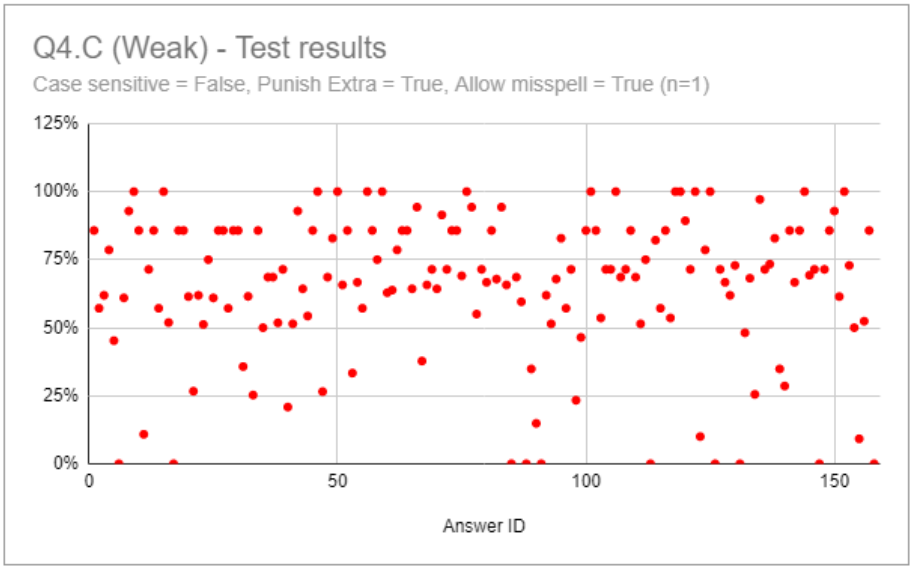


Figure 28: Q4.C Test results

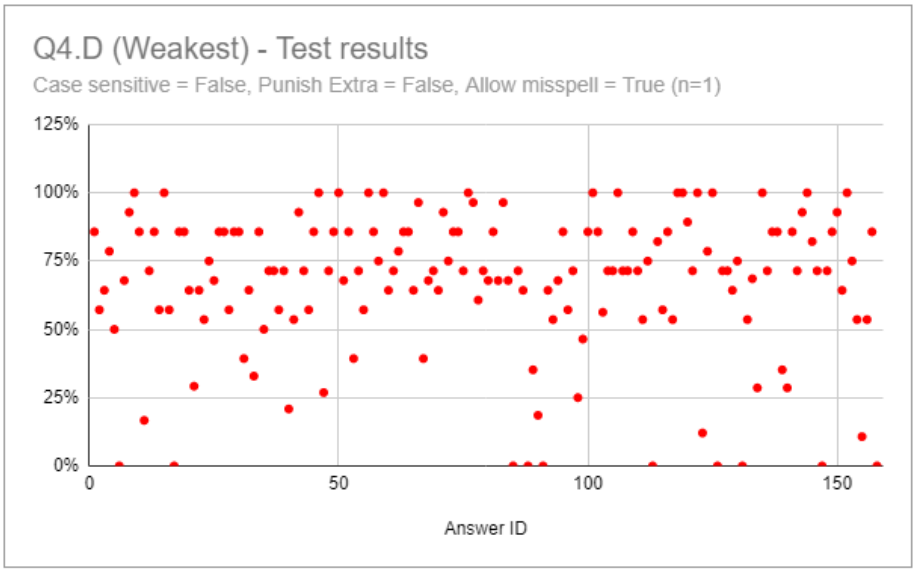


Figure 29: Q4.D Test results

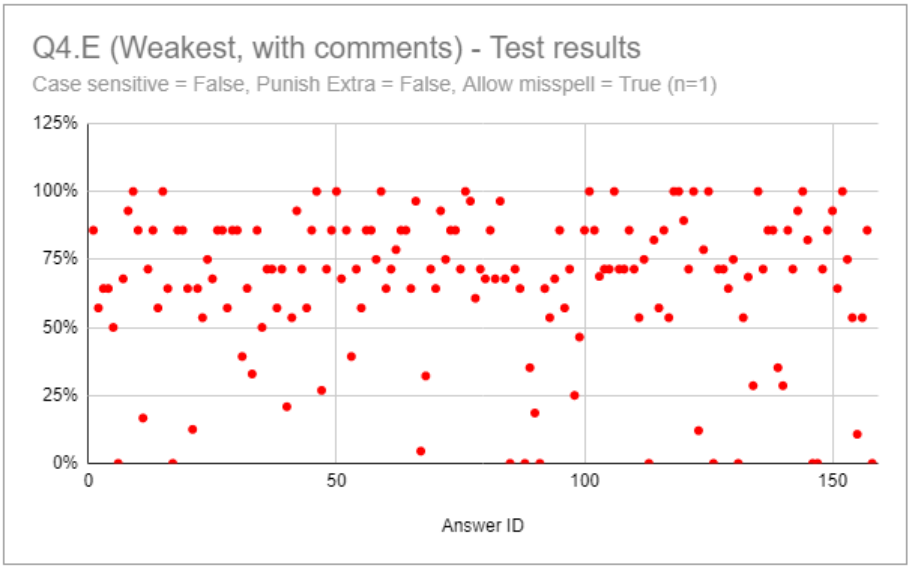


Figure 30: Q4.E Test results

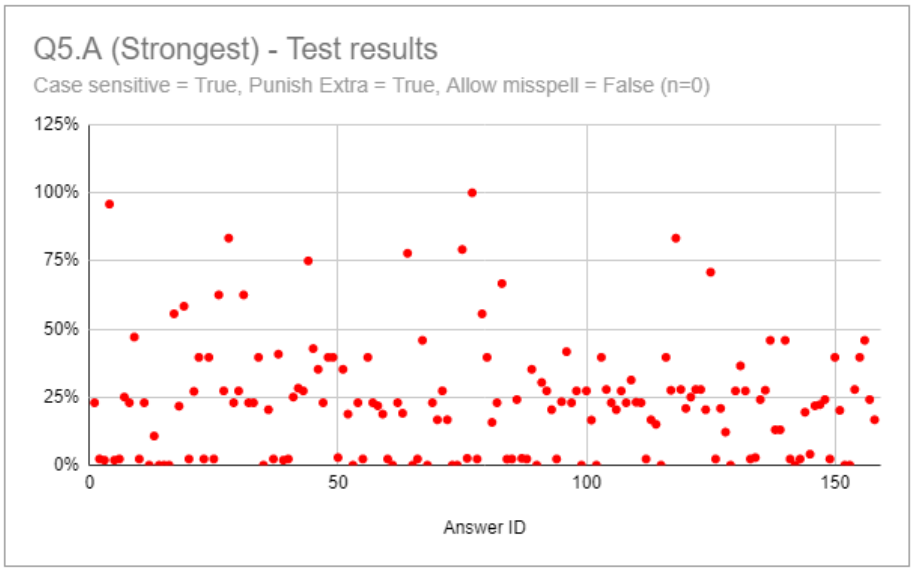


Figure 31: Q5.A Test results

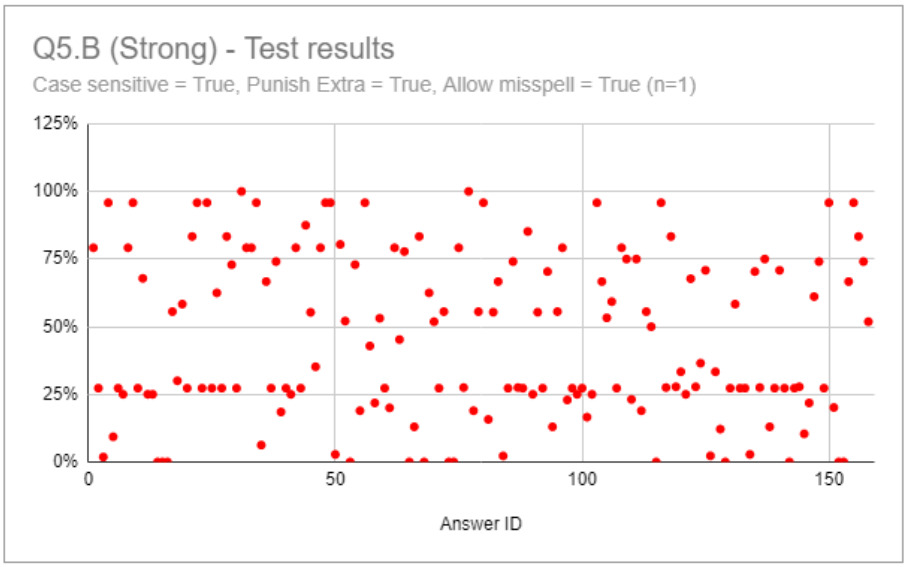


Figure 32: Q5.B Test results

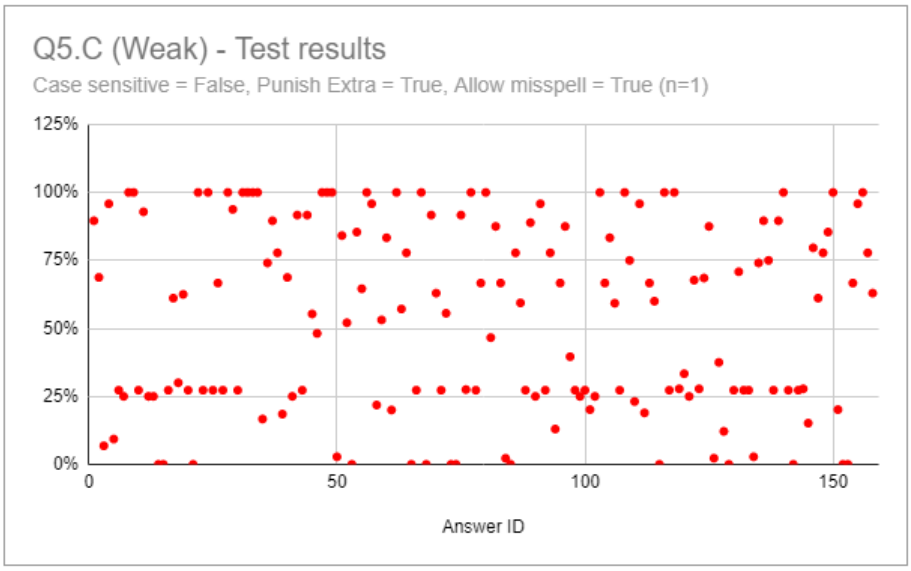


Figure 33: Q5.C Test results

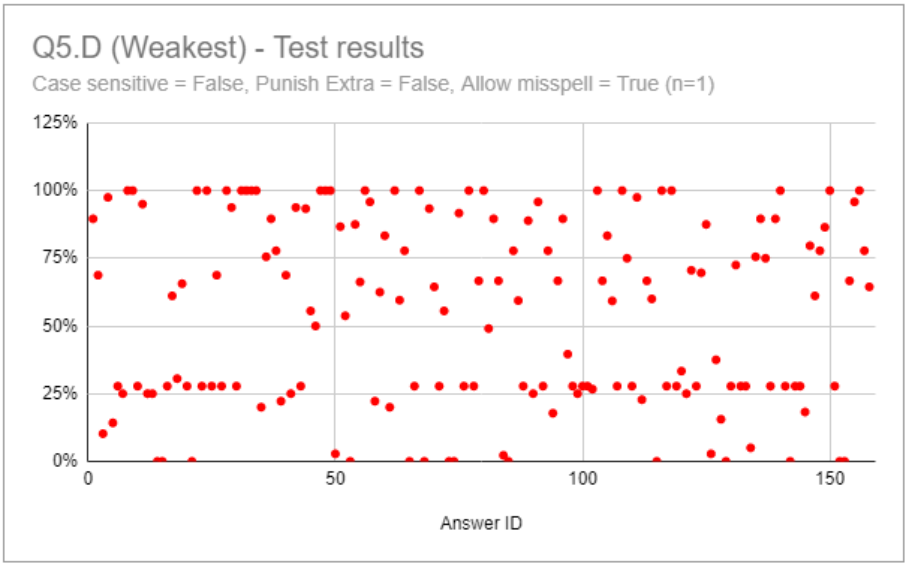


Figure 34: Q5.D Test results

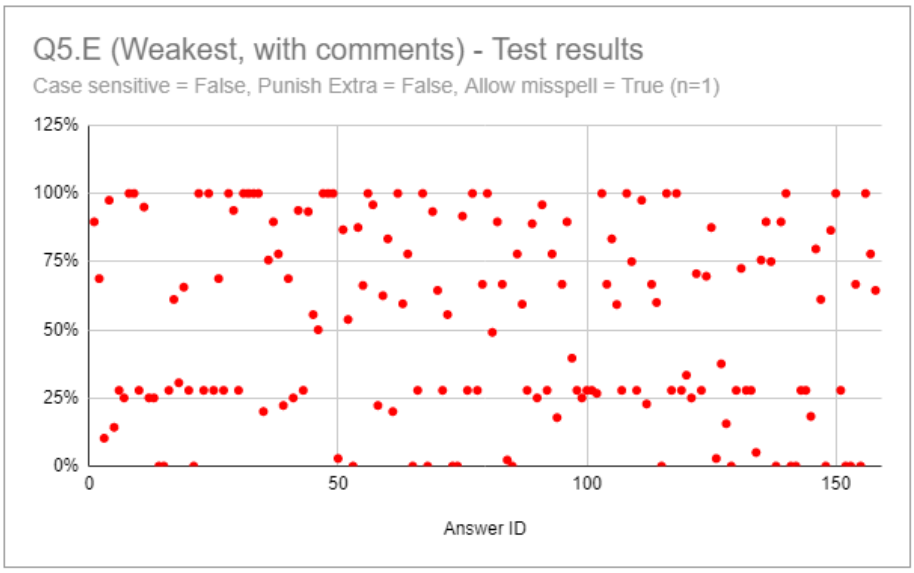


Figure 35: Q5.E Test results

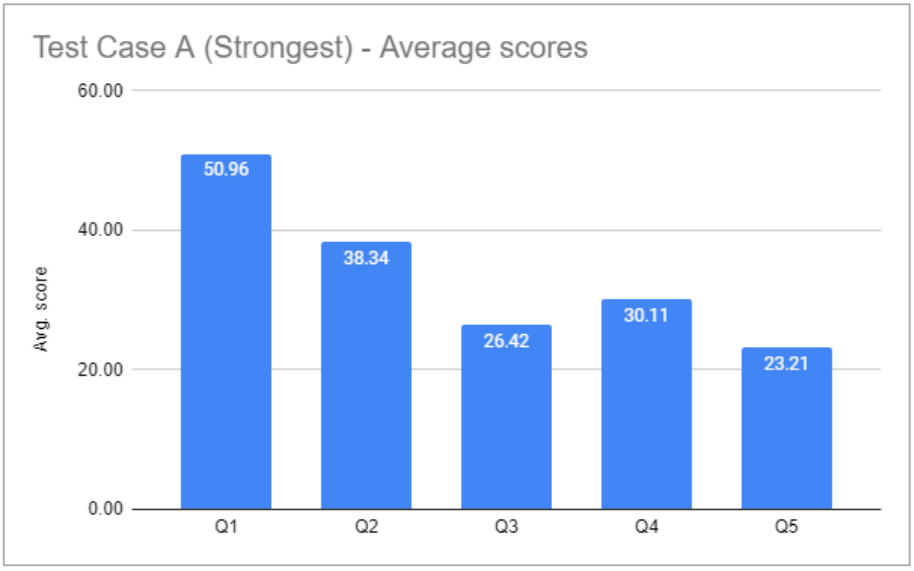


Figure 36: Average results from Test Case: A

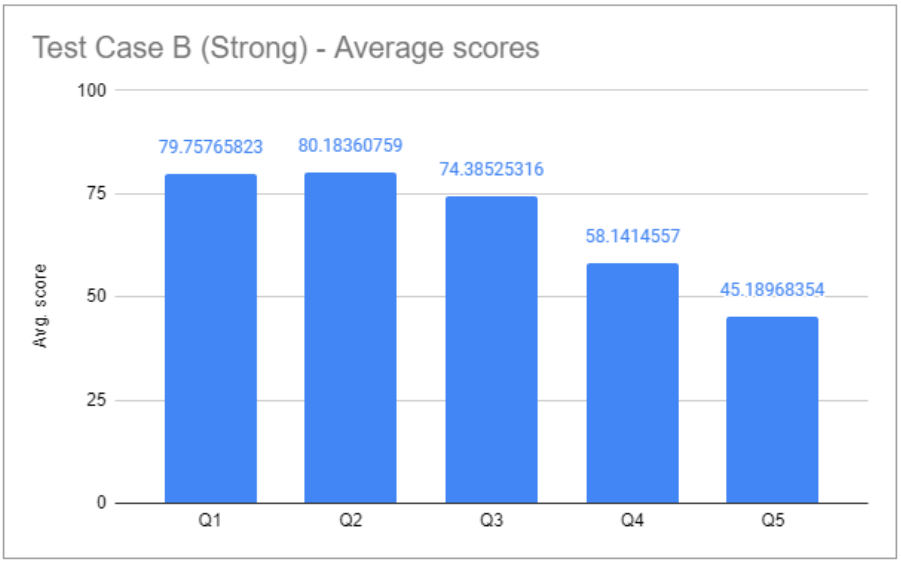


Figure 37: Average results from Test Case: B

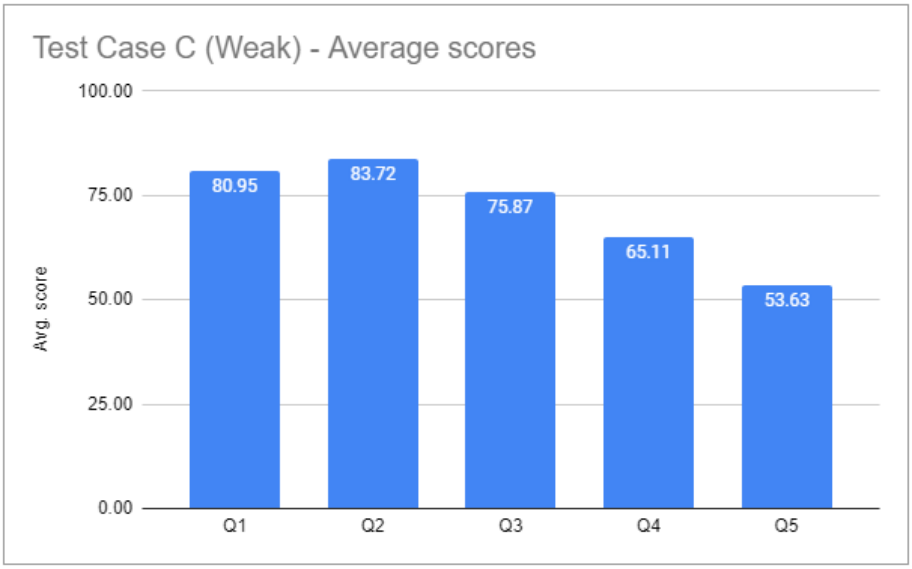


Figure 38: Average results from Test Case: C

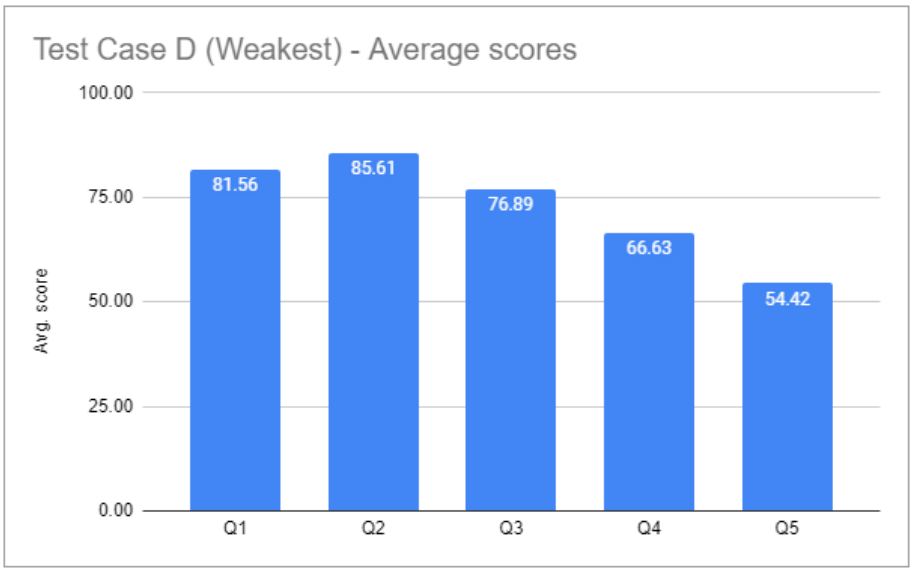


Figure 39: Average results from Test Case: D

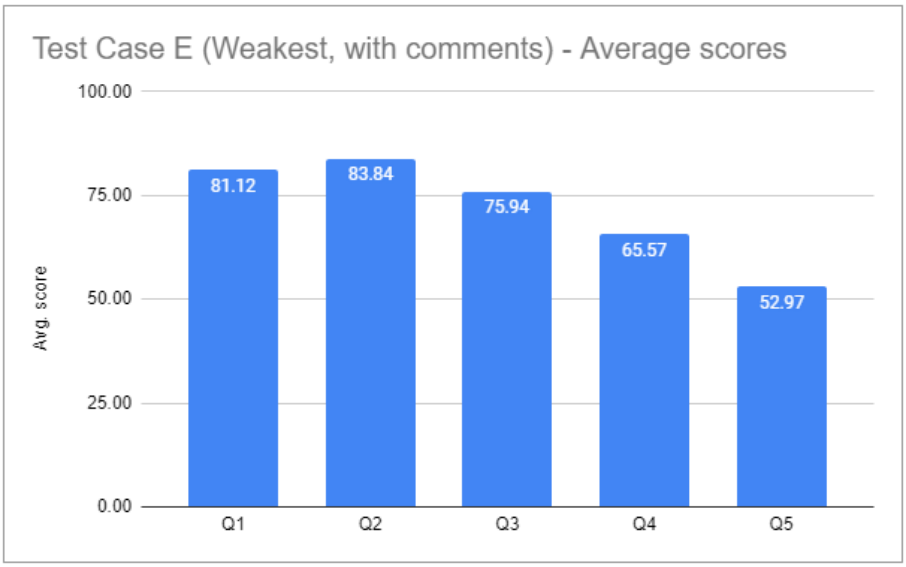


Figure 40: Average results from Test Case: E

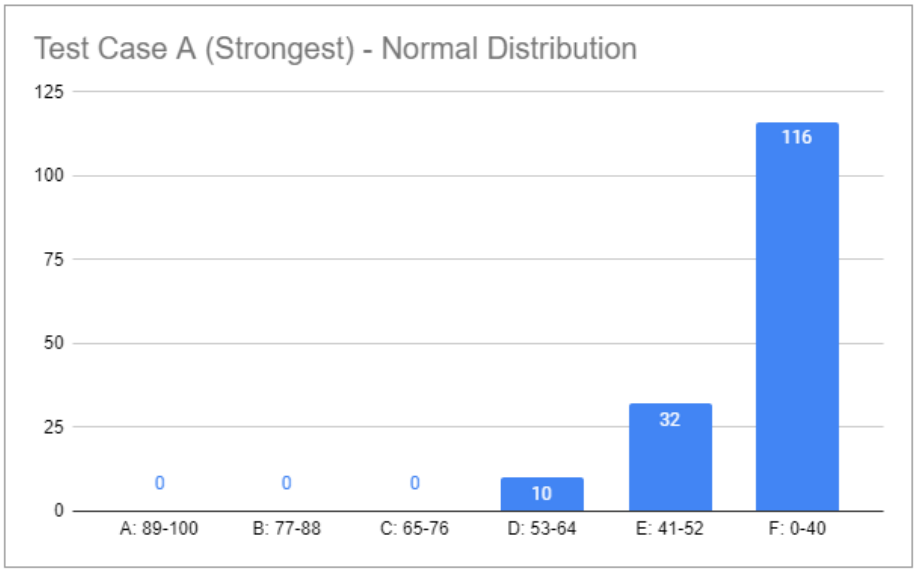


Figure 41: Normal Distribution from Test Case: A

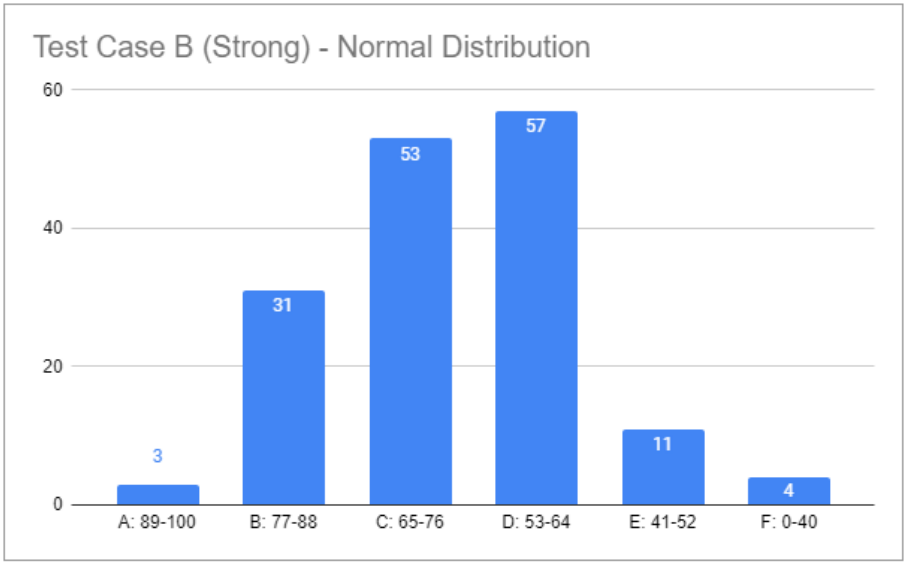


Figure 42: Normal Distribution from Test Case: B

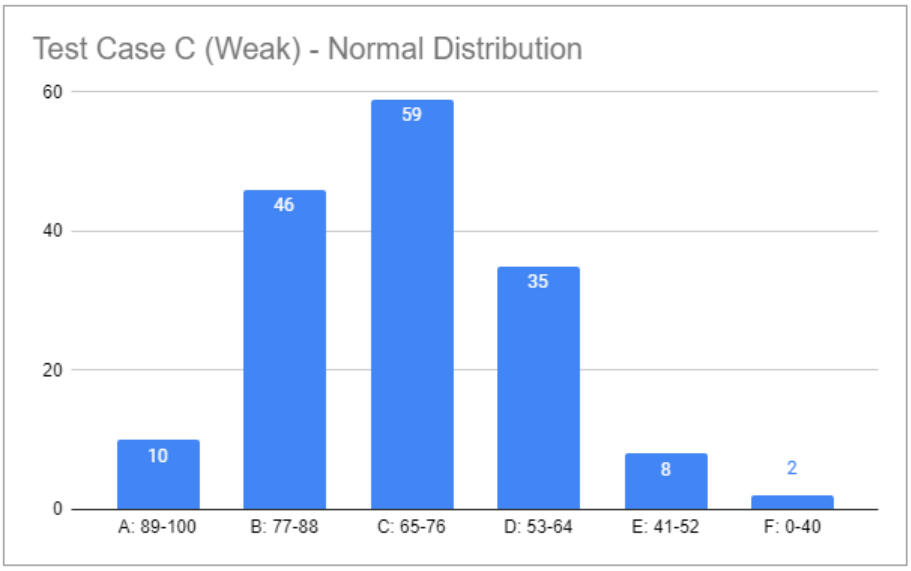


Figure 43: Normal Distribution from Test Case: C

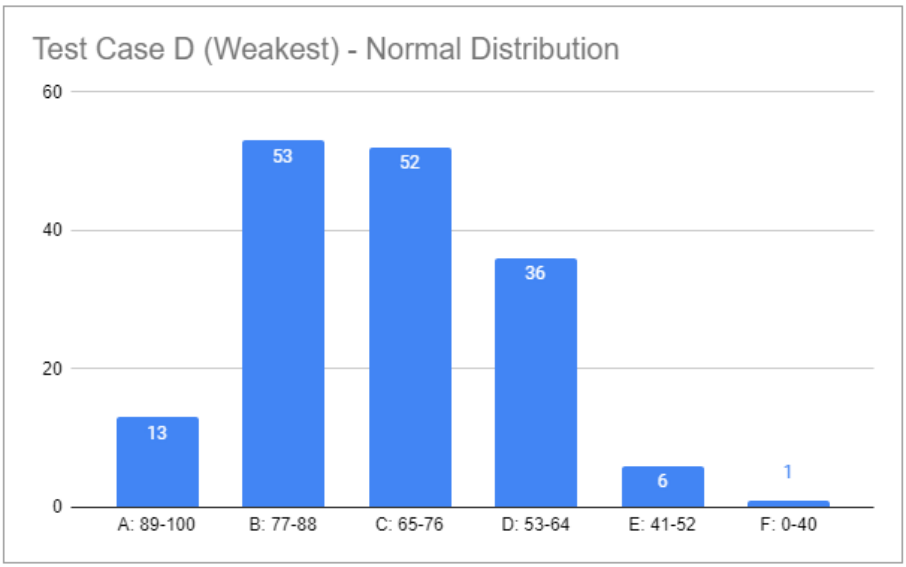


Figure 44: Normal Distribution from Test Case: D

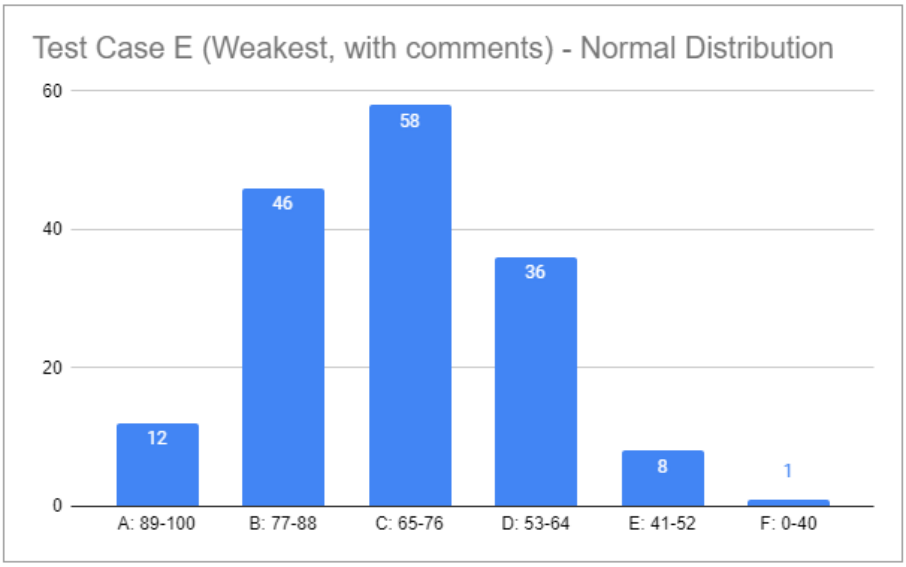


Figure 45: Normal Distribution from Test Case: E

References

Abelló, A., M. E. Rodríguez, T. Urpí, X. Burgués, M. J. Casany, C. Martín, and C. Quer (2008). Learn-sql: Automatic assessment of sql based on ims qti specification. In *2008 Eighth IEEE International Conference on Advanced Learning Technologies*, pp. 592–593. IEEE.

Anley, C. (2002). Advanced sql injection in sql server applications.

Beaulieu, A. (2009). *Learning SQL: master SQL fundamentals.* " O'Reilly Media, Inc.".

Big Data Zone (2020). Join types venn diagram. `https://dzone.com/articles/how-to-perform-joins-in-apache-hive`, Last accessed on 2020-06-10.

Bourhis, P., J. L. Reutter, F. Suárez, and D. Vrgoč (2017). Json: data model, query languages and schema specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems*, pp. 123–135.

Ceri, S. and G. Gottlob (1985). Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries. *IEEE Transactions on software engineering* (4), 324–345.

Chandra, B., A. Banerjee, U. Hazra, M. Joseph, and S. Sudarshan (2019). Automated grading of sql queries. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1630–1633. IEEE.

Danaparamita, J. and W. Gatterbauer (2011). Queryviz: helping users un-

derstand sql queries and their patterns. In *Proceedings of the 14th International Conference on Extending Database Technology*, pp. 558–561. ACM.

David, M. M. (1999). *Advanced ANSI SQL data modeling and structure processing*. Artech House.

Hursch, C. J., J. L. Hursch, and C. J. Hursch (1988). *SQL, the Structured Query Language*. Tab Books.

Jdmcpeek (2017). Pretty print binary tree. `https://github.com/jdmcpeek/pretty-print-binary-tree`, Last accessed on 2020-06-10.

Kjerstad, J. (2019). A study on intelligent tutor systems and exercisegeneration in sql.

Klug, A. (1982). Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM (JACM) 29*(3), 699–717.

Mitrovic, A. (2003). An intelligent sql tutor on the web. *International Journal of Artificial Intelligence in Education 13*(2-4), 173–197.

NTNU (2020). Grading scale using percentage points. `https://innsida.ntnu.no/wiki/-/wiki/English/Grading+scale+using+percentage+points`, Last accessed on 2020-06-07.

Python (2001). Grading scale using percentage points. `https://docs.python.org/3/library/shlex.html`, Last accessed on 2020-05-29.

Srinath, K. (2017). Python–the fastest growing programming language. *International Research Journal of Engineering and Technology (IRJET) 4*(12), 354–357.

Van Rossum, G. and F. L. Drake (1995). Python library reference.

Widenius, M., D. Axmark, and K. Arno (2002). *MySQL reference manual: documentation from the source*. " O'Reilly Media, Inc.".

Yergeau, F. (2003). Utf-8, a transformation format of iso 10646. Technical report, STD 63, RFC 3629, November.

NTNU

Norwegian University of
Science and Technology