TIC2003 Software Development Project

Static Programme Analyzer

Project Report - Iteration 3

AY 2021/2022 Semester 2

TEAM 17

DARREN SIM YI KANG (A0227083H)
+65 9457 9118
e0648984@u.nus.edu

LEE JOEL (A0227031W)
+65 9118 3543
e0648932@u.nus.edu

1. Project Information

1.1. Background

Very often a programmer will be examining a large amount of code. This may result in the programmer overlooking certain errors while trying to locate the codes during program maintenance. Therefore, the design of our Static Program Analyzer (SPA) will help us as programmers understand programs better. The idea is to design a tool for programmers that is able to automatically answer the queries on the program using SPA for a simple source language. This will allow us to efficiently locate the particular statement that is causing errors which needs to be fixed and also check for possible unwanted effects of changes that are to be made on any particular statement.

1.2. Prototype Implementation Progress - Iteration 1

In Iteration 1, as the team are unfamiliar with the SPA implementation and SIMPLE language, our objective is to successfully initialize the database, create tables for the insertion of the following entities: procedure, variable, constant, read, print and statement. After which, the team aims to successfully parse the SIMPLE source code and queries to correctly populate the database and extract from it to answer the queries input. To achieve this, we split the weeks into several mini-iterations and assigned functionality-specific tasks for each one.

Week	**Tasks**
3	• Source Processor should be able to parse procedures, print, read, and assignment statements • Database should be able to implement methods to populate design entities in the database tables • Query Processor should be able to parse Select clauses
4	• Database should be able to implement methods to retrieve design entities from the database tables

	● Query Processor should be able to evaluate Select clauses ● Write report and documentation ● Write system test cases

1.3. Prototype Implementation Progress - Iteration 2

In iteration 2, our team aims to build a more advanced SPA application that is able to handle program design abstractions. “while” loops, “if” statements as well as Parent/Parent*, Modifies and Uses relationships.

Week	**Tasks**
5	● Convert Source Processor to recursive descent ● Ensure all design entities are stored correctly in the database ● Query Parser is able to support Such That and Pattern parsing
6	● Source Processor is able to handle while and if statements, as well as assignment statements with expressions ● Source Processor, is also able to handle relationships: Parent/Parent*, Modifies and Uses ● Ensure new tables are created and all design entities are stored correctly in the database ● Query Processor can now evaluate Parent, Parent*, Modifies and Uses Such That Clauses, and
7	● Database to be able to implement methods to retrieve design entities and design abstractions from the database. ● Write the report and documentation. ● Write system test cases and perform testing to ensure all possible test cases are covered. ● Debugging of Query Processor evaluation for all design abstractions (Iter2). ● Query Parser and Query Processor is able to parse queries with a combination of

	Such That and Pattern Clause.

1.3. Prototype Implementation Progress - Iteration 3

In iteration 3, our team improved the implementations in iteration 2 to handle multiple procedures. This also includes more advanced design abstractions with Modifies for procedures, Uses for Procedures, Calls, Calls*, Next and Next*.

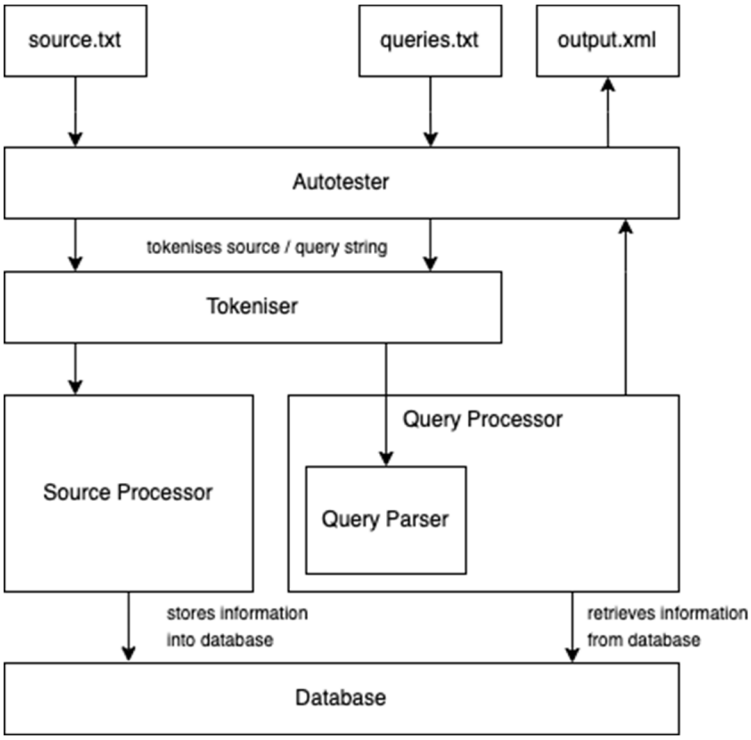
Week	**Tasks**
9	● Ensure that Source Processor is able to take in multiple procedures ● Implemented the design abstraction Calls and ensure that the procedure correctly call its direct procedure ● Implemented the design abstraction Calls* and ensure that the procedure correctly call its indirect procedure ● Ensure both Calls and Calls* are stored correctly into the database ● Query Parser is able to support Select clause tuple, and Pattern clause expression specification and exact matching. ● Query Processor is able to get Calls and Calls* data from the database. ● Implement Query Table to aid in filtering synonyms from multiple Select, Such That and Pattern clauses.
10	● Create new database tables to accommodate Modifies for procedure (modifiesproc) and Uses for procedure (usesproc) ● Implement and ensure that “modifiesproc” is able to store direct Modifies for procedure into the database ● Implement and ensure that “modifiesproc” is able to store indirect Modifies for procedure into the database using Calls* ● Implement and ensure that “usesproc” is able to store direct Uses for procedure into the database ● Implement and ensure that “usesproc” is able to store indirect Uses for procedure

	into the database using Calls* ● Query Processor is able to get ModifiesP and UsesP data from the database.
11	● Implement Next and Next* ● Ensure edge cases for while loops (last line of while loop) and if-then-else statements (last line of “if-then” and “if-else”) are correctly identified and point to the correct next element. ● Query Processor is able to get Next and Next* data from the database. ● Write system test cases and perform testing to ensure all possible test cases are covered. ● Write the report and documentation.

2. SPA Design

2.1. Overview

In this section, we will discuss the main components of the SPA and how they interact with each other with a diagram to help illustrate this.



There are 3 main components to the SPA, the Source Processor, Database and Query Processor. Each component plays a role in the stack by interacting with each other to ensure that the SPA is working as intended.

- Source Processor

  The Source Processor is the front end of the SPA, where it analyzes the SIMPLE source code, parses it and stores the design entities into the database.

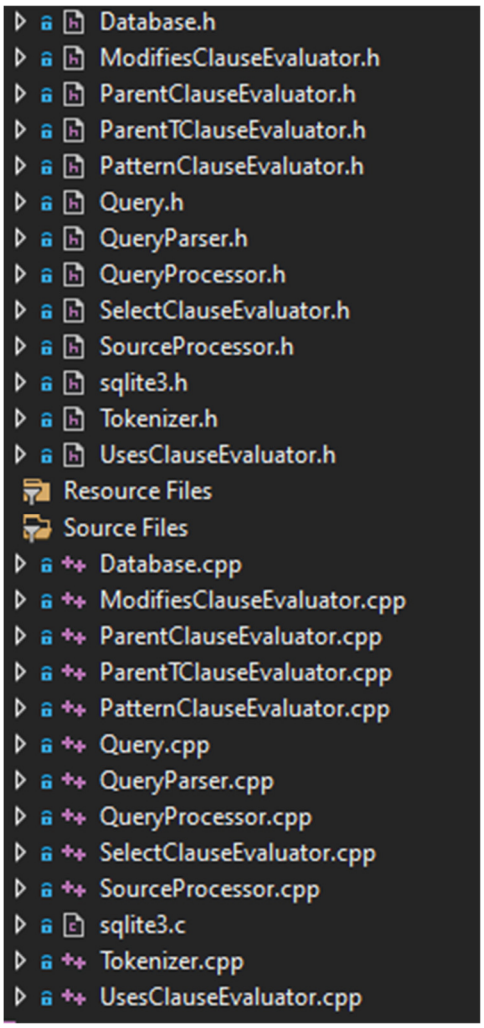
- Database

This component stores design entities and synonyms in their respective tables obtained from the Source Processor, only to be retrieved by the Query Processor.

- Query Processor

  The Query Processor analyzes the PQL queries by parsing them. Next, it evaluates these queries using the information of the source code stored in the database, before returning the results of the query.

2.2. File Hierarchy



We have decided to break the Query Processor into smaller sub-components for better scalability and making it easier to debug the program via unit testing. It is also in accordance with the Single Responsibility Principle, which states that every class in the

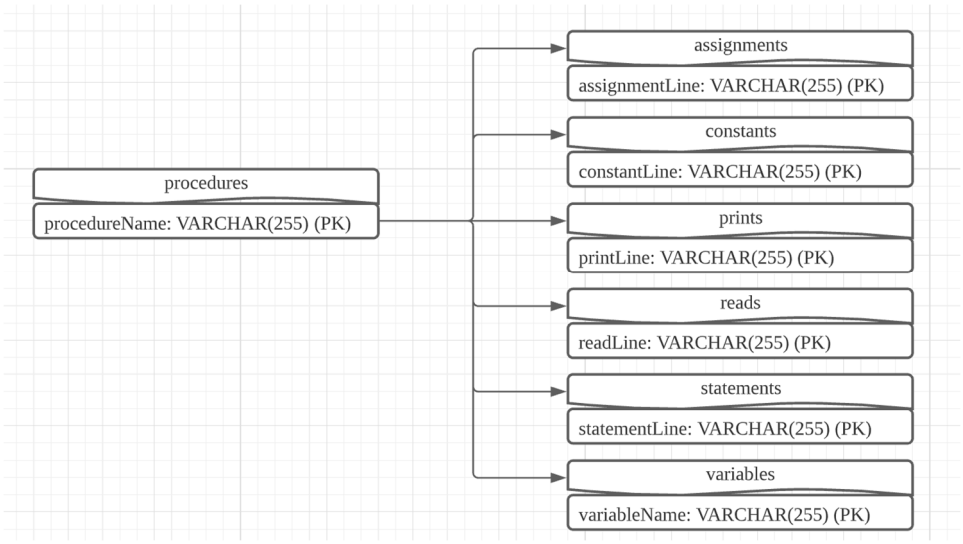
SPA program should have responsibility over a single part of SPA's functionality. For iteration 2, we also have a few "ClauseEvaluator"-type classes, which will handle the evaluation of each clause in the PQL query. Details about how the ClauseEvaluator classes operate with the Query Processor would be explained in the Query Processor section of the report.

2.3. Tokenizer

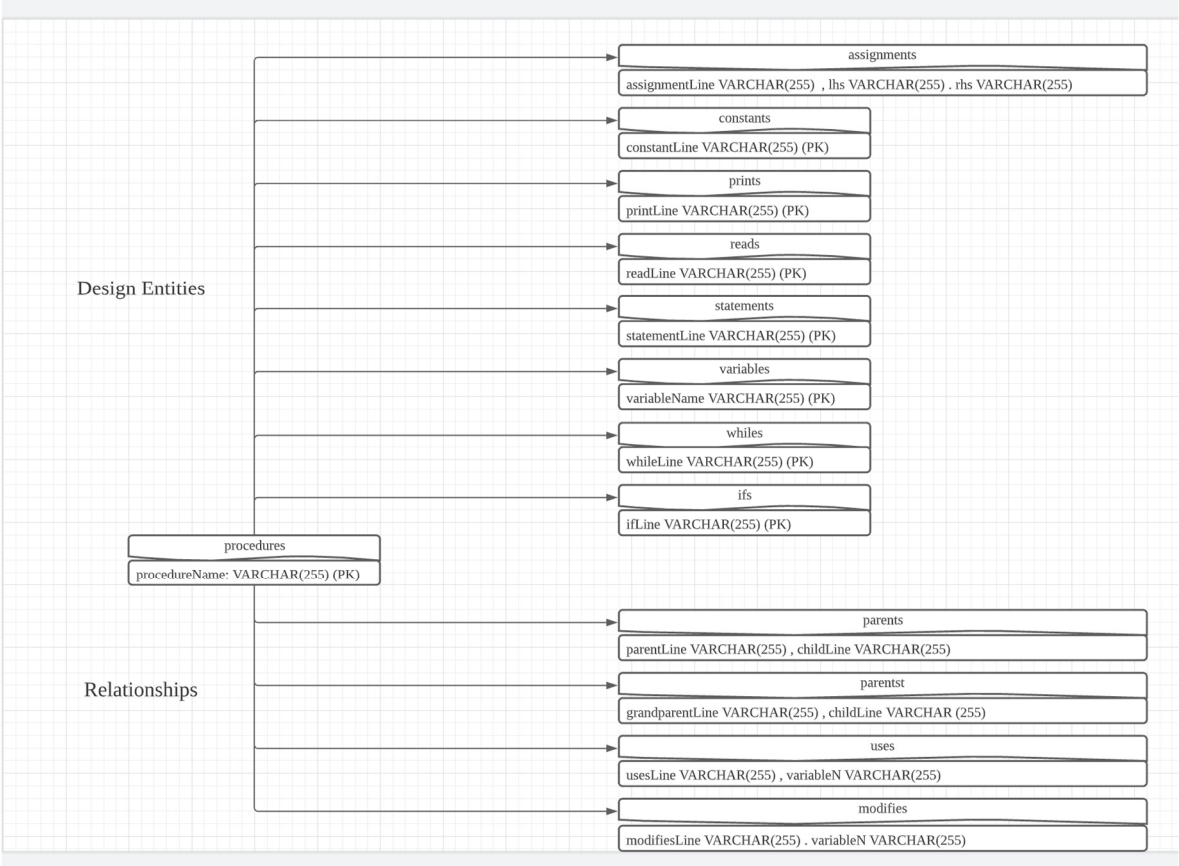
The Tokenizer class is responsible for tokenizing the source or query string into individual tokens that would be further parsed (and validated in later iterations) by either Source Processor or Query Processor. The Tokenizer works by removing whitespaces in the source and query string, and splitting the string into a vector of strings, called tokens. These tokens can either be an alphanumeric word (e.g. "main", "a1") or a symbol (e.g. ";").

2.4. Database

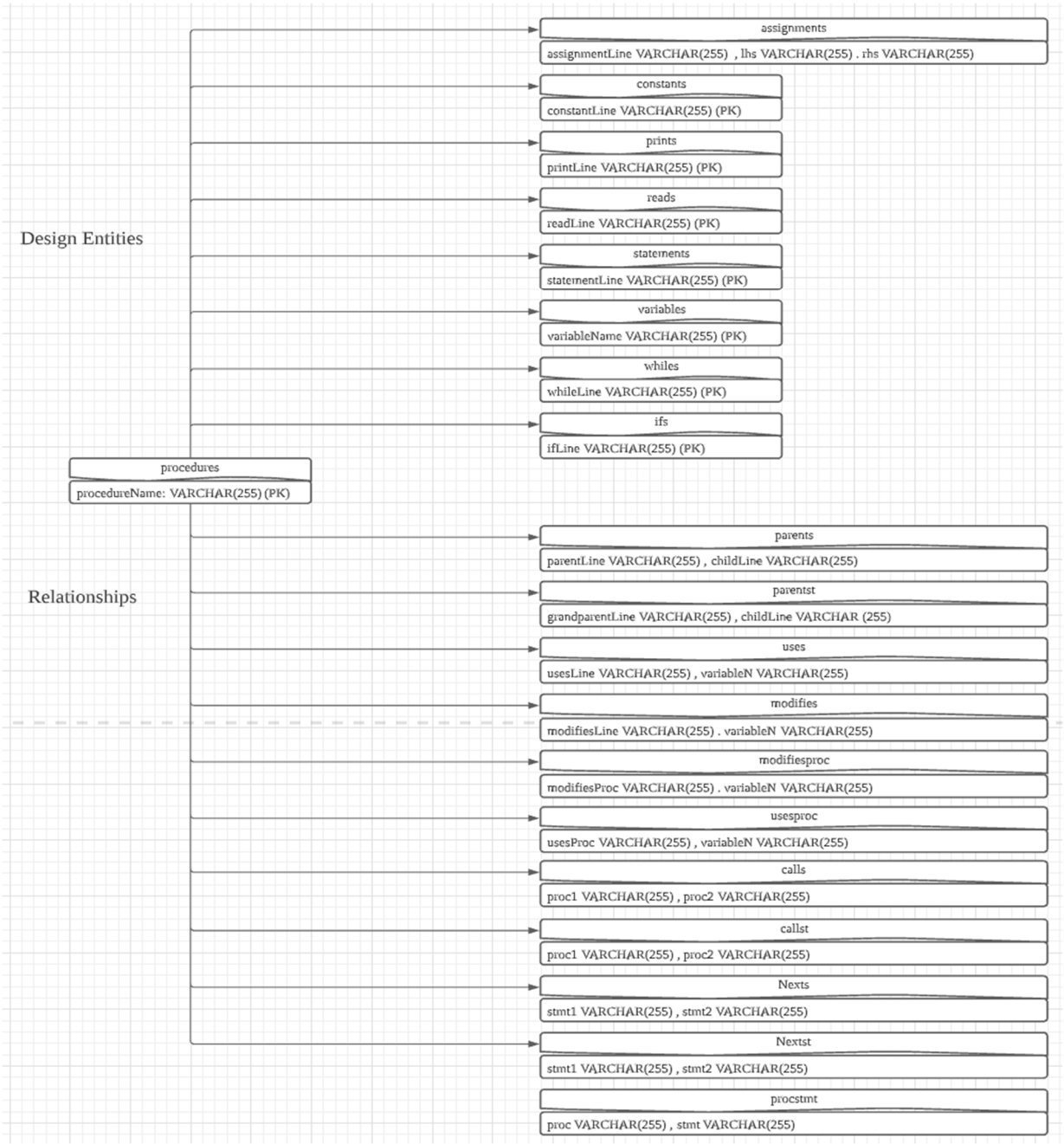
The Database class opens and closes the database connection. The database contains methods to drop any existing tables and create new ones. The database consists of the following entities 'assignments', 'constants', 'prints', 'procedures', 'reads', 'statements' and 'variables' with the following attributes and type as their primary keys respectively 'assignmentLine VARCHAR(255)', 'constantName VARCHAR(255)', 'printLine VARCHAR(255)', 'procedureName VARCHAR(255)', 'readLine VARCHAR(255)', 'statementLine VARCHAR(255)' and 'variableName VARCHAR(255)'. When the Source Processor takes in the source program, the Source Processor calls the Database class 'insert' method to insert the content into the correct tables in the database. Likewise, the Query Processor calls the Database class 'get' method to retrieve the correct query from the respective tables. The relational schema of the database as shown in the figure below:



In iteration 2, two more design entities ‘whiles’ and ‘ifs’ have been added. Relationships have also been introduced, namely ‘parents’, ‘parents*’, ‘uses’ and ‘modifies’. These changes have been reflected in our database as per the updated relational scheme below:



In iteration 3, 8 more design abstractions are being added to the database. The database now consists of the following entities ***'assignments', 'calls', 'callst', 'constants', 'ifs', 'modifies', 'modifiesproc', 'nexts', 'nextst', 'parents', 'parentst', 'prints', 'procedures', 'procstmt', 'reads', 'statements' 'uses', 'usesproc', 'variables'*** and ***'whiles'.*** The changes are reflected in the database schema below:



2.5. Source Processor

The role of the Source Processor in the SPA is to analyze the SIMPLE program and perform parsing to extract meaningful information from the source file and store it accurately into the database. The Source Processor component takes in the source program and calls on the Tokenizer class to tokenize the source into individual tokens. For iteration 1, a simple while loop with nested if statements is able to iterate through the tokens, identify them and correctly store them into the database accordingly. Stringstream is used to convert integers into strings as well as to convert strings to integers as constant, print, statement and assignment requires us to store the line number in the procedure into the database. The source code requirements in iteration 1 is basic enough for the use of multiple while loops and if statements. In iternation 2, we will be introducing Source Parser to correctly parse more complex source codes and check for potential grammar and syntax errors. Our team has decided to do so to show the progression from iteration 1 to iteration 3.

2.5.1 Source Parser

For iteration 2 onwards, the Source Parser is added into the Source Processor to parse more advanced source codes and check for both grammar and syntax errors. Recursive descent is introduced with a top-down traversal of the grammar specification, it allows us to create multiple functions and methods for the parsing of the grammar rules. This also allows us to reduce the amount of code to write as the functions are reused accordingly. This has also made it easier for us to debug the codes. The functions and methods are as follow:

- void process(string program);
- void parse(list<string> tokens);
- void parseProgram();
- void parseProcedure();
- void parseStatement();
- void parseVariable();
- void parseFactorCondition();

- void parseFactor();
- void parseConstant();
- void parseExpression();

In order to check and ensure that the grammar NAME or INTEGER is correct, we included the standard C++ library “regex” to define the correct syntax for both NAME (#define INTEGER_PATTERN "[0-9]+") and INTEGER (#define NAME_PATTERN "[a-zA-Z][a-zA-Z0-9]*"). The following function and methods takes in the above input and checks if the token is either a name or an integer:

- bool checkName(string token);
- bool checkInteger(string token);

The following functions and methods then aim to check for the expected symbols such as “{“ and “}” for the start and end of a statement list, “;” for the end of each statement line and “(“ and “)” for the start and end of a condition. Other symbols include the arithmetic and comparison operator signs.

- void expect(string symbol);
- bool match(string symbol);

Lastly, to iterate through the tokens, a list of strings “remainingTokens” is used to take in the vector of strings such that the list method pop.front() can be used to remove the head of the list until it is empty. This allows us to iterate through the entire list of tokens:

- void next();

The following will illustrate how the Source Parser works:

procedure	q	{	read	a	;	a	=	a	+	1	;	print	a	;

1. Call parseProcedure();
    a. Check current token if it is “procedure”

b. Call next();

q	{	read	a	;	a	=	a	+	1	;	print	a	;

1. Call parseVariable();
    a. Check current token if it is the correct NAME grammar
    b. If it is, insert into the database
    c. Call next();

{	read	a	;	a	=	a	+	1	;	print	a	;

1. Call expect();
    a. Check current token if it is the expected symbol “{“

read	a	;	a	=	a	+	1	;	print	a	;

1. Call parseStatement(); as it is the opening of the statement list
    a. A while loop is set here to check if it is the end of the statement list by check if the current token is a closing parenthesis (“}”)
    b. At the start of each statement line, the line number will be stored into the database
    c. It will proceed to check if the current token is any of the design entities, which is ‘read’ in this example
    d. The current line number will be stored in the database under the reads table
    e. Call next();

a	;	a	=	a	+	1	;	print	a	;

1. Call parseVariable()
    a. Check current token if it is the correct NAME grammar
    b. If it is, insert into the database
    c. Call next();
    d.

In iteration 3, the source parser, parseStatement() function is improved to include handling of multiple procedures, calls, calls*, modifiesproc, usesproc, Next and Next*.

- Multiple Procedures

  This is done by checking if the remaining tokens left to be parsed, after a procedure ends, is empty. If it is empty, the function exits and the program finishes, otherwise, continue on and check if the next token is a procedure. Each time a procedure is parsed, the procedure will be added into a vector of strings to maintain a vector of procedures (vector<string> procedureList;).

- Calls and Calls*

  Call is added into parseStatement() to check if there are any procedure calls within the statements. If so, the current procedure, extracted from the procedureList vector, will be inserted into the database together with the current token, which is the procedure name being called.

  A vector of vector strings (vector<vector<string>> procedureCalls;) is used to store all procedure calls. A temporary vector will be used to store the current procedure calls, checks if the first element exists in the procedureCalls vector, if it does, add the callee into the procedureCalls vector. It also checks if its callee in the temp vector exist in the procedureCalls. If it is, include the callee's callee into the temp vector and insert it into the procedureCalls. An illustration below:

Statements	Temp Vector
Procedure "first" call Procedure "second"	{first,second}

Procedure “first” call Procedure “second” Procedure “first” call Procedure ”fourth”	{first, second, fourth}
Procedure “second” call Procedure “third”	{second, third}
Procedure “fourth” call Procedure “fifth”	{fourth, fifth}

**temp vector is cleared at the end of each procedure before parsing the next procedure*

procedureCalls	Temp Vector / Remarks
Empty	{first,second, fourth}
first | second | fourth	{second, third}
first | second | fourth | third second | third	{fourth, fifth}
first | second | fourth | third | fifth second | third fourth | fifth	Empty

**each line represents 1 procedure cycle*

Lastly, iterate through the procedureCalls vector with the first element as the caller, and the remaining element as its callee.

- Modifies Procedure and Uses Procedure

  Both direct modifies and uses for procedure are able to tap on the existing parsing for modifies and uses for statements. Instead of inserting the statementLine into the database, insert the procedure instead which can be called out from the procedureList vector.

For indirect modifies and uses for procedure, two get functions are introduced to extract all the direct modifies and uses variables from the modifies procedure and uses procedure table that correspond to the callee of the current procedure:

- void Database::getCallsTmodifies(vector<string>& results, string callee)
  "SELECT variableN FROM modifiesproc WHERE modifiesProc = '" + callee + "';"
- void Database::getCallsTuses(vector<string>& results, string callee)
  "SELECT variableN FROM usesproc WHERE usesProc = '" + callee + "';"

Add the resulting results vector as the callee of the current procedure.

- Next and Next*

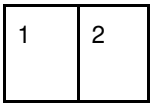
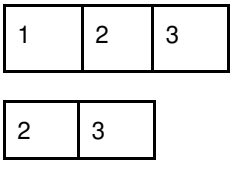
  Next and Next* is implemented similarly to Call and Calls* by using a vector of vector strings as well as a few temporary vectors for the checking of edge cases. A new table “procstmt” is created to store the corresponding procedurename to each statement line. This will allow us to check if the statement before or after is within the same procedure as Next and Next* only applies within its own procedure.

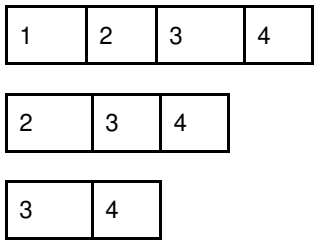
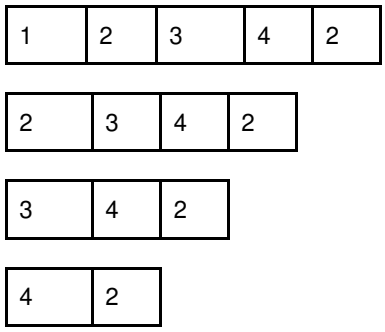
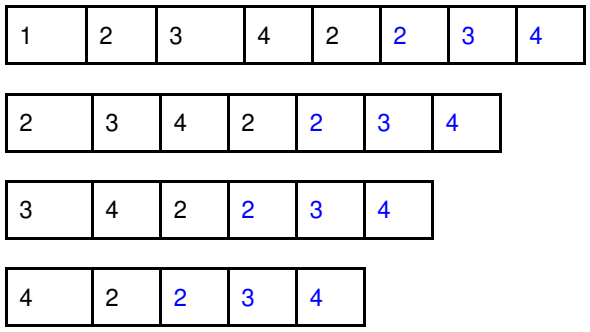
  Next is added into the database by checking if the previous statement line has the same procedure with the current statement line. If it is, both lines will be added into the database “next(prevLine,currentLine)”. However, when a While or If statement is detected, a check is required to determine if the current statement line is at the last line of the While statement, the last line of the If-Then statement or the last line of the If-Else statement. If it is, then the following actions will occur:

Scenario	Actions
Detects last line of While statement	Direct the last statement line to the statement line of the While statement
Detects last line of If-Then statement	Direct the statement line of the If statement to the current line (first line

	of the else statement) Store the last If-Then line statement
Detects last line of If-Else statement	Direct the last If-Then line to the current line
Detects last line of If-Then statement nested in a While loop	Direct the statement line of the If statement to the current line (first line of the else statement) Store the last If-Then line statement
Detects last line of If-Else statement nested in a While loop	Direct the last If-Then line to the statement line of the While statement Direct the last If-Else line to the statement line of the While statement

For Next*, a vector of vector strings “nextst” is used to store all the next* relation. The temp vector will collect the next relationship for each cycle of a statement line, the temp vector will be used to compare against the nextst vector and check if the first element in the temp vector exists in nextst. If it is, the next element will be added into each row of the nextst vector. This process processes the temp vector in pairs, until the temp vector is empty before going through any statement cycle with temp values in the temp vector:

Vector of vector strings	Temp Vector / Remarks
Empty	{1,2,2,3}
	{2, 3}
	{3, 4}

	{4,2}
	Empty
	The vector detects a while loop when the right element is less than the left element. This means that the element between these two numbers can be accessed again due to the loop. Therefore, they are added back into the vector.

2.5.2 Program Design Abstractions

The following illustrates how the relationships are stored in the database:

- Parents Table

```
procedure q {

1     read x;
2     read y;
3     read h;
4     print n;

5     while (y < 5) {
6           print y;
7           if (y > 1) then {
8                 print g;
9                 read f;
            } else {
10                print k;
            }
      }

11    read z;
12    x = 2 + 3;
13    print a;
14    print b;

15    if (x > 1) then {
16          read d;
      } else {
17          read s;
      }

18    read m;

}
```

parentLine	childLine
Filter	Filter
5	6
5	7
7	8
7	9
7	10
15	16
15	17

For the Parents Table, for any statement s1 and s2, Parent(s1, s2) holds if s2 is directly nested in s1. Line 5 is a statement with line 6 and line 7 directly nested in line 5.

- Parents* Table

```
procedure q {

1     read x;
2     read y;
3     read h;
4     print n;

5     while (y < 5) {
6           print y;
7           if (y > 1) then {
8                 print g;
9                 read f;
            } else {
10                print k;
            }
      }

11    read z;
12    x = 2 + 3;
13    print a;
14    print b;

15    if (x > 1) then {
16          read d;
      } else {
17          read s;
      }

18    read m;

}
```

grandparentLine	childLine
Filter	Filter
5	8
5	9
5	10

For the Parents* Table, for any statements s1 and s2, Parents*(s1, s2) holds if Parent(s1, s2) OR Parent(s1, s) and Parents*(s, s2). This can be seen from line 5 with line 8, line 9 and line 10 indirectly nested in line 5.

- Modifies Table

```
procedure q {

1     read x;
2     read y;
3     read h;
4     print n;

5     while (y < 5) {
6           print y;
7           if (y > 1) then {
8                 print g;
9                 read f;
            } else {
10                print k;
            }
      }

11    read z;
12    x = 2 + 3;
13    print a;
14    print b;

15    if (x > 1) then {
16          read d;
      } else {
17          read s;
      }

18    read m;

}
```

modifiesLine	variableN
Filter	Filter
1	x
2	y
3	h
9	f
7	f
5	f
11	z
12	x
16	d
15	d
17	s
15	s
18	m

For the Modifies Table, Modifies(a, v) holds if variable v appears on the left hand side of an assignment statement. This can be seen in line 12, which stores the variable ‘x’ with it in the database. Modifies(r, v) holds if a variable v appears in r which is a read statement. This can be seen in line 1 which stores the variable ‘x’ with it in the database. Modifies(s, v) holds if there is a statement s1 in the container such that the Modifies(s1, v) holds. This can be seen in line 9, which stores the variable ‘f’ along with it in the database.

- Uses Table

```
procedure q {

1     read x;
2     read y;
3     read h;
4     print n;

5     while (y < 5) {
6           print y;
7           if (y > 1) then {
8                 print g;
9                 read f;
            } else {
10                print k;
            }
      }

11    read z;
12    x = 2 + 3;
13    print a;
14    print b;

15    if (x > 1) then {
16          read d;
      } else {
17          read s;
      }

18    read m;

}
```

usesLine	variableN
Filter	Filter
4	n
6	y
8	g
7	g
5	g
10	k
7	k
5	k
7	y
5	y
13	a
14	b
15	x

For the Uses Table, Uses(a, v) holds if variable v appears on the right hand side of an assignment statement. This can be seen in line 12, where there is no variable on the right hand side of the assignment statement, hence nothing was added into the database. Uses(p, v) holds if a variable v appears in p which is a print statement. This can be seen in line 4 which stores the variable ‘n’ with it in the database. Uses(s, v) holds if v appears in the condition of s, or there is a statement s1 in the container such that Uses(s1,v) holds. This can be seen in line 5 where the variable ‘y’ appears in the condition statement and is stored into the database. Line 5 also stores the variable ‘g’ as print is nested in if which is nested in line 5.

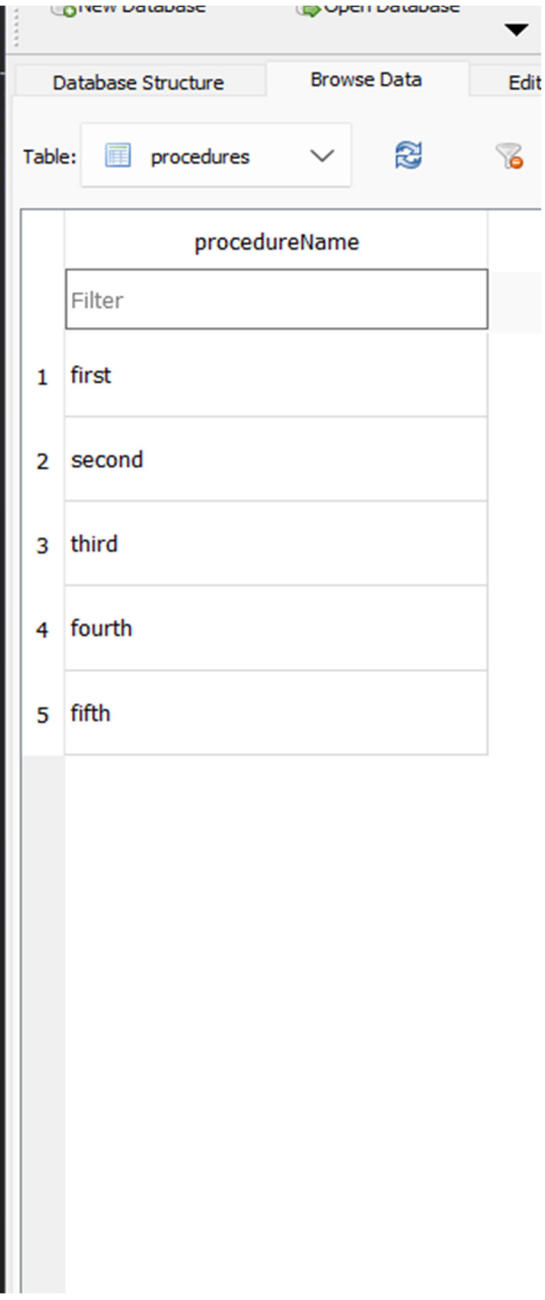
- Procedure Table

```
procedure first {
      z = (2-3+4)*(5+6*7);
      while (g < 8) {
            print k;
            read j;
            if (x > 1) then {
                  read d;
                  read t;
            } else {
                  read s;
                  print y;
            }
      }
      call second;
      call fourth;
}

procedure second {
      read b;
      print y;
      call third;
      while (g < 8) {
      print k;
      read j;
      }
      call third;
}
procedure third {
      print t;
      read c;
      a = (r+4)/(e-7);
}

procedure fourth {
      print h;
      call fifth;
}

procedure fifth {
      read y;
}
```



Procedure table contains the name of the procedure in the source code.

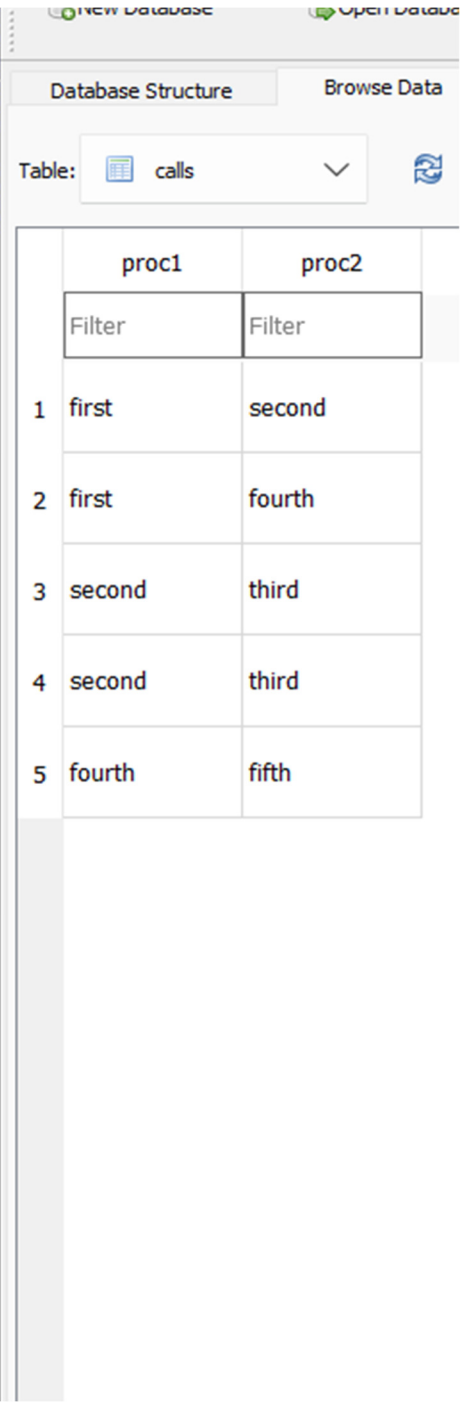
- Calls Table

```
procedure first {
      z = (2-3+4)*(5+6*7);
      while (g < 8) {
            print k;
            read j;
            if (x > 1) then {
                  read d;
                  read t;
            } else {
                  read s;
                  print y;
            }
      }
      call second;
      call fourth;
}

procedure second {
      read b;
      print y;
      call third;
      while (g < 8) {
      print k;
      read j;
      }
      call third;
}
procedure third {
      print t;
      read c;
      a = (r+4)/(e-7);
}

procedure fourth {
      print h;
      call fifth;
}

procedure fifth {
      read y;
}
```



Calls table contains 2 fields with procedure 1 as the caller and procedure 2 as the callee for direct procedure calls.

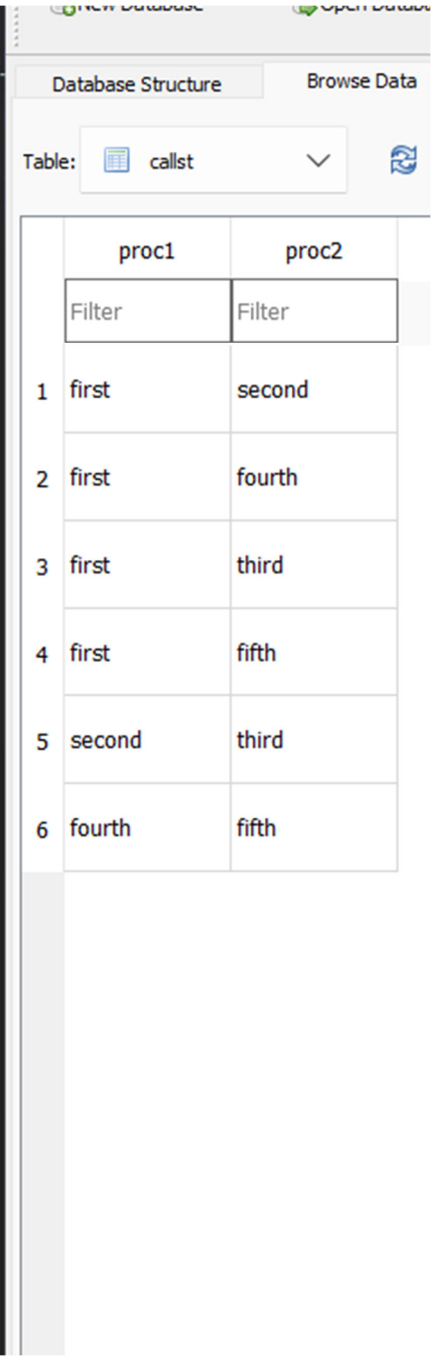
- Calls* Table

```
procedure first {
      z = (2-3+4)*(5+6*7);
      while (g < 8) {
            print k;
            read j;
            if (x > 1) then {
                  read d;
                  read t;
            } else {
                  read s;
                  print y;
            }
      }
      call second;
      call fourth;
}

procedure second {
      read b;
      print y;
      call third;
      while (g < 8) {
      print k;
      read j;
      }
      call third;
}
procedure third {
      print t;
      read c;
      a = (r+4)/(e-7);
}

procedure fourth {
      print h;
      call fifth;
}

procedure fifth {
      read y;
}
```



	proc1	proc2
	Filter	Filter
1	first	second
2	first	fourth
3	first	third
4	first	fifth
5	second	third
6	fourth	fifth

Calls table contains 2 fields with procedure 1 as the caller and procedure 2 as the callee for indirect procedure calls.

- Modifies for Procedure

```
procedure first {
	z = (2-3+4)*(5+6*7);
	while (g < 8) {
		print k;
		read j;
		if (x > 1) then {
			read d;
			read t;
		} else {
			read s;
			print y;
		}
	}
	call second;
	call fourth;
}

procedure second {
	read b;
	print y;
	call third;
	while (g < 8) {
	print k;
	read j;
	}
	call third;
}
procedure third {
	print t;
	read c;
	a = (r+4)/(e-7);
}

procedure fourth {
	print h;
	call fifth;
}

procedure fifth {
	read y;
}
```

Database Structure | Browse Data

Table: modifiesproc

	modifiesProc	variableN
	Filter	Filter
1	first	z
2	first	j
3	first	d
4	first	t
5	first	s
6	second	b
7	second	j
8	third	c
9	third	a
10	fifth	y
11	first	b
12	first	a
13	first	c
14	first	y
15	second	a
16	second	c
17	fourth	y

- Uses for Procedure

```
procedure first {
	z = (2-3+4)*(5+6*7);
	while (g < 8) {
		print k;
		read j;
		if (x > 1) then {
			read d;
			read t;
		} else {
			read s;
			print y;
		}
	}
	call second;
	call fourth;
}

procedure second {
	read b;
	print y;
	call third;
	while (g < 8) {
	print k;
	read j;
	}
	call third;
}
procedure third {
	print t;
	read c;
	a = (r+4)/(e-7);
}

procedure fourth {
	print h;
	call fifth;
}

procedure fifth {
	read y;
}
```

Database Structure | Brows

Table: usesproc

	usesProc	variableN
	Filter	Filter
1	first	k
2	first	y
3	first	x
4	first	g
5	second	y
6	second	k
7	second	g
8	third	t
9	third	r
10	third	e
11	fourth	h
12	first	h
13	first	e
14	first	r
15	first	t
16	second	e
17	second	r
18	second	t

- Next Table

```
procedure first {
	z = (2-3+4)*(5+6*7);
	while (g < 8) {
		print k;
		read j;
		if (x > 1) then {
			read d;
			read t;
		} else {
			read s;
			print y;
		}
	}
	call second;
	call fourth;
}

procedure second {
	read b;
	print y;
	call third;
	while (g < 8) {
	print k;
	read j;
	}
	call third;
}
procedure third {
	print t;
	read c;
	a = (r+4)/(e-7);
}

procedure fourth {
	print h;
	call fifth;
}

procedure fifth {
	read y;
}
```

Database Structure | Browse Data

Table: nexts

	stmt1	stmt2
	Filter	Filter
1	1	2
2	2	3
3	3	4
4	4	5
5	5	6
6	6	7
7	5	8
8	8	9
9	7	2
10	9	2
11	2	10
12	10	11
13	12	13
14	13	14
15	14	15
16	15	16
17	16	17
18	17	15
19	15	18
20	19	20
21	20	21
22	22	23

2.6. Query Processor

The Query Processor component is broken into four classes: Query Parser, Query Processor, Query Plan and Query Table. As mentioned, we broke Query Processor into smaller sub-components for better scalability and to adhere to the Single Responsibility Principle.

2.6.1. Query Parser

The Query Parser takes in the vector of string tokens from the Tokenizer class, which would then be parsed. As the query string itself is made up of multiple sections or clauses, we have separate methods to parse the different sections:

- parseDeclarationList(string tokens)
- parseSelectClause(string tokens)
- parseSuchThatClause(string tokens)
- parsePatternClause(string tokens)

We know from the grammar syntax that a query string would always start with a declaration list. Hence, the QueryParser parse() method would first call parseDeclarationList() to parse the declaration of synonyms. While parsing the declaration list, we will keep the information of the synonyms and their related design entities in a map. This would be used later when parsing the Select clause, to ensure that the synonyms used in the Select clause are present in the declaration list. Else, the query would be semantically invalid. In iteration 2, we added a while loop to parse through the declaration of synonyms to account for multiple declarations.

Next, the parse() method would call the parseSelectClause() method to parse the Select clause. While parsing the Select clause, we will add the information of the clause into a class called Query, which will be used later by the Query Processor to evaluate the query.

To better illustrate how the Query Parser works, we will look at an example query:

`assign a, variable v; Select <a, v> pattern a (v, “x + y”)`

1. The list of tokens retrieved from the Tokeniser class would look like below. We will first parse the declaration list. We know that the first part of the declaration list is the design entity. So, we will pop the token and move on to process the next token.

assign	a	,	variable	v	;	Select	<	a	,	v	>	pattern	a	(	v	,	“	x	+	y	“	)

2. After popping the design entity, we move on to the next token in the string, the synonym of the design entity. The synonym will be stored into a map which consists of a synonym-design entity, key-value pair. In this case, the token is valid and will pop the synonym ‘a’.

a	,	variable	v	;	Select	<	a	,	v	>	pattern	a	(	v	,	“	x	+	y	“	)

3. After the synonym, we know that there can either be (1) more synonyms or (2) the declaration list has ended. Since we detect a comma, we know that there are more synonyms so we would expect the same thing as before - a design entity. After parsing this synonym, we would then expect a semicolon.

,	variable	v	;	Select	<	a	,	v	>	pattern	a	(	v	,	“	x	+	y	“	)

4. The next section in the query would be the Select clause. The first token in the Select clause is a “<”, so we know that we need to expect a tuple. Hence, we would then check for a synonym. After getting a synonym, we would check for a comma, followed by another synonym. After parsing the second synonym, we would expect a comma yet again, but we are greeted with a “>”, signifying the end of the tuple and Select clause. Before exiting the parseSelectClause() method, the synonym and its design entity will be pushed into the Select clause vector that is stored in the Query object, to be used by the Query Processor.

Select	<	a	,	v	>	pattern	a	(	v	,	“	x	+	y	“	)

5. After parsing the Select Clause, the method would then call parseSuchThatClause() or parsePatternClause() to parse the Such That Clause and Pattern Clause respectively. In this case, the query has a Pattern Clause. Then, the parsePatternClause() will first expect to see one token - “pattern”.

pattern	a	(	v	,	“	x	+	y	“	)

6. We know from the syntax grammar that the synonym comes after, hence, we add the “a” synonym to the PatternClause struct stored in the Query object.

a	(	v	,	“	x	+	y	“	)

7. After expecting and popping the opening bracket, we should expect to see the first argument of the Pattern clause. In this case, the argument is “v”. We will also add the first argument to our PatternClause struct.

(	v	,	“	x	+	y	“	)

8. Then, we pop the comma and start parsing the second argument. We know from the grammar syntax that it can either be a _, “<expression>”, or _”<expression>”_, where <expression> is an expression specification. Since we see an open inverted comma, we know that this is an exact match expression specification. Then, we will parse the expression using the shunting yard algorithm and store it as a postfix expression in the PatternClause struct. After that, we should also expect to see the close inverted comma.

,	“	x	+	y	“	)

2.6.2. Query Processor

To evaluate such a query, we would be splitting the query into its different clauses and evaluating them individually.

To better illustrate how the Query Processor works, we will look at an example query:

assign a, variable v; Select <a, v> such that Modifies(a, v) pattern a (v, “x + y”)

First, we evaluate the Select clauses. To evaluate the clauses, we would take a look at the information provided in the Select clause struct, which is stored in the Query class as “designEntity”. Then, in the Query Processor, we will check the Select clause’s design entity and compare it to the list of valid design entities (i.e. “assignment”, “procedure”, etc.) that are stored in the database. If the design entity matches any of the given design entities, data for that corresponding design entity type will be pulled from the database. The Query Processor then checks for duplicate output from the results in the database. If there is, we would remove the duplicate output.

Next, we evaluate the Such That clause. In this case, the Such That clause has a Modifies relationship reference, hence, ModifiesClauseEvaluator class would be created and its evaluate() method would be called. In the evaluate() method, we would check the argument types of the clause, which would determine which SQL statement will be run by the Database. For the above example, the first argument is a synonym while the second argument is an IDENT. Hence, we will run the following query: "`SELECT * FROM modifies WHERE variableN = “x” AND modifiesLine IN (SELECT * FROM assignment);`". The query will return a list of assignment statements that are modifying a variable “x” (i.e. “x” on the left hand side of the assignment). The results will be stored in a vector of vector strings where each vector is a row in the result, and added to the QueryTable.

Then, we evaluate the Pattern Clause. In this case, we would retrieve the list of assignment statements from the database and check each row whether the right-hand

side equals “xy+” (i.e. the postfix expression of “x+y”). If it does equal, then we would add the row to the vector of results. The results would then be added to the QueryTable.

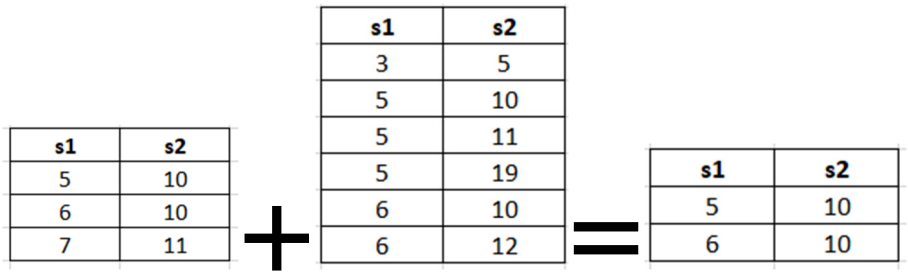
Finally, to filter the results from the Select, Such That, and Pattern clauses based on its common synonyms, we would rely on QueryTable operations to do so.

2.6.4. Query Table

In Iteration 3, we have set up a Query Table class in order to handle the different operations required by PQL such as inner join, insert column, cross product, etc. We decided to change our approach as our original Query Processor could only handle two Clauses (1 Such That and 1 Pattern) with limited number of Such That combinations (Modifies + Uses, Parent* + Modifies, etc). This approach will help counter that limitation.

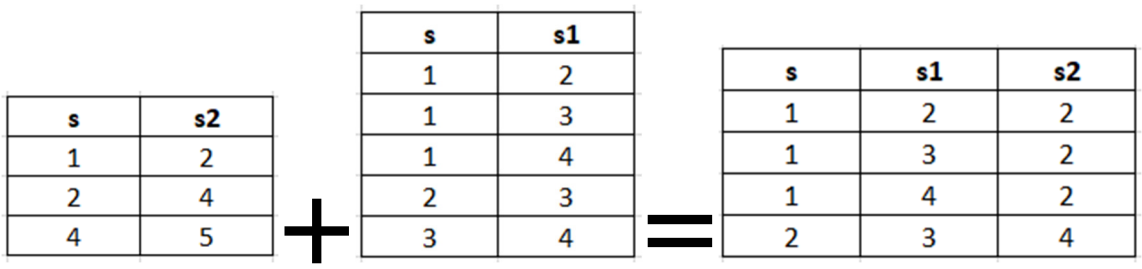
Join

Just like SQL, (inner) join occurs when the query table contains every column header that the incoming data has. The current table and the incoming table data will be compared, and rows that are not present in both tables will be subsequently deleted.



Insert

Insert will occur when the table has at least one identical column, but not all column headers that the incoming data has. A new table will be created based on the shared columns.



s	s2
1	2
2	4
4	5

+

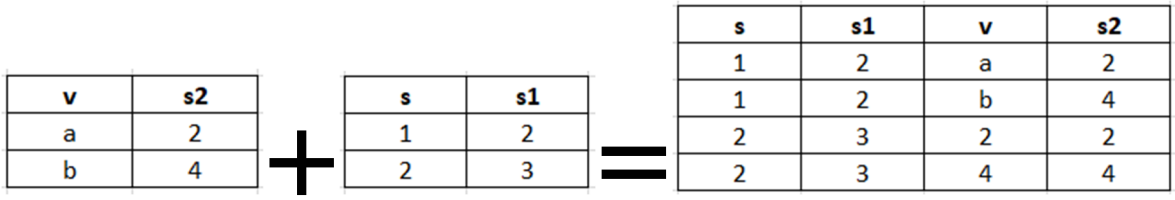
s	s1
1	2
1	3
1	4
2	3
3	4

=

s	s1	s2
1	2	2
1	3	2
1	4	2
2	3	4

Cross Product

Cross Product will occur when the table and incoming data do not have any shared column headers. A new table will be created by doing a cartesian product between the current table and incoming data table.



v	s2
a	2
b	4

+

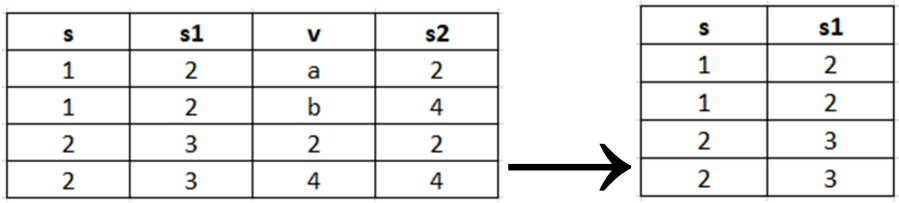
s	s1
1	2
2	3

=

s	s1	v	s2
1	2	a	2
1	2	b	4
2	3	2	2
2	3	4	4

Drop

Drop will occur when the QueryEvalutor needs to drop specific column(s) based on the Select clause synonyms.



s	s1	v	s2
1	2	a	2
1	2	b	4
2	3	2	2
2	3	4	4

→

s	s1
1	2
1	2
2	3
2	3

3. Testing

In Iteration 1, the team has only done system testing for Select clauses to validate the program according to the SPA requirements. In Iteration 2, we have done testing for Select, Such That and Pattern clauses to validate the program according to the SPA requirements. We have also tested a combination of Such That and Pattern clauses i.e. Modifies and Pattern, Uses and Pattern, etc.

3.1. System Testing

There is two main aspects when it comes to writing system test cases:

1. Designing source programs
2. Designing queries

When writing the test source program, the team made sure that all of the design entities are accounted for — procedure, statement, print, assignment, read, variables, and constants. For the variables, we included alphanumeric and alphabetical names, which are supposed to be allowed as per the grammar syntax. The source program also included multiple whitespaces, since multiple whitespaces are allowed as per the grammar syntax and should be handled properly by the Tokenizer. Similarly, when writing the test query, the team made sure to test all the different design entities. An example of the system test cases created by our team can be seen below.

Source program - Iteration 1

```
procedure lambda {
	num1 = 3000;
	num2 =    2;
	read r;
	read   a1;
	read num9;

	num3 = 9999;
	print p;
	print num;
	print a1  ;

}
```

Query - Iteration 1

```
1 - Procedure
procedure p;
Select p;
lambda
5000
2 - Variable
variable v;
Select v
num1, num2, r, a1, num9, num3, p, num
5000
3 - Constant
constant c;
Select c
3000, 2, 9999
5000
4 - Assignment
assign a;
Select a
1, 2, 6
5000
5 - Print
print pr;
Select pr
7, 8, 9
5000
6 - Read
read rd;
Select rd
3, 4, 5
5000
7 - Statement
stmt s;
Select s
1, 2, 3, 4, 5, 6, 7, 8, 9
```

```
5000
```

Source program 1 - Iteration 2

```
procedure q {

      read x;
      read y;
      read h;
      print n;

      while (y < q) {
            print y;
            if (y > 1) then {
                  print g;
                  read f;
            } else {
                  print k;
            }
      }

      read z;
      x = r + 2;
      print a;
      print b;

      if (x > 1) then {
            read d;
      } else {
            read s;
      }

      read m;

}
```

Query program 1 - Iteration 2

```
1 - Gets all statements
stmt s;
Select s
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18
5000
2 - Gets all read statements
```

```
read r;
Select r
1, 2, 3, 9, 11, 16, 17, 18
5000
3 - Gets all print statements
print p;
Select p
4, 6, 8, 10, 13, 14
5000
4 - Gets all while statements
while w;
Select w
5
5000
5 - Gets all procedures
procedure p;
Select p
q
5000
6 - Gets all variables
variable v;
Select v
x, y, h, n, g, f, k, q, z, a, b, d, s, m, r
5000
7 - Select all read statement that Modifies something
variable v; read r;
Select r such that Modifies(r, _)
1, 2, 3, 9, 11, 16, 17, 18
5000
8 - Select read statements that Modifies a certain variable
variable v1, v; read r;
Select r such that Modifies(r, "y")
2
5000
9 - Select all while statements that Modifies something
while w; read r; variable v;
Select w such that Modifies(w, v)
5
5000
10 - Select all children of while statements
while w; stmt s;
Select s such that Parent(5, s)
6, 7
5000
11 - Select all children and grandchildren of while statements
while w; stmt s;
Select s such that Parent*(5, s)
6, 7, 8, 9, 10
5000
```

```
12 - Select all variables being Modified by a Read
variable v; read r;
Select v such that Modifies(r, v)
x, y, h, f, z, d, s, m
5000
13 - Select all assignment statements that has r on RHS
assign a;
Select a pattern a(_, _"r"_)
12
5000
14 - Select all assignment statements that has x on LHS
assign a;
Select a pattern a("x", _)
12
5000
```

Source program 2 - Iteration 2

```
procedure aran {
      x = y;
      a = b + 1;
      print k;
      while (x < a) {
            print a;
            while (a < 1) {
                  a = d + c;
                  print c;
                  read b;
            }
            if (y > a) then{
                  read y;
                  print k;
            } else {
                  print a;
            }
      }
      print s;
      s = s + y;
      read y;
}
```

Query program 2 - Iteration 2

```
1 - Select all print statement that Uses something
variable v; print r;
Select r such that Uses(r, v)
3, 5, 8, 12, 13, 14
5000
2 - Select specific print statement that Uses something
variable v; print r;
Select r such that Uses(3, r)
3
5000
3 - Select specific assignment statement that Uses an ident
variable v1, v; assign r;
Select r such that Uses(r, "b")
2
5000
4 - Select variables being Used in a specific assignment statement
variable v1, v; assign r;
Select v1 such that Uses(7, v1)
d, c
5000
5 - Select variables being Used in a in all while loops
while w; variable v;
Select v such that Uses(w, v)
x, a, c, d, y, k
5000
6 - Select variable being Used in a specific if
if ifs; variable v;
Select v such that Uses(10, v)
y, k, a
5000
7 - Select all reads if there is a assignment statement in a while
read r; while w; assign a;
Select r such that Parent(w, a)
9, 11, 16
5000
8 - Select all reads if line 7 is a descendent of line 4
read r; while w; assign a;
Select r such that Parent*(4, 7)
9, 11, 16
5000
9 - Select all assignments if line 7 is a child of line 4 (meaningless
query)
read r; while w; assign a;
Select r such that Parent(4, 7)

5000
10 - Select all descendents of line 4
```

```
while w; stmt s;
Select s such that Parent*(4, s)
5, 6, 7, 8, 9, 10, 11, 12, 13
5000
```

Source program 3 - Iteration 2

```
procedure melange {
      if (a > 10) then {
            read c;
            d = b;
            while (f > 5) {
                  print g;
                  read h;
                  i = a + 1;
                  a = 1 + b;
            }
            if (l > f) then {
                  m = a + 1;
            } else {
                  read p;
            }
      } else {
            while (r < s) {
                  if (s > t) then {
                        t = b + 1;
                  } else {
                        print y;
                  }
            }
            print y;
      }
      a = a;
}
```

Query program 3 - Iteration 2

```
1 query
assign a; variable v;
Select a such that Uses(a, v) pattern a (v, _)
17
5000
2 query
```

```
assign a; variable v;
Select a such that Uses(a, v) pattern a (_, _"b"_)
3, 8, 14
5000
3 query
assign a; variable v;
Select a such that Uses(a, v) pattern a ("a", _"b"_)
8
5000
4 query
assign a; variable v; while w;
Select a such that Parent(w, a) pattern a (_, _)
7, 8
5000
5 query
assign a; variable v; while w;
Select a such that Parent(w, a) pattern a ("a", _)
8
5000
6 query
assign a; variable v; while w;
Select a such that Parent(w, a) pattern a ("a", _"b"_)
8
5000
7 query
assign a; variable v; while w;
Select a such that Parent*(w, a) pattern a (v, _"b"_)
8, 14
5000
8 query
assign a; variable v; while w;
Select a such that Parent*(w, a) pattern a (_, _"b"_)
8, 14
5000
9 query
assign a; variable v; while w;
Select a such that Parent*(w, a) pattern a ("t", _"b"_)
14
5000
10 query
assign a; variable v; while w;
Select a such that Modifies(a, v) pattern a ("a", _"b"_)
8
5000
11 query
```

```
assign a; variable v; while w;
Select a such that Modifies(a, v) pattern a (v, _"b"_)
3, 8, 14
5000
12 query
assign a; variable v; while w;
Select a such that Modifies(a, v) pattern a (_, _"b"_)
3, 8, 14
5000
```

Source program 1 - Iteration 3

```
procedure satu {
	print mula;
	call tiga;
}

procedure dua {
	print a;
	print b;
	print c;
	print mula;
}

procedure tiga {
	x = a;
	y = b;
	z = c;

	if (x < y) then {
		mula = 0;
	} else {
		mula = 1;
	}
	while(y > 0){
		z = (x+y)*x;
		y = y % x;
		while (mula < 100) {
			y = y-x;
			x = z;
			while ((z + 1) > (y + 2)) {
				z = z % 2;
			}
			mula = 1;
		}
	}
```

```
      call dua;
}


procedure fifth {
      read y;
}
```

Query program 1 - Iteration 3

```
1 - simple uses p
procedure p;
Select p such that Uses(p, "a")
tiga, dua
5000
2 - simple uses p
variable v;
Select v such that Uses("tiga", v)
x, y, z, a, b, c, mula
5000
3 - uses p multiple clause
variable v; while w; assign a;
Select v such that Uses("tiga", v) such that Parent(w, a) pattern a (v, _)
y, z
5000
4 - tuple uses p
variable v; print pr;
Select <pr, v> such that Uses("dua", v)
3 a, 3 b, 3 c, 3 mula, 4 a, 4 b, 4 c, 4 mula, 5 a, 5 b, 5 c, 5 mula, 6 a, 6
b, 6 c, 6 mula
5000
5 - tuple uses p multiple clause
variable v; while w; assign a; if ifs;
Select <ifs, v> such that Uses("tiga", v) such that Parent(w, a) pattern a
(v, _)
10 y, 10 z
5000
6 - simple modifies p
procedure p;
Select p such that Modifies(p, "z")
tiga
5000
7 - simple modifies p
procedure p;
Select p such that Modifies(p, "a")

5000
8 - modifies p multiple clause
```

```
variable v; while w; assign a;
Select v such that Modifies("tiga", v) such that Parent(w, a) pattern a (v,
_)
x, y, z
5000
9 - tuple modifies p
variable v; print pr;
Select <pr, v> such that Modifies("tiga", v)
3 x, 3 y, 3 z, 3 mula, 4 x, 4 y, 4 z, 4 mula, 5 x, 5 y, 5 z, 5 mula, 6 x, 6
y, 6 z, 6 mula
5000
10 - tuple modifies p multiple clause
variable v; while w; assign a; if ifs;
Select <ifs, v> such that Modifies("tiga", v) such that Parent(w, a)
pattern a (v, _)
10 x, 10 y, 10 z
5000
```

Source program 2 - Iteration 3

```
procedure main {
      call calculate;
}

procedure input {
      read num1;
      read num2;
}

procedure output {
      print num1;
      print num2;
}

procedure calculate {
      call input;
      x = num1;
      y = num2;
      while(y > 0){
            y = x % y;
      }
      final = (num1 * num2) / 500;
      call output;
}
```

Query program 2 - Iteration 3

```
1 - simple call
procedure p;
Select p such that Calls("calculate", p)
input, output
5000
2 - simple call
procedure p;
Select p such that Calls(p, "input")
calculate
5000
3 - simple call*
procedure p1, p2;
Select p2 such that Calls(p1, p2)
calculate, input, output
5000
4 - simple call*
procedure p;
Select p such that Calls*("main", p)
calculate, output, input
5000
5 - simple call*
procedure p;
Select p such that Calls*(p, "input")
main, calculate
5000
6 - tuple call*
procedure p1, p2, p3;
Select <p1, p2, p3> such that Calls*(p1, p2) such that Calls*(p2, p3)
main calculate input, main calculate output
5000
7 - simple next
assign a; while w;
Select a such that Next(w, a)
10, 11
5000
8 - tuple next
stmt s; while w
Select <w, s> such that Next(w, s)
9 10, 9 11, 9 12, 9 4, 9 5
5000
9 - next with pattern expression
assign a; while w
Select a such that Next(w, a) pattern a (_, _"num1 * num2"_)
11
5000
10 - tuple next with pattern expression
assign a; while w
Select a such that Next(w, a) pattern a (_, _"num1 * num2"_)
9 11
```

```
5000
```

Source program 3 - Iteration 3

```
procedure melange {
      if (a > 10) then {
            read c;
            d = b;
            while (f > 5) {
                  print g;
                  read h;
                  i = a + 1;
                  a = 1 + b;
            }
            if (l > f) then {
                  m = a + 1;
            } else {
                  read p;
            }
      } else {
            while (r < s) {
                  if (s > t) then {
                        t = b + 1;
                  } else {
                        print y;
                  }
            }
            print y;
      }
      a = a;
}
```

Query program 3 - Iteration 3

```
1 query
assign a; variable v;
Select a such that Uses(a, v) pattern a (v, _)
17
5000
2 query
assign a; variable v;
Select a such that Uses(a, v) pattern a (_, _"b"_)
3, 8, 14
```

```
5000
3 query
assign a; variable v;
Select a such that Uses(a, v) pattern a ("a", _"b"_)
8
5000
4 query
assign a; variable v; while w;
Select a such that Parent(w, a) pattern a (_, _)
7, 8
5000
5 query
assign a; variable v; while w;
Select a such that Parent(w, a) pattern a ("a", _)
8
5000
6 query
assign a; variable v; while w;
Select a such that Parent(w, a) pattern a ("a", _"b"_)
8
5000
7 query
assign a; variable v; while w;
Select a such that Parent*(w, a) pattern a (v, _"b"_)
8, 14
5000
8 query
assign a; variable v; while w;
Select a such that Parent*(w, a) pattern a (_, _"b"_)
8, 14
5000
9 query
assign a; variable v; while w;
Select a such that Parent*(w, a) pattern a ("t", _"b"_)
14
5000
10 query
assign a; variable v; while w;
Select a such that Modifies(a, v) pattern a ("a", _"b"_)
8
5000
11 query
assign a; variable v; while w;
Select a such that Modifies(a, v) pattern a (v, _"b"_)
3, 8, 14
```

5000
12 query
assign a; variable v; while w;
Select a such that Modifies(a, v) pattern a (_, _"b"_)
3, 8, 14
5000