CS3203 Software Engineering Project

AY21/22 Semester 1

Project Report

Team 9

Team Members	Student No.	Email
Ang Song Yi	A0197082X	e0392524@u.nus.edu
Chan Wa Wai	A0200190B	e0407171@u.nus.edu
Cheng Le Da, Clement	A0183909R	e0310704@u.nus.edu
Leong En Ze, Hope	A0199596Y	e0406577@u.nus.edu
Oh Jun Ming	A0197781L	e0398574@u.nus.edu
Wong Kok Ian	A0202776A	e0418183@u.nus.edu

Consultation Hour: Tuesday (16:00)

Tutor: Raivat Shah

Abstract

The Static Program Analyser (SPA) analyses a source program written in SIMPLE source language, and extracts and stores relevant design abstractions into a Program Knowledge Base (PKB). Users can then ask questions regarding the program in a Program Query Language (PQL) and receive their corresponding answers.

Optimisations

In order for the SPA to respond to queries quickly and correctly, there is a need for query processing and evaluation to be optimised effectively. Below are the main optimisations implemented in our final iteration of the project. The full list of implemented optimisations can be found in *Annex A: Complete List of Optimisations for SPA Program*.

1. Terminating early when encountering syntactic or semantic errors in source or query
2. Storing of two-way relationships in PKB storages
3. Grouping queries based on inter-connected synonyms
4. Implementing a “NO Cartesian-Product” synonym storage

Extensions

Our team will be implementing Extension B: General Case of `CFGBip` and `AffectsBip`. This version of the extension deals with repeated procedure calls and introduces 4 unique program design abstractions. A more detailed explanation of the extension and its implementation can be found at *Section 3*.

Part 1 – Technical report

1. SPA Design

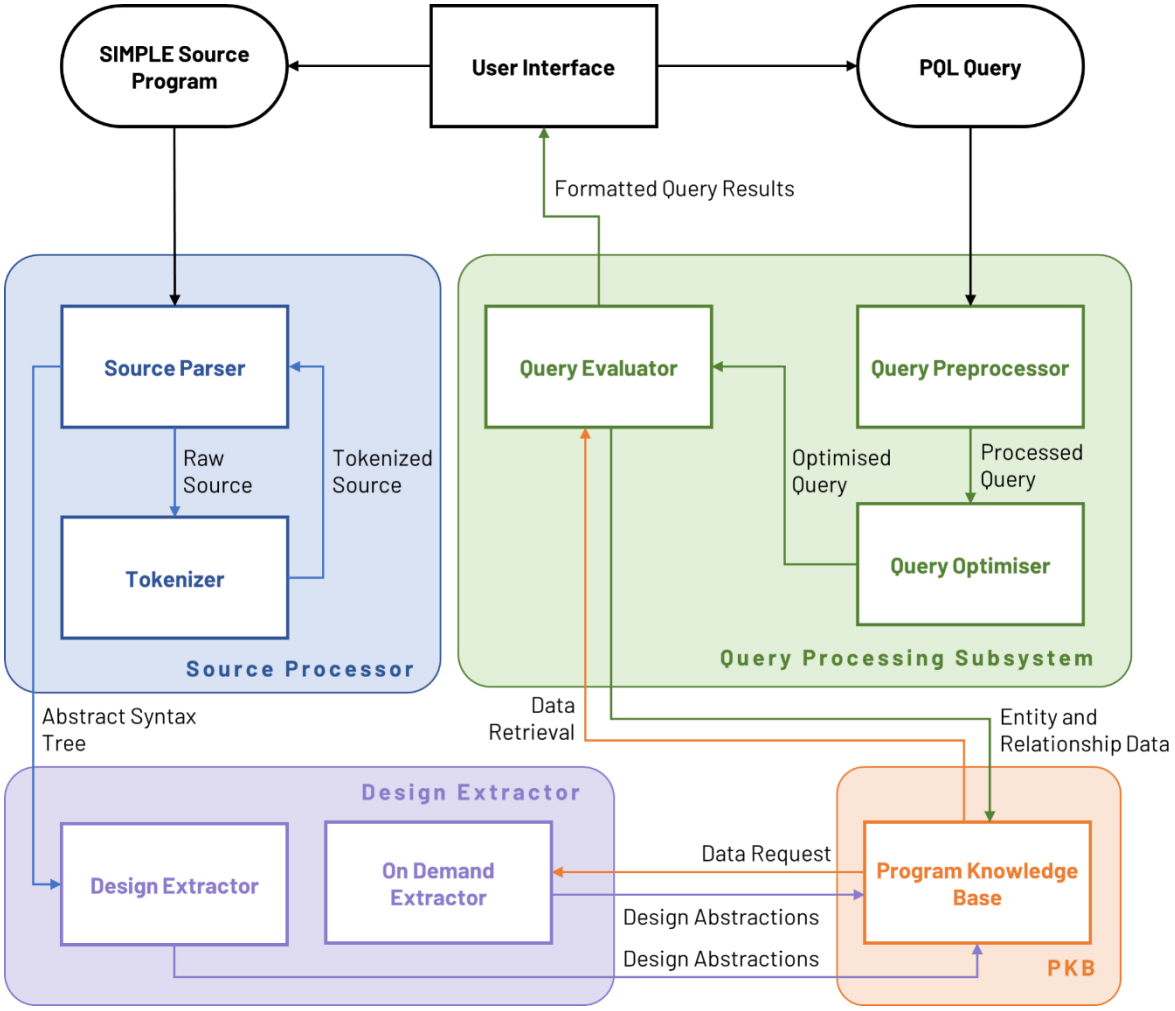


Figure 1: SPA Architecture Diagram

Figure 1 shows the main architecture diagram of our SPA. Our SPA can be categorised into **five** main components:

1. **User Interface (UI)** – Receives the SIMPLE source program and the set of PQL queries from the user, and displays the corresponding query results.
2. **Source Processor (SP)** – Parses and validates the given SIMPLE source program.
3. **Design Extractor (DE)** – Extracts the design entity and relationship information from the source program, and on demand during query evaluation.
4. **Program Knowledge Base (PKB)** – Stores the entity and relationship information extracted from the DE. Information is later retrieved by the QPS.
5. **Query Processing Subsystem (QPS)** – Parses, validates and processes the given PQL query. Query is then evaluated by utilising the data stored in the PKB and the formatted result is passed to the UI to display to the user.

Each of the five components perform their own specific roles and interact with each other only through API calls. As such, there is reduced coupling between the main components. It also provides a layer of abstraction to hide implementation details of the various components.

Each main component is further subdivided into sub-components, with each sub-component performing an even more specific task. This is important as the resulting sub-components can become more cohesive and we wanted to enforce the Single Responsibility Principle as much as possible to reduce potential coupling.

Excluding the UI, the detailed implementation and design decisions for the other four main components are documented in their respective subsections in the report.

1.1. Sample SIMPLE program

The sample SIMPLE program below is a function that finds and prints the maximum sum of the digits in a number from a list of numbers. It will be used as a running example to supplement our team's illustration of how the various components are being implemented.

```
      procedure FindMaxSumOfDigits {
 1.     ans = 0;
 2.     flag = 0;
 3.     count = 0;
 4.     error = 0 - 1;
 5.     while (flag == 0) {
 6.       read number;
 7.       print number;
 8.       if (number < 0) then {
 9.         flag = 1;
        }   else {
10.         count = count + 1;
11.         sum = 0;
12.         while (!(number == 0)) {
13.           sum = sum + (number % 10);
14.           number = number / 10;
          }
15.         if (sum > ans) then {
16.           ans = sum;
          }   else {
17.           ans = ans;
          }
        }
18.     print count;
      }
19.   call PrintAnswer;
    }
```

```
        procedure PrintAnswer {
20.       if ((!((flag == 0) || (count > 0))) && (number < 0)) then {
21.         print error;
          } else {
22.         print ans;
          }
        }

        procedure a {
23.       read a;
24.       print a;
25.       call b;
        }

        procedure b {
26.       read b;
27.       print cs3203;
28.       print fun;
        }

        procedure c {
29.       read c;
30.       print c;
31.       call a;
        }

        procedure d {
32.       read d;
33.       print d;
34.       call b;
35.       call a;
        }

        procedure e {
36.       read e;
37.       print e;
38.       call d;
        }

        procedure f {
39.       read f;
40.       print f;
41.       call a;
42.       call b;
43.       call c;
        }
```

Figure 2: Sample SIMPLE Program

1.2. Source Processor

The source processor component consists of two sub-components:

1. Tokenizer
2. Source Parser

1.2.1. Tokenizer

The tokenizer is responsible for validating lexical token rules. It will convert the raw source programs into a sequence of tokens with specific types for the parser to validate concrete syntax grammar. The tokenizer will keep consuming characters from the source program and convert them into `Token` and emplace them into a `queue<Token>` until the file stream is empty at which it will return the `queue`, in which the order of characters in the form of tokens is preserved with respect to the file stream.

A `Token` is a struct that contains value and `TOKEN_TYPE` which is an enumeration. The full list of enumerations can be referred in *Appendix A: Tokenizer Symbols to Type Mapping*.

```
Token = { value, Token Type }
```

The list below summarises the lexical token rules that the tokenizer validates.

1. `NAME: Letter (Letter | Digit)`
2. `INTEGER: Digit+` (with the first digit being a non-zero)
3. Logical symbols
4. Unknown symbol

The main purpose of the `Tokenizer` is to group symbols together and assign the type as stated above. The `Tokenizer` continuously appends alphanumeric characters together and validates rule 1 and 2 using `REGEX`. Afterwards, it will classify the string as either `VARIABLE` or `INTEGER`. All delimiters will be map to their value. For conditionals, the only accepted symbol directed after a conditional symbol is an equal symbol or a white space. For logical symbols, it must be followed by another same logical symbol. The tokenizer will ignore white spaces and any symbols that does not follow the rules mentioned, throwing an unexpected token error.

Given the source program mentioned in *Section 1.1*, tokenizing line 5-7 will result in the following sequence of tokens in the queue of tokens.

…
<WHILE, “while”>
<LEFTBRACKET, “(“>
<VARIABLE, “flag”>
<CONDITIONAL, “==”>
<CONSTANT, “0“>
<RIGHTBRACKET, “)”>
<LEFTBRACE, “{“>
<READ, “read”>
<VARIABLE, “number”>
<SEMICOLON, “;”>

<PRINT, “print”>
<VARIABLE, “number”>
<SEMICOLON, “;”>
…

The following activity diagram summarizes how the `Tokenizer` works.

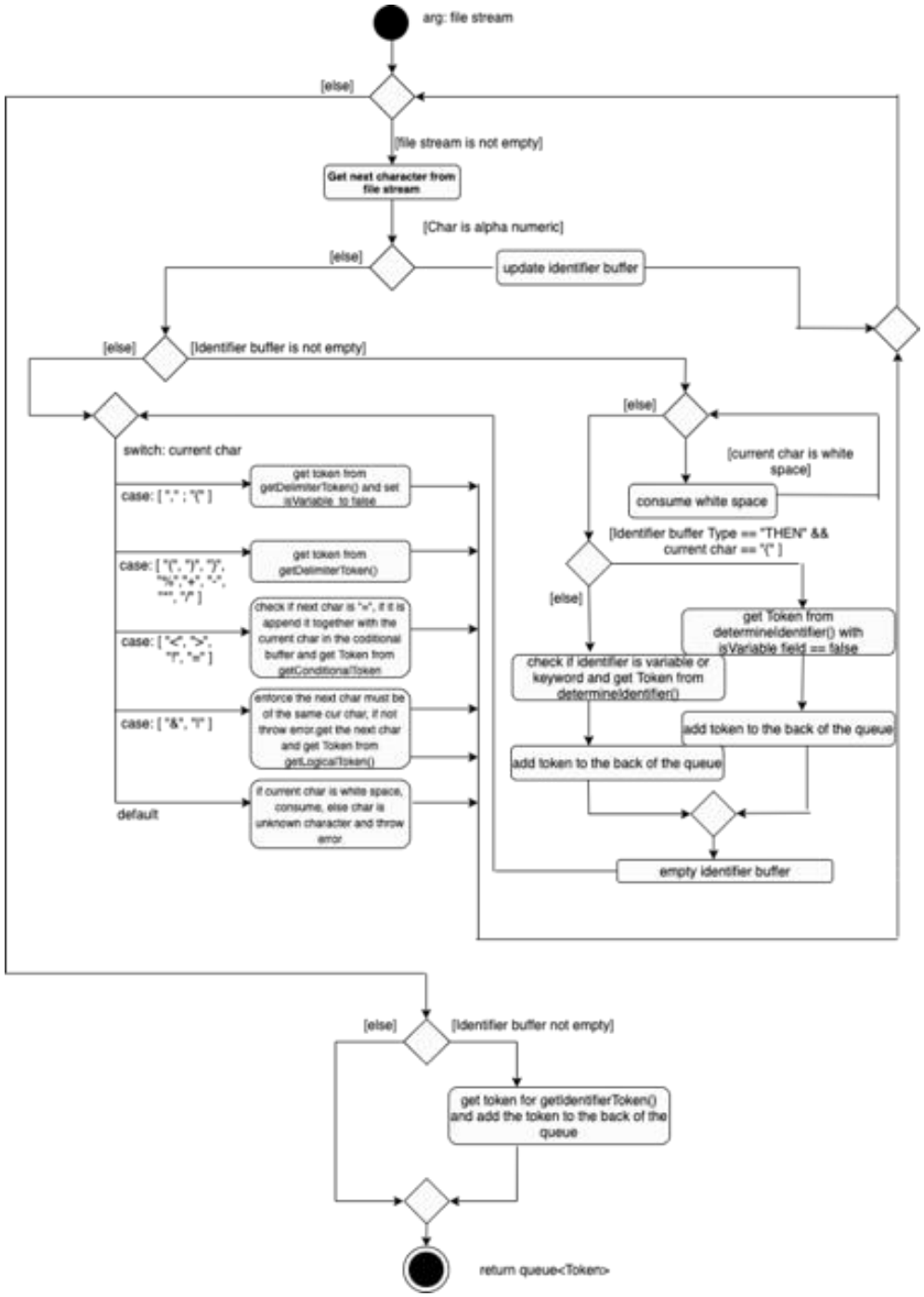


Figure 3: Activity diagram for Tokenizer

Given the source program mentioned in *Section 1.1*, tokenizing line 5-7 will result in the following sequence of tokens in the queue of tokens.

1.2.2. Source Parser

The source parser is responsible for the validating the program with the set of concrete grammar syntax plus almost all the additional rule that are not captured by the concrete grammar syntax. The source parser implements the recursive descent algorithm which makes use of an expect function and sub parser to validate the various grammar syntax. Throughout the parsing, the parser will also maintain and build 3 data structure: Abstract Syntax Tree, Procedure Call Matrix and Procedure List. The AST and Procedure Call Matrix is then passed to the Design Extractor for extraction of relationships. The details of each component are described below.

1. Abstract Syntax Tree

Our team decided to use the abstract syntax tree (AST) as choice of data representation of the SIMPLE program. The information stored in each node can be referred in Section 10.

Section 10 contains the AST diagram that will be produced after parsing the Sample Simple Program in *Section 1.1*. Due to the size of the AST, we will show the root node and its immediate child and the sub-AST for procedure `FindMaxOfSum` and the value of the relevant attributes. The details of the value of each AST Node Type and the value they encapsulate will be describe in the parser component. All relationship is extracted from the structure of the AST by the Design Extractor, we do not store any relationship information in the AST.

2. Procedure Call Matrix

It is an adjacency matrix that keeps tracks of the procedure that has call statement and the various procedure that they have called. The row in the matrix represents the Caller while the column of the matrix represents the Callee. The purpose of the call matrix will be explained with more details in the Design Extractor section.

Referring to Figure 2, the source parser will maintain and build the following Call Matrix.

FindMaxSumOfDigits	{ printAns }
c	{ a }
e	{ d }
a	{ b }
d	{ b ,a }
f	{ a, b , c}

3. Procedure List

It is a list that keeps track of the different procedures that the parser has parsed at various point in time. At the end of the parsing, it is a list of all procedures declared in the program.

Referring to Figure 2, when the source parser is parsing procedure c, the procedure list at that point of time is as follows:

<FindMaxSumOfDigits, PrintAnswer, a, b ,c>

At the end of the parsing, the complete procedure list will be as follows:

<FindMaxSumOfDigits, PrintAnswer, a, b, c, d, e, f>

These data structures will help to validate the additional rules that are not captured by the syntax grammar rule.

Recursive Descent

The algorithm makes use of an `expects(Token t)` function and sub parser to parse the different grammar structures. With the data structure and these sub parsers, we can validate all the concrete syntax grammar and additional rules. Refer to *Appendix A: Validation of Grammar by Sub Parser* for the table that summarises the concrete syntax grammar rule and each sub parser validates.

Expects(Token) function

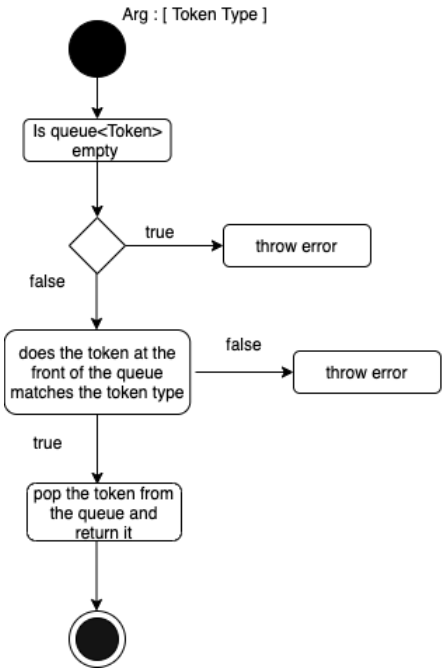


Figure 4: Activity diagram for Expects(Token) function

This function checks if the current token at the front of the queue that is current being validated matches the token type in the argument. If it matches, it will consume the token in the queue and return the value of the token. If it does not match, it implies that the program has violated the syntax grammar rule and an error will be thrown. A series of expects can be concatenate together to enforce a certain grammar rule structure. The strict arrangement of expects for different structure enforce that only the required tokens are at where they are supposed to be and must be there. This will ensure that there will be no extra and no missing token if not an error signal will be thrown.

The following details how the individual sub parser validate the grammar rule they are responsible for.

1. Expr Parser

< expr: expr ‘+’ term | expr ‘-’ term | term >

< term: term ‘*’ factor | term ‘/’ factor | term ‘%’ factor | factor >

< factor: factor: var_name | const_value | ‘(’ expr ‘)’ >

The expr parser has 3 helper functions to parse **Factor**, **Term** and **Expr**. Combined, they can validate the recursive definition of an expr recursively.

Afterwards, shunting yard algorithm is used to convert the infix mathematical expression into postfix notation, which is also known as AST and we add it to the assign AST Node. The detailed run through is found in the appendix.

The validation of the cond expr is similar to the expr parser.

< cond_expr: rel_expr | '!' '(' cond_expr ')' | '(' cond_expr ') '&&' '(' cond_expr ')' | '(' cond_expr ')' '||' '(' cond_expr ')' >

< rel_expr: rel_factor '>' rel_factor | rel_factor '>=' rel_factor | rel_factor '<' rel_factor | rel_factor '<=' rel_factor | rel_factor '==' rel_factor | rel_factor '!=' rel_factor >

< rel_factor: var_name | const_value | expr >

2. Assignment Parser

< assign: var_name '=' expr ';'' >

To validate for the grammar rule above, the assign parser will validate and expects for all the token above except for the expr which it will reply on the expr parser to validate.

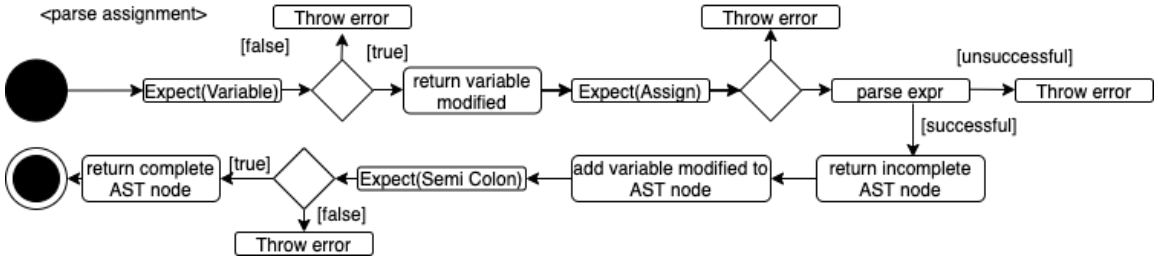


Figure 5: Activity diagram for parsing assignment

1. Expects(VARIABLE)
2. Expects(ASSIGNMENT)
3. Calls Expr Parser
4. Expects(SEMI COLON)

The Expr Parser will return an incomplete Assignment AST Node in which the assignment parser will update the used variable attribute with `var_name` and the type set to `ASSIGN` and returns the completed AST node.

Given line 13 in the sample program: `sum = sum + (number % 10);`

It will be translated into the AST Statement Node below.

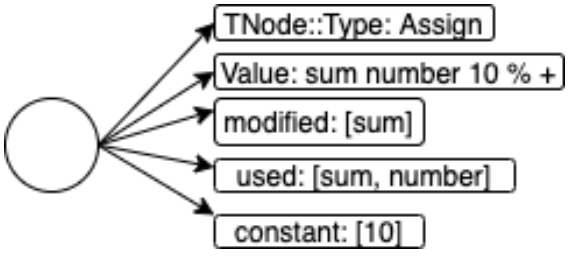


Figure 6: Assignment AST Node

The validation of the remaining three statement types is similar to assignment

< **call**: ‘call’ **proc_name** ‘;’ >
< **read**: ‘read’ **var_name** ‘;’ >
< **print**: ‘print’ **var_name** ‘;’ >

3. Statement List Parser

Validates Grammar Rule: **stmt+**

As observed, a statement list always starts and ends with LEFT BRACE (‘{‘) and RIGHT BRACE (‘}’). The container statement will have validated the LEFT BRACE, thus the statement list parser will only have to peek at the queue to check for the RIGHT BRACE to determine the end of the right brace. If it does not see a statement keyword before encountering the RIGHT BRACE, it means that the rule that a statement list must contain one or more statements has been violated and thus returns an error. If not, it will continuously peek at the queue to check for the statement keyword to determine which sub parser to use and validate the statement list. It will create a Statement List AST Node at the start with the type STMTLIST and append every successful statement validation returned statement AST Node to its children node. Once it peeks a RIGHT BRACE, it will return the updated Statement List AST Node.

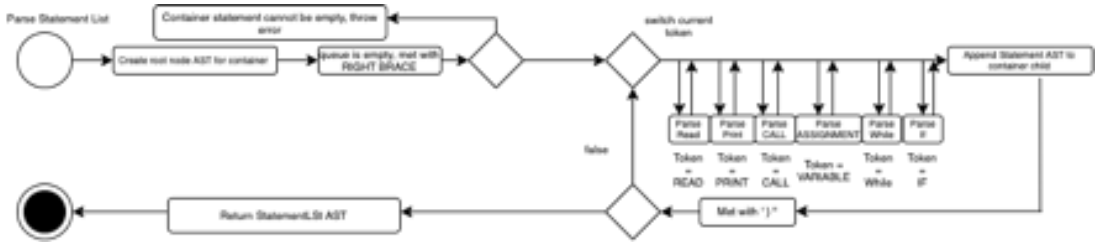


Figure 7: Activity diagram Statement List

9. If Parser

< **if**: ‘if’ ‘(‘ **cond_expr** ‘)’ ‘{‘ **stmtLst** ‘}’ **‘else’** ‘{‘ **stmtLst** ‘}’ >

1. Expects(IF)
2. Expects (LEFT BRACKET)
3. Calls Cond Expr Parser
4. Expects (RIGHT BRACKET)
5. Expects (LEFT BRACE)
6. Calls Statement List Parser
7. Expects (RIGHT BRACE)
8. Expects (ELSE)
9. Expects (LEFT BRACE)
10. Calls Statement List Parser
11. Expects (RIGHT BRACE)

To validate for the grammar rule above, the if parser will expect and validate for all the token above except for the cond_expr which it will reply on the cond expr parser to validate and the statement list parser to parse the stmtLst. It will first create an IF AST Node with the type set to IF, if the grammar and statements has been successfully validated, it will receive two Statement List AST node from the ‘then’ and ‘else’ block in which it will append it to its children attribute and return the IF AST Node.

The validation of while, procedure container statement is similar to If container statement except that we do not need to validate cond expr for procedure statement:

< **while**: ‘while’ ‘(‘ **cond_expr** ‘)’ ‘{‘ **stmtLst** ‘}’ ‘;’>

< **procedure**: ‘procedure’ **proc_name** ‘{‘ **stmtLst** ‘}’ >

11. Program parser

Validates Grammar Rule : **< program**: procedure+ **>**

To validate for this grammar rule, the parser simply has to ensure that the queue <Token> that the tokenizer has passed is not empty. And if it is not empty, it will rely on the procedure parser to continuously parse for procedure until the queue is empty.

The parser will then create a root AST node which appends all the procedure node to its children.

Figure 8: Activity diagram for Program Parser

Semantic Checks

The Recursive Descent Algorithm will only ensure that the concrete grammar syntax is followed strictly but it misses to spot the additional rules as mentioned in the introduction. Therefore, we have various data structure to help us enforce semantic rules.

1. Duplicate Procedure Declaration

This additional rule will be check in the procedure parser.

If there are two or more procedure in the program with the same `proc_name`, when we are parsing the program, the first procedure will have added into the `Procedure List` before parsing the remaining duplicated procedure declaration. Thus, we simply check if the `proc_name` of the procedure that is currently being parsed exist in the `Procedure List`. If there exist a procedure with the same `proc_name` in the `Procedure List`, it implies that there is a duplicated procedure declaration, and an error will be thrown.

2. Undeclared Procedure Calls

This additional rule will be check after the program parser.

After a successful parsing of the program, the `Call Matrix` and `Procedure List` will have been fully populated. To check if any of the procedure makes a call to an undeclared procedure, we simply have to check if all the procedures in every row of the `Call Matrix` exist in the `Procedure List`. Since the column of the `Call Matrix` represents the `proc_name` of the procedure that was being called, if the `proc_name` does not exist in the `Procedure List`, none of the procedure being parsed in the program matches the `proc_name` and thus it implies an undeclared procedure call. Therefore, an error will be thrown.

3. Cyclic Calls
This additional rule will be checked in the design extractor.

1.2.3. Design Decisions

Tokenizer
We decide to separate the validation of variable and constant and removal of white space to tokenizer so that it will be more efficient and easier for the parser to parse for grammar rules as it has less things to consider. We can identify keyword of the statement early, delimiters and different type of symbols and encapsulate it in Tokens such that the parser can parse for grammar at a high level of abstraction.

Parser
We choose to use recursive descent for the parsing as the requirements of the SPA is complete and final, and we do not have to consider portability issue. Recursive descent is very intuitive and easy to debug and we can ensure the program follows the given rule strictly as long we adhere to the rule tightly when we are implementing the parser.

Abstract Syntax Tree
Our team chose to use an abstract syntax tree as choice of data representation because we feel that it is the most appropriate data structure for this project. Not only is it used widely in numerous programming languages, it is also the clear data structure choice as the nature of SIMPLE code is very similar to that of a tree. Moreover, one advantage of having an abstract syntax tree is that it would make debugging and verification of correctness much simpler. We would not be bogged down by concrete syntax such as semicolons and we would also be able to visually spot mistakes made in the generated abstract syntax tree. Lastly, trees are a very familiar data structure to many of us and we are experienced with the various traversal algorithms to extract the relationships.

Another design decision regarding the abstract syntax tree is the storing of constants and variables modified and used in the node objects. In most implementations of abstract syntax trees, this information is not stored and are only part of the children of the node. However, our team realized that the extraction and storing of these variables and constants while parsing would increase the efficiency of extracting relationships in the design extractor at little additional cost.

An additional design decision made was the increased specification of node type. In most implementations of abstract syntax trees, statement lists are not differentiated by the type of statement list. However, our team recognized that by specifying the type of statement list, e.g. while statement list vs procedure statement list, we can increase the efficiency of extracting the appropriate relationships in the design extractor.

1.2.4. Abstract API

SourceParser API	Description
TNODE `parseProgram()`	After a `SourceParser` object has been created with a SIMPLE code, `parseProgram` can be called to generate the abstract syntax tree and return it.
PROC_MATRIX `getMatrix()`	Returns an adjacency matrix that will the row representing the procedures will call statement and the column representing the procedure that they call.

1.3. Design Extractor

1.3.1. Design Extractor

Implementation

The Design Extractor is responsible for the extraction of relationships from the AST. It takes in an AST and the Call Adjacency Matrix from the parser, traverses through the AST and calls the appropriate PKB API to populate the tables.

There are 3 main types of functions within the API and their uses are stated below

Parser-PKB functions	These are the only public functions in the design extractor class. The roles of these functions are to call the appropriate PKB API when a relationship is identified.
Type-checking functions	Type checking functions are private functions that check a node for its type.
Extractor functions	Extractor functions are private functions that are called to extract specific relationships when the traversal algorithm identifies that a section of the tree might contain a particular relationship.

Pre-Traversal

Currently, the root of the AST contains all the procedure in the program and they are arranging in the order of their position in the program. That means that the DE will traverse the procedure in the order of the position. However, there are merits in traversing the procedure in the reverse topological sorted order of the Call Matrix which brings about helps with extending the Modifies and Uses with calls statement. The details will be explained in the Extend Modifies/ Uses section.

We can notice that the procedure calls form a call graph in which the procedure is the node and the call statement represents an edge from the caller to the callee. The adjacency matrix is essentially the call graph and we will use this data structure to sort the procedures in the root AST topologically. The entire call graph may not be a connected one but it does not affect

the result as the individual components and their relative position in the topological sort order will still be correct. We will also include procedures without call statements as the parser did not include it during the parsing. Thereafter, we will rearrange the procedures in the root in the reverse order of the topological sort.

Additional Rule: Detecting of Cyclic Call

The topological sort will help us detect any cyclic call too, when we are sorting the calls, if any of the procedure has a rank that is higher than its children node in terms of the position it was pop from the stack will indicate that there is a cyclic call and an error signal will then be thrown.

With this final check, we have ensured that the entire program has met and fulfilled all the concrete syntax grammar and additional rule.

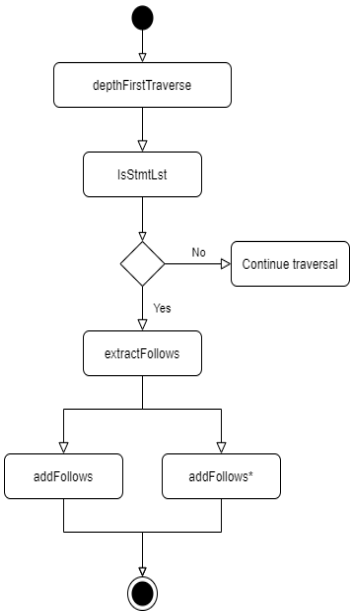


Figure 9: Activity diagram of extraction of follows

Traversal

To extract relationships from the AST, the main job of the Design Extractor is to traverse the tree and find patterns which indicate that a particular relationship might exist. Our team decided to use a depth first traversal approach, and this is accomplished by having a traversal function which can be recursively called.

When traversing, the algorithm looks out for container nodes which indicate that recursion is necessary to traverse the tree. When a container node is identified, the algorithm recursively calls the traversal function on the untraversed child nodes of the parent node. This process is repeated until no more nesting is found and the algorithm traverses back up the tree, extracting all the relationships that it finds.

`Follows` and `Follows*`

The `Follows` and `Follows*` relationship can only be found within statement lists. Thus, we look for statement lists within the AST to extract all `Follows` and `Follows*` relationships. The activity diagram below shows the various steps of our code. To extract all possible `Follows` and `Follows*` relationship from a statement list, we use two iterators to iterate through the list. The diagram below is a visual representation of the algorithm used.

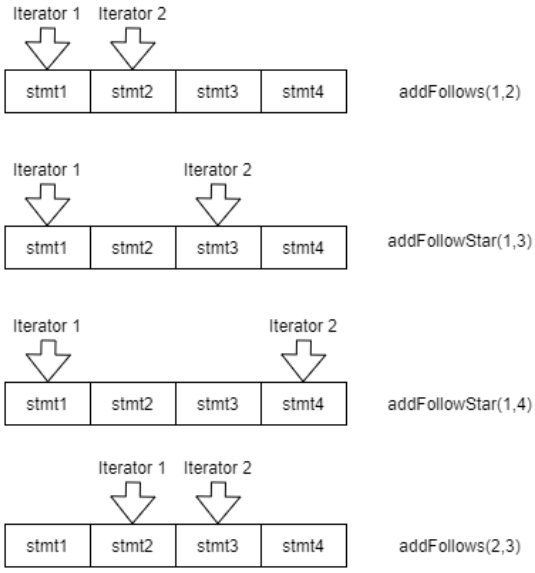


Figure 10: Visual representation of algorithm

Step 1: Assign iterator 1 to the first element and iterator 2 to the next element.

Step 2: Add Follows relationship.

Step 3: Repeatedly increment iterator 2 until last element in list.

Step 4: Add Follows* relationship after every increment.

Step 5: Increment iterator 1 by 1.

Step 6: Repeat steps 1 – 5 until iterator 1 reaches the last element in list.

Following these steps would extract every `Follows` and `Follows*` relationship in the statement list.

Parent and Parent*

The `Parent` and `Parent*` relationship can only be found within container blocks. Thus, the code looks for container blocks within the AST to extract all `Parent` and `Parent*` relationships. In order to handle nested container blocks, the code maintains a vector of statement numbers to help keep track of which nodes are parents to the container block.

When the code finds a container block, it first adds the parent to the back of the parent vector. This indicates the start of a container block in the AST. Next, the code recursively calls the traversal function on all the children of the container parent node. As the vector of parent statement numbers would increase in size for every nested while/if, it keeps track of all parents of the container, and this allows it to extract the appropriate `Parent*` relationship.

Finally, when the parent extractor function is called in the extract statement function, the code iterates through the entire vector of parent statement numbers and adds all appropriate `Parent` and `Parent*` relationships.

Modifies and Uses

As the implementation of extracting modifies and uses relationships are extremely similar, we will be addressing them in the same section.

In our implementation of the abstract syntax tree, the nodes of the tree contain a set of all variables modified in the statement. Thus, the role of the Design Extractor is simply to get the set of modified variables from the node and add all the relationships to the PKB.

However, this is not always the case as the Design Extractor must account for container statements like while and if-else. To solve this issue, our code contains various copy functions.

The copy function calls the copy function in the PKB and copies all variables modified in a statement to another statement. For example, if the PKB stores the information that variables `var1`, `var2` and `var3` are modified in statement 1, copying from 1 to 2 would store the information that variables `var1`, `var2` and `var3` are modified in statement 2.

This copy function is called in the Design Extractor, in the parents' extractor function, whenever a container relationship is identified. This allows the code to copy all variables modified in the children statements to the parent statement.

Extended `Modifies` and `Uses`

Since there are not cyclic calls, that means there are nodes in the call graph without outgoing edges. Since we will traverse in the reverse order, we will always traverse this node without any outgoing edges first. This will mean that their modifies and uses will be final as they do not have to consider any other procedure's uses and modifies. Any procedure that copies all their uses and modifies from the procedures the call will ensure that those procedures would have been traversed first thus their uses and modifies will be updated for the Caller to copy. Therefore, by induction, we can correctly extend the modifies and uses of each procedure. The PKB has a function that allow us to copy all the uses and modifies from a procedure to a particular statement. This function will help facilitate the updating of modifies and uses of the call statements.

`Calls` And `Calls*`

Extracting the `Calls` relationship is trivial as the parser has done all the work building the call matrix. All the Design Extractor has to do is to iterate through all the row and columns, adding all the entries of the column to the row. `Calls*` is simply the transitive closure of `Calls`. We will iterate the procedure in their reverse Topological Order and add their neighbour to `Calls*` and copy all their neighbour `Calls*`. The diagrams below show how `Calls` and `Calls*` is being extracted for the source program above. This algorithm works as the reason stated in extended modifies and uses because we will always guarantee that we have the most updated `Calls` and `Calls*` for the neighbour of any procedure we are updating as we traverse in the reverse topological order.

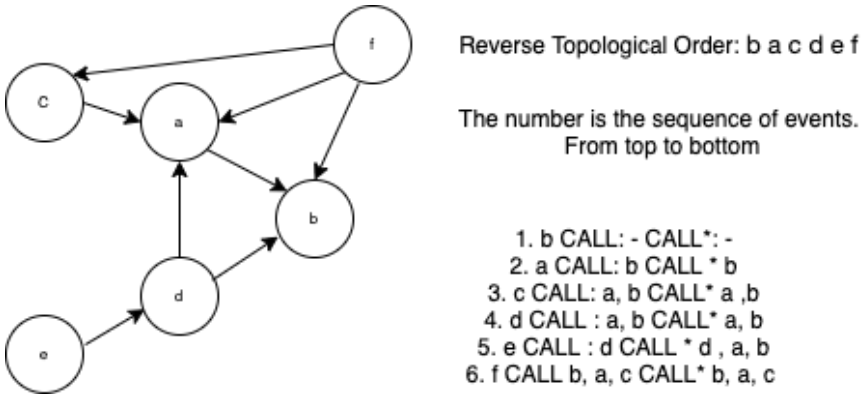


Figure 11: Diagram showing extraction of Calls relationship for procedure a – f in SIMPLE program*

`Next`

As this project is done iteratively over a semester, we decided to extract the `Next` relationship by building on to existing extractor functions as well as adding new extractor functions for while/if nodes in the AST.

When the algorithm identifies a While node during its traversal, it would call a While-Next extractor function. We are not concerned about the statement after this While statement as this would be handled in the statement list portion. The While-Next extractor function takes in two nodes as parameters, the first node is the While node while the second node is a current node to allow recursion. This is applicable when the last statement in the While loop is an If-Else statement, thus both branches would have to point back to the start of the While loop. In a normal case, the current node is set to the While node by default and only changes during the recursion. The extractor function then adds the appropriate `Next` relationship for the first and last statement of the While loop.

Next, when the algorithm identifies a statement list node in the AST during its traversal, it would call the Follows extractor function to extract all Follows relationship in the list. As the `Next` relationship is similar to the Follows relationship, we can extract `Next` easily by following the iteration of the statement list in the Follows extractor function shown in Figure 9.

However, when the iterator encounters an If node in the statement list, it would call an If-Else-Next extractor function and not add a `Next` relationship between the If node and the following statement. Instead, it passes the following statement number as a parameter of the If-Else-Next extractor function, together with the If node and a current node to allow recursion. This is applicable when the last statement in the If or Else branch is an If-Else statement, as both branches would have to point to the following statement. In a normal case, the current node is set to the If node by default and only changes during the recursion. The extractor function then adds the appropriate `Next` relationship for the first and last statements of the If-Else branches.

An activity diagram showing the whole algorithm process at each stage of traversal will be shown in the appendix Section 9.8.

1.3.2. On Demand Extractor

The On Demand Extractor (ODE) is a static class under the Design Extractor component and its main purpose is to handle the extraction of relationships that are required on demand during the query phase. These relationships are `Next*`, `Affects` and `Affects*`. After extraction, these relationships will then be added to the PKB and the PKB will return the appropriate information to the query processor.

As all processes in the ODE are done on demand, the time complexity of the algorithms would have to be low to ensure we do not run into timeout issues.

`Next*`

To extract the `Next*` relationship, we are doing a basic depth first traversal of the control flow graph (CFG) and extracting all `Next*` relationships. The CFG in this case is represented by the `Next` table in the PKB. In iteration 2, we initially did a one pass traversal once a single query is

called. For iteration 3, we implemented 2 different types of extractors, a partial extractor, and a complete extractor. The general approach used for both extractors are the same.

First, we start by creating an unordered set of integers to keep track of all nodes traversed. Whenever we traverse to a new node, we check whether we have traversed to this node before. If we have traversed to a node before, this could mean two things, 1. We reached the end of a while loop. 2. We reached the end of the else branch. For each case, we handle them differently to allow for the continued traversal of the CFG.

For while loops, when we encounter a node that has been traversed before, we will do a reverse traversal of the while loop and at each node, we will copy the `Next*` relationship from one node to another. This method allows us to extract all `Next*` relationship within a while loop by traversing only twice, thus ensuring the algorithm runs at optimal efficiency.

For if-else branches, we will first do a full traversal of the If branch. At the end of the if branch, we will return to the else branch and traverse until we encounter a node that has been traversed before, indicating the end of the branch. At this point, we will again do a back traversal and copy the `Next*` relationship from one node to another.

This concept of identifying cycles and back-tracking is also used in our implementation of the partial extractor. One main problem we identified when planning the partial extractor is that the presence of while loops can result in false positives, where the PKB would think the `Next*` relationship for a program line is fully populated when in fact it has not. We combat this by making the algorithm check whether a particular line is within a cycle, and then back-track to ensure all `Next*` relationship for that node is fully extracted.

`Affects`

Our team uses the one-pass algorithm mentioned during consultation to implement extraction of the `Affects` relationship in one pass. The main idea is to use a hash table to keep track of all the variables that are modified by an assignment statement such that when we meet an assignment statement, we tally all the variable used by that particular assignment statement with the hash table to check if the variable it uses are last modified by any assignment statement, if there exist an entry it implies that the assignment statement affects the current assignment statements.

The modified history table has the structure below:

{key = variable, value = {all assignment that modifies that variable} }

<u>Case 1: Statements</u>

- **Read Statement**
  Since read statement modifies a variable, it will block any assignment that last modify that variable from affecting any other assignment statement, thus we will remove the entry with key equals to that variable in the table.

- **Call Statement**
  Similarly, a call statement modifies multiple variables, so we will remove all the entries with the key matching the variables that the call statement modifies.
- **Assignment Statement**
  Similarly, it modifies a variable, thus blocking any assignments that last modified the variable it modifies from affecting. However, we also check for all the used variable by the assignment statement to determine which assignment statement affects it. However, we need to take note to do the checking first before removing any entries from the table

Case 2: If Statements

- **If – Else branch**
  There are two valid and independent control paths, thus we must consider the different assignments and variables modified in the two paths as they both are able to affect any statements outside of the if container statement, thus before we traverse the two control paths, we will duplicate the modified history table and use it to traverse the two separate path. After the traversing, we combine the information gain from going into the two paths into a new modified history table and use it to continue traversing.

Case 3: While Statements

- **While – branch**
  At a while node, the valid control path can either to skip the entire while container statement and continue traversing or to enter the while loop. Thus, we must split the modified history table, one that retains the original information that is not modified from going into the while loop and another used to traverse the while loop.

  After each iteration, the control path can choose to either exit the while loop or continue looping, thus we will duplicate the table again, and combine it with the original table and use the duplicate table to traverse the while loop. The table will stop updating when all possible path in the while loop is visited, thus we can then use the updated table to continue traversing.

`Affects*`

It is simply the transitive closure of the `Affects` table, we conduct a DFS on all assignment as the start node to fully populate the `Affects*` table. We have both functions to do the forward and reverse transitive closure for a given program line as it is computationally expensive to populate the entire table if the query is simple something of the form like `Affects*(stmt_num, synonym)`.

1.3.3. Design Decisions

Design Extractor

A possible implementation of the Design Extractor is to not have a design extractor at all. Instead of writing a parser to produce an AST which is sent to the design extractor, a possible

implementation is to work directly with the SIMPLE code, extracting relationships as the code body is parsed. Although this implementation might sound simple, our group decided on our current implementations as we felt that they are more in line with software engineering principles.

1. **Single Responsibility Principle**: By having the parser and design extractor as separate classes, we can define very specific purposes for each class. In this case, the parse is responsible for parsing the SIMPLE code and creating an AST, while the design extractor is responsible for extracting relationships from the AST. This helps our code to follow the principle of separation of concerns and have high modularity.
2. **Open-Closed principle**: By separating the source parser into two components, we can ensure our code is open to extension but closed for modification. This is especially relevant for this project as we will be working on different iterations, each having different requirements such as new relationships or new queries. However, since the SIMPLE source will not change, our current implementation allows us to extend our design extractor to extract other relationships, without touching the parser or our previous iteration's code.
3. **High Cohesion, Low Coupling**: Our implementation has low coupling as the two classes within the source parser are not reliant on each other and they would be able to work by themselves. The two classes also have high cohesion as the responsibilities are highly focused and this allows for the code to be easy to comprehend, reusable, extendable and maintainable.
4. **Testing**: By separating the source parser into two components, we will be able to test the components more robustly and be able to identify errors and debug with greater precision.

On Demand Extractor

As all extraction must be done on demand, the algorithms used must be efficient to ensure no timeout errors occur. One design decision we had to make was regarding whether we should traverse the CFG multiple times in sections based on the type of query, or a single full traversal. Initially in iteration 2 we did a one full traversal to get all Next Star relationship as we identified that many queries the PQL would ask from the PKB would require a close to full traversal of the CFG. In iteration 3, we further optimized our system to allow for both full and partial extraction for all relationships depending on the type of query. This has further optimized the efficiency of the On Demand Extractor especially for queries of program lines that are located close together in the CFG. Separate abstract API for Next*, Affects and Affects* are used, but since all of them work the same way, we will only be listing down the Next* abstract API below.

Another design decision we made was to do the traversal of Affects and Affects* on the AST instead of the CFG. This decision was made after our team planned out the traversal algorithm and realised that majority of the information required for algorithm has already been

conveniently extracted and stored within the nodes of the AST. Such information includes variables used and modified for assignment and call statements. Thus, we would not have to query the PKB for the variables and can efficiently traverse the AST. Furthermore, the structure of the various container statements is encapsulated in the form of vectors. For a statement list, we can make use of these vectors and the statements' relative positions will be preserved. Thus, we can simply loop through the vector to traverse the statement in a statement list and when we encounter a container statement, we traverse it recursively.

1.3.4. Abstract API

On Demand Extractor API	Description
`VOID extractNextStar(&CACHE cache)`	The PKB cache can call this function from the on-demand extractor with a reference to the cache as parameter to populate all Next Star relationships.
`VOID extractNextStar(INDEX programLine, &CACHE cache)`	The PKB cache can call this function from the on-demand extractor with an index and a reference to the cache as parameters to populate Next Star relationships for that program line.

1.4. PKB

The following class diagram shows the structure of the PKB. Methods have been left out from the diagram as they will be documented in *Section 1.4.2* and *Appendix A: PKB Abstract APIs*.

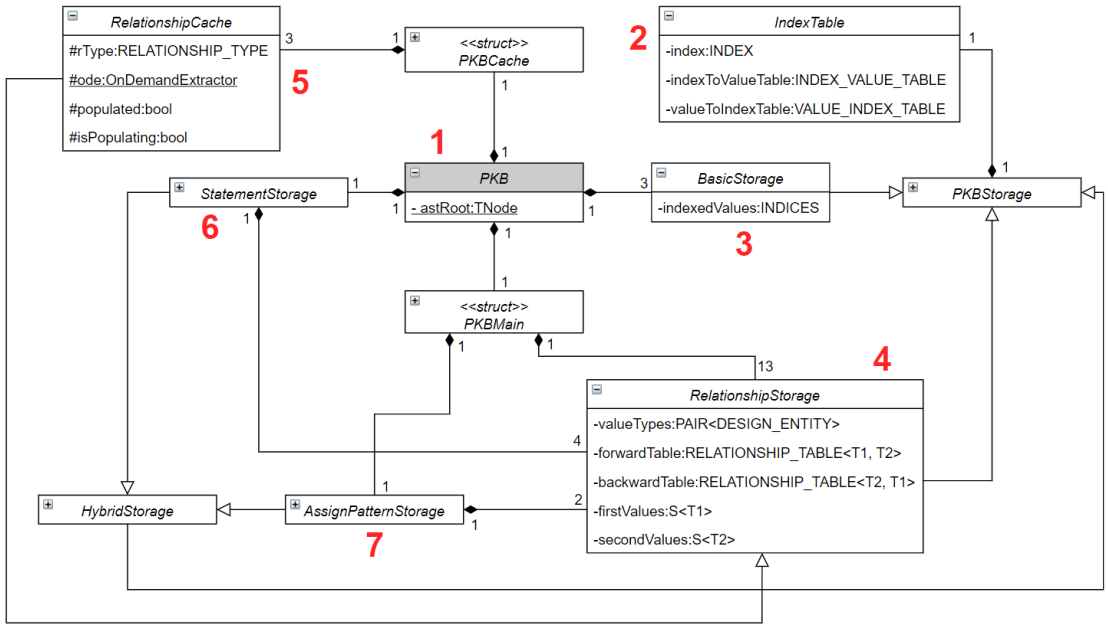


Figure 12: PKB Class Diagram (w/o methods)

The PKB is responsible for the storage of information passed down by the Source Processor. It provides the API for the Source Processor to populate the PKB and for the Query Processing Subsystem to retrieve the relevant information needed to answer queries. The following describes the roles of the main components (numbered in red above) of the PKB:

1. The **PKB** consists of multiple sub-components, each responsible for the storage of a particular design entity/relationship. It also contains the root node of the AST.
2. The **IndexTable** is a static member in the **PKBStorage** class and it is used to store the index-to-string mappings for every design entity type (statement, variable, constant, procedure). Indexing is based on the order in which the design entities are added into the PKB, regardless of type.
3. There are 3 **BasicStorages** within the PKB which are used to store procedures, variables, and constants respectively. They do so by storing the indices of the design entities in which they are responsible for.
4. There are 13 **RelationshipStorages** within the PKB, 11 of which are responsible for the storage of the various relationship types (Follows, Parent, Calls, etc.) that need not be computed on demand. The remaining 2 are responsible for the storage of the control variables of while and if statements. A **RelationshipStorage** contain 2 tables, 1 for forward relationships (e.g. `Parent(1, 2)`), the other for backward relationships (e.g. `Child(2, 1)`) to allow for faster lookup.
5. The 3 **RelationshipCaches** within the PKB are similar to the `RelationshipStorages`, the only difference being that they used to cache the relationships that are to be computed on demand (`Next*`, `Affects`, `Affects*`). Each of these caches contain a static `OnDemandExtractor` that extracts their respective relationships to be stored in the cache. All 3 caches are cleared after the evaluation of every particular query.
6. The **StatementStorage** is responsible for the storage of statements. It contains 4 `RelationshipStorages` (stmt-type, readStmt-var, printStmt-var, callStmt-proc).
7. The **AssignPatternStorage** is responsible for the storage of assign statement patterns. It has 2 `RelationshipStorages`, 1 is used to map of every assign statement to its left-hand-side (LHS) variable, and the other maps every assign statement to its pattern (as a string).

Assume that there is a query, based on the sample SIMPLE source program in *Section 1.1*, that has a `Next* (1, 4)` clause. The following sequence diagram shows the interaction between the Query Evaluator and the PKB when the Query Evaluator requests to check if the relationship exists in the PKB. It also shows the interaction between the PKB and the `OnDemandExtractor` when the `Next*` cache is to be populated on demand.

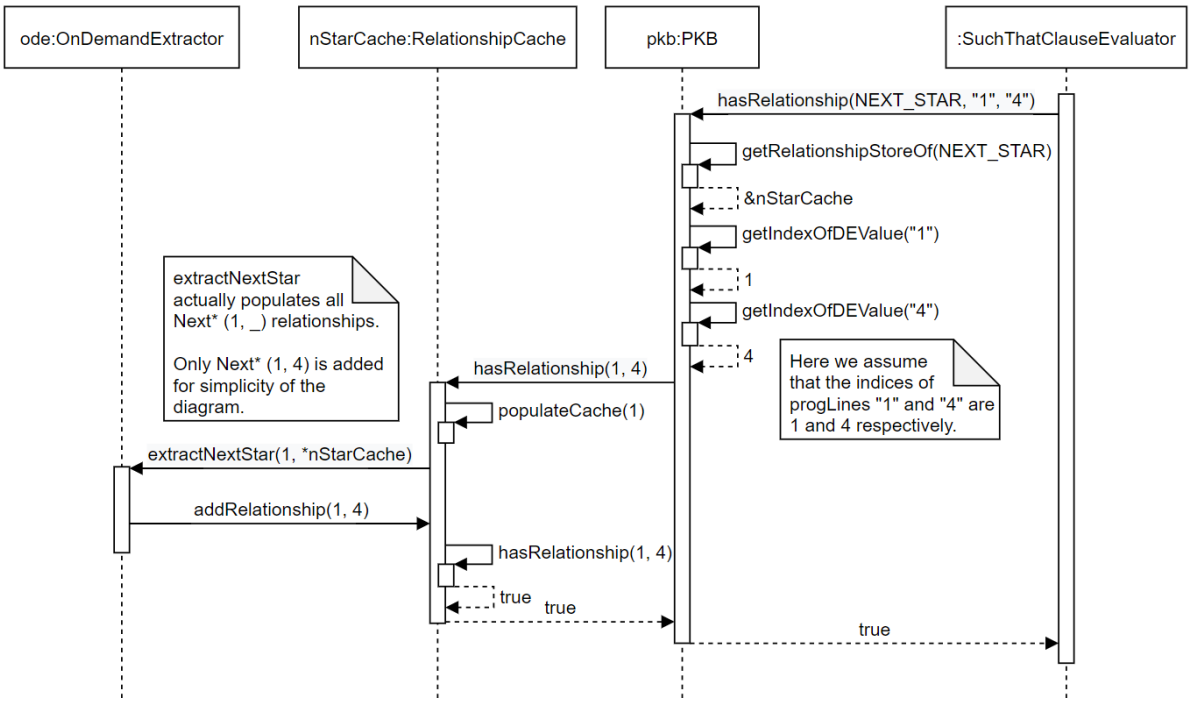


Figure 13: PKB Sequence Diagram

1.4.1. Design Decisions

Initial proposed STL data structures to store entities/relationships (Iterations 1 and 2)

Criteria	Array	Vector	Set	List	Map
Size	Fixed	Dynamic	Dynamic	Dynamic	Dynamic
Duplicates	Yes	Yes	No	Yes	No
Retrieval Time Complexity	`O(n)`	`O(n)`	`O(log (n))`	`O(n)`	`O(log (n))`
Additional Info	By linear search	By linear search	-	-	Able to store key-value pairs

Chosen data structure(s)

1. Sets
2. Maps

Justification (ranked by importance)

Reason 1: Sets and Maps do not allow duplicates. This is important because under no circumstance should there be any duplicate relationships/entities within the PKB. For example, there should not be multiple variables with the same name or multiple Follows relationship with the same pair of statement numbers.

Reason 2: The search/retrieval methods of Sets and Maps are of `O(log(n))`, which are faster than those of Arrays, Vectors and Lists which are of `O(n)` by linear search. This is crucial as this can greatly reduce the query evaluation time.

Reason 3: Both Sets and Maps use allocators to dynamically handle their storage needs. Unlike Array's fixed memory, the size of Sets and Maps can be re-sized depending on the size of the source program.

Optimizations (Iteration 3)

Changing to unordered versions of Sets and Maps

As the orders of the stored elements in the tables are not required to be maintained, we used the unordered versions of Sets and Maps as the main data structures instead. This change has improved the search time complexity from to `O(log(n))` to `O(1)` on average.

Indexing of stored design entities

Strings are inherently more complicated to evaluate compared to integers as they are essentially lists of numbers encoded to characters. For this reason, we decided to index the stored design entities within the PKB by introducing index-to-string tables for every entity type (e.g. variables, procedures), so that the Query Evaluator only has to work with integers instead of strings. With this change, the query evaluation time has improved significantly.

Generalized storage classes

From Iterations 1 and 2, we have noticed that many of the PKB API shared similar structures. For example, `addFollows(s1, s2)` and `addParent(s1, s2)` both add a relationship between statements `s1` and `s2`, the only difference being the type of relationship. As such, we created a generalized storage class from which these individual relationship storages can inherit from.

This also allows us to implement generic API methods such as `addRelationship(relationshipType, s1, s2)` which applies to any relationship type, thereby reducing the amount of repetition within the PKB API as per the DRY (Don't Repeat Yourself) principle. By having generic parent classes, the amount of coupling between the various PKB components is also significantly reduced.

1.4.2. Abstract API

API for Source Processor	Description
VOID setASTRoot(**TNODE** node)	Sets the AST root node in the PKB.
TNODE getASTRoot()	Returns the AST root node.
VOID addStmt(**DESIGN_ENTITY** stmtType, **STMT_NUM** stmtNum)	Stores a statement of stmtType into the PKB.
VOID addStmtVar(**DESIGN_ENTITY** stmtType, **STMT_NUM** stmtNum, **VAR** var)	Stores a statement-to-variable mapping into the PKB.
VOID addDEValue(**DESIGN_ENTITY** deType, **VAR** value)	Stores a design entity of deType into the PKB.
VOID addRelationship (**RELATIONSHIP_TYPE** rType, **VAR** firstValue, **VAR** secondValue)	Stores a relationship of rType between firstValue and secondValue into the PKB.

VOID addPattern(**STMT_NUM** stmtNum, **VAR_NAME** varName, **PATTERN** pattern)	Stores an assign statement pattern into the PKB.
VOID addCondVar(**RELATIONSHIP_TYPE** rType, **STMT_NUM** stmtNum, **VAR_NAMES&**condVars)	Stores a set of control variables from a while/if statement into the PKB.

API for Query Processing Subsystem	Description
INDEX getIndexOfDEValue(**VAR** value)	Returns the index of a design entity.
VAR getStringValueOfDE(**INDEX** index)	Returns the string of a design entity.
INDICES& getIndexedStmtsOf (**DESIGN_ENTITY** stmtType)	Returns the indices of all statements of stmtType.
BOOLEAN hasRelationship (**RELATIONSHIP_TYPE** rType, **VAR** firstValue, **VAR** secondValue)	Returns TRUE if there is a relationship of rType between firstValue and secondValue, FALSE otherwise.
VALUES getFirstValuesOf (**RELATIONSHIP_TYPE** rType, **DESIGN_ENTITY** firstValueType, **VAR** secondValue)	Returns the set of indices of values x for Relationship(x, secondValue) where Relationship is of rType and x is of firstValueType.
VALUES getSecondValuesOf (**RELATIONSHIP_TYPE** rType, **DESIGN_ENTITY** secondValueType, **VAR** firstValue)	Returns the set of indices of values x for Relationship(firstValue, x) where Relationship is of rType and x is of firstValueType.
VALUES getAssignOfPattern (**VAR_NAME** varName, **PATTERN** pattern, **BOOLEAN** isExactMatch)	Returns the set of indices of assign statements a for a(varName, pattern) or a(varName, _pattern_).

1.5. Query Processing Subsystem

The query processing subsystem consists of 3 components:

1. **Query Pre-processor** parses and validates a query string while extracting data from the query string to form a query object.
2. **Query Optimiser** optimises the query object by grouping and ordering query clauses.
3. **Query Evaluator** evaluates an optimised query object to produce an answer.

1.5.1. Query Pre-Processor

The class diagram below describes the structure of `QueryPreprocessor` and `QueryObject`.

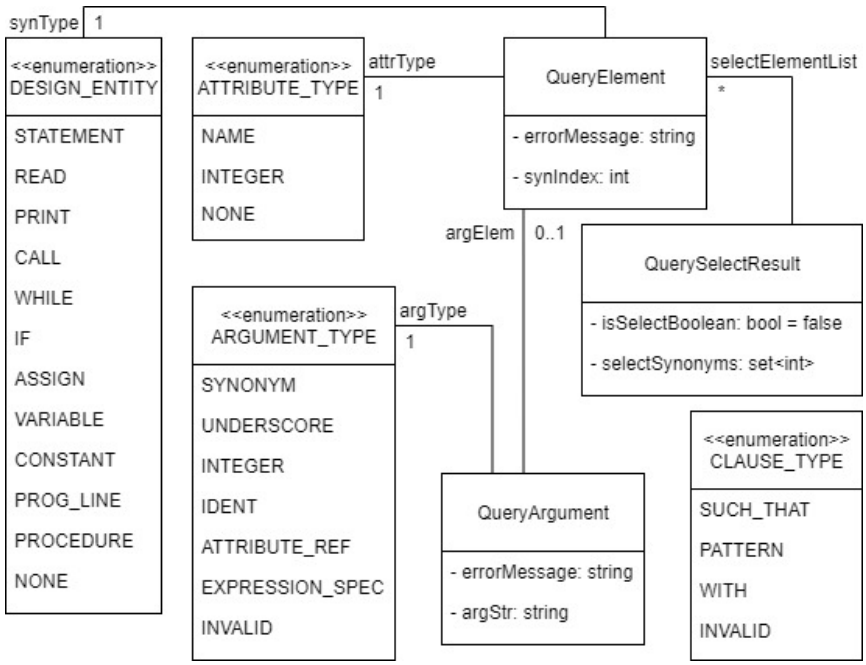


Figure 14a: Class diagram of query pre-processor

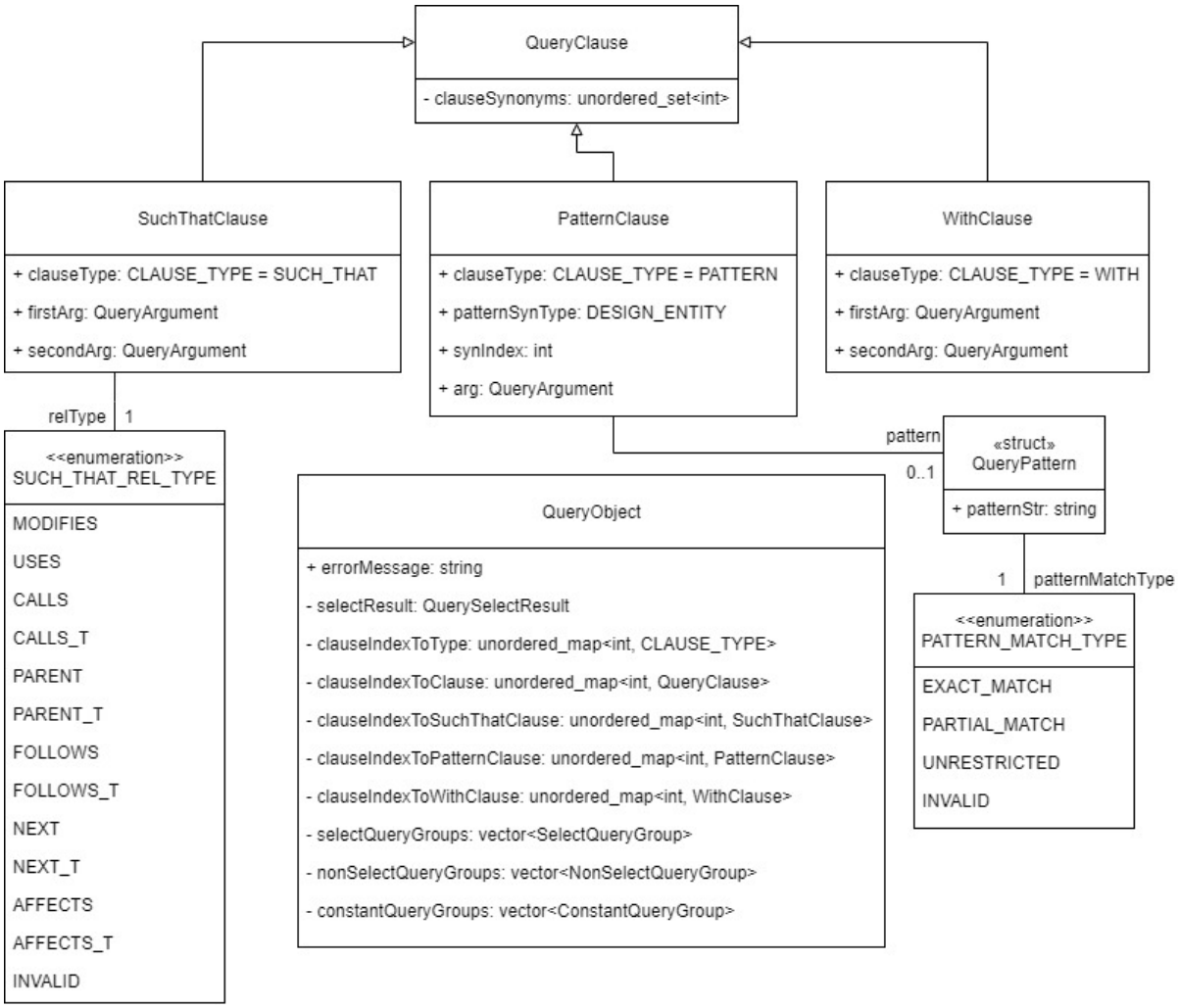


*Figure 14**b**: Class diagram of query pre-processor **(with query object)***

The general approach in parsing and validating a query is to tokenise the query string before checking that each string token follows the grammar rule in the wiki. With reference to the class diagram in Figure 14 the following activity diagrams in Figures 15 and 16 describe how a **syntactically and semantically valid** query string is parsed and validated and how a query object is created with its attributes assigned.

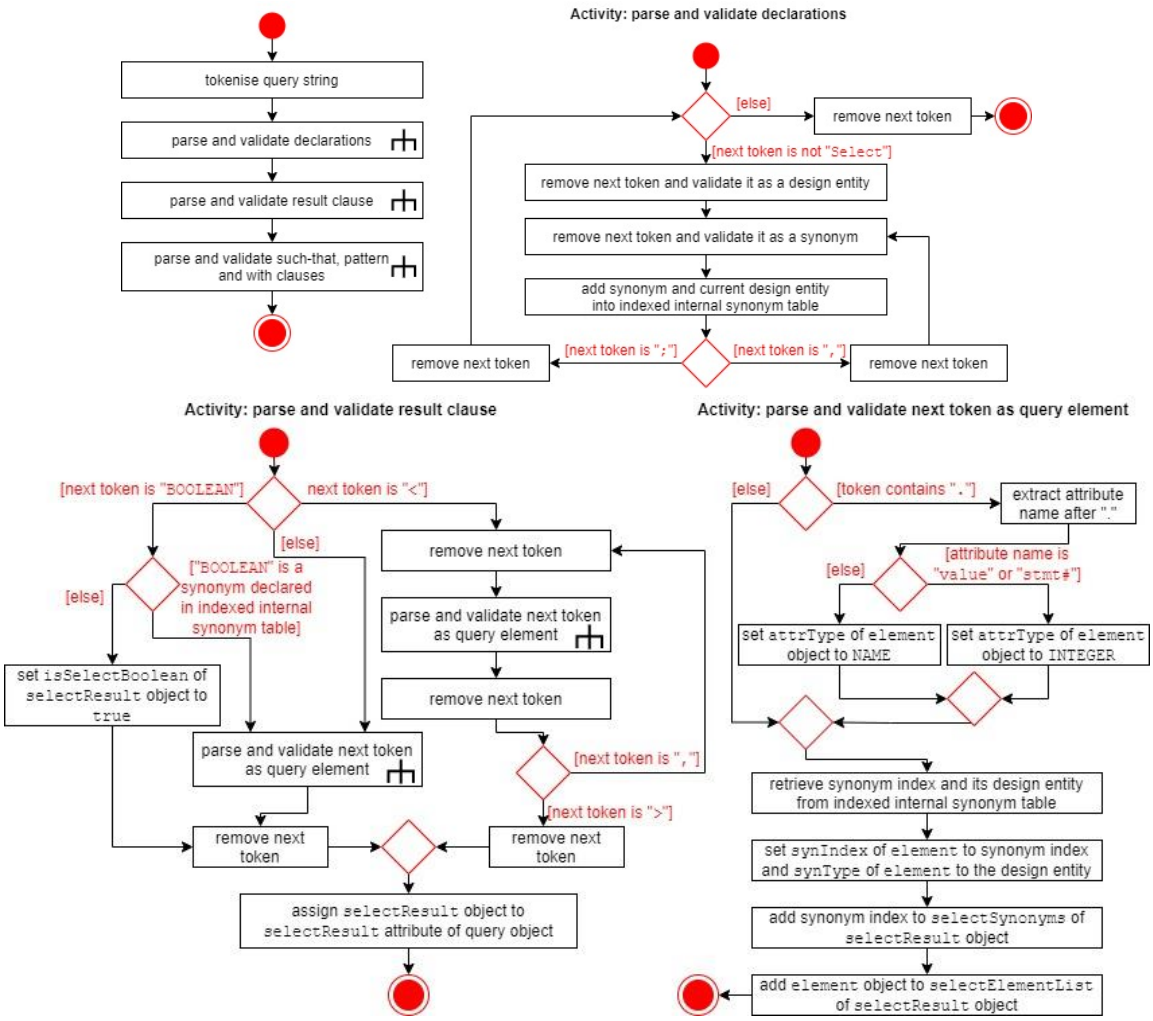


Figure 15: Activity diagram for query pre-processor (Part 1)

The indexed internal synonym table mentioned in Figure 15 is implemented using two unordered maps – `declaredSynonymsToIndex` and `declaredSynIndexToDesignEntity`.

`declaredSynonymsToIndex` maps a synonym string to its synonym index. We can use it to check that the synonyms used in the result, such that, pattern and with clauses have already been declared, before mapping them to the corresponding synonym index to be used as information for the query object as shown in Figures 15 and 16.

`declaredSynIndexToDesignEntity` maps a synonym index to its design entity. If argument of a such that, pattern or with clause is of a synonym type, this map can be used to validate that the correct synonym type is being used. This validation of a synonym argument type is explained in *Section 1.5.4*.

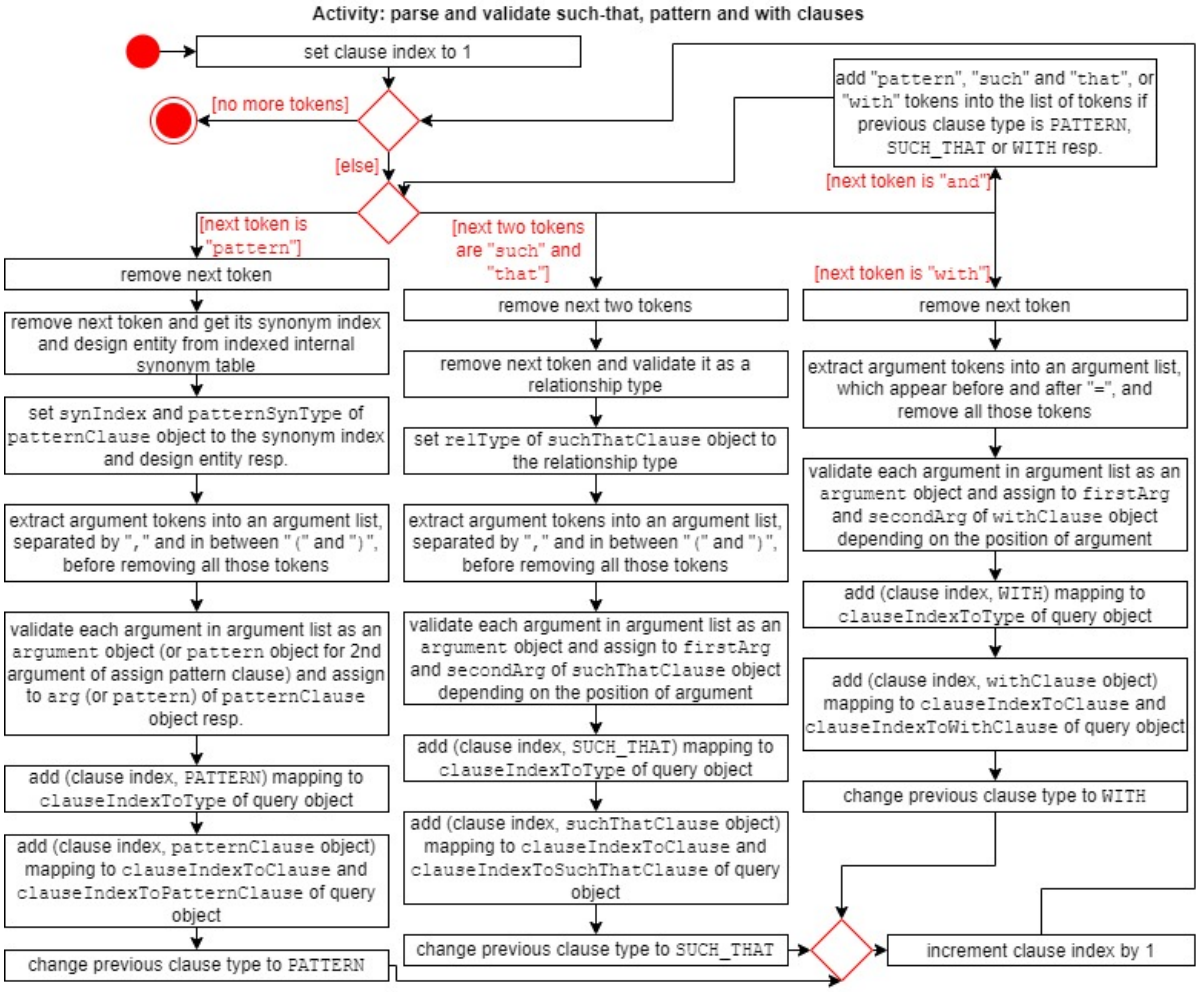


Figure 16: Activity diagram for query pre-processor (Part 2)

Once parsing and validation of a query string is complete, data from the query is extracted into a query object from the process outlined in Figures 15 and 16. The query object is then passed to the query optimiser. **Design decisions involving the query object is detailed in** *Section 1.5.4*. With reference to Figure 14, the attributes of a query object are as follow:

1 `errorMessage` is an empty string for a valid query or stores an error message (syntax error or semantic error) for an invalid query
2 `selectResult` stores information related to the result to be selected
3 `clauseIndexToType` stores a mapping of an index to the clause type
4 `clauseIndexToClause` stores a mapping of an index to a clause
5 `clauseIndexToSuchThatClause`, `clauseIndexToPatternClause` and `clauseIndexToWithClause` stores a mapping of an index to a such that, pattern and with clause respectively
6 `selectQueryGroups`, `nonSelectQueryGroups` and `constantQueryGroups` store a vector of grouped clauses and are only populated during optimisation by the query optimiser as explained in Section 1.5.2

Example of parsing and validating of PQL query

```
assign a1; prog_line n1; Select a1 such that Parent(8, a1) pattern a1(“flag”, _) with n1 = a1.stmt#
```

We use the query example above to show how attributes of a query object is populated, with reference to the class diagram (Figure 14) and activity diagrams (Figures 15 and 16). **The resulting object diagram of the populated query object is found in Figure 29.**

First, the query is tokenised into a tokens list. After activity “parse and validate declarations” in Figure 15, the indexed internal synonym table consists of `declaredSynonymsToIndex` with mappings: `(a1, 1)` and `(n1, 2)` and `declaredSynIndexToDesignEntity` with mappings: `(1, ASSIGN)` and `(2, PROG_LINE)`.

Next, from activity “parse and validate result clause” in Figure 15, “`a1`” token is extracted and parsed as a synonym based on activity “parse and validate next token as query element” in Figure 15. A query element object is created with the synonym index of “`a1`” being retrieved from `declaredSynonymsToIndex`.

Lastly, each query clause is parsed and validated based on activity “parse and validate such-that, pattern and with clauses” in Figure 16. Each argument of theses clauses is validated using a table-driven method. **This design decision is explained in Section 1.5.4.** Data extracted from each clause is contained in either a `SuchThatClause`, `PatternClause` or `WithClause` object depending on clause type. These clause objects, together with their corresponding clause indices, are inserted into relevant mappings as described in Figure 16.

Data extracted from this query is contained within a query object and is passed to the query optimiser as detailed in the following section.

Parsing and validating invalid query

`errorMessage` of `QueryObject` class is either `NO_ERROR` for a valid query, `SYNTAX_ERROR` for a query with syntax error and `SEMANTIC_ERROR` for a query with semantic error. The semantic errors are:

1. integer digit starts with zero
2. same synonym is declared more than once
3. synonym syntax is valid but not declared in the declarations part of the query
4. invalid combination of synonym and attribute name in a reference e.g. assign.value
5. invalid synonym type in argument
6. first argument of Modifies and Uses relationship is an underscore
7. both arguments of with clause are not of the same type

Parsing and validation terminate once a syntax error is found. If a semantic error is found, parsing and validation will only terminate once the result clause is parsed since if it is a SELECT BOOLEAN query, the query returns FALSE as mentioned in wiki.

1.5.2. Query Optimiser

`QueryOptimiser` will be called to optimise `QueryObject` generated from the `QueryPreprocessor` by grouping and ordering the query clauses. The following class diagram describes the structure of `QueryGroup`.

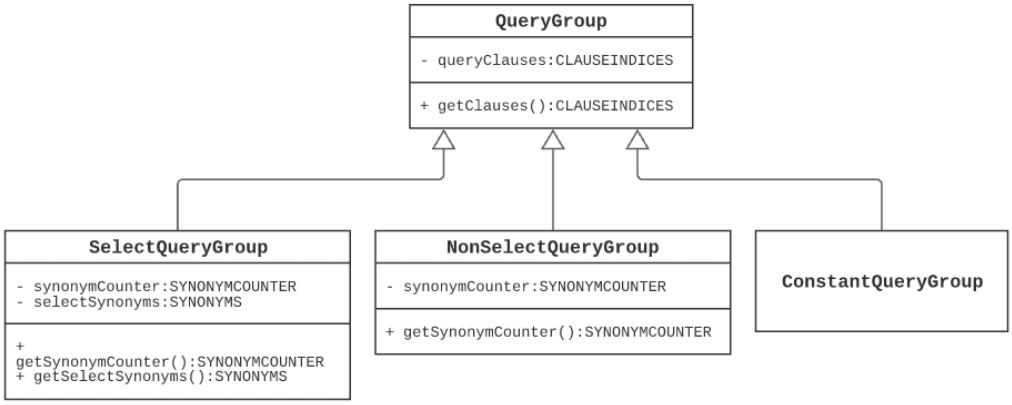


Figure 17: Query Group Class Diagram

Query Optimisation Strategy

In general, we aim to reduce the overall query evaluation time by 2 ways:

Early termination of query evaluation

With the nature of query evaluation where all clauses are implicitly linked by “and”, once it's determined that the current clause has an empty result, we can terminate early and ignore the remaining yet to be evaluated clauses, improving overall evaluation time.

Scenario 1:

- `ConstantQueryGroup {empty result}`
- `NonSelectQueryGroup`
- `SelectQueryGroup`

Scenario 2:

- `SelectQueryGroup {empty result}`
- `NonSelectQueryGroup`
- `ConstantQueryGroup`

Assuming for both scenarios, the first query group evaluates to an empty result and can be terminated early, scenario 1 would be more ideal than scenario 2, as `ConstantQueryGroup` have a relatively faster evaluation time compared to `NonSelectQueryGroup` and `SelectQueryGroup`. Step 1 and step 2 contain further explanations on how this is achieved.

Minimise the `SynonymStorage` size during evaluation of clauses with synonyms

The size of the `SynonymStorage` is positively corelated to the number permutations of synonyms required if new synonyms were added to the `SynonymStorage`, which have a significant impact on overall evaluation time based on AutoTester run time results. Step 1 and step 3 contain detailed explanations on how this is achieved.

Steps:

1. Divide all QueryClause into 3 sub-classes of QueryGroup

ConstantQueryGroup	Contains QueryClause that do not contain synonyms
NonSelectQueryGroup	Contains QueryClause that contains synonyms which are not connected to synonyms in the select clause
SelectQueryGroup	Contains QueryClause that contains synonyms which are connected to synonyms in the select clause

2. Sort order of QueryGroup for Query Evaluator to evaluate in order:

ConstantQueryGroup	Evaluated first as the evaluation outcome would just be a BOOLEAN result and no intermediate SynonymStorage will need to be generated.
NonSelectQueryGroup	Intermediate SynonymStorage result generated can be discarded after evaluating the group since these synonyms are not connected to the synonyms in the select clause. Thus, we can save on the storage space and reduce the size of the final SynonymStorage
SelectQueryGroup	Evaluated last as SynonymStorage generated needs to be preserved until final evaluation of QuerySelectResult

3. Sort order of QueryClause within each QueryGroup

   For ConstantQueryGroup implement a prioritisation of QueryClause based on CLAUSE_TYPE and RELATIONSHIP_TYPE. Relationship types (Next*, Affect, Affects*) that require on demand evaluation will be of the lowest priority due to them being significantly time intensive, while the rest of the relationship types would be ranked but in no meaningful order as they have similar evaluation times.

   For NonSelectQueryGroup and SelectQueryGroup groups, implement a prioritisation of QueryClause by strictly enforcing (1) while applying (2) whenever possible, as illustrated in Figure 18.

   (1) Ensuring adjacent QueryClause that have overlapping synonyms are prioritised as this will reduce when the size of the intermediate SynonymStorage when combining the results of individual QueryClause.

   (2) QueryClause based on CLAUSE_TYPE and RELATIONSHIP_TYPE and number of synonyms as described in clauesPriorityMatrix. For Relationship types (Next*, Affect, Affects*) that require on demand evaluation, will be of the lowest priority. QueryClause with double synonyms will be prioritised over QueryClause with single synonym. QueryClause of specific CLAUSE_TYPE and RELATIONSHIP_TYPE that generally return lesser results will be prioritised.

clausePriorityMatrix		
Priority Rank	Relationship Type	No. of Synonyms
1	Calls	1
2	With	1
3	Follows	1
4	Parent	1
5	Next	1
6	Modifies	1
7	Uses	1
8	assign pattern	1
9	if pattern	1
10	while pattern	1
11	Calls*	1
12	Follows*	1
13	Parent*	1
14	Calls	2
15	With	2
16	Follows	2
17	Parent	2
18	Next	2
19	Modifies	2
20	Uses	2
21	assign pattern	2
22	if pattern	2
23	while pattern	2
24	Calls*	2
25	Follows*	2
26	Parent*	2
27	Next*	1
28	Affect	1
29	Affect*	1
30	Next*	2
31	Affect	2
32	Affect*	2

Figure 18: Clause Priority Diagram

Query Optimiser Example

```
stmt s1, s2, s3; assign a; prog_line n; variable v;
Select s1 such that Follows (s1, s2) and Parent (s2, a) and Parent*(5, 8)
pattern a(v, _) such that Follows* (s3, n) and Next(n, 5) with n=10
```

Before Optimisation:

`QueryClause(s): Follows (s1, s2), Parent (s2, a), Parent* (5, 8), pattern a(v, _), Follows* (s3, n), Next (n, 5) with n = 10`

After Optimisation:

- `ConstantQueryGroup: Parent* (5, 8), with n = 10`
- `NonSelectQueryGroup: Follows* (s3, n), Next (n, 5)`
- `SelectQueryGroup: Follows (s1, s2), Parent (s2, a), pattern a (v, _)`

1.5.3. Query Evaluator

Helpful UML diagrams for Query Evaluator can be found in Appendix A: Query Evaluator UML Diagrams

The `QueryEvaluator` evaluates the PQL query based on the optimised `QueryObject` generated from the `QueryOptimiser`. `QueryEvaluator` interacts with the `PKB` to retrieve the relevant stored design entities and relationships for query evaluation. Subsequently, the retrieved values will be merged and stored in a `SynonymStorage`, which stores the possible values of the synonyms evaluated in the previous query clauses.

The query evaluation process can be divided into **four** distinct sub-processes:

1. Evaluation of `ConstantQueryGroup`
2. Evaluation of **ALL** `NonSelectQueryGroup`
3. Evaluation of **ALL** `SelectQueryGroup`
4. Evaluation of `QuerySelectResult`

The overall flow for query evaluation is illustrated in the activity diagram below.

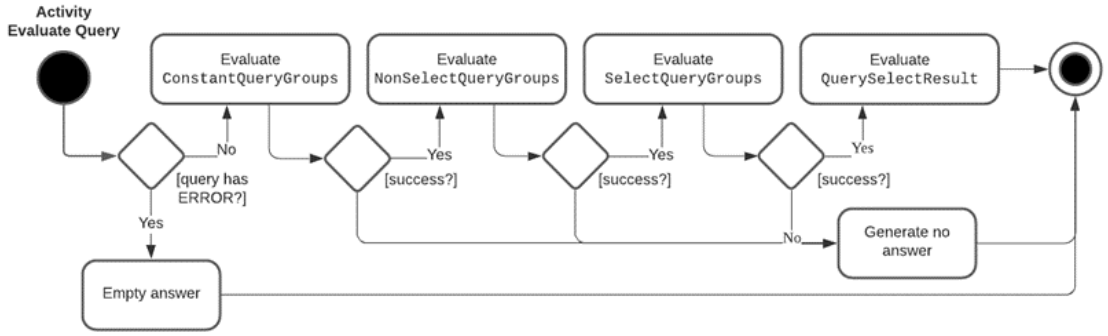


Figure 19: Evaluation Activity Diagram

The `QueryEvalautor` first checks if the `QueryObject` is free from error. Then, it will evaluate all the `QueryClause` in each `QueryGroup` in turn. Any `QueryClause` in the `QueryGroup` that returns an empty result will trigger an early termination in the query evaluation and generates a negative answer. Otherwise, `QueryEvalautor` will finally evaluate the `QuerySelectResult` and returns the answer to the query.

The sub-sections below detail the different steps in the query evaluation process.

Evaluation of each `QueryClause` in a `QueryGroup`

With respect to the first three sub-processes from above, each `QueryClause` in the corresponding `QueryGroup` will be evaluated based on their respective `CLAUSE_TYPE`.

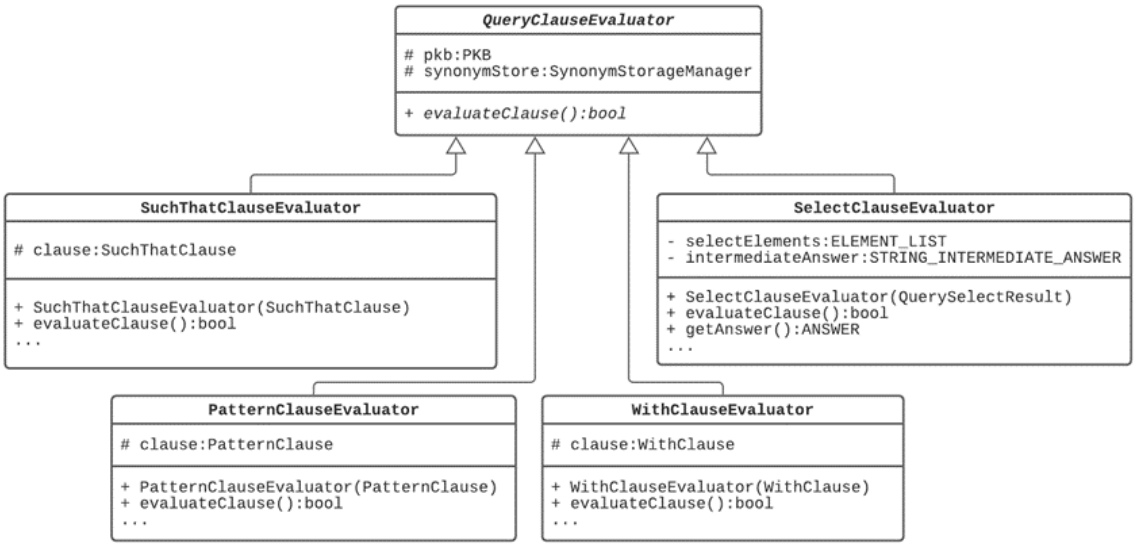


Figure 20: QueryClauseEvaluator Class Diagram

The `QueryClauseEvaluator` is an abstract class, with four subclasses, namely the `SuchThatClauseEvaluator`, the `PatternClauseEvaluator`, the `WithClauseEvaluator` and the `SelectClauseEvaluator`. Each of the subclasses inherits the abstract `evaluateClause()` method from the parent `QueryClauseEvaluator` class. The `evaluateClause()` method evaluates the given `QueryClause`.

We can observe visually how the QueryClause from each QueryGroup is being evaluated in the activity diagram below.

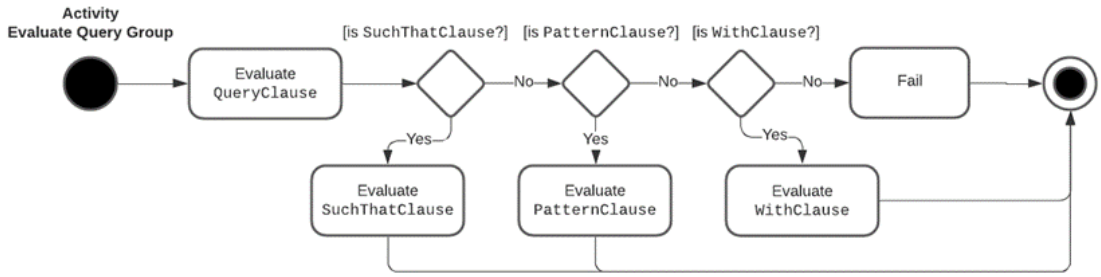


Figure 21: Query Group Evaluation Activity Diagram

Recall from the QueryOptimiser that each QueryGroup stores the CLAUSE_INDICES of the different QueryClause. The type of QueryClause can be determined by mapping the CLAUSE_INDEX to its CLAUSE_TYPE via the clauseIndexToType map in the QueryObject. Based on its CLAUSE_TYPE, the appropriate QueryClauseEvaluator will then evaluate the QueryClause, and returns true if successful.

It is **important** to note that validation of ARGUMENT_TYPE and DESIGN_ENTITY for each argument in the QueryClause is not performed by the QueryClauseEvaluator. The reason is that validation should have been done in the QueryPreprocessor. As such, we are able to generalise the evaluation of each clause just by their CLAUSE_TYPE alone.

We will now zoom in on each individual type of QueryClauseEvaluator, and explain how their functionalities are being implemented.

Evaluation of SuchThatClause

The evaluation of the SuchThatClause is performed by the SuchThatClauseEvaluator.

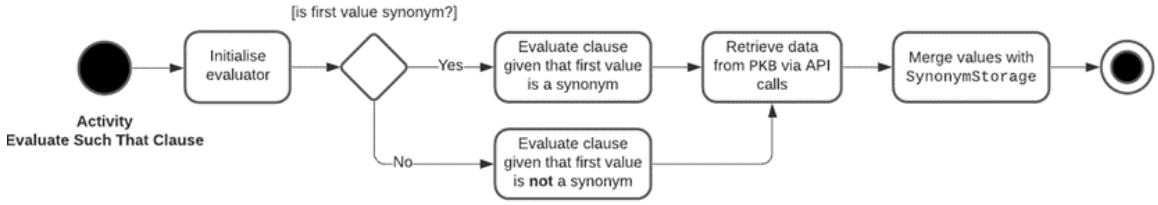


Figure 22: SuchThatClauseEvaluator Evaluation Activity Diagram

When evaluateClause() is called, the SuchThatEvaluator will first determine the RELATIONSHIP_TYPE of the SuchThatClause. This is so that the design abstractions can be retrieved from the correct PKBStorage later.

The evaluation process is divided based on whether the first QueryArgument has ARGUMENT_TYPE::SYNONYM. The evaluation then continues to the second QueryArgument. This helps to ensure that the code length of the method is not overly extensive, and keeps our code tidy. The SuchThatClauseEvaluator then retrieves the corresponding data from the PKB via the PKB's API calls *(the full list of PKB abstract APIs can be found in the Appendix)*. Subsequently, the retrieved values (if any) will be merged into the synonymStore.

Evaluation of `PatternClause`

The evaluation of the `PatternClause` is performed by the `PatternClauseEvaluator`. Evaluation for assign patterns and conditional (`WHILE` and `IF`) patterns are performed by calling the `evaluateAssignPattern()` and `evaluateConditionalPattern()` methods respectively. They are done separately as the structures of the `PatternClause` for the two forms are different *(pattern parameter for conditional patterns are always blank)*.

The rest of the evaluation is similar to the `SuchThatClauseEvaluator`. The evaluation is processed by first and second `ARGUMENT_TYPE`, then the design abstractions are retrieved from the PKB, and finally the values are merged into the `SynonymStorage`.

Evaluation of `WithClause`

The evaluation of the `WithClause` is performed by the `WithClauseEvaluator`.

Since the possible `ARGUMENT_TYPE` for both the first and second `QueryArgument` can be any one of the four – `INTEGER`, `IDENT`, `SYNONYM` and `ATTRIBUTE_REF`, the number of possible combinations for the `WithClause` arguments is increased significantly. Furthermore, the `ATTRBUTE_REF` can be further split into `INTEGER` and `NAME` as well. A comprehensive set of methods is used to divide and conquer this problem by breaking the evaluation process by validating the first `QueryArgument`, followed by the second `QueryArgument` as before. The subsequent steps follow likewise the previous two `QueryClauseEvaluator`.

Evaluation of `QuerySelectResult`

After all the `QueryGroup` have been evaluated, and there are no `QueryClause` that returns a negative result, the `SelectClauseEvaluator` evaluates the `QuerySelectResult`.

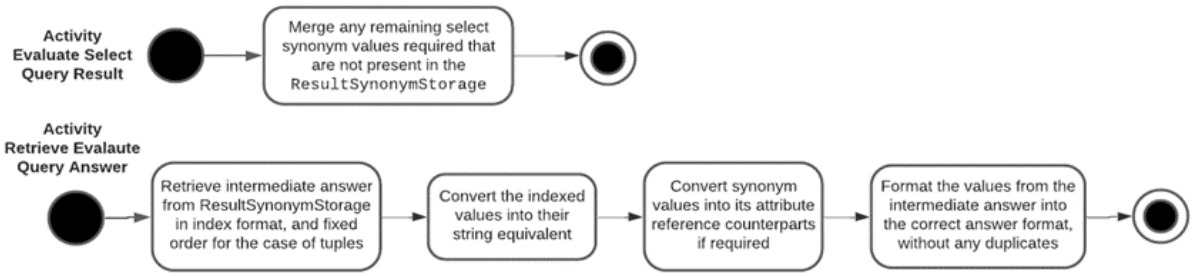


Figure 23: SelectClauseEvalautor Evaluation and Answer Retrieval Activity Diagrams

The `ResultSynonymStorage` stores all the synonym values from the previous clause evaluations. The `SelectClauseEvaluator` first merges values for any remaining synonyms not present in the storage. This is done in the `evaluateClause()` method.

The `getAnswer()` method is called to obtain the final query answer. The `SelectClauseEvaluator` retrieves the indexed intermediate answer from the `ResultSynonymStorage`. The results are then converted into their equivalent string format, converted further into their attribute reference values if required. This applies to `READ`, `PRINT` and `CALL` statements, since they can be attributed to *variables* or *procedures*. The updated intermediate answer is then collated and converted into the correct query answer format.

Implementation of `SynonymStorage`

There `SynonymStorage` class governs the merging and storing of synonym values retrieved from the PKB. It has three subclasses, namely `SelectSynonymStorage`, `TempSynonymStorage` and `ResultSynonymStorage`.

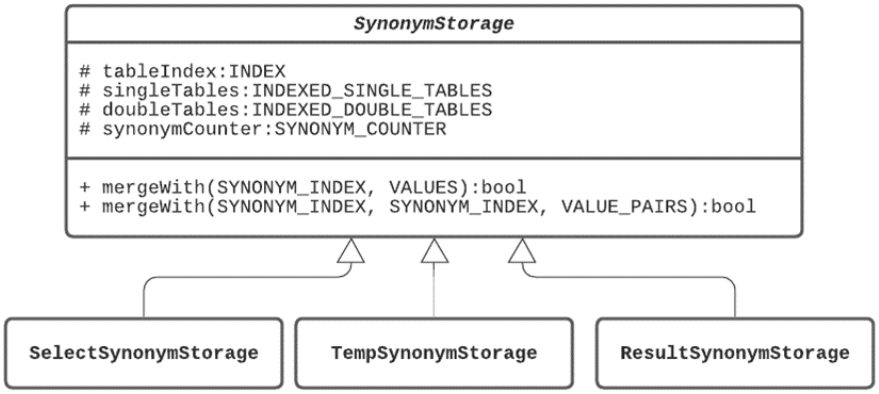


Figure 24: Simplified SynonymStorage Class Diagram

Each subclass inherits the `mergeWith()` methods from the base class. Within the base class, there are methods to intersect and union values into the `various` *single* and *double* tables. Intersect and union of values can be performed on one or two synonyms simultaneously.

The data structure of the `SynonymStorage` is uniquely designed such that cartesian products across values are vastly minimised. In fact, other than obtaining the intermediate answer, cartesian products are restricted to only two synonyms.

The `SynonymStorage` adopts the following data structure:

1. A list of `MiniSingleTable`
2. A list of `MiniDoubleTable`

`MiniSingleTable`	**`MiniDoubleTable`**
Table that contains **one** column that corresponds to values of a **single** synonym	Table that contains **two** columns that corresponds to values of a **pair** of synonyms
One table for each synonym	One table for each pair of *"connected"* synonyms
Maximum number of tables: **`N`**	Maximum number of tables: **`0.5N(N-1)`**$\approx N^2$

The key design decisions for having such a data structure for the `SynonymStorage` will be elaborated in the *Section 1.5.4*.

Another interesting point is the `synonymCounter` variable, which tracks the remaining number of each synonym to be merged into the table. If it reaches zero for a particular synonym, it means that synonym will not be in subsequent clause evaluations. The synonym is considered redundant, and its values can be removed safely from the storage. This saves not only memory space, but also time needed to merge other synonym values into fewer tables.

The implementations of the three subclasses of the `SynonymStorage` are summarised below:

`SelectSynonymStorage`	**`TempSynonymStorage`**	**`ResultSynonymStorage`**
Merges synonym values from `QueryClause` in **`SelectSynonymGroup`**	Merges synonym values from `QueryClause` in **`NonSelectSynonymGroup`**	Merges synonym values from all **`SelectSynonymStorages`**, and values of any **missing** synonyms
Contains synonym(s) from the select clause	Contains **no** synonym from the select clause	Contains **all** synonym(s) from the select clause
All expired synonym values, **except** those of **select synonyms**, are removed	**All** expired synonym values are removed	Generates the intermediate answer which is later formatted into the final query answer

Merging of Synonym Values into `SynonymStorage`

Helpful visuals for the implementation of the *`SynonymStorage`* can be found in *Appendix A: Synonym Storage Implementation Visuals*.

The following table summarises the different types of merges in the `SynonymStorage` class. *Union* refers to the addition of values of a synonym that is **not** currently in the storage. *Intersect* refers to the addition of values of a synonym that **is** currently in the storage.

Type of Merge	Implementation
Union of **single** synonym	Adds **one** `MiniSingleTable`
Union of **paired** synonyms	Adds **one** `MiniDoubleTable` and **two** `MiniSingleTable`
Intersect of **single** synonym	Intersects **one** `MiniSingleTable` and keep track of removed values in a set Removes values in `MiniDoubleTables` containing synonym Recurses using values of other synonym in `MiniDoubleTable`
Intersect of **paired** synonym (already connected)	Intersects **one** `MiniDoubleTable` Intersects **two** single synonym using values of each of the synonyms in the `MiniDoubleTable`
Intersect of **paired** synonym (**not** already connected)	Intersects **two** single synonyms Adds **one** `MiniDoubleTable` Intersects **one** `MiniDoubleTable` using values from `MiniSingleTables` Intersects **two** single synonyms again Adds `MiniDoubleTables` to connect synonyms
Intersect union	Intersects **one** single synonym Adds **one** `MiniDoubleTable` Intersects **one** `MiniDoubleTable` using values from `MiniSingleTables` Adds **one** `MiniSingleTable` Adds `MiniDoubleTables` to connect synonyms

Implementation of SynonymStorageManager

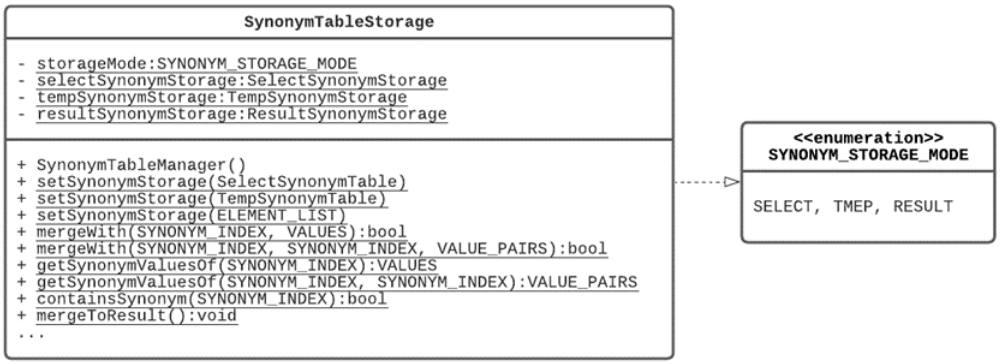


Figure 25: Simplified SynonymStorageManager Class Diagram

The SynonymStorageManager manages which SynonymStorage subclass to merge the values fetched from the PKB into. Its class variables and methods are mostly static since the QueryClauseEvaluator subclasses all depend on the SynonymStorageManager for the merge. This would serve very well the purpose of merging the values into the same SynonymStorage regardless where the SynonymStorageManager is being instantiated.

The key methods in this class are the mergeWith() methods. They are responsible for calling the respective SynonymStorage to perform a merge with the given synonym values. Also, there are getValuesOfSynonym() methods for retrieving the values of synonyms.

There are three different SYNONYM_STORAGE_MODE enumeration values – SELECT, TEMP and RESULT; they correspond to the type of SynonymStorage to be used for the merge. The static storageMode variable toggles the current working SynonymStorage. In this way, every time a mergeWith() method is called, the values will be merged with the correct SynonymStorage.

After each SelectQueryGroup is evaluated, the selectSynonymStorage will be merged immediately into the resultSynonymStorage variable via the mergeResult() method. The ResultSynonymStorage is later used to generate the intermediate answer at the end to obtain the final query answer as discussed in the previous subsection.

The inner workings of the different SynonymStorage will be explained below.

PQL Query Example

```
assign a1; prog_line n1;
Select a1 such that Parent(8, a1)
pattern a1("flag", _) with n1 = a1.stmt#
```

Answer: 9

Parent(8, a1)
a1
9

pattern a1("flag", _)
a1
2
9

n1 = a1.stmt#	
n1	a1
1	1
2	2
3	3
4	4
9	9
10	10
11	11
13	13
14	14
16	16
17	17

Interaction between Query Evaluator and PKB (Evaluation of Such-That clause)

With reference to *PQL Query Example* and *Section 1.1* sample *SIMPLE* program, the sequence diagram in *Figure 26* below illustrates the interaction between `QueryEvaluator` and `PKB` for Such-That clause. The `QueryEvaluator` calls the `SuchThatClauseEvaluator` which then calls handle the evaluation of parent clause, `Parent(8, a1)`.

Initially, `SuchThatClauseEvaluator` will attempt to retrieve existing values of `a1` from the `synStorage`. However, `synStorage` is empty at the moment as `Parent (8, a1)` is the first clause to be evaluated. As a result, `SuchThatClauseEvaluator` will obtain the values of `a1` by retrieving the child statements of stmt #8 from `pStore` of the `pkb` and verify that the design entity type of each child statement value retrieved is `ASSIGN`. Finally, merge the remaining assign statement values back into the `synStorage`.

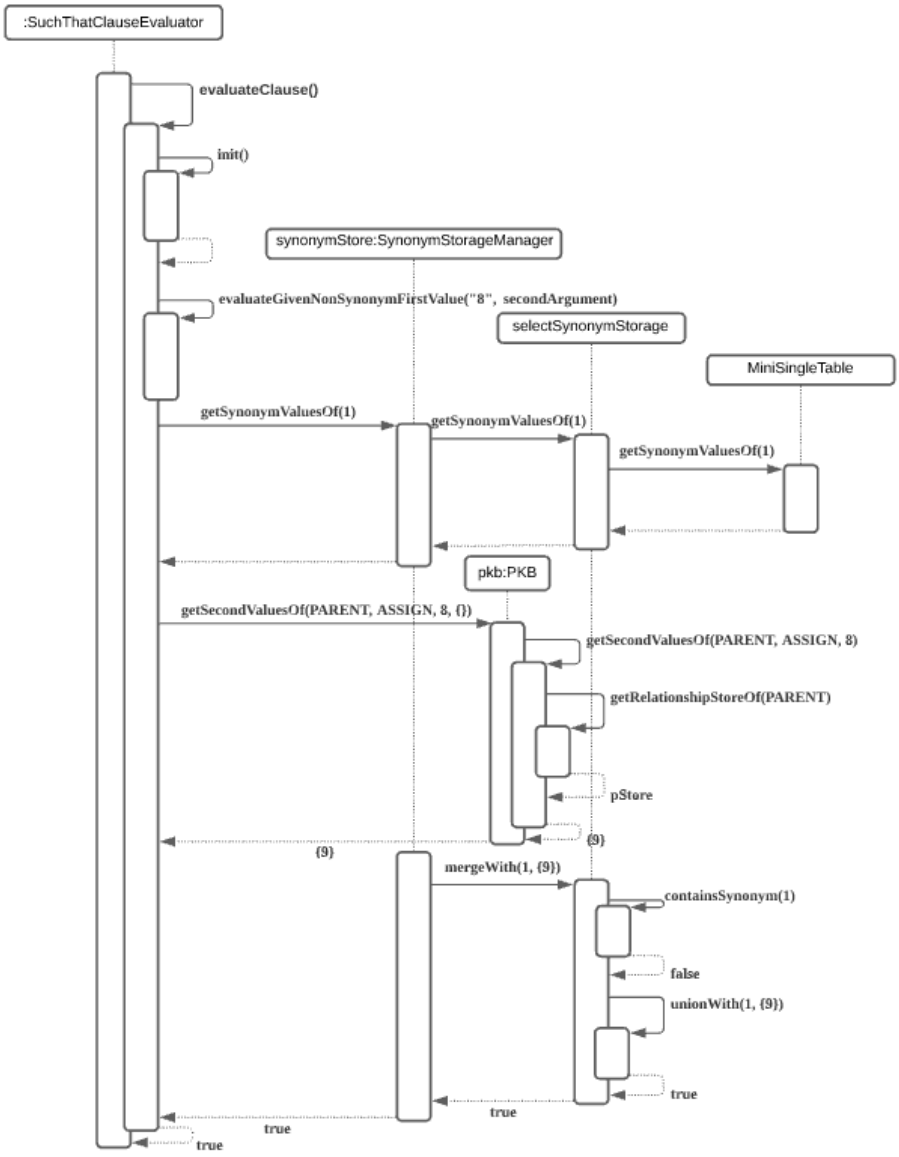


Figure 26: Evaluation of Such-That clause (PQL Query Example 1) Sequence Diagram

Interaction between Query Evaluator and PKB (Evaluation of Pattern clause)

With reference to PQL Query Example and *Section 1.1* sample SIMPLE program, the `QueryEvaluator` calls the `PatternClauseEvaluator` to handle the evaluation of the assign pattern clause, `pattern a1 ("flag", _)`.

The `PatternClauseEvaluator` will retrieve assign statement values of assign synonym `a1` which currently exist in the `synStorage`. Subsequently, for each assign statement value retrieved, verify that the variable used is "`flag`" by consulting the `aPatStore` of `pkb`. Finally, merge the remaining assign statement values back into the `synStorage`.

Interaction between Query Evaluator and PKB (Evaluation of With clause)

With reference to PQL Query Example and *Section 1.1* sample SIMPLE program, the `QueryEvaluator` calls the `WithClauseEvaluator` to handle the evaluation of with clause, with `n1 = a1.stmt#`.

The `WithClauseEvaluator` will retrieve assign statement values of assign synonym `a1` from `synStorage` which already existed in the `synStorage` from the earlier evaluation of `Parent (8, a1)` and `pattern a1 ("flag", _)`. Subsequently, for each assign statement value retrieved, verify that the design entity type of the statement value is `PROG_LINE` and `ASSIGN`. Finally, merge the remaining assign statement values back into the `synStorage`.

1.5.4. Design Decisions

Query Pre-processor: Validation of Argument

As mentioned in Section 1.5.1, a table-driven method is used to validate each argument of a clause. For brevity, validating of argument type in such that clause is explained. (Idea is the same for pattern and with clauses.) The allowed argument types are stored in an `unordered_map` mapping a relationship type to `vector<set<ARGUMENT_TYPE>>`.

The vector contains two sets of argument types (one set for each argument). (Vector for `if` pattern clause has three sets.) Vector is used as it allows duplicate sets of argument types e.g. first and second argument of `Parent` relationship allow the same argument types. Each set of argument types lists down the allowed argument types for each argument. Figure 27 describes how `Parent` relationship is mapped to a vector of its allowed argument types.

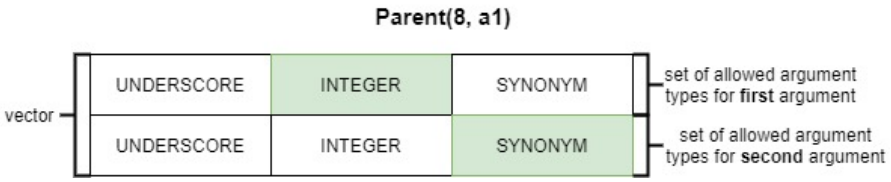


Figure 27: Validating argument types using a table-driven method

If the argument type is `SYNONYM`, the synonym type is validated using the same idea but by using `vector<set<DESIGN_ENTITY>>` instead of `vector<set<ARGUMENT_TYPE>>`. Figure 28

shows how `ASSIGN` synonym type of `a1` is validated. Recall from Section 1.5.1 that the synonym type of a synonym can be obtained from `declaredSynIndexToDesignEntity`.

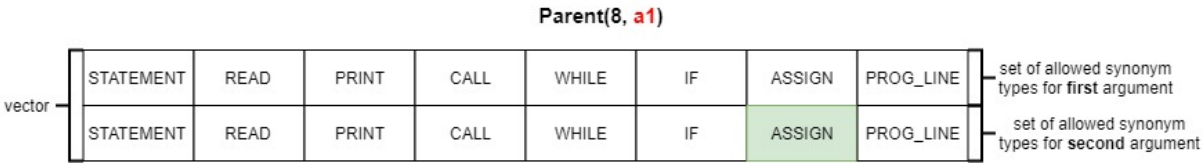


Figure 28: Validating synonym types using a table-driven method

By using this table-driven method, we better adhere to the open-closed principle. Validation of argument for new relationship types or even clause types can easily be done by adding new `vector<set<ARGUMENT_TYPE>>` and `vector<set<DESIGN_ENTITY>>` for validating `SYNONYM` argument type, without modifying the code used to validate the arguments.

Query Object

Decision 1: Storing synonyms and clauses as integers instead of strings

Recall from Section 1.5.1 that each synonym declared has an integer index and is stored in the indexed internal synonym table using two unordered maps – `declaredSynonymsToIndex` and `declaredSynIndexToDesignEntity`. The synonym is stored as an integer index for better performance. When `QueryOptimiser` groups the clauses, clauses with the same synonyms are grouped together. This means synonyms used in a query need to be compared. Comparing two integers take constant time but comparing two strings has the time complexity of the length of the shorter string. This reason also applies to the storing of clauses using clause index instead of the clause string. Performance is the reason behind this design decision.

Decision 2: Identifying synonyms used in result, such that, pattern and with clause

Synonyms involved in clauses are needed by `QueryOptimiser` for grouping of clauses. With reference to Figure 14, synonyms involved in a result clause can be obtained using `synIndex` of each `QueryElement` in the `selectElementList` of `QuerySelectResult`. However, this involves looping through each element in `selectElementList` which is time inefficient when the result clause is a tuple of many elements. Instead, the synonym index of each synonym involved in the result clause is added to `selectSynonyms` set during the parsing and validation of result clause based on activity "parse and validate next token as query element" in Figure 15. For example, `selectSynonyms` will contain one integer "`1`" for the query "`stmt s; Select <s, s, s, s, s>`" since set does not allow for duplicate values. The integer "`1`" represents the index for synonym "`s`". This is more time efficient than looping through each element in `selectElementList` before realising that only one synonym index is involved in the result clause for this example. This reason also applies to identifying synonyms that are used in such that, pattern and with clauses. In this case, instead of going through the `argElem` of each `QueryArgument` object in the query clause, `clauseSynonyms` set is used. Performance is also the reason behind this design decision.

Decision 3: Storing of clauses

With reference to *Figure 16*, data from each such that, pattern or with clause is stored as mappings and inserted into the relevant unordered maps as described in the next paragraph. Unordered maps are used because insertion of mappings by `QueryPreprocessor` and retrieval of mappings by `QueryOptimiser` and `QueryEvaluator` take average constant time and the order of mappings do not matter since a mapping is identified by the clause index. This is better compared to using maps where insertion and retrieval take average logarithmic time and the order of mappings matters.

`clauseIndexToType` stores a mapping of clause index to clause type. The clause type is then used by `QueryEvaluator` to identify which of `clauseIndexToSuchThatClause`, `clauseIndexToPatternClause` and `clauseIndexToWithClause` to use. The more general `clauseIndexToClause` cannot be used as methods in `SuchThatClause`, `PatternClause` and `WithClause` subclasses are needed by QueryEvaluator to evaluate the specified clause.

Instead, the more general `clauseIndexToClause` is used by `QueryOptimiser` to group the cluases. `QueryOptimiser` requires only the synonyms involved in each clause and this is provided by `clauseSynonyms` attribute of the parent `QueryClause` class.

Query Optimiser

With reference to *Section 1.5.2*, for query optimisation step 3, ordering of the query clauses in a query group. We have identified 2 methods

Method 1: Prioritise adjacent clauses that have overlapping synonyms

E.g., Ordering of clauses within each query group

Before Optimisation Step 3: `Follows (s1, s2) Follows*(s3, s4) Parent (s2, s3)`
After Optimisation Step 3: `Follows (s1, s2) Parent (s2, s3) Follows* (s3, s4)`

By swapping `Follows*(s3, s4)` with `Parent (s2, s3)`, we would limit the costly intersection of synonym values at a later stage when the synonym storage gets too big. In our current synonym storage model, intersection of synonym values is a lot more costly than union.

Method 2: Prioritise clauses based on the type of relationship and number of synonyms

E.g., Ordering of clauses within each query group

Before Optimisation Step 3: `Follows (s1, a1) Affects (a1, a2) Parent (a1, s2)`
After Optimisation Step 3: `Follows (s1, a1) Parent (a1, s2) Affects (a1, a2)`

By swapping `Affects (a1, a2)` with `Parent (a1, s2)`, we can delay or possibly avoid the evaluation of `Affects` clause which would take a significantly much longer evaluation time than `Parent` clause due to the on-demand evaluation feature implemented.

We have chosen to go with a hybrid approach of strictly enforcing method 1 while applying method 2 whenever possible, for a list of adjacent `QueryClause` that directly overlap with the current `QueryClause`, based on 2 criteria:

1. **Overall evaluation time**
   In most scenarios, the query evaluation will pass, that means all clauses would have to be evaluated sooner or later. As such, delaying the expensive evaluation of relationships that require on demand computing would be less relevant.

2. **Memory Space**
   In Method 1, we will be able to perform the expensive intersection of values right at the start when the storage size is still small. The much cheaper union of values *(independent of storage size)* can be prioritised last. This fulfils both criteria, since now, the storage size will be maintained relatively small, and evaluation time is shortened by prioritising expensive methods first.

Query Evaluation

As reiterated in the *Abstract*, we have implemented a trigger to terminate the query evaluation early if the intermediate evaluation of a clause returns an empty result. This would save the unnecessary effort and time of the evaluating the remaining clauses.

For the actual query evaluation, we have chosen to use a rather unique data structure for our `SynonymStorage` class to store the values of the synonyms. In Iteration 1 and 2, we opted for the one table approach to store all the synonym values when evaluating a `QueryGroup`. However, for this Iteration 3, multiple *single* and *double* tables are utilised instead.

Below, we compare the time/space complexity between the two data structures for a given state or action. Here, we consider the worst-case time complexities, with **K** being the number of synonyms in the table, and **N** being the number of design entity values *(e.g. statements, variables, constants, etc.).*

State/Action	One table	Multiple tables
Number of entries *(or cells)*	$O(K \cdot N^K)$	$O(K^2 \cdot N^2)$
Retrieval of values for **one** synonym	$O(N^K)$	$O(N)/O(1)$*
Retrieval of values for **two** connected/non-connected synonyms	$O(N^K)$ $O(N^K)$	$O(N^2)$ $O(N^2)$
Union of **single** synonym	$O(N^{K+1})$	$O(N)$
Union of **paired** synonyms	$O(N^{K+2})$	$O(N^2)$
Intersection of **single** connected/non-connected synonym	$O(N^K)$ $O(N^K)$	$O(K^2 \cdot N^2)$ $O(N)$
Intersection of **paired** connected/non-connected synonyms	$O(N^K)$ $O(N^K)$	$O(K^2 \cdot N^2)$ $O(K^2 \cdot N^2)$

Intersection-union of **paired** connected/non-connected synonyms	$O(N^{K+1})$ $O(N^{K+1})$	$O(K^2 \cdot N^2)$ $O(N^2)$
Retrieval of intermediate answer	$O(L \cdot N^K)$**	$O(L \cdot N^K)$**

**Call by reference*

***$L \geq K$ is number of synonyms per answer*

Clearly, for large **K** and **N**, the multiple tables data structure is more superior in terms of time and space complexities. Thus, we adopted the multiple tables approach. Perhaps, an interesting thing to note is that time complexity for the one table structure for union is bigger than for intersection, but the converse is true for the multiple tables structure.

Helpful visuals for the implementation of the `SynonymStorage` can be found in Appendix A: Synonym Storage Implementation Visuals.

To further optimise the query evaluation process, we included **three** further optimisation strategies involving the `SynonyStorage`.

1. **Discard redundant synonym values**
   Remove synonym values that are not used in subsequent clauses. This is implemented through a counter, which tracks the remaining number of each synonym left in a `QueryGroup`. If the counter reaches 0, it means that that particular synonym will no longer appear in subsequent clauses, and can be safely removed from the tables. This effectively removes every `MiniDoubleTable` containing that redundant synonym in the storage. The number of tables in the storage will maintain significantly small, especially if the clauses are well sorted and there are not many connected synonyms.
2. **Scope retrieved `PKB` values using `SynonymStorage` values**
   Consider the evaluation of the clause `Modifies (s, _)`. Instead of obtaining the full set of statement numbers for synonym `s` from the `PKB`, we can first check if `s` exists in the `SynonyStorage`, and using those values of `s`, check if the relationship holds. This would reduce the number of retrieved values for `s` by a fair amount on the average case, which translate to less values to merge, and hence faster merging.
3. **Discard values pre-emptively**
   There are two conditions to this. The synonym must not already be existing in the table, and that the synonym whose values are to be merged only appear once in the entire query. This would mean that merging of the values of that synonym would be pointless, since it would simply be removed right after.

Another possible optimisation not implemented in the current iteration is the possible caching of retrieved synonym values if the number of these values is considerably large, and there are many tables currently in the storage. The values can then be merged later once there are fewer tables, or left to last. This would reduce the number of tables required to perform an intersection.

1.5.5. Abstract API

Query Pre-Processor API	Description
`QUERY_OBJECT parseQuery(QUERY query)`	Parse and validate query while extracting data from query to form a query object
Query Optimiser API	**Description**
`QUERY_OBJECT optimiseQuery()`	Optimise the query object
Query Evaluator API	**Description**
`ANSWER evaluateQuery()`	Evaluate query Object and consult the PKB when necessary to return Answer

2. Testing

2.1. Unit testing

Design Extractor

Our group decided that there was no need to unit test the Design Extractor due to various reasons.

1. The Design Extractor does not store or return any information that can be unit tested. As the role of the Design Extractor is to extract relationships from an AST and populate the PKB tables, it does not store or return any information regarding the relationships. This creates numerous problems when it comes to unit testing as we would have to verify that the correct Parser-PKB functions are called the correct number of times with the right parameters. One possible solution of this would be to mock the Design Extractor using a third-party mocking library and we might explore these options in future iterations.
2. The Parser-PKB integration tests are sufficient to prove the correctness of the Design Extractor. Since the role of the Design Extractor is to extract relationships from an AST and populate the PKB tables, unit testing of the Design Extractor and integration testing of Parser-PKB have very similar cases and outputs. By ensuring the population of table is done correctly in integration testing, we can also verify the correctness of the Design Extractor. This point is further supported by the extensive unit testing of the parser and PKB, thus enabling the Parser-PKB integration test to remove the need of Design Extractor unit test.

Due to the above 2 reasons, our team decided to forgo the unit testing of the Design Extractor.

PKB sample unit test case #1 (`RelationshipStorage`)

Test purpose

The purpose is to test if `RelationshipStorage` allows the insertion of duplicate relationships.

Required test input

Component(s) tested: RelationshipStorage

For the purpose of this test, a `RelationshipStorage` stub named `relStore` is created and pre-populated with a single relationship pair `(1, 2)`.

Then the following lines will be executed:

```
int initialSize = relStore.getTableSize(); // intialSize will be 1
relStore.addRelationship(1, 2);
return relStore.getTableSize() == initialSize;
```

Expected test result

TRUE will be returned as the relationship pair (1, 2) will not be added into relStore and its size remains the same.

PKB sample unit test case #2 (BasicStorage)

Test purpose

The purpose is to test that the set of indices retrieved from the `BasicStorage` is correct.

Required test input

Component(s) tested: `BasicStorage`, `IndexTable`

For the purpose of this test, a `BasicStorage` named `bStore` will be created. A set of expected indexed values is also created as follows:

```
INDICES expectedIndexedValues = { 1, 2, 3 }
```

Then the following lines will then be executed:

```
bStore.addValue(“a”); // index 1 mapped to “a” is added into bStore
bStore.addValue(“b”); // index 2 mapped to “b” is added into bStore
bStore.addValue(“c”); // index 3 mapped to “c” is added into bStore
return bStore.getIndexedValues() == expectedIndexedValues;
```

Expected test result

`TRUE` will be returned as the returned indices from `getIndexedValues()` will match that of `expectedIndexValues`.

Query processor sample unit test case #1 (Query Pre-processor)

Test purpose

The purpose is to test that the query object created during the parsing and validation of a query is correctly populated.

Required test input

The query pre-processor component is tested. The test case input is a query with one such that, one pattern and one with clause. An example is:

```
assign a1; prog_line n1; Select a1 such that Parent(8, a1) pattern a1(“flag”, _) with n1 = a1.stmt#
```

Expected test result

Based on the example, a query object is created as shown below. (Refer to Figure 14 for the structure of a query object.) Note that `selectQueryGroups`, `nonSelectQueryGroups` and `constantQueryGroups` are only populated during query optimisation and not query pre-processing.

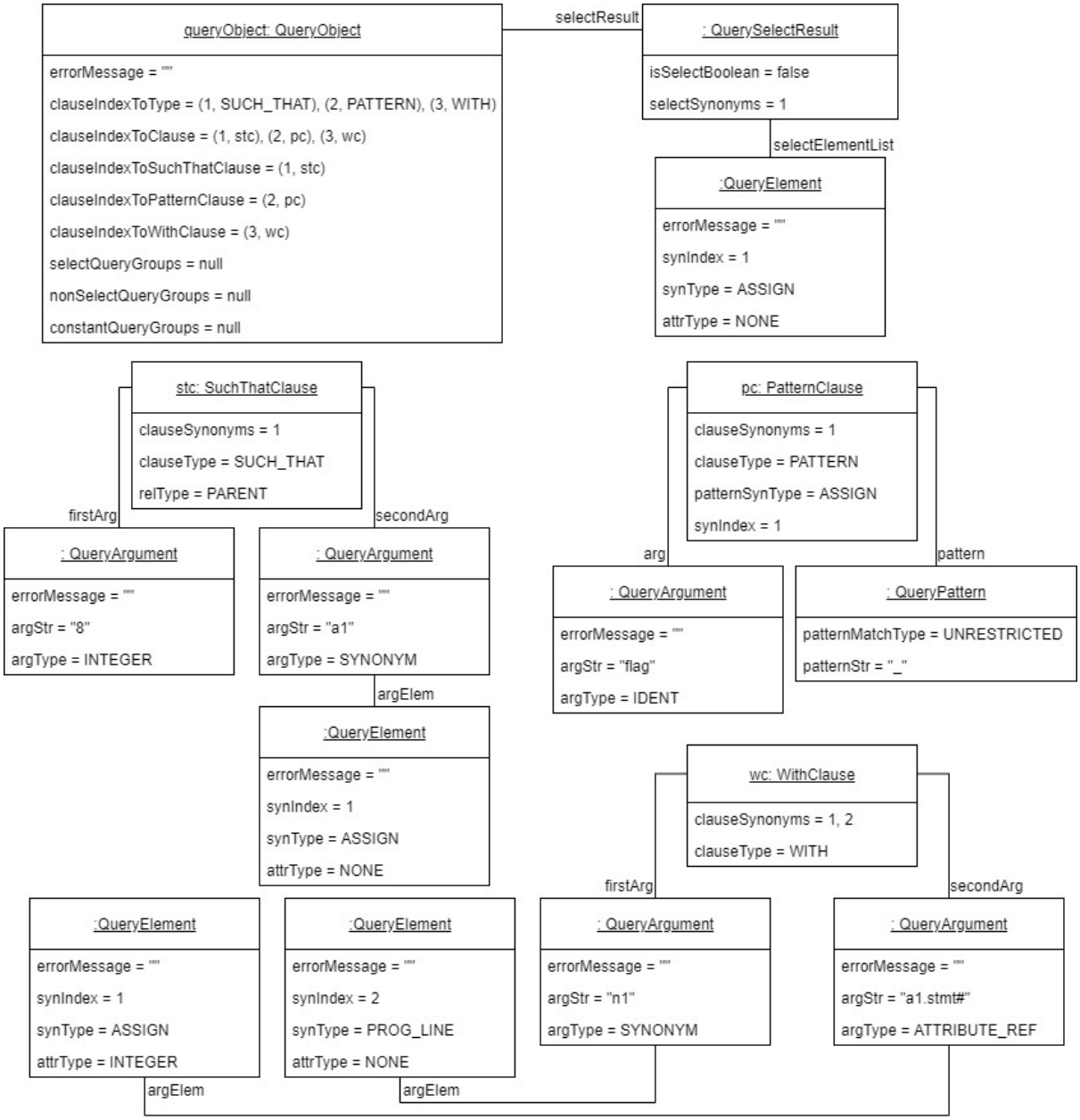


Figure 29: Object diagram of a query object produced by the test input

Query processor sample unit test case #2 (Query Evaluator)

Test purpose

The purpose is to test that the `QueryEvaluator` can successfully evaluate a `QueryClause` stub that focuses on `Calls` relationship.

Specifically, the `SuchThatClauseEvaluator` class will be tested whether it a evaluate a `SuchThatClause` with `RELATIONSHIP_TYPE` of `CALLS`.

Required test input

Component(s) Tested: `QueryEvaluator`

A `SuchThatClause` stub is created from the following PQL query

```
procedure pd; Select pd such that Calls (“FindMaxSumOfDigits”, pd)
```

Expected test result

TRUE as possible values of `pd` is “`PrintAnswer`”

Other Requirements

A PKB stub populated with sample program stated in *Section 1.1*.

2.2. Integration testing

Parser-PKB Integration Test case

Our team will be doing Parser-PKB integration test for 5 different types of SIMPLE code. This integration testing will be more detailed to ensure the Design Extractor and On Demand Extractor is working correctly as we chose to bypass their unit testing. The table below shows the different codes and interactions tested. The full list of test cases is in the *Appendix B: Parser-PKB Integration test cases*.

Code Type	Code specifications	Interactions tested
Basic Procedure	Contains 1 procedure which has 3 statements, a read, print and assignment. No container statements. This basic code will be used as a building block in the other code types.	Ensures the correct extraction and population of `Follows`, `Follows*`, `Next`, `Next*`, `Affects`, `Affects*`, statements, constants, variables modified, variables used and pattern tables in the PKB.
While/If Procedures	Contains 1 procedure, each with 1 container statement which contains a basic block. Basic code blocks are used to ensure that the different possible locations of the container statement within the procedure is accounted for.	Ensures the correct extraction and population of parent table. Verify the correctness of `Next` ,`Next*`, `Affects` and `Affects*` tables in container statements. Also ensures the variables modified and variables used tables account for container statements. All interactions mentioned in the basic procedure are also tested.

Nested While/If Procedures	Contains 1 procedure, each with 2 container statements nested in each other, containing multiple basic code blocks. All variations of procedure are tested (e.g. while in if, while in else, while in while).	Ensures the correct extraction and population of `Parent*` table. Verify the correctness of `Next` ,`Next*`, `Affects` and `Affects*` tables in nested statements. All interactions mentioned in the basic procedure are also tested.
Complicated procedures	Contains 1 procedure, each with multiple levels of deep nesting.	Ensuring the correct extraction and population of all tables for deep nesting. All interactions mentioned in the basic procedure are also tested.
Multiple procedures	Contains multiple procedures, covering various code types, including all mentioned above. Additionally, the procedures would contain call statements.	Ensures the correct extraction and population of `Calls` and `Calls*` tables. All interactions mentioned in the basic procedure are also tested across multiple procedures.

Our implementation of this integration testing will be done by parsing a SIMPLE source in the parser, extracting all relationships in the design extractor, and populating the `PKBStorages` in the `PKB` with these relationships.

Next, we will call the `OnDemandExtractor` through the `PKB` to populate the `RelationshipCache` tables to simulate on demand extraction by the PQL.

We will then verify the correctness by ensuring every table in the `PKB` is populated with the correct data. This is done by creating expected `PKB` table stubs, and then using the equal to comparison operator to compare the actual `PKB` tables with the expected tables. A list of the tables we check is in the test example below.

Test Purpose

Ensure the extraction and population of `PKB` tables by the `SourceParser` is correct for nested if-while procedure.

Required test input: nestedIfWhileElseBottomTest

```
    procedure proc1 {
1.      if ( var1 > 1) then {
2.         print var1;
3.         read var2;
4.         var2 = var2 * 3;
5.         while ( var2 >= 2) {
6.             print var3;
7.
```

```
 8.              read var4;
 9.              var5 = 1 + var6;
                 }
10.          } else {
11.              print var6;
12.              read var7;
13.              var8 = 1 / var9;
14.          }
         }
```

Figure 30: nestedIfWhileElseBottomTest input SIMPLE code

Expected test result

Every table in the PKB is equal to the expected table. Namely, we verify the correctness of the following PKBStorage:

1. BasicStorage
   procStore, varStore, constStore
2. StatementStorage
   stmtStore
3. RelationshipStorage
   fStore, fStarStore, pStore, pStar, uProcStore, uStmtStore, mProcStore, mStmtStore, cStore, cStarStore, nStore, wPatSore, iPatStore
4. AssignPatternStorage
   aPatStore
5. RelationshipCache
   nStarCache, aCache, aStarCache *(nBipCache, nBipStarCache, aBipCache, aBipStarCache)**

*Extensions

PKB-QPS Integration Test case

PKB-QPS integration test was conducted mainly in *black-box* testing design approach, where QPS would be given a PQL query string and is expected to evaluate correctly with a PKB stub of a sample program to retrieve the relevant data.

The test cases were generated based on 4 attributes:

- **Number of clauses**
  Single, Multiple clauses were tested
- **Parameters**
  Constants only, synonyms only, integers only, mix of different types were tested
- **Type of relationships**
  Coverage of all relationship types of Such-That clauses (Follows/Follows*/Parent/Parent*/Uses/Modifies/Calls/Calls*/Next/Next*/Affects/Affects*) and pattern clause (Assign/While/If)
- **Order of Clauses**

The order of clauses was random, with both cases of `<pattern Clause> <such that clause> <with clause>` **OR** `<such that clause> <pattern clause> <with clause>`

Test purpose

The purpose is to test that the Query Processor System can consult the PKB appropriately during the evaluation of the Query Object which was generated from a PQL query String.

Required test input

Component(s) Tested: `PKB`, `QueryEvaluator`, `QueryOptimiser`, `QueryPreprocessor`

Input PQL query:

```
assign a1, prog_line n1; Select a1 such that Parent(8,a1) pattern a1("flag", _) with n1 = a1.stmt#
```

Expected test result

We will ensure that the answer generated from the `QueryEvaluator` matches our expected answer.

ANSWER: {"9"}

Other Requirements

A `PKB` stub populated with sample program stated in *Section 1.1*.

2.3. System testing

The system test cases are designed systematically. The test suites used for system testing and their descriptions are listed below.

Test suite 1: The source contains a single procedure with each type of statement included. It allows keywords such as "procedure" to be used as variable names. The queries focus on different combinations of arguments in such that and assign pattern clauses introduced in Iteration 1 and the overlapping of synonyms within a query.

Test suite 2: The source contains a single procedure with multiple nesting of while and if statements and includes complex conditional expressions used in those container statements. The queries test the correctness of the Follows and Parent relationships and their transitive based on the nesting of container statements in the source.

Test suite 3: The source contains a single procedure of multiple while and if statements arranged one after another with an additional nesting of a container statement at different positions of the parent container. The queries test the evaluation of a combination of one such that and one assign pattern clause as allowed in Iteration 1. Both source and queries test for whitespace correctness.

Test suite 4: The source contains multiple procedures with call statements. The queries test the correctness of selecting BOOLEAN and a tuple as the result.

Test suite 5: The source contains multiple procedures with call statements. Each query contains multiple such that/pattern/with clauses and test the correctness of those clauses which are introduced in Iteration 2.

Test suite 6: The source and queries are used for stress testing query evaluation based on what is implemented in Iteration 2. Multiple clauses with the same overlapped synonyms are chained together to form a query. This is to test the efficiency of the synonym table used by `QueryEvaluator` to evaluate the clauses.

Test suite 7: The source and queries test the correctness of Affects relationship and its transitive. The handling of semantic errors in queries is also tested.

Test suite 8: The source and queries test for the correctness of new relationship types as introduced in the Extension.

Test suite 9: Similar to test suite 6, the source and queries are used for stress testing query evaluation based on what is implemented in Iteration 3. Emphasis is placed on stress testing the speed of evaluating Next*, Affects and Affects* relationship types since they cannot be pre-computed.

Throughout these test suites, the ability to parse and validate SIMPLE source codes and PQL queries are checked. With reference to the respective grammar rules in the wiki, a list of syntax to be tested is generated. SIMPLE source codes and PQL queries are then written to ensure that the list of syntax is being tested by at least one SIMPLE source code or one PQL query.

Next, we focus on how queries' results can be affected by the structure of the SIMPLE source code. An example is the nesting of container (while and if) statements in test suite 2. A SIMPLE source code with multiple levels of nesting is created, where queries are created to focus on the testing for the Parent and Follows relationships.

Lastly, queries are created to focus on the testing of queries with multiple combinations of such that, pattern and with clauses where synonyms involved may or may not overlap at different parts of the quey.

As a result, the list of syntax that is being tested for SIMPLE source codes and PQL queries, and the different arrangements and positions of nested container statements is produced in *Appendix B: List of Syntax for System Testing*. The files used for each test suite is in *Appendix B: Files used in System Test*.

We provide two sample test cases in `AutoTester` format below. Both PQL queries refer to the SIMPLE source code in Figure 2.

Sample test case #1

```
1. 1 – Multiple combinations of clauses
2. stmt s; while w;
3. Select s such that Parent(s, _) pattern w(“flag”, _) with 1 = 1
4. 5, 8, 12, 15, 20
5. 5000
```

Figure 31: Sample test case #1 code

Sample test case #2

```
1. 2 - Overlapping of synonyms
2. stmt s; assign a; variable v;
3. Select s such that Modifies(s, v) pattern a(v, _"sum"_)
4. 1, 11, 13, 16, 17
5. 5000
```

Figure 32:Sample test case #2 code

3. Extensions to SPA

Our team will be doing Extension Version B: General case of `CFGBips` and `AffectsBip`.

3.1. Implementation Changes to SPA Components

Source Parser

For the extension, no changes are required to be made to the `SourceParser` as the Abstract Syntax Tree generated is sufficient for the extraction of all extension relationships.

Design Extractor

The main changes to the SPA components for the extension are in the Design Extractor. We will be doing the extraction of all relationships inside the `OnDemandExtractor`. Thus, we must ensure that the algorithm used is efficient and able to run at runtime.

Since the new relationships in the extension are cross-procedural, we now have to consider the effect of call statements.

`NextBip` and `NextBip*`

When extracting `NextBip`, **three** additional information about the procedures in the program are stored. They are:

1. The first line of the procedure
2. The last line(s) of the procedure
3. The list of lines in the procedure

This information is stored as a struct `{ firstLine, lastLines, lines }` mapped to each procedure.

The first two are performed before the extraction. The first line of a procedure can be obtained trivially through the AST, which can be retrieved by a PKB API call. The last line(s) of a procedure is taken by looking up the PKB's `Next` storage. Program lines with no next lines are the last lines of a procedure. This is also true for while statements with only one next line.

Extraction of `NextBip` is performed one procedure at a time. The order of the procedures is determined by their call dependency. Procedures that do not call other procedures are prioritised first, and procedures that are not called by any other procedure are prioritised last. A possible ordering of procedures with reference to the sample SIMPLE program is `PrintAnswer → FindMaxSumOfDigits → b → a → c → d → e → f`. The ordering is done by checking the `Calls*` relationships between procedures.

For each procedure, we begin a DFS starting from the `firstLine` of that procedure. The next line to traverse to is determined by the `Next` relationship stored in the `Next` storage. The lines in the procedure are also added into the procedure's `lines` during the traversal.

The NextBip relationship is mostly the same as of the Next relationship. Thus, it stores the same relationship as in the Next storage during the traversal in most cases. However, when encountering a call statement in the procedure during the traversal, we instead do the following two actions:

1. Store the NextBip relationship between the call statement to the firstLine of the called procedure
2. Store the NextBip relationship(s) between the lastLines of the called procedure to the next line of the call statement as in the normal Next relationship

Using this strategy, the entire NextBip relationship can be achieved through the traversal. Additionally, we would also copy all the lines in any called procedures into the current procedure's lines.

For NextBip*, the extraction process is simply a modification of the transitive closure in the extraction of the basic Next*. The only difference is again, we extract the relationship one procedure at a time in the same procedure order. Also, upon encountering a call statement, just like in NextBip, we perform the same two actions to link the program lines to the next line across procedures.

During the backtracking of the DFS for call statements, we will instead copy all the program lines stored in lines of the called procedure to the caller statement as well. The rest of the extraction will follow quite similarly to that of a Next* extraction.

AffectsBip and AffectsBip*

Since our original traversal algorithm of Affects and AffectsStar was implemented to traverse the Abstract Syntax Tree instead of a CFG, we decided to modify it to fit the requirements of the extension. One particular benefit of doing this is that we can extract the NextBip relationships and AffectsBip relationships independently and this can improve the runtime of the system if only one type of query is specified.

Thus, the algorithm used for AffectsBip extraction is similar to that of the basic Affects relationship as mentioned in *Section 1.3.2*. The main difference is that when we encounter a Call node, we ignore all variables used and modified in the node but instead call the function to traverse the AST of the procedure called.

As for AffectsBipStar, the original method for AffectsStar was to make use of the transitive closure property of this relationship to extract this relationship. However, in the extension, this transitive closure property no longer applies due to the inter-procedural calls. Thus, the method we are using to extract this relationship is to keep an ordered vector of all affects relationship for every independent CFGBip and we back-traverse this vector to get the AffectsStarBip relationship. Every program line in this vector is numbered to account for

inter-procedural calls which will result in the same program line being traversed multiple times.

PKB

As all cache tables in the PKB are created using inheritance from the main `RelationshipCache` class, the changes required for the extension is simply to create 4 new tables that extend from the `RelationshipCache` class. These for tables are `nBipCache`, `nBipStarCache`, `aBipCache` and `aBipStarCache`, each responsible for storing the relationships of a particular type.

Query Pre-processor

With reference to Figure 14, NEXTBIP, NEXTBIP_T, AFFECTSBIP, AFFECTSBIP_T are added to `SUCH_THAT_REL_TYPE` enumeration.

As explained in "Query Pre-processor: Validation of Argument" sub-section of Section 1.5.4, new relationship types are easily handled by mapping the new relationship types to `vector<set<ARGUMENT_TYPE>>` and `vector<set<DESIGN_ENTITY>>` for validating `SYNONYM` argument type. Figure 33 shows how these vectors of sets look like.

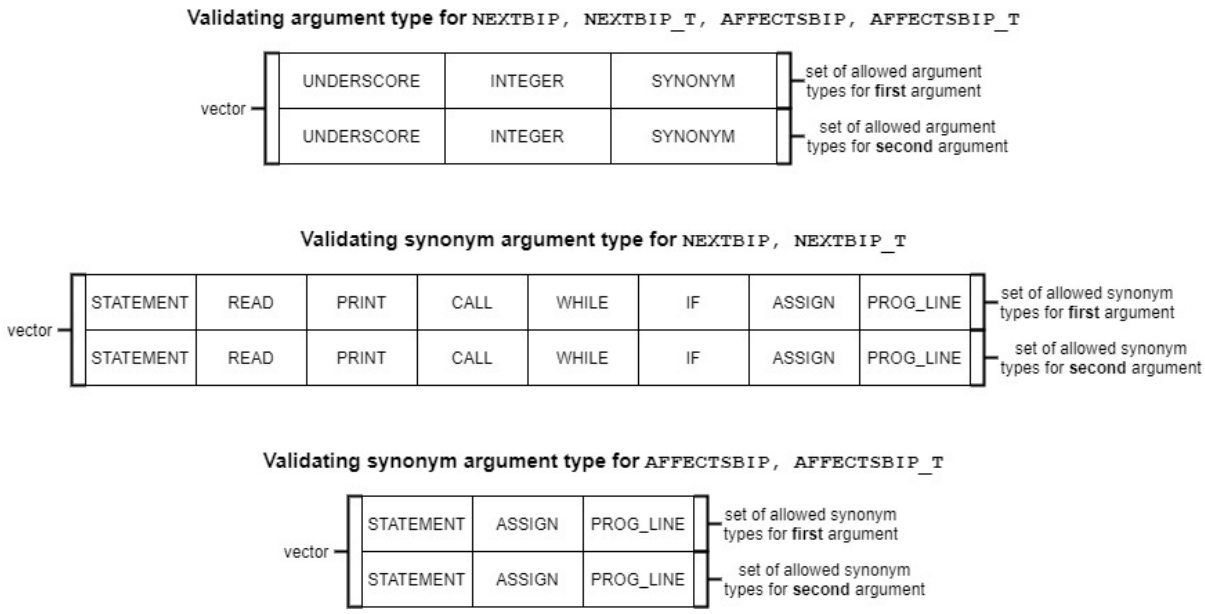


Figure 33: Validating argument and synonym argument type for new relationship types

Query Optimiser

For optimisation of the 4 new relationship types (`NextBip`, `NextBip*`, `AffectsBip`, `AffectsBip*`), the changes required for the extension would simply be expanding the `OptimiserUtility` to include the specific priority ranking of these new relationships, as already implemented in *Section 1.5.2 Query Optimisation* (Query Optimisation strategy)

Query Evaluator

For evaluation of the 4 new relationship types (`NextBip`, `NextBip*`, `AffectsBip`, `AffectsBip*`), the changes required for the extension would simply be expanding the

SuchThatClauseEvaluator to handle SuchThatClause of the four new RELATIONSHIP_TYPE namely, NEXT_BIP, NEXT_BIP_STAR, AFFECTS_BIP, AFFECTS_BIP_STAR.

The evaluation of suchThatClause of these newly added RELATIONSHIP_TYPE would be the similar to the evaluation of suchThatClause of the RELATIONSHIP_TYPE (NEXT, NEXT_T, AFFECTS, AFFECTS_T) respectively, which was implemented in iteration 2, as described in *Section 1.5.3 Query Evaluation* (Evaluation of suchThatClause).

3.2. Abstract API

Design Extractor API	Description
VOID extractNextBip(&CACHE cache)	The RelationshipCache will contain a static OnDemandExtractor object and call this API to populate the nBipCache
VOID extractNextBipStar(&CACHE cache)	The RelationshipCache will contain a static OnDemandExtractor object and call this API to populate the nBipStaCache
VOID extractAffectsBip()	The RelationshipCache will contain a static OnDemandExtractor object and call this API to populate the aBipCache
VOID extractAffectsBipStar()	The RelationshipCache will contain a static OnDemandExtractor object and call this API to populate the aBipStarCache

Testing

Our team carried out testing of the extension in 2 parts.

First, we do integration testing of the OnDemandExtractor and PKB to verify correctness of the extraction and integration of the two components.

There are 5 different test cases for this section,

- Basic Test
- While Test
- If Test
- Nested Test
- Complicated Test

The testing method used here is similar to that mentioned in Section 2.2. Below is the SIMPLE source for the While Test.

```
    procedure A {
1.      while(x==1) {
2.          x = y +1;
3.          x = x +1;
```

```
4.          y = 2;
5.          call C;
        }
    }

    procedure B {
6.      while(x==1) {
7.          call C;
8.          read y;
        }
9.      call A;
    }

    procedure C {
10.     y = x + 1;
    }
```

Figure 34: While Test input SIMPLE code

For system testing, we will also be focusing on the correctness of the new relationships as well as the handling of queries that contain such that clauses with these relationship types. The snippet of SIMPLE code and PQL queries shown below are taken from system test suite 8. This is part of the comprehensive system test plan found at *Section 2.3*.

```
    procedure C {
1.      d = a;
2.      a = b;
3.      b = c;
4.      c = d;
5.      call D;
    }

    procedure D {
6.      call E;
7.      while (i > 0) {
8.          i = 100;
9.          call Q2;
        }
10.     if (i > 0) then {
11.         call Q2;
12.         i = 2000;
        } else {
13.         i = i + 20;
14.         a = 10;
15.         call Q3;
16.         i = i - a;
        }
    }
```

Figure 35: A section of SIMPLE code from system test suite 8

```
11 - NextBip* (synonym, INTEGER)
assign a;
Select a such that NextBip*(a, 4)
1, 2, 8, 9, 10, 11, 15, 19, 20, 21, 23, 26, 27, 28, 29, 32, 33, 35, 38, 39
5000
```

Figure 36: Sample query #11 from system test suite 8

```
29 - AffectsBip (synonym1, synonym2)
assign a1, a2;
Select a1 such that AffectsBip(a1, a2)
1, 2, 8, 9, 10, 11, 15, 19, 20, 21, 23, 26, 27, 28, 29, 32, 33, 35, 38,
39
5000
```

Figure 37: Sample query #29 from system test suite 8

Part 2 – Project management

4. Project Planning

In every project, planning is crucial to ensure its success. Prior to commencing the writing of code for the SPA program, it was important to first decide on the work allocation for each team member. In addition, while individual members may be involved in separate tasks, it is beneficial to be aware of the progress of other team members as well.

In this section, we will document our work allocation, project timeline and our weekly meetings as part of our effort to achieve good planning for our project.

4.1. Work Allocation

Team members were divided across the three main components in the SPA program based on their preferences, with some involved in more than one component. By having a few of our team members managing two different components, they would be in a better position to understand the needed APIs necessary for the interactions between the components once the components are actually being integrated together.

The component(s) that the team member is working on may not be fixed, and can possibly change after each iteration.

Regarding the allocation for the testing of the program, in general, team members who are tasked with a particular component(s) of the program will subsequently conduct unit and integration testing for their respective component(s).

4.1.1. Work Allocation by Team Member

The tables below show the work allocation for the SPA program and the testing respectively.

Team Members	Components	Allocation
Ang Song Yi [QPS IC] [System Test IC]	**Query Preprocessor**	Implement parsing and validation of PQL queries including the new relationship types in Extension Create abstractions and data structures to represent a query, query element, query clause and query object
Chan Wa Wai [SP IC]	**Tokenizer**	Translate original SIMPLE source program into tokens Implement check for unknown characters
	Source Processor	Implement parsing and syntax validation of procedure and all statements in program Build AST based on parsed statements Build AM based on call statements

	Design Extractor	Extract C/C* relationship information from the AM
	On Demand Extractor	Extract A/A* relationships on demand
Clement Cheng [Team Leader]	**Source Processor**	Implement parsing and validation of arithmetic and conditional expressions
	PKB	Create APIs for retrieving design entities and relationships from their respective tables for the Query Evaluator
	Query Evaluator	Implement evaluation of such that, pattern and with clauses Implement synonym table to store possible values for the different synonyms in a query Optimise synonym table
	Query Optimiser	Optimise evaluation of queries by categorising query clauses into different groups
	On Demand Extractor	Extension: extraction of NextBip and NextBip* relationship
Hope Leong [SP 2IC] [PKB 2IC]	**Design Extractor**	Extract design abstractions and F/F*/P/P*/U/M/N relationships from the AST using DFS Populate PKB correctly with all the extracted design abstractions
	On Demand Extractor	Extract N* relationships based on N table in PKB on demand. Extension: extraction of AffectsBip and AffectsBip* relationship
Oh Jun Ming [QPS 2IC]	**Query Evaluator**	Implement evaluation of F/F*/P/P* relationships Implement validation of parameters of PQL clauses
	Query Optimiser	Optimise evaluation of queries by categorising query clauses into different groups Order and prioritise clauses in each group
Wong Kok Ian [PKB IC] [System Test 2IC]	**PKB**	Design all tables of appropriate data structures for storing of design abstractions Create APIs for the DE to add design entities and relationships into their respective tables Create APIs to interact with the ODE

Testing Components	Team Members	Allocation
Source Parser	Chan Wa Wai	Create unit test for various sub parsers using tokenizer stub
Tokenizer	Chan Wa Wai	Create unit test to ensure the strings/ char are tokenized properly. Create unit test to test if the generated AST is correct

PKB	Wong Kok Ian	Create unit test cases to check that the PKB's and its sub-components' methods populate the PKB correctly by comparing with stubs
Query Preprocessor	Ang Song Yi	Create unit test cases to cover the possible variations of PQL queries
Query Evaluator	Clement Cheng	Create unit test cases to cover the different combinations for the parameters in the queried relationship
	Oh Jun Ming	
SP-PKB Integration	Hope Leong	Create source program to test if the PKB is populated properly
	Chan Wa Wai	
PKB-QPS Integration	Oh Jun Ming	Create integration test cases for PKB stub and PQL queries
System	Ang Song Yi [IC]	Create 9 sets of system test cases for SIMPLE source programs and PQL queries
	Wong Kok Ian	

4.1.2. Tasks Conducted during SPA Project

The detailed list of all tasks carried out during the duration of the project is recorded in the tables below. Additionally, the following colour codes and legends are used in the table:

Colour Codes
Source Processor
Design Extractor
Program Knowledge Base
Query Processing Subsystem
Testing
Documentation

Legend
[P] – **P**lanning/Design Activity
[C] – **C**oding/Implementation Activity
[T] – **T**esting Activity
[D] – **D**ocumentation Activity

Tasks	Activity ID	Activities
1 Implement parsing of SIMPLE source	1.1 [P]	Research Recursive Descent parsing algorithm
	1.2 [P]	Design APIs for Source Parser
	1.3 [C]	Implement tokenization
	1.4 [C]	Implement basic skeleton for Source Parser
	1.5 [C]	Implement parsing and validating of basic statements (read, print, call)
	1.6 [C]	Implement parsing and validating of assign statements
	1.7 [C]	Implement parsing and validating of container statements and statement lists
	1.8 [P]	Research on Shunting Yard Algorithm
	1.9 [C]	Implement parsing and validating of arithmetic expressions
	1.10 [C]	Implement parsing and validating of conditional expressions

	1.11 [P]	Research on Abstract Syntax Tree (AST)
	1.12 [P]	Design APIs for AST
	1.13 [C]	Implement building of AST from parsed program
	1.14 [T]	Unit test Tokenizer
	1.15 [T]	Unit test Source Parser
	1.16 [T]	Unit test AST
	1.17 [D]	Document Tokenizer implementation
	1.18 [D]	Document Source Parser implementation for Iteration 1
	1.19 [P]	Research on Adjacency Matrix (AM)
	1.20 [C]	Implement creation of AM
	1.21 [C]	Implement validating for procedures (duplicates)
	1.22 [D]	Document Source Parser implementation for Iteration 2
2 Implement extraction of design entities and relationships from AST	2.1 [P]	Research on tree traversal algorithms
	2.2 [P]	Design APIs for Design Extractor
	2.3 [P]	Design APIs for PKB to store extracted design entities and relationships 1
	2.4 [C]	Implement basic skeleton for Design Extractor
	2.5 [C]	Implement AST traversal using an improvised Depth-First Search
	2.6 [C]	Implement APIs in PKB to store extracted design abstractions from the Design Extractor 1
	2.7 [C]	Extract design entities such as procedure names, variables, constants and statement types
	2.8 [C]	Extract `Follows/Follows*` relationships
	2.9 [C]	Extract `Parent/Parent*` relationships
	2.10 [C]	Extract `Uses` and `Modifies` relationships
	2.11 [C]	Extract pattern from `Assign/While/If` statements
	2.12 [T]	Unit test Design Extractor (attempted)
	2.13 [D]	Document Design Extractor implementation 1
	2.14 [P]	Research on reverse topological order tree traversal
	2.15 [C]	Implement reverse topological order tree traversal
	2.16 [P]	Design APIs for PKB to store extracted design entities and relationships 2
	2.17 [C]	Implement APIs in PKB to store extracted design abstractions from the Design Extractor 2
	2.18 [C]	Extract `Calls/Calls*` relationships
	2.19 [C]	Implement cyclic call check
	2.20 [C]	Extract basic `Next` relationships
	2.21 [C]	Extract advanced `Next` relationships (with conditionals)
	2.22 [D]	Document Design Extractor implementation 2

3 Create PKB data structures and functionalities	3.1 [P]	Research on suitable data structures for storing the different design entities and relationships
	3.2 [P]	Design tables for storing each different design abstraction and corresponding APIs in Iteration 1
	3.3 [C]	Create storage for design entities (procedures, variables, constants and statements)
	3.4 [C]	Create storage for `Follows/Follows*` relationships
	3.5 [C]	Create storage for `Parent/Parent*` relationships
	3.6 [C]	Create storage for `Uses` relationships
	3.7 [C]	Create storage for `Modifies` relationships
	3.8 [C]	Create storage for Assign patterns
	3.9 [C]	Implement APIs to store extracted design abstractions from the Design Extractor 1
	3.10 [C]	Implement APIs to retrieve the stored design abstractions for the Query Evaluator 1
	3.11 [T]	Unit testing for PKB `Follows/Follows*` storage
	3.12 [T]	Unit testing for PKB `Parent/Parent*` storage
	3.13 [T]	Unit testing for PKB `Uses` storage
	3.14 [T]	Unit testing for PKB `Modifies` storage
	3.15 [T]	Unit testing for PKB Pattern storage
	3.16 [D]	Document PKB implementation 1
	3.17 [P]	Design tables for storing each different design abstraction and corresponding APIs in Iteration 2
	3.18 [C]	Create storage for `Calls/Calls*` relationships
	3.19 [C]	Create storage for `Next` relationships
	3.20 [C]	Create cache storage for `Next*` relationships
	3.21 [C]	Create cache storage for `Affects/Affects*` relationships
	3.22 [C]	Implement storing of variables/procedures for read/print/call statements
	3.23 [C]	Implement APIs to store extracted design abstractions from the Design Extractor 2
	3.24 [C]	Implement APIs to retrieve the stored design abstractions for the Query Evaluator 2
	3.25 [C]	Implement referencing to On Demand Extractor (ODE)
	3.26 [C]	Implement APIs in PKB to store extracted design abstractions from the ODE
	3.26 [T]	Unit testing for PKB `Calls/Calls*` storage
	3.27 [T]	Unit testing for PKB `Next` storage
	3.28 [T]	Unit testing for PKB cache
	3.29 [D]	Document PKB implementation 2

	3.30 [P]	Design data structures to store different design abstractions in a more generalised manner
	3.31 [C]	Implement generalised Basic Storages for procedures, variables and constants
	3.32 [C]	Implement generalised Relationship Storages
	3.33 [C]	Implement generalised Relationship Caches
	3.34 [C]	Implement generalised Hybrid Storages for statements and assign patterns
	3.35 [C]	Update APIs for Design Extractor to store design abstractions into the generalised tables
	3.36 [C]	Update APIs for Query Evaluator to retrieved stored design abstractions from the generalised tables
	3.37 [C]	Implement Index Table to map string values of procedures, variables, constants and statement numbers to integers
	3.38 [C]	Update PKB storages to store indices instead of strings
	3.39 [T]	Unit testing for PKB Basic Storage
	3.40 [T]	Unit testing for PKB Relationship Storage
	3.41 [T]	Unit testing for PKB Cache
	3.42 [T]	Unit testing for Index Table
	3.43 [D]	Document PKB implementation 3
4 Implement ODE	4.1 [P]	Research on efficient traversal algorithms to extract `Next*` relationships
	4.2 [P]	Design APIs for extracting `Next*` relationships on demand
	4.3 [C]	Implement extraction of `Next*` relationships
	4.4 [P]	Design APIs for PKB to store on demand extracted design entities and relationships
	4.5 [C]	Implement APIs in PKB to store on demand extracted design abstractions from the ODE
	4.6 [T]	Unit testing for `Next*` extraction
	4.7 [D]	Document ODE implementation 2
	4.8 [P]	Design algorithm to extract `Affects` relationships
	4.9 [C]	Implement extraction of `Affects` relationships
	4.10 [C]	Design algorithm to extract `Affects*` relationships
	4.11 [C]	Implement extraction of `Affects*` relationships
	4.12 [T]	Unit testing for `Affects` extraction
	4.13 [T]	Unit testing for `Affects*` extraction
	4.14 [D]	Document ODE implementation 3
5 Implement	5.1 [P]	Research on parsing algorithms for Query Preprocessor
	5.2 [P]	Design APIs to process input PQL query
	5.3 [C]	Implement parsing and validation of declarations
	5.4 [C]	Implement parsing and validation of queries with only select clause

	5.5 [C]	Create abstractions and data structure to represent query clause and query object
	5.6 [C]	Implement parsing and validation of queries with one such that **or** one pattern clause
	5.7 [C]	Implement parsing and validation of queries with one such that **and** one pattern clause
	5.8 [T]	Unit testing for Query Preprocessor (only select clause)
	5.9 [T]	Unit testing for Query Preprocessor (either one such that or one pattern clause)
	5.10 [T]	Unit testing for Query Preprocessor (one such that and one pattern clause)
	5.11 [D]	Document Query Preprocessor implementation 1
	5.12 [P]	Design APIs for validating query clause arguments
	5.13 [C]	Split Query Clause class into subclasses
	5.14 [C]	Implement whitelist validating of such that clause arguments
	5.15 [C]	Implement whitelist validating of pattern that clause arguments
	5.16 [C]	Implement parsing of all such that clause types for Iteration 2
	5.17 [C]	Implement parsing of all pattern clause types
	5.18 [C]	Implement parsing of select tuple
	5.19 [C]	Implement parsing of select Boolean
	5.20 [C]	Implement parsing of “and” keyword
	5.21 [C]	Implement whitelist validating of with clause arguments
	5.22 [C]	Implement parsing of with clause
	5.23 [T]	Unit testing for parsing new select clause types
	5.24 [T]	Unit testing for single query clause
	5.25 [T]	Unit testing for multiple query clauses
	5.26 [D]	Document Query Preprocessor implementation 2
	5.27 [P]	Design APIs to tokenise of PQL queries
	5.28 [C]	Implement tokenising of PQL queries
	5.29 [C]	Categorise query errors into syntax or semantic errors
	5.30 [T]	Unit testing for updated Query Preprocessor
	5.31 [D]	Document Query Preprocessor implementation 3
6 Implement evaluation of	6.1 [P]	Design APIs for Query Evaluator to evaluate Query Object from Query Preprocessor
	6.2 [P]	Design APIs to retrieve stored design abstractions from PKB 1
	6.3 [C]	Implement evaluation of queries with only select clause
	6.4 [C]	Implement validation of parameters of PQL clauses
	6.5 [P]	Design data structure and APIs for Synonym Table that manages and stores possible values of synonyms in the query

	6.6 [C]	Implement using Synonym Table to merge different possible synonym values
	6.7 [C]	Implement APIs in PKB to retrieve stored design abstractions for Query Evaluator 1
	6.8 [C]	Implement evaluation of Follows/Follows* such that clause
	6.9 [C]	Implement evaluation of Parent/Parent* such that clause
	6.10 [C]	Implement evaluation of Use such that clause
	6.11 [C]	Implement evaluation of Modifies such that clause
	6.12 [C]	Implement evaluation of Assign pattern clause
	6.13 [C]	Implement selecting answer of select clause from Synonym Table
	6.14 [T]	Unit testing of `Follows/Follows*` such that clause evaluation
	6.15 [T]	Unit testing of `Parent/Parent*` such that clause evaluation
	6.16 [T]	Unit testing of `Use` such that clause evaluation
	6.17 [T]	Unit testing of `Modifies` such that clause evaluation
	6.18 [T]	Unit testing of assign pattern clause evaluation
	6.19 [D]	Document Query Evaluator implementation 1
	6.20 [P]	Design APIs to retrieve stored design abstractions from PKB 2
	6.21 [C]	Implement APIs in PKB to retrieve stored design abstractions for Query Evaluator 2
	6.22 [C]	Split Query Evaluator class into subclasses
	6.23 [C]	Implement evaluation of `Calls/Calls*` such that clause
	6.24 [C]	Implement evaluation of While pattern clause
	6.25 [C]	Implement evaluation of If pattern clause
	6.26 [C]	Implement evaluation of `Next/Next*` such that clause
	6.27 [C]	Implement evaluation of `Affects/Affects*` such that clause
	6.28 [C]	Implement evaluation of While pattern clause
	6.29 [C]	Implement evaluation of If pattern clause
	6.30 [C]	Implement evaluation of with clause
	6.31 [C]	Implement evaluation of clauses by groups
	6.32 [C]	Implement selecting answers for new select types
	6.33 [T]	Unit testing of `Calls/Calls*` such that clause evaluation
	6.34 [T]	Unit testing of `Next/Next*` such that clause evaluation
	6.35 [D]	Document Query Evaluator implementation 2
	6.36 [P]	Design APIs to generalise the different Query Clause Evaluators
	6.37 [C]	Implement generalised Such That Clause Evaluator
	6.38 [C]	Implement generalised Pattern Clause Evaluator
	6.39 [T]	Unit testing of `Affects/Affects*` such that clause evaluation
	6.40 [T]	Unit Testing of While/If pattern clause evaluation

	6.41 [T]	Unit Testing of With clause evaluation
	6.42 [D]	Document Query Evaluator implementation 3
7 Optimisations	7.1 [P]	Research and plan query optimisation strategies
	7.2 [P]	Design APIs for Query Optimiser
	7.3 [C]	Implement basic skeleton for Query Optimiser
	7.4 [C]	Create Query Group classes to contain group of related clauses
	7.5 [C]	Implement grouping of clauses with no synonyms
	7.6 [C]	Implement grouping of clauses with select synonym(s) and other interconnected clauses
	7.7 [C]	Implement grouping of clauses with no select synonym and other interconnected clauses
	7.8 [P]	Design APIs for optimised Synonym Table
	7.9 [C]	Implement Synonym Table Manager to manage intermediate synonym tables
	7.10 [C]	Implement removal of values of redundant synonyms
	7.11 [C]	Implement merging of intermediate tables
	7.12 [P]	Design APIs to use synonym values from Synonym Table to scope data retrieved from PKB
	7.13 [C]	Implement scoping of retrieved data from PKB for all evaluators
	7.14 [D]	Document Query Optimiser
	7.15 [D]	Document Synonym Table Manager
	7.16 [P]	Design data structures for more optimised Synonym Storage
	7.17 [C]	Implement single synonym table
	7.18 [C]	Implement double synonym table
	7.19 [C]	Integrate new Synonym Storage optimisations
	7.20 [C]	Prioritise queries in query groups with no synonyms
	7.21 [C]	Prioritise queries in query groups with synonyms
	7.22 [C]	Update PKB to use integer indexing instead of string values of design abstractions
	7.23 [C]	Update Query Preprocessor to use integer indexing instead of string values of synonyms
	7.24 [C]	Update Synonym Storage to store values and synonyms as integer indices
	7.25 [T]	Unit testing of Query Optimiser
	7.26 [T]	Unit testing of improve Synonym Storage
	7.27 [D]	Document updated Query Optimiser
	7.28 [D]	Document updated Synonym Storage
8 Extensi	8.1 [P]	Design APIs in ODE to extract new relationships in extension
	8.2 [P]	Design data structure in PKB to store new relationships
	8.3 [P]	Design APIs for QP to process clauses with new relationships

	8.4 [P]	Design APIs for QO to optimise clauses with new relationships
	8.5 [C]	Implement APIs in PKB to store relationships extracted from ODE
	8.6 [C]	Implement APIs in PKB to retrieve stored new relationships for QE
	8.7 [C]	Implement APIs in QP to process clauses with new relationships
	8.8 [C]	Implement APIs for QO to optimise clauses with new relationships
	8.9 [C]	Implement extraction of `NextBip/NextBip*` in ODE
	8.10 [C]	Implement extraction of `AffectsBip/AffectsBip*` in ODE
	8.11 [D]	Document Extension
9 Others	9.1 [T]	Integration test SP-PKB 1
	9.2 [T]	Integration test PKB-QPS 1
	9.3 [T]	Create list of syntax to test against for system testing 1
	9.4 [T]	Create System Test Suite 1 (Single Procedure)
	9.5 [T]	Create System Test Suite 2 (Nested While-If)
	9.6 [T]	Create System Test Suite 3 (While-If Arrangements)
	9.7 [T]	Resolve test bugs 1
	9.8 [D]	Document testing 1
	9.9 [D]	Other documentation 1
	9.10 [T]	Integration test SP-PKB 2
	9.11 [T]	Integration test PKB-QPS 2
	9.12 [T]	Create list of syntax to test against for system testing 2
	9.13 [T]	Create System Test Suite 4 (Attribute Reference, Boolean, Tuple)
	9.14 [T]	Create System Test Suite 5 (Multiple Procedures)
	9.15 [T]	Create System Test Suite 6 (Stress Test)
	9.16 [T]	Resolve test bugs 2
	9.17 [D]	Document testing 2
	9.18 [D]	Other documentation 2
	9.19 [T]	Integration test SP-PKB 3
	9.20 [T]	Integration test PKB-QPS 3
	9.21 [T]	Integration test PKB-Extension
	9.22 [T]	Create list of syntax to test against for system testing 3
	9.23 [T]	Update System Test Suite 6 (Stress Test)
	9.24 [T]	Create System Test Suite 7 (Affects)
	9.25 [T]	Create System Test Suite 8 (Extension)
	9.26 [T]	Regression Testing
	9.27 [T]	Resolve test bugs 3
	9.28 [D]	Document testing 3
	9.29 [D]	Other documentation 3

4.2. Project Timeline

The SPA project consists of 3 main iterations. In each of the iteration, we performed various tasks to meet the criteria set out at the end of each iteration. This included planning, implementing and documenting the tasks we have done.

The following table summarises the tasks completed per week.

Week	Tasks
2	Planning and designing of data structures and APIs for the different components (**SP**, **DE**, **PKB**, **QPS**)
3	Further planning and designing of APIs
4	Implementing very basic features of each component **SP** – Tokenizing and parsing of simple statements (read, print) **DE** – Extracting simple design abstractions and relationships (F/F*/P/P*) **PKB** – Creating storages to store design abstractions, APIs for DE and QE **QPS** – Processing and evaluating very simple queries (select clause only)
5	Extending implementations to cover more features in Basic SPA **SP** – Parsing all other statements (assign, while, if) **DE** – Extracting the rest of the relationships (U/M) **PKB** – Finalising data structures and APIs, and unit testing **QPS** – Processing and evaluating the rest of the clauses (such that, pattern)
6	Unit, integration and system testing for all components
Recess	Documenting implementations and design decisions for all components
7	Planning and designing APIs for Advanced SPA requirements
8	Extending implementations to cover features in Advanced SPA **SP** – Parsing of call statements **DE** – Extracting new relationships and on demand (C/C*/N/N*) **PKB** – Creating storages to store new relationships and caches **QPS** – Processing and evaluating other clause variants (Boolean, tuple)
9	Optimising current implementations and testing/documenting **QPS** – Optimise query evaluation process by grouping queries
10	Planning and designing APIs for Advanced SPA requirements **DE** – Extracting remaining relationships (A/A*) on demand and plan extensions **PKB** – Generalising storages **QPS** – Designing a more optimised model for synonym storage
11	Wrapping up Advanced SPA requirements and implementing extensions **DE** – Extracting extension relationships (NB/NB*) **PKB** – Creating storages to store extension relationships, storing values as INT **QPS** – Optimising synonym storage and processing/evaluating extensions
12	Implementing extensions, testing and documenting

	DE – Extracting extension relationships (AB/AB*)

A visual representation of the detailed timeline can be found in *Section 5.2*.

4.3. Team Meetings

To support our project planning, our team conducted weekly sprint meetings over Zoom to discuss about any issues faced with the implementation and progress of our individual components in the SPA program. Thereafter, we would plan for the tasks we aim to complete by the next meeting. The weekly sprints are beneficial to ensure that each team member is on an even footing, and are clear in the direction that that the project is heading towards.

Furthermore, the tasks discussed will be added into the TODO list under out team's GitHub project management tool. Our team also utilised Telegram as a mean for easy informal communication and clarification as we worked on the project.

5. Test Strategy

5.1. Testing Method and Resolution

Two `.bat` files are written to automate system testing. The command to use `AutoTester` given the file paths of SIMPLE source codes and their corresponding PQL queries are added to the `.bat` files. The name of these source code and query files can be found in *Appendix A: Files used in System Testing*. The `.bat` files are then executed, which produces `.xml` files. The `.xml` files are then cross-checked to ensure that all PQL queries are correctly evaluated. The first `.bat` file is called `execute_system_test.bat` and is used to test the main system (without extension). The second `.bat` file is called `execute_system_test_extension.bat` and is used to test the extension.

During integration and system testing, any defect found is first analysed to determine which component is causing the error. Once determined, the person working on the component will be alerted via Telegram and is responsible for fixing the bug causing the defect. Regression testing is then conducted by that person to ensure that fixes in the component does not have any unintended side effects. The average defect resolution time is **1 working day**.

Unit testing is done concurrently with the implementation of the components in order to verify the correctness of the component. After each individual unit testing is completed, we then do integration testing to test the abstract API and relationships between the separate components. As system testing requires a more robust and systematic test plan to ensure maximum coverage, our team had worked on the system test planning since early in the iterations. The result of this planning has been described earlier in the report in *Section 2.3*.

Regression testing is done after the implementation of each new component to ensure that the existing system does not break after the addition before merging into the master branch.

A common technique our team employs to debug is using the visual studio debugger tool to crawl through of the code. This is a very efficient method and has allowed us to have an average defect resolution time of 1 working day.

5.2. Project Tasks and Testing Timeline

The figures below illustrate visually the individual activities and testing carried out by each member of the team, organised by weeks. The colour codes correspond to those used in *Section 4.1.2* for easy reference.

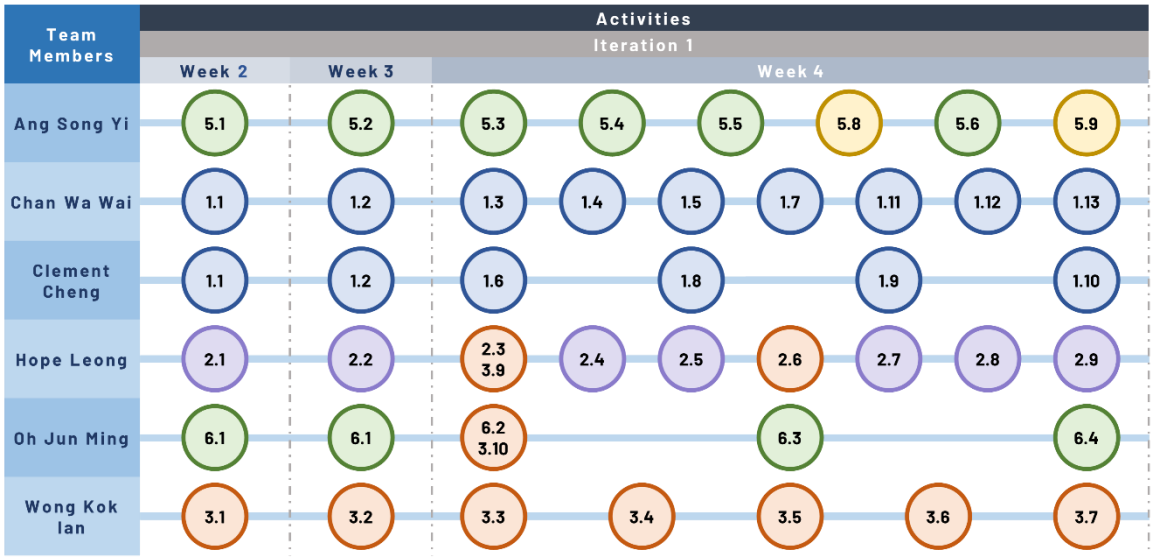


Figure 38a: Project Timeline Weeks 2, 3 and 4

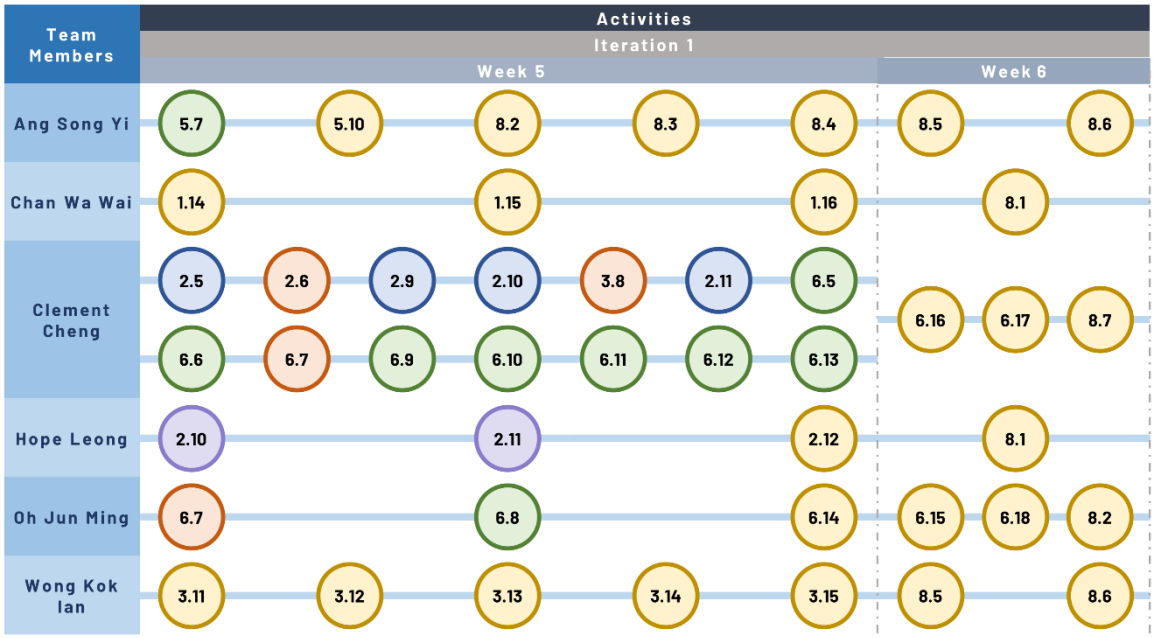


Figure 38b: Project Timeline Weeks 5 and 6

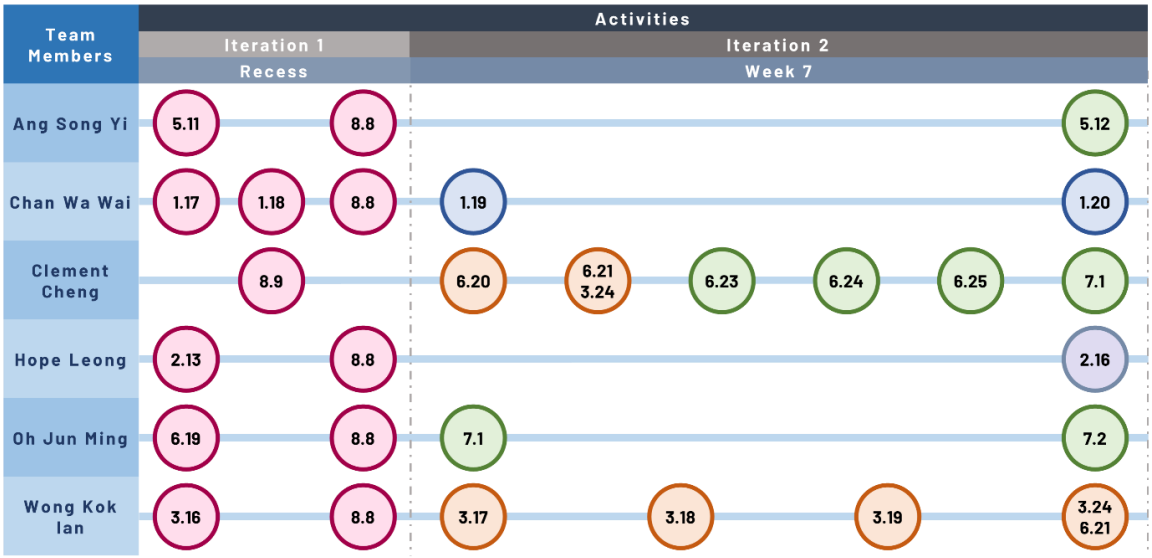


Figure 38c: Project Timeline Weeks Recess and 7

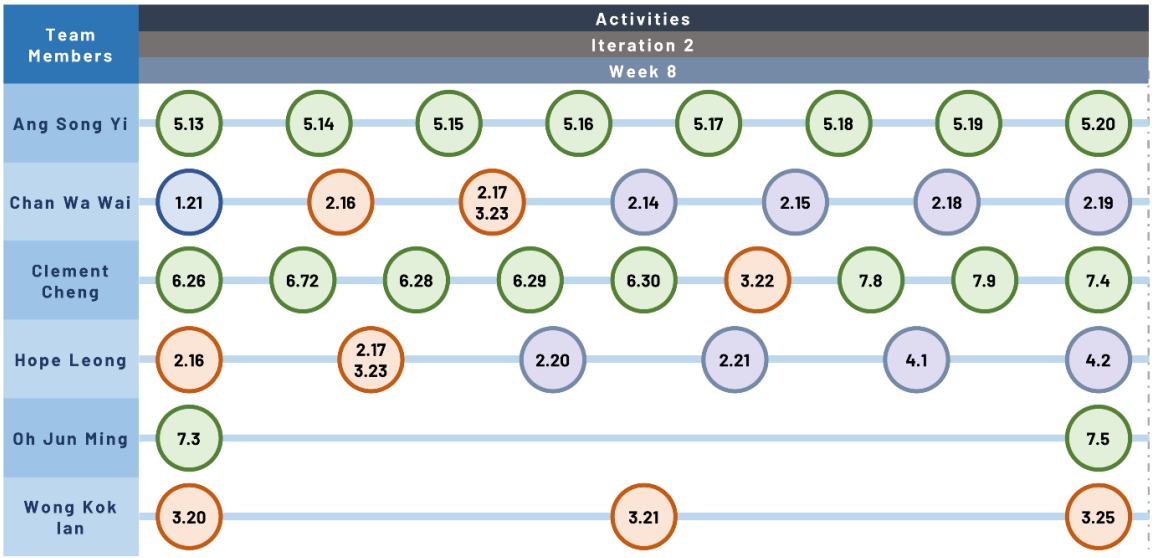


Figure 38d: Project Timeline Week 8

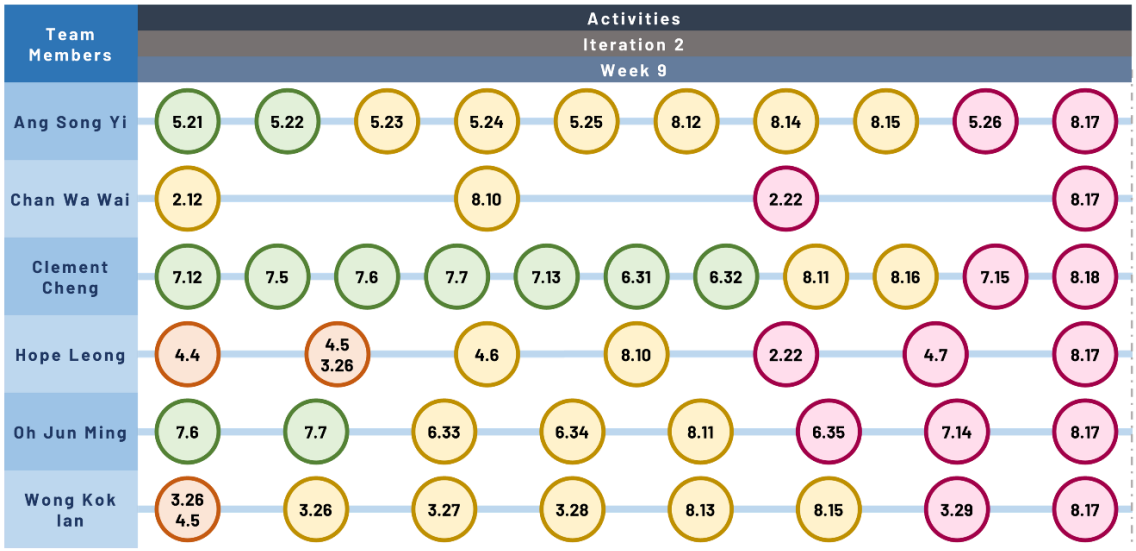


Figure 38e: Project Timeline Week 9

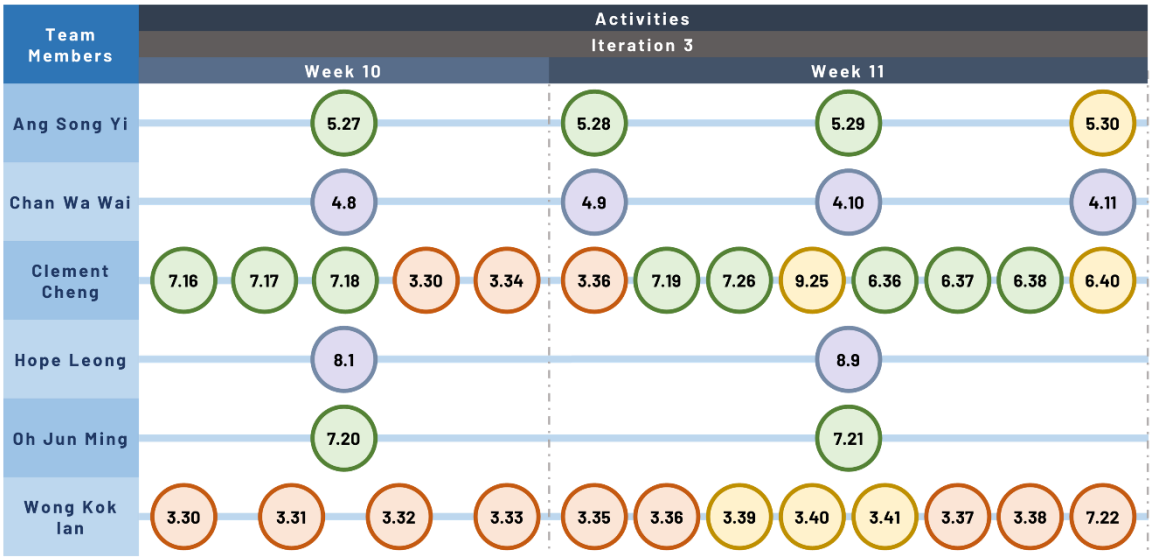


Figure 38f: Project Timeline Week 10-11

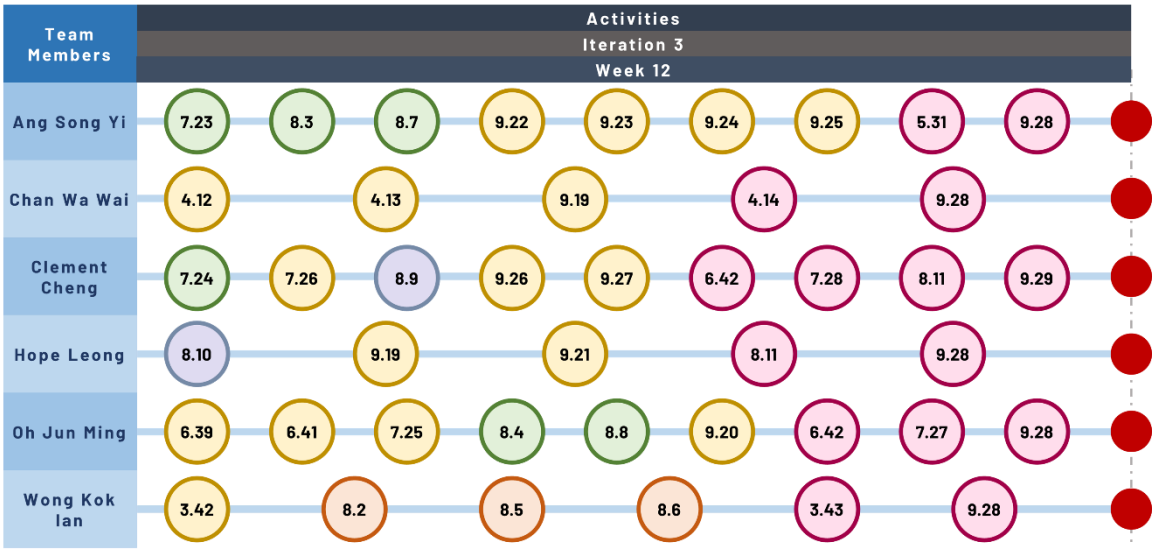


Figure 38g: Project Timeline Week 12 (Final)

6. Coding standards

In our SPA project, our team adhered to several coding standards and good practices as we write our code. The following are some of the actions that our team has taken:

Consistent Naming Conventions

One of the main coding standards we have adhered to in this project is the use of a consistent naming convention. Our team made use of various word boundaries, and this allows the code to become more readable and understandable with reference different entities.

Class names

For class names, we will be adhering to the Pascal Case format where the first letter of each word is capitalized.

```
class OnDemandExtractor {
     . . . .
}
```

Procedure Names

For procedure names, we will be adhering to the Camel Case format where the first letter of each word is capitalized, except the first word. This applies to all API used as well.

```
SYNONYM_INDEX getSynIndex() {
   . . .
}
```

Variables

For variables, we will also be adhering to Camel Case format.

```
VALUE_PAIRS emptyValuePairs;
```

Enumerations

For enumerations, we will be adhering to the Upper Snake format, where are letters are capitalized and separated by an underscore. This allows for easy recognition of enumerations in the code.

```
enum class DESIGN_ENTITY {
   STATEMENT, READ, PRINT, CALL, WHILE, IF, ASSIGN, VARIABLE, CONSTANT,
  PROG_LINE, PROCEDURE, NONE
};
```

Type Definition

For type definitions, we will be adhering to the Upper Snake format, where every letter is capitalized and separated by an underscore.

```
typedef S<STMT_NUM> STMT_NUMS;
```

Conditionals

For conditionals and methods to check conditions of variables, we will be adhering to the Camel case format as well, and the name will typically start with the word "is" or "has".

```
bool isWhile() {
   . . . .
}
```

Organisation of Repetitive Headers and Type Definitions

Our team noticed during our implementation of our code that we had utilised many similar C++ library header files or type definitions. As such, instead of having to constantly enter the same set of headers and type definitions for each file, we created a file, Common.h, that contains those headers and type definitions.

Common include headers are: <string>, <vector>, <list>, <unordered_map>, <unordered_set>, <stack>, <queue>

Common typedefs are: STMT_NUM, PROC_NAME, VAR_NAME, PROG_LINE, PATTERN, CONST, VALUE, INDEX, SYNONYM_INDEX

Thus, for each .cpp file, we would include the Common.h header file, as well as other header files specific to the implementation of the classes in that file. A benefit for containing all of these headers and definitions in a single file is when changing the definition for a certain typedef. For example, if we want to change the type for STMT_NUM from string to int, we just have to change the single typedef of STMT_NUM in the Common.h file, instead of searching through each file in our project to change the definition.

Aside from headers and type definitions, our team also included the common enumerations that are used in many of the components. Two examples are DESIGN_ENTITY and RELATIONSHIP_TYPE.

Documentation and Comments

As this project is being done in groups of 6, it is crucial that our code is readable and understandable to people aside from the person that implemented it. Therefore, one coding standard we are adhering to is the documentation and commenting of code. Procedures that are not self-explanatory are required to have appropriate documentation explaining what the procedure does. All documentation for procedures will be done in the header file for easy access. For complex logic within procedures, our team will also be adding comments regarding what that section of the code does.

Besides having sufficient documentation and comments, our team will also be using the region and endregion pragma to specify blocks of code that can be expanded or collapsed. This helps readers be able to clearly identify different blocks of code within a file.

7. Correspondence of the abstract API with the relevant C++ classes

In our project, we use the same naming conventions in the abstract API and our C++ program mainly for the PKB component and its sub-components. We do so by mapping the abstract API name types to `C++ types` using `typedef`. A few examples are shown below.

Example 1 (PKB):

In abstract API:

VALUES getFirstValuesOf (**RELATIONSHIP_TYPE** rType, **DESIGN_ENTITY** firstValueType, **VAR** secondValue)	Returns the set of indices of values x for Relationship(x, secondValue) where Relationship is of rType and x is of firstValueType.

In C++ program:

```
// Definitions
typedef int VALUE;
typedef unordered_set<VALUE> VALUES;
typedef string VAR;

// Method
VALUES    getFirstValuesOf(RELATIONSHIP_TYPE    rType,    DESIGN_ENTITY
firstValueType, VAR secondValue);
```

Figure 39: PKB getFirstValuesOf Method Code Snippet

Example 2 (StatementStorage):

In abstract API:

INDEX getIndexedVarOf(**DESIGN_ENTITY** stmtType, **INDEX** indexedStmt)	Returns the indexed variable of the indexed statement of stmtType.

In C++ program:

```
// Definitions
typedef int INDEX;

// Method
INDEX getIndexedVarOf(DESIGN_ENTITY stmtType, INDEX indexedStmt);
```

Figure 40: StatementStorage getIndexedVarOf Method Code Snippet

Example 3 (AssignPatternStorage):

In abstract API:

PATTERN getPatternOf (**INDEX** indexedStmtNum)	Returns the pattern associated with the indexed assign statement.

In C++ program:

```
// Definitions
typedef string PATTERN;
typedef int INDEX;

// Method
PATTERN getPatternOf(INDEX indexedStmtNum);
```

Figure 41: AssignPatternStorage getPatternOf Method Code Snippet

Part 3 – Conclusion

8. Reflection

Our team has very strong and capable members who are able to deliver their assigned tasks efficiently. In general, we were able to adhere to our planned project schedule in a timely manner. The weekly meetings we held enabled us to be aware of our other members' progress and gain a better understanding of the implementations and designs of the components we were not working on. This allowed us to better integrate the different components together and help other members who are working on other components.

Yet not all teams are perfect, though we aspire to be one. We have learnt from our experiences in previous iterations and made significant improvements.

For example, with regards to project management, we pushed our commits to GitHub on a regular basis instead of making one big commit. This reduces the chance of potential merge conflicts, and enabled our workflow to be much smoother.

Our team also placed higher emphasis towards the testing of our program. Before committing, we would tend to run our unit and integration testing again to make sure previous functionalities are still working correctly, and we did not break something accidentally. Also, we planned more stress tests to ensure the correctness and efficiency our program to parse big source files and evaluate highly complex queries.

Of course, there are definitely still challenges that we faced in this final iteration of the project.

One issue faced was poor communication of refactoring changes in the code that could potentially indirectly affect other components. An example is `QueryObject` which is populated by `QueryPreprocessor` and used by `QueryOptimiser` and `QueryEvaluator`. With each iteration, the structure of the `QueryObject` changes to adapt to new requirements. During iteration 2, the refactoring changes made were not communicated clearly, resulting in a lot of merge conflicts and failures in regression testing. This was improved in iteration 3, with members communicating their refactoring changes needed in advance so that those working on the affected components can be aware and make the necessary changes. This has led to less merge conflicts and less occurrence of failed regression testing.

In our endeavour to optimise our implementation of query evaluation, our team made considerable changes to our existing code. For instance, the generalisation of the PKB storage structures resulted in massive changes in the APIs. Consequently, major components, such as the Design Extractor and Query Evaluator, which relied heavily on the PKB's APIs have to be replaced. This also led to delays in the other components while waiting for the PKB to finalise the updated storage structure. Our team felt that this could be better managed, but also

understood that there really was not much to be done since the PKB is inherently coupled with multiple components and was not due to poor design.

In conclusion, our team felt that we have certainly took away key experiences from undertaking the SPA project till its end. Working together as a team, and managing a project of considerable size, were clearly no easy feats, and there have been a few setbacks that we faced within the duration of the project. Nevertheless, we took things one step at a time, consistently learning form past mistakes, and affirming ourselves that we will do better the next time. Good communication and proper planning are absolutely essential to ensure success in a team project, and we have worked consistently to achieve them. Through this project, we had the opportunity to learn more about each other and to take advantage of each member's strengths. In the same manner, our team hopes that with the experiences we have gained from this SPA project, we can grow to become more adept in working with or even leading a team at a professional level in our future careers as software engineers.

Appendix

9. Appendix A: SPA

9.1. Tokenizer Symbols to Type Mapping

Token Type	Character/Strings
PROCEDURE (keyword)	“procedure”
READ (keyword)	“read”
PRINT (keyword)	“print”
CALL (keyword)	“call”
WHILE (keyword)	“while”
IF (keyword)	“if”
THEN (keyword)	“then”
ELSE (keyword)	“else”
OPERATOR_1	“+”, “-“
OPERATOR_2	“%”, “/”, “*”
CONDITIONAL	“<”, “<=”, “>”, “>=”, “==”, “!=”
LOGICAL	“&&”, “||”
NOT	“!”
LEFTBRACE	“{“
RIGHTBRACE	“}”
LEFTBRACKET	“(“
RIGHTBRACKET	“)”
SEMICOLON	“;”
ASSIGNMENT	“=”
VARIABLE	NAME
CONSTANT	INTEGER

9.2. AST Node Attributes

Type	Description of the type of node
Children Nodes	A vector containing all children node
Parent	An integer of the statement number of the parent node
Statement Number	An integer of the statement number of the node.
Name	A string of the name of the node / Value of the node which depends on the type.
Variables Used	A set of string storing the variables used in the statement or its direct children
Variables Modified	A set of string storing the variables modified in the statement
Constants	A set of string storing the constants used in the statement

Traversed	A Boolean indicating whether the Design Extractor has visited this node during its traversal
Type	Type of AST Node

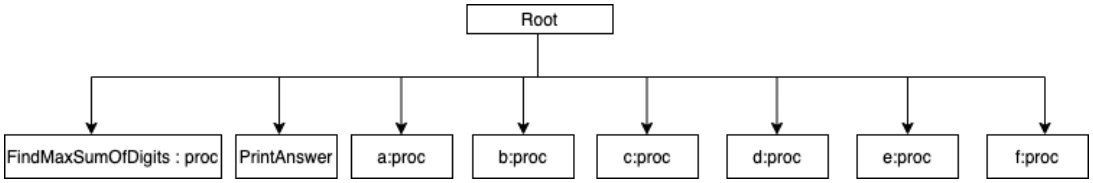


Figure 42: Truncated AST diagram showing the root and its children

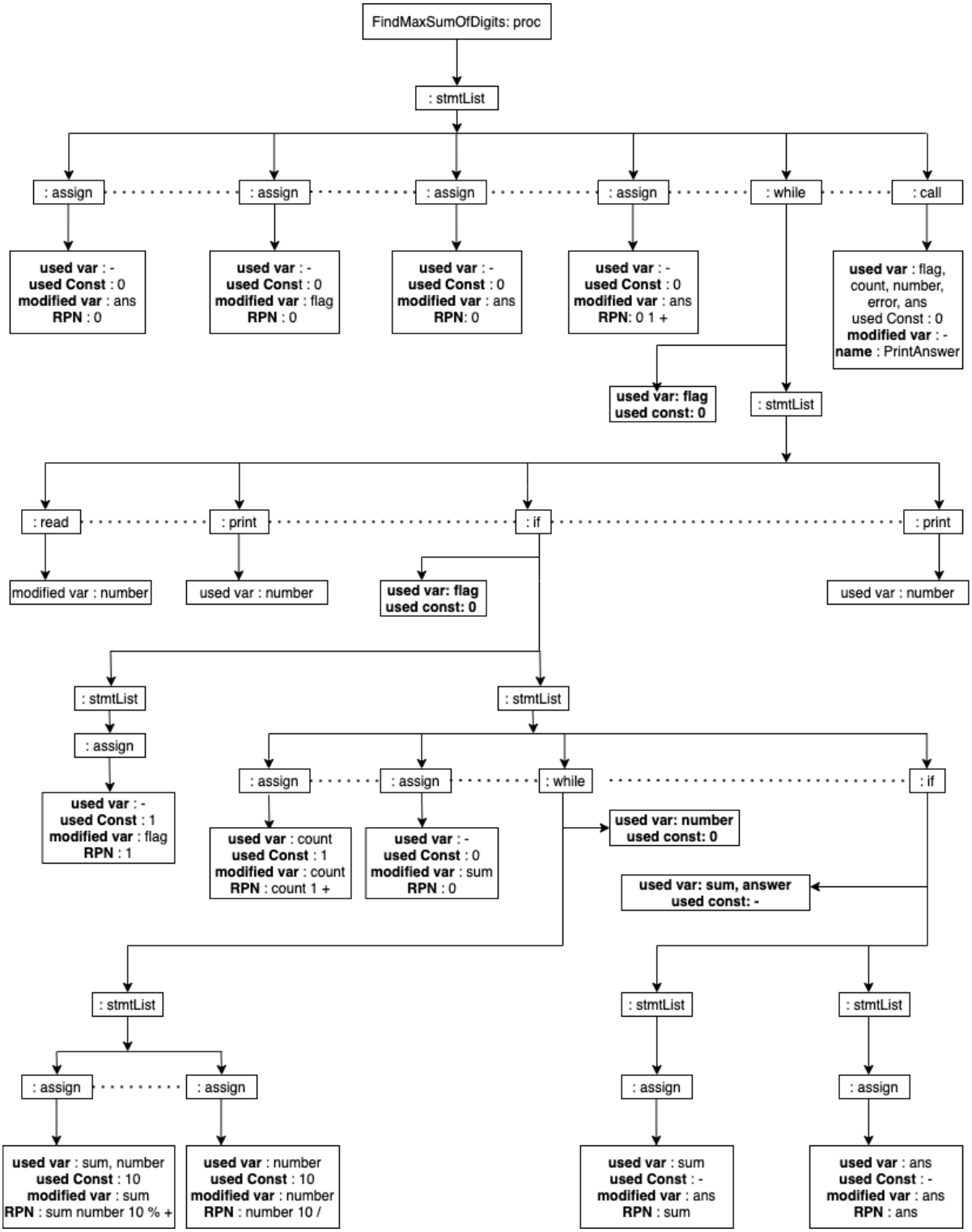


Figure 43: Truncated AST diagram showing the FindMaxSumOfAllDigits

9.3. Validation of Grammar by Sub Parser

program: procedure +	program parser
procedure: ‘procedure’ proc_name ‘{‘ stmtLst ‘}’	procedure parser
stmtLst: stmt+	stmtLst parser
read: ‘read’ var_name ‘;’	read parser
print: ‘print’ var_ame ‘;’	print parser
call: ‘call’ proc_name ‘;’	call parser
while: ‘while’ ‘(‘ cond_expr ‘)’ ‘{‘ stmtLst ‘}’	while parser
if: ‘if’ ‘(‘ cond_expr ‘)’ ‘then’ ‘{‘ stmtLst ‘}’ ‘else’ ‘{‘ stmtLst ‘}’	if parser
assign: var_name ‘=’ expr ‘;’	Assignment Parser
cond_expr	Cond Expr Parser
rel_expr	Cond Expr Parser
rel_factor	Cond Expr Parser
expr	Expr Parser
term	Expr Parser
factor	Expr Parser
var_name, proc_name	Tokenizer
const_value	Tokenizer

9.4. Detailed Run Through of Expr Parser

The following is a run through of how the expr in the assignment statement 13 is being parsed and validated.

```
sum = sum + (number % 10);
```

```
expression queue<Tokens> =
```

```
[“VARIABLE: sum”, “OPERATOR_1: +”, “LEFT BRACKET”, “VARIABLE: number”, “OPERATOR_2: %”, “CONSTANT: 10”, “RIGHT BRACKET]
```

1. An empty expression string is created: “”

2. Calls `Expr Parser`

3. Calls `Term Parser` and `Expr Parser`

4. `Term Parser` calls `Factor Parser` and `Term Parser`

5. Factor Parser peeks at “`VARIABLE: sum`”, thus it will consume and append the token to the empty expression.

6. `Expression string = “sum”`

7. Term parser peeks at "OPERATOR_1: +", thus it will not continue recursing through Term and the term structure has ended.

8. The Expr Parser will then peek at "OPERATOR_1: +", thus it will consume and append it to the expression. It will then call Factor Parser and Term Parser.

9. Expression string = "sum +"

10. Factor parser will peek at LEFT BRACKET, thus it will enforce the bracketing rule by calling expects(LEFT BRACKET), calls Expr Parser and expects(RIGHT BRACKET). Since the current token is a bracket, it will be consumed and append to the expression string.

11. Expression string = "sum + ("

12. Expr Parser will call Term Parser and Expression Parser

13. Term Parser will call Factor Parser and Term Parser

14. Factor Parser will peek at "VARIABLE: number" thus it will append it to the expression string.

15. Expression string = "sum + (number"

16. The Term Parser will then peek and see OPERATOR_2 and thus it will append it to the expression string and calls Factor Parser and Term Parser.

17. Expression string = "sum + (number %"

18. The Factor Parser will peek at the "CONSTANT: 10" and appends it to the expression string.

19. Expression string = "sum + (number % 10"

20. The Term Parser will peek at the RIGHT BRACKET and do nothing since it signifies the end of the term expression.

21. The remaining expect(RIGHT BRACKET) from line 10 will consume and append the bracket to the expression.

22. Expression string = "sum + (number % 10)"

23. The Expr Parser call from line 12 will do nothing as it has met the end of the queue. This concludes the end of the validation of the expr.

24. The expression string will then be converted into RPN.

25. RPN = sum number 10 % +

9.5. PKB Abstract APIs

Design Entity API (for VarTable and ProcTable)	Description
VOID addDEValue (**DESIGN_ENTITY** deType, **VAR** value)	Stores a design entity of deType into the PKB.
BASICSTORAGE& getDEStoreOf (**DESIGN_ENTITY** deType)	Returns a reference to the storage of deType.
INDICES& getIndexedDEValuesOf (**DESIGN_ENTITY** deType)	Returns the set of indices of all design entities of deType.
INDEX createIndexOfDEValue (**VAR** value)	Creates an index-to-value mapping in the index table and returns the index.
VAR getStringValueOfDE (**INDEX** index)	Returns the string of a design entity.
INTEGER getDEStoreSizeOf (**DESIGN_ENTITY** deType)	Returns the size of the storage of deType.

AST API	Description
VOID setASTRoot(**TNODE** node)	Sets the AST root node in the PKB.
TNODE getASTRoot()	Returns the AST root node.

Relationship API (for all relationships)	Description
VOID addRelationship (**RELATIONSHIP_TYPE** rType, **VAR** firstValue, **VAR** secondValue)	Stores a relationship of rType between firstValue and secondValue into the PKB.
VOID copyValues(**RELATIONSHIP_TYPE** fromRType, **VAR** from, **RELATIONSHIP_TYPE** toRType, **VAR** to)	Copies the values x from the relationship fromRType(from, x) and add the relationships toRType(to, x).
BOOLEAN hasRelationship (**RELATIONSHIP_TYPE** rType, **VAR** firstValue, **VAR** secondValue)	Returns TRUE if there is a relationship of rType between firstValue and secondValue, FALSE otherwise.
VALUES getFirstValuesOf (**RELATIONSHIP_TYPE** rType, **DESIGN_ENTITY** firstValueType, **VAR** secondValue)	Returns the set of indices of values x for Relationship(x, secondValue) where Relationship is of rType and x is of firstValueType.
VALUES getSecondValuesOf (**RELATIONSHIP_TYPE** rType, **DESIGN_ENTITY** secondValueType, **VAR** firstValue)	Returns the set of indices of values x for Relationship(firstValue, x) where Relationship is of rType and x is of firstValueType.

VALUE_PAIRS getValuePairsOf (**RELATIONSHIP_TYPE** rType, **DESIGN_ENTITY** firstValueType, **DESIGN_ENTITY** secondValueType)	Returns the set of indices of the value pairs {x, y} for Relationship(x, y) where x is of firstValueType and y is of secondValue type.
VALUES getRelationshipToSelfOf (**RELATIONSHIP_TYPE** rType, **DESIGN_ENTITY** valueType)	Returns the set of indices of values x for Relationship(x, x) where Relationship is of rType and x is of valueType.
RELATIONSHIPSTORAGE<INDEX, INDEX>& getRelationshipStoreOf (**RELATIONSHIP_TYPE** rType)	Returns a reference to the storage of relationships of rType.
INTEGER getRelationshipStoreSize (**RELATIONSHIP_TYPE** rType)	Returns the size of the storage of relationships of rType.
VOID populateNext*Cache()	Populates the Next* cache if it has not been populated.
VOID populateAffectsCache()	Populates the Affects cache if it has not been populated.
VOID populateAffects*Cache()	Populates the Affects* cache if it has not been populated.
VOID clearCache()	Clears all caches.

9.6. Query Evaluator UML Diagrams

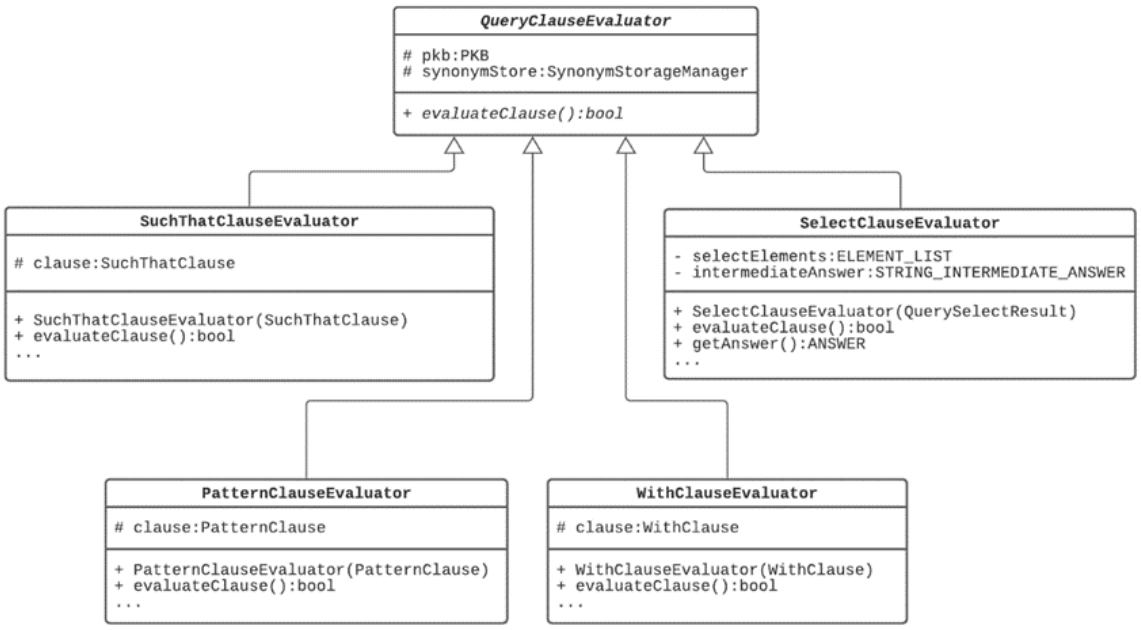


Figure 44: QueryClauseEvaluator Class Diagram

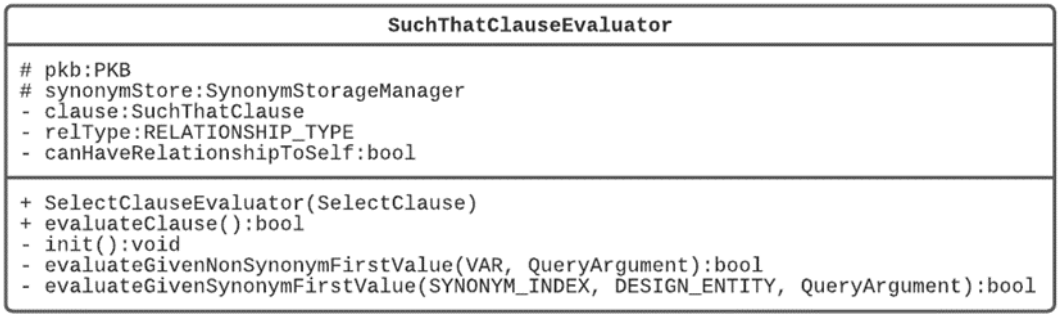


Figure 45: SuchThatClauseEvaluator Class Diagram

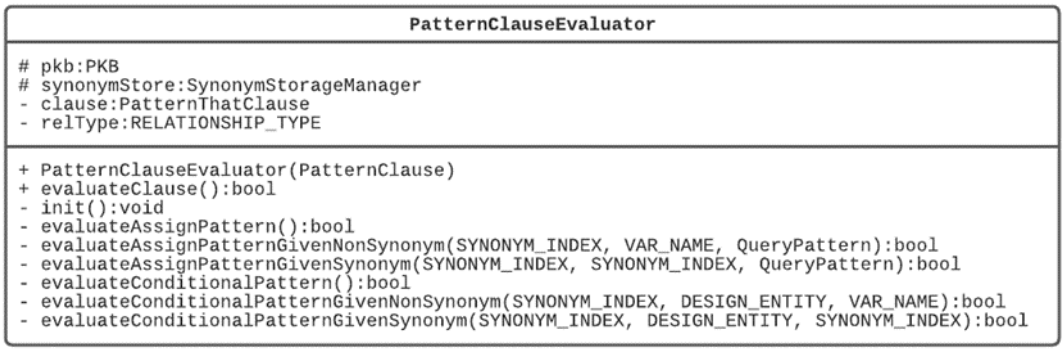


Figure 46: PatternClauseEvaluator Class Diagram

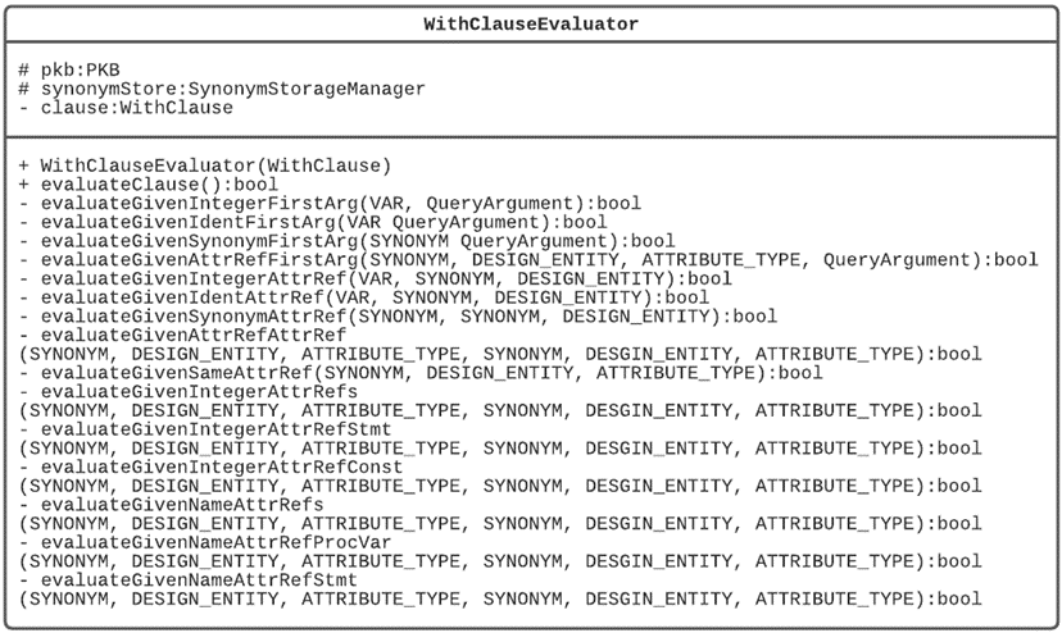


Figure 47: WithClauseEvaluator Class Diagram

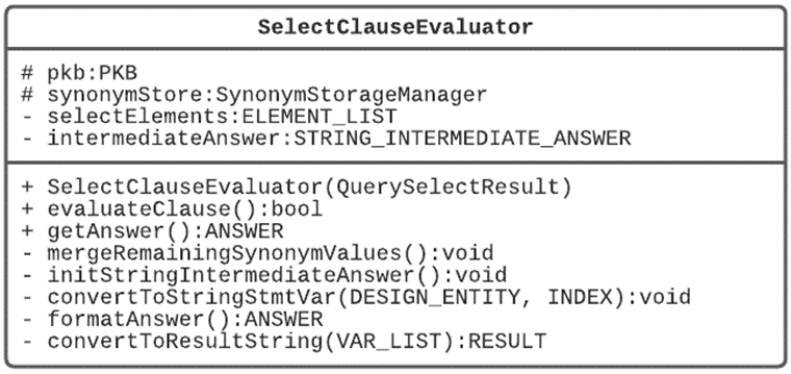


Figure 48: SelectClauseEvaluator Class Diagram

Figure 49: MiniTable Class Diagram

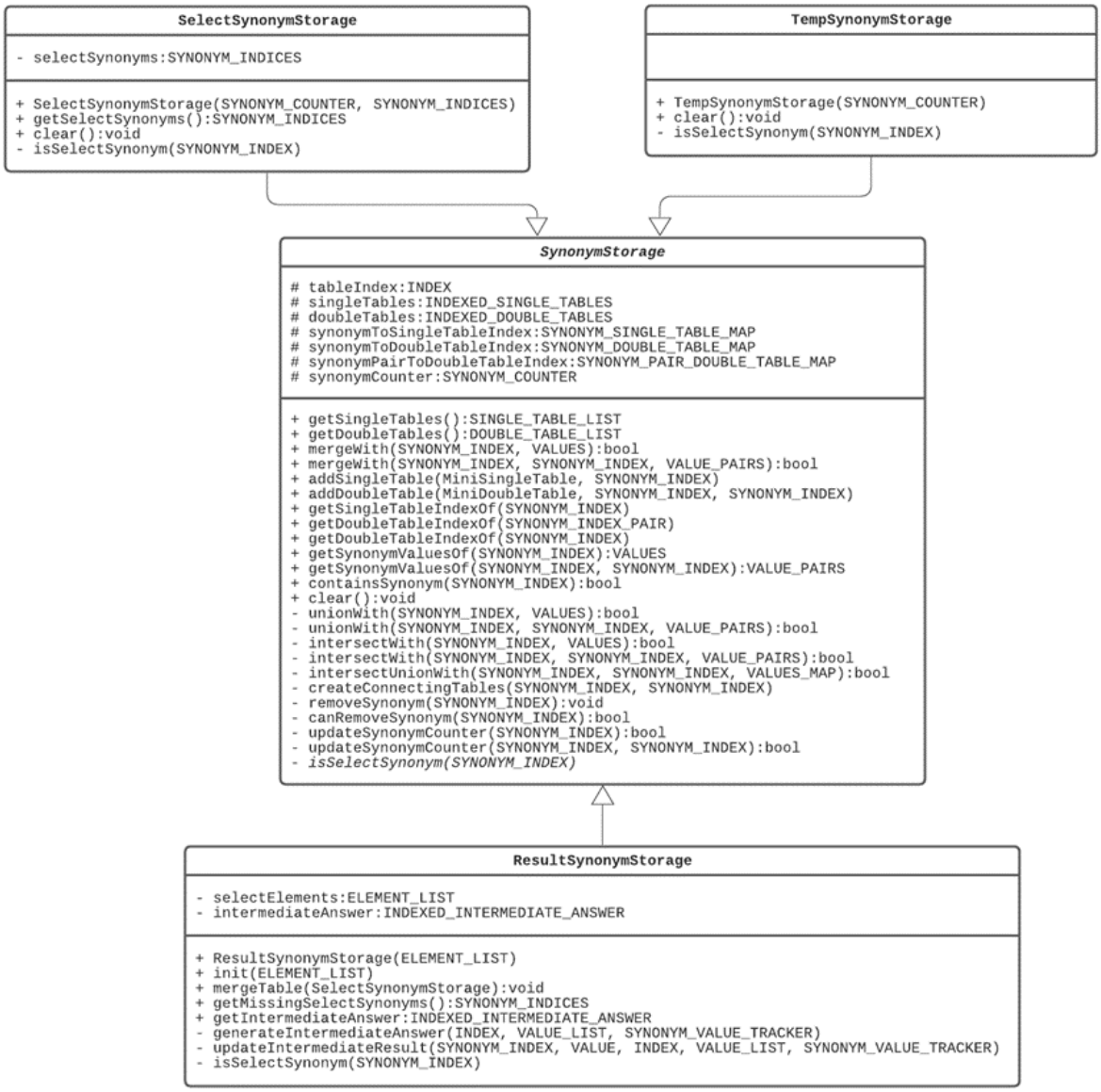


Figure 50: SynonymStorage Class Diagram

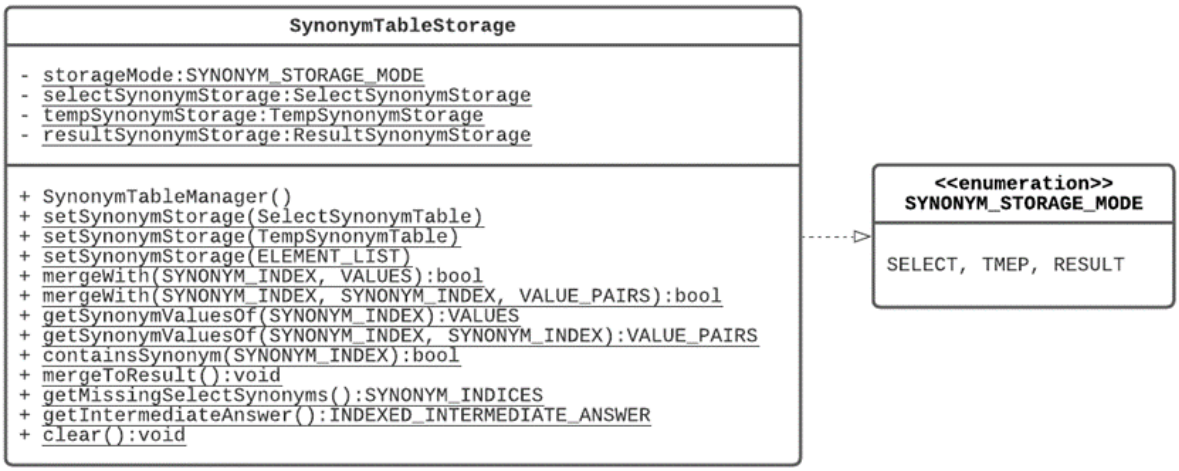


Figure 51: SynonymStorageManager Class Diagram

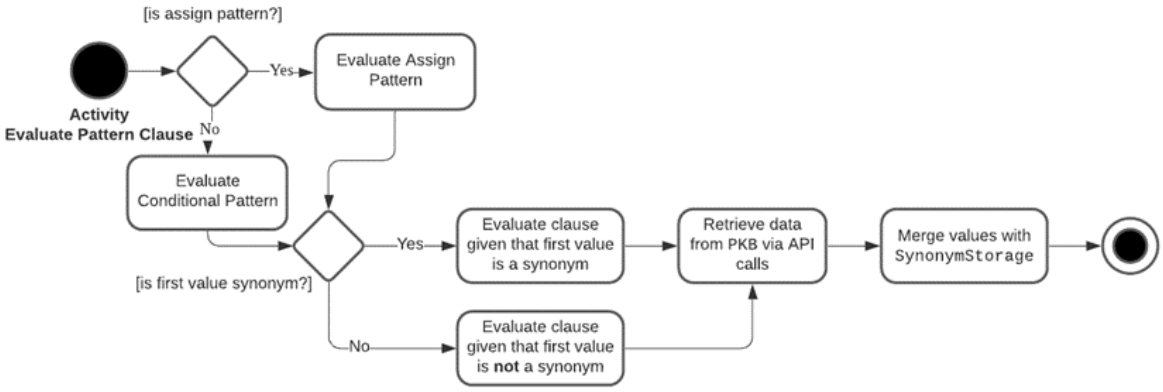


Figure 52: PatternClauseEvaluator Evaluation Activity Diagram

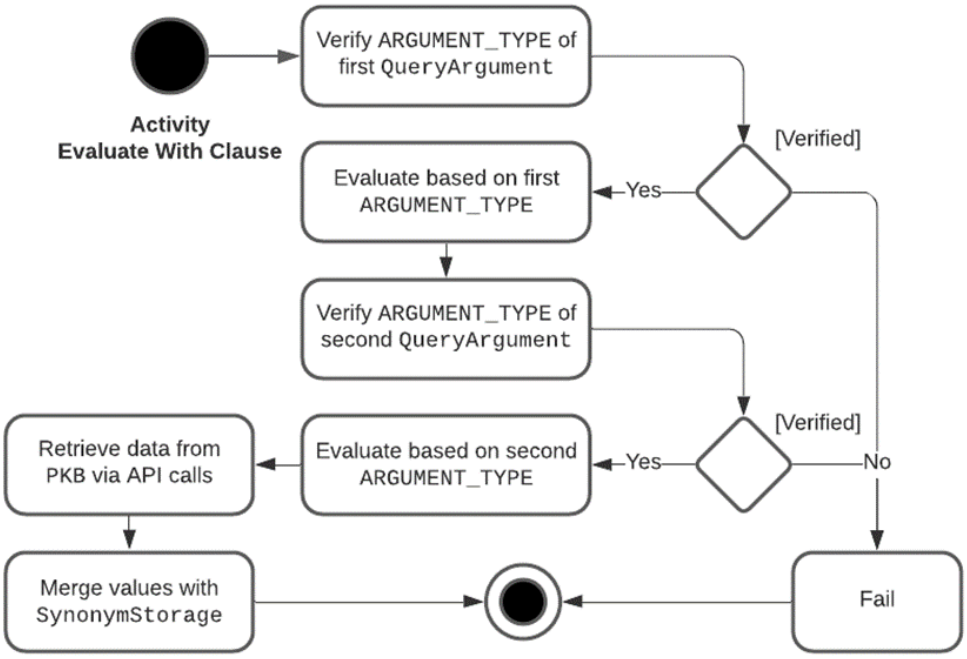


Figure 53: Simplified WithClauseEvaluator Evaluation Activity Diagram

9.7. Synonym Storage Implementation Visuals

In this section, we compare the implementations between the one-table structure and the multi-table (*single* and *double*) structure. The latter structure is our team's current implementation for this Iteration.

In the illustrations, we calculate the time complexities based on the worst-case scenario, with **K** being the number of synonyms in the table, and **N** being the number of design entity values *(e.g. statements, variables, constants, etc.).*

Merging of Synonym Values

For the following two scenarios, assume that the `SynonymStorage` is currently empty.

Union of **single** synonym in empty storage

Incoming

a
1
2

One-Table	**2 entries**	**O(N)**
Multi-Table	**2 entries**	**O(N)**

Union of **paired** synonyms in empty storage

Incoming

b	c
1	1
1	2
2	1
2	2

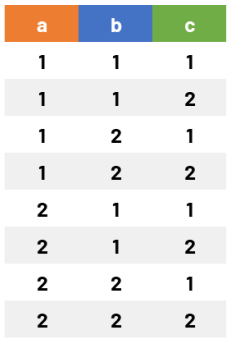
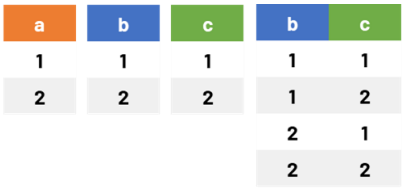
One-Table	**8 entries**	**$O(N^2)$**
Multi-Table	**12 entries**	**$O(N^2)$**

Union of **paired** synonyms in storage with **one** synonym

For this scenario, assume that the synonym values for **a** are already in the storage.

Incoming

b	c
1	1
1	2
2	1
2	2

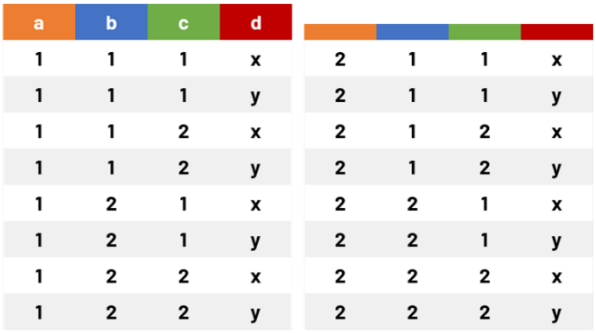
One-Table		**24 entries** **$O(N^{K+2})$**
Multi-Table		**14 entries** **$O(N^2)$**

Union of **single** synonym in non-empty storage

For the remaining scenarios, assume that the synonym values for **a**, **b** and **c** are already in the storage. The initial state of the storage would be the same as the resulting merge of the previous scenario.

Incoming

d
x
y

One-Table		**64 entries** **$O(N^{K+2})$**

Multi-Table

a	b	c	d	b	c
1	1	1	x	1	1
2	2	2	y	1	2
				2	1
				2	2

16 entries $O(N^2)$

Union of **paired** synonyms in non-empty storage

Incoming

d	e
x	y
x	z
y	y
y	z

One-Table

a	b	c	d	e					
1	1	1	x	y	2	1	1	x	y
1	1	1	x	z	2	1	1	x	z
1	1	1	y	y	2	1	1	y	y
1	1	1	y	z	2	1	1	y	z
1	1	2	x	y	2	1	2	x	y
1	1	2	x	z	2	1	2	x	z
1	1	2	y	y	2	1	2	y	y
1	1	2	y	z	2	1	2	y	z
1	2	1	x	y	2	2	1	x	y
1	2	1	x	z	2	2	1	x	z
1	2	1	y	y	2	2	1	y	y
1	2	1	y	z	2	2	1	y	z
1	2	2	x	y	2	2	2	x	y
1	2	2	x	z	2	2	2	x	z
1	2	2	y	y	2	2	2	y	y
1	2	2	y	z	2	2	2	y	z

160 entries $O(N^{K+2})$

Multi-Table

a	b	c	d	e
1	1	1	x	y
2	2	2	y	z

b	c	d	e
1	1	x	y
1	2	x	z
2	1	y	y
2	2	y	z

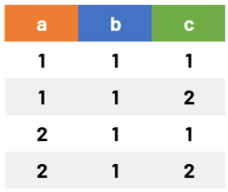
26 entries $O(N^2)$

Intersection of **single connected** synonym in non-empty storage

Incoming

b
1
3
5

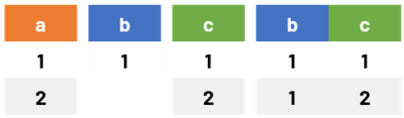
One-Table



a	b	c
1	1	1
1	1	2
2	1	1
2	1	2

12 entries | **O(N^K)**

Multi-Table



a	b	c	b	c
1	1	1	1	1
2		2	1	2

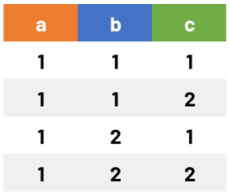
9 entries | **O(N)**

Intersection of **single non-connected** synonym in non-empty storage

Incoming

a
1
3
5

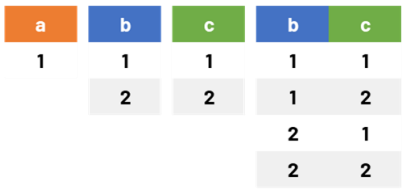
One-Table



a	b	c
1	1	1
1	1	2
1	2	1
1	2	2

12 entries | **O(N^K)**

Multi-Table

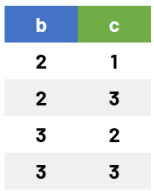


a	b	c	b	c
1	1	1	1	1
	2	2	1	2
			2	1
			2	2

13 entries | **O(K²·N²)**

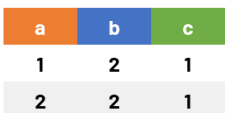
Intersection with **paired connected** synonyms in non-empty storage

Incoming



b	c
2	1
2	3
3	2
3	3

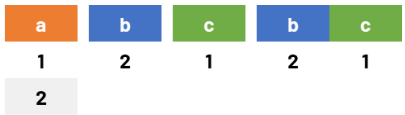
One-Table



a	b	c
1	2	1
2	2	1

6 entries | **O(N^K)**

Multi-Table



a	b	c	b	c
1	2	1	2	1
2				

6 entries | **O(K²·N²)**

Intersection with **paired non-connected** synonyms in non-empty storage

Incoming

a	c
2	1
2	3
3	2
3	3

One-Table	**6 entries**	**$O(N^K)$**
Multi-Table	**14 entries**	**$O(K^2 \cdot N^2)$**

One-Table

a	b	c
2	1	1
2	2	1

Multi-Table

a	b	c
2	1	1
	2	

a	b
2	1
2	2

b	c
1	1
2	1

a	c
2	1

Union-intersection with **connected** synonyms in non-empty storage

Incoming

c	d
1	x
1	y
1	z
3	x
3	y

One-Table	**48 entries**	**$O(N^{K+1})$**
Multi-Table	**30 entries**	**$O(K^2 \cdot N^2)$**

One-Table

a	b	c	d
1	1	1	x
1	1	1	y
1	1	1	z
1	2	1	x
1	2	1	y
1	2	1	z

2	1	1	x
2	1	1	y
2	1	1	z
2	2	1	x
2	2	1	y
2	2	1	z

Multi-Table

a	b	c	d
1	1	1	x
2	2		y
			z

b	c
1	1
2	1

c	d
1	x
1	y
1	z

b	d
1	x
1	y
1	z
2	x
2	y
2	z

Union-intersection with **non-connected** synonyms in non-empty storage

Incoming

a	d
1	x
1	y
1	z
3	x
3	y

One-Table

a	b	c	d
1	1	1	x
1	1	1	y
1	1	1	z
1	1	2	x
1	1	2	y
1	1	2	z

1	2	1	x
1	2	1	y
1	2	1	z
1	2	2	x
1	2	2	y
1	2	2	z

48 entries $O(N^{K+1})$

Multi-Table

a	b	c	d
1	1	1	x
	2	2	y
			z

a	d
1	x
1	y
1	z

b	c
1	1
1	2
2	1
2	2

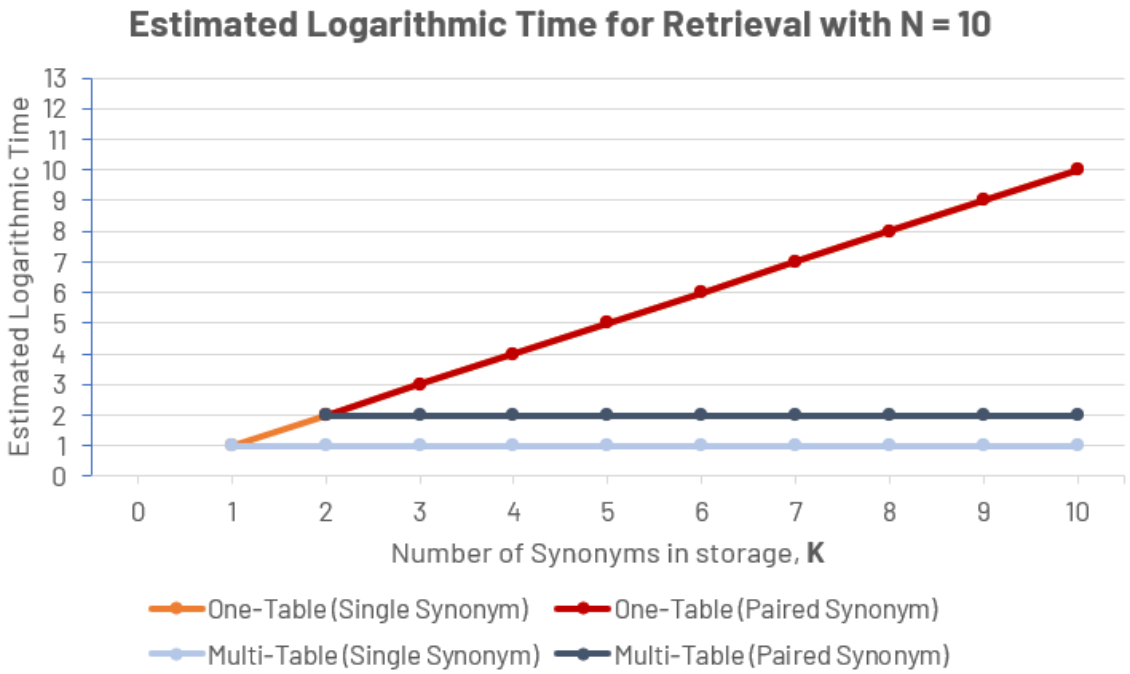
22 entries $O(N^2)$

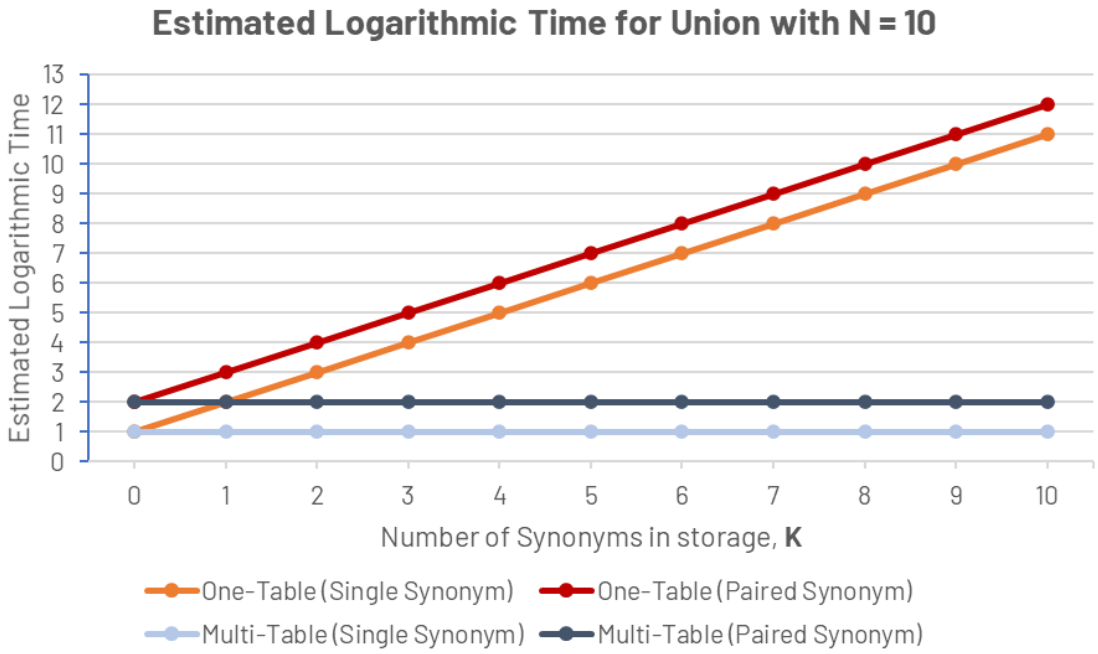
Estimated Merging Efficiency by Worst-Case Time Complexity

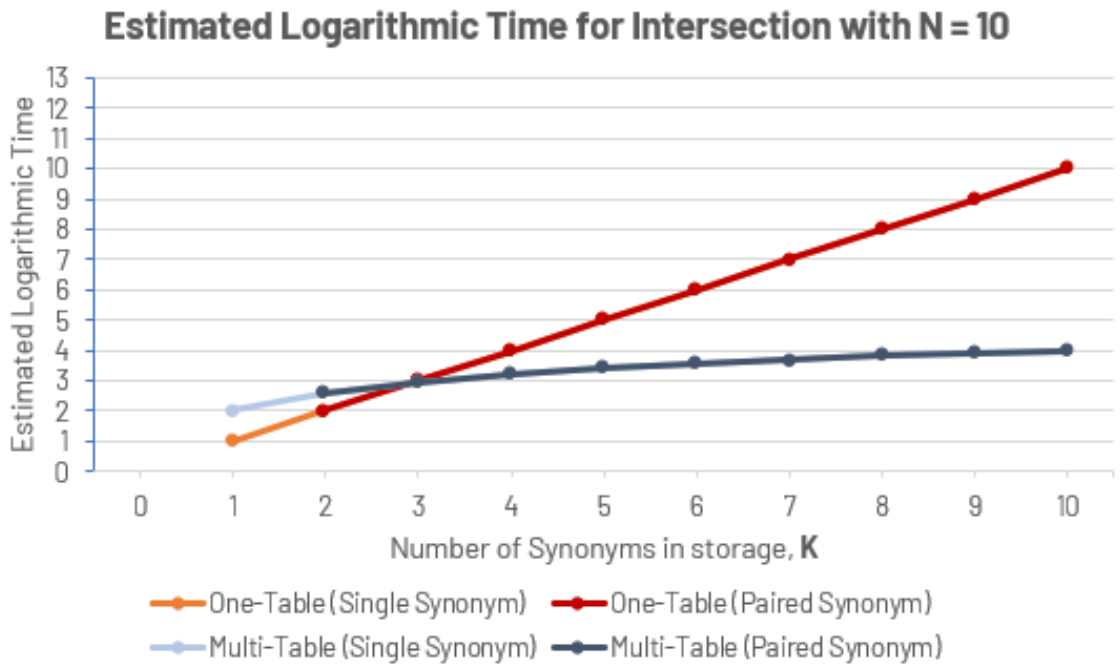
Below are some graphs to visually show the differences between the one-table and multi-table, concerning the relationship between the estimated time taken to perform an action for different number of synonyms, **K**, currently present in the storage.

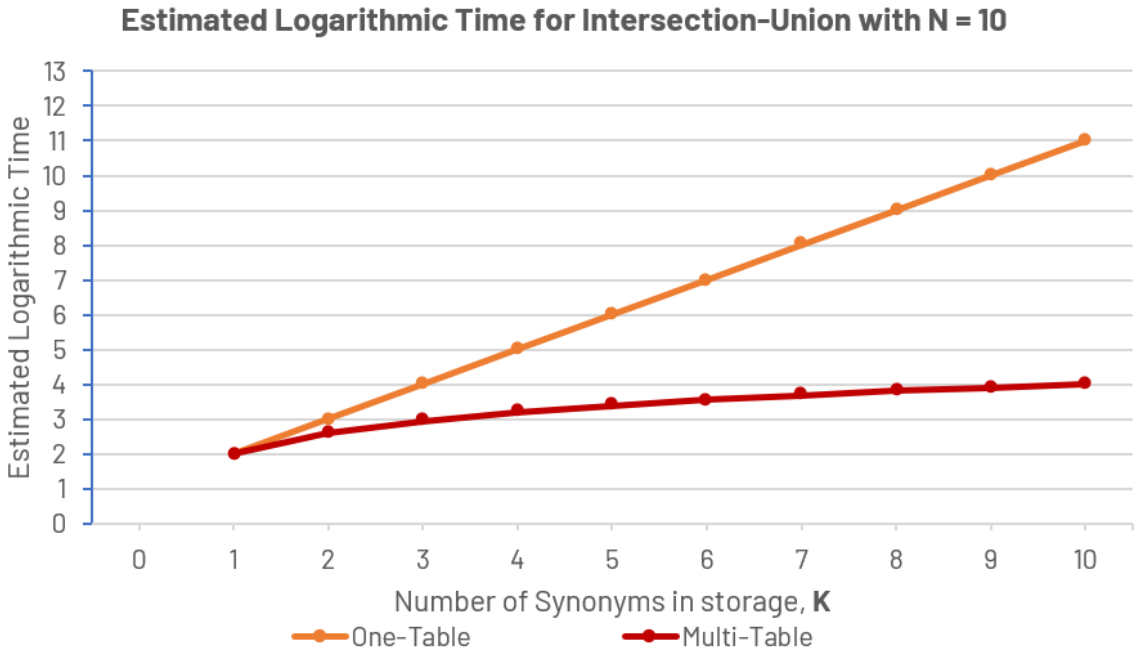
Note that we use the number of design entity values, **N** = 10 for our examples, and increasing the value for **N** will increase the discrepancies between the timings of the two models exponentially. Also, the estimated time taken is in terms of logarithmic scale, so values on the graphs are increasing *exponentially*.

Notice that when **K** > 3, the estimated time to perform any of the storage actions by the multi-table is **always** faster than that of the one-table model.


Estimated Logarithmic Time for Retrieval with N = 10
Estimated Logarithmic Time
0
1
2
3
4
5
6
7
8
9
10
11
12
13
0
1
2
3
4
5
6
7
8
9
10
Number of Synonyms in storage, K
One-Table (Single Synonym)
One-Table (Paired Synonym)
Multi-Table (Single Synonym)
Multi-Table (Paired Synonym)


Estimated Logarithmic Time for Union with N = 10
Estimated Logarithmic Time
0
1
2
3
4
5
6
7
8
9
10
11
12
13
0
1
2
3
4
5
6
7
8
9
10
Number of Synonyms in storage, K
One-Table (Single Synonym)
One-Table (Paired Synonym)
Multi-Table (Single Synonym)
Multi-Table (Paired Synonym)


Estimated Logarithmic Time for Intersection with N = 10
Estimated Logarithmic Time
0
1
2
3
4
5
6
7
8
9
10
11
12
13
0
1
2
3
4
5
6
7
8
9
10
Number of Synonyms in storage, K
One-Table (Single Synonym)
One-Table (Paired Synonym)
Multi-Table (Single Synonym)
Multi-Table (Paired Synonym)


Estimated Logarithmic Time for Intersection-Union with N = 10
Estimated Logarithmic Time
0
1
2
3
4
5
6
7
8
9
10
11
12
13
0
1
2
3
4
5
6
7
8
9
10
Number of Synonyms in storage, K
One-Table
Multi-Table

9.8. Activity diagram of Next extraction in the Design Extractor

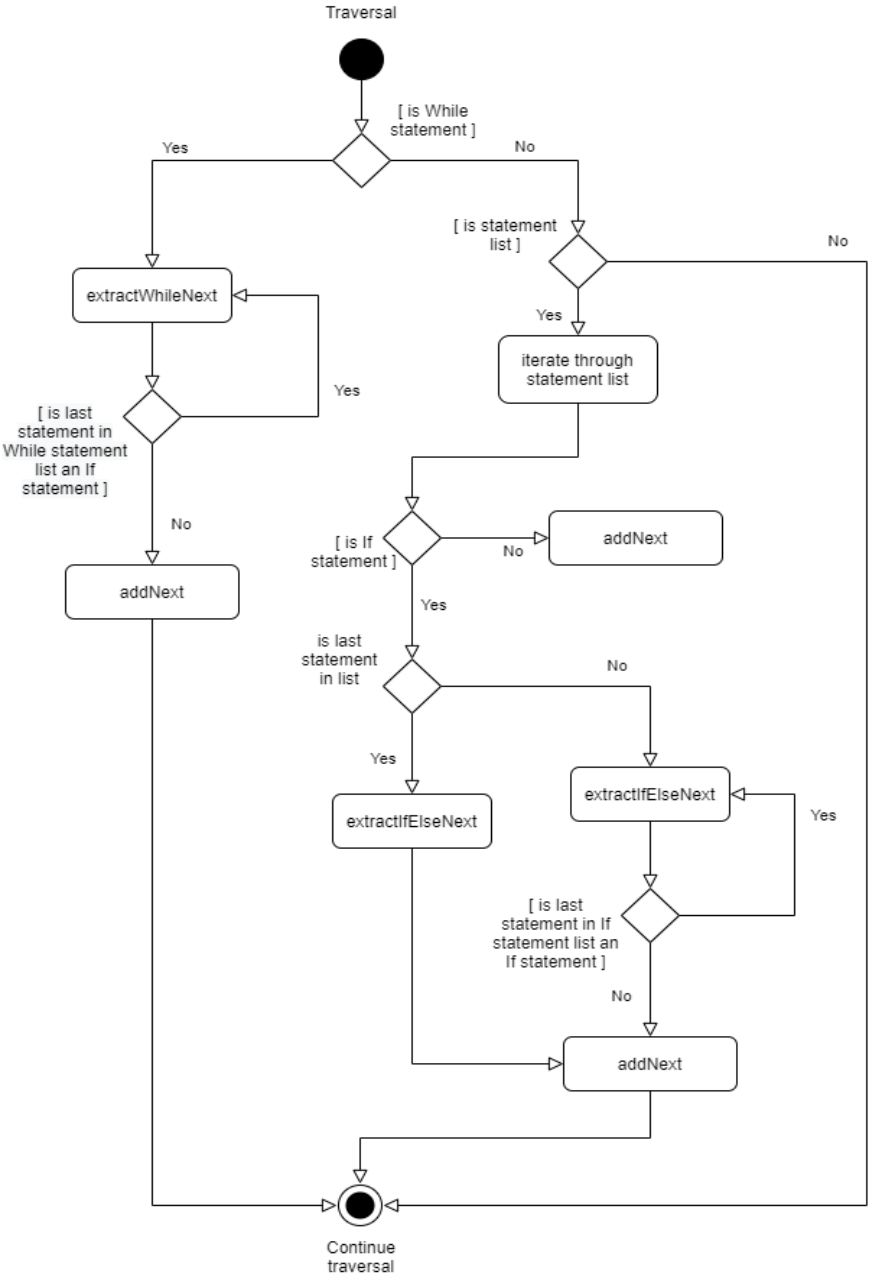


Figure 54: Activity diagram of Next extraction in Design Extractor

9.9. Complete List of Optimisations for SPA Program

PKB

1. Use generalised tables
2. Store design abstractions as INTEGERS instead of STRINGS
3. Use unordered versions of sets and maps to store design abstractions

QPS

1. Early termination for queries with syntactic or semantic errors
2. Interpret synonyms as INTEGERS instead of STRINGS
3. Organise query clauses into groups of interconnected synonyms
4. Sort queries within groups according to interconnection to previous clause
5. Create a no cartesian-product data structure and implementation for synonym storage

10. Appendix B: Testing

10.1. List of Syntax for System Testing

Simple source code

Procedure	Variable
1. empty file 2. missing procedure keyword 3. missing procedure name 4. missing opening brace 5. missing closing brace 6. missing statement list 7. invalid procedure keyword 8. invalid procedure name 9. invalid duplicate procedure name 10. invalid recursive and cyclic call 11. nesting of procedure 12. valid procedure name 13. keyword as procedure name 14. single procedure 15. multiple procedures	1. case-sensitive variables are treated as different variables 2. procedure and variable have same name 3. keyword as variable name 4. invalid variable name 5. valid variable name
Constant	**Read statement**
1. invalid first digit of zero 2. invalid constant	1. missing read keyword 2. missing read variable

3. valid constant	3. missing read semicolon 4. invalid read keyword 5. invalid read variable name 6. valid read statement
Print statement	**Call statement**
1. missing print keyword 2. missing print variable 3. missing print semicolon 4. invalid print keyword 5. invalid print variable name 6. valid print statement	1 missing call keyword 2 missing call variable 3 missing call semicolon 4 invalid call keyword 5 invalid call to non-existing procedure 6 valid call statement
While statement	**If statement**
1. empty while statement 2. missing while keyword 3. missing opening bracket 4. missing closing bracket 5. missing opening brace 6. missing closing brace 7. invalid while keyword 8. invalid conditional expression 9. valid conditional expression	1 empty then statement 2 empty else statement 3 missing else statement 4 missing if keyword 5 missing then keyword 6 missing else keyword 7 missing opening bracket 8 missing closing bracket 9 missing opening brace for then 10 missing closing brace for then 11 missing opening brace for else 12 missing closing brace for else 13 invalid if keyword 14 invalid then keyword 15 invalid else keyword 16 invalid conditional expression 17 valid conditional expression
Assign statement	**Whitespace variants**
1 missing equal sign 2 missing assign semicolon 3 invalid equal sign 4 invalid expr	1. minimal whitespace 2. multiple whitespace

Arrangement of container (while and if) statements
1. procedure with only while statement 2. procedure with only if statement 3. if statement right after while statement

4. while statement right after if statement 5. while statement nested at the start/middle/end of a container statement 6. if statement nested at the start/middle/end of a container statement 7. multiple nesting of container statements

PQL query

Select synonym as result	**Select attribute reference as result**
1. select statement 2. select read 3. select print 4. select call 5. select while 6. select if 7. select assign 8. select variable 9. select constant 10. select prog_line 11. select procedure	1. select `procedure.procName` 2. select `call.procName` 3. select `variable.varName` 4. select `read.varName` 5. select `print.varName` 6. select `constant.value` 7. select `statement.stmt#` 8. select `read.stmt#` 9. select `print.stmt#` 10. select `call.stmt#` 11. select `while.stmt#` 12. select `if.stmt#` 13. select `assign.stmt#`
Select BOOLEAN as result	**Select tuple with multiple elements as result**
1. select `BOOLEAN` with NO such that and pattern and with clause (should evaluate to `TRUE`) 2. select `BOOLEAN` that evaluates to `TRUE` 3. select `BOOLEAN` that evaluates to `FALSE`	1. select with duplicate synonyms e.g. `Select <p, p>` 2. select with duplicate attribute references e.g. `Select <s1.stmt#, s1.stmt#>` 3. select <synonym, attribute reference> where synonym and attribute reference have no overlap in synonym e.g. `Select <s1, s2.stmt#>` 4. select <synonym, attribute reference> where synonym and attribute reference have overlap in synonym e.g. `Select <s1, s1.stmt#>`
Such that clause (`Follows, FollowsT, Parent, ParentT, Next, NextT, Affects, AffectsT, NextBip, NextBipT, AffectsBip, AffectsBipT`)	**Such that clause (`Calls, CallsT`)**
1. `(synonym1, synonym2)`	1. `(synonym1, synonym2)`

• both do not overlap with select synonym • synonym1 overlaps with select synonym • synonym2 overlaps with select synonym 2. (synonym, _) • synonym does not overlap with select synonym • synonym overlaps with select synonym 3. (synonym, INTEGER) • synonym does not overlap with select synonym • synonym overlaps with select synonym 4. (_, synonym) • synonym does not overlap with select synonym • synonym overlaps with select synonym 5. (_, _) 6. (_, INTEGER) 7. (INTEGER, synonym) • synonym does not overlap with select synonym • synonym overlaps with select synonym 8. (INTEGER, _) 9. (INTEGER, INTEGER)	• both do not overlap with select synonym • synonym1 overlaps with select synonym • synonym2 overlaps with select synonym 2. (synonym, _) • synonym does not overlap with select synonym • synonym overlaps with select synonym 3. (synonym, “IDENT”) • synonym does not overlap with select synonym • synonym overlaps with select synonym 4. (_, synonym) • synonym does not overlap with select synonym • synonym overlaps with select synonym 5. (_, _) 6. (_, “IDENT”) 7. (“IDENT”, synonym) • synonym does not overlap with select synonym • synonym overlaps with select synonym 8. (“IDENT”, _) 9. (“IDENT”, “IDENT”)
Such that clause (Modifies, Uses)	**Pattern clause (Assign)**
1. (synonym1, synonym2) • both do not overlap with select synonym • synonym1 overlaps with select synonym • synonym2 overlaps with select synonym 2. (synonym, _) • synonym does not overlap with select synonym	1. pattern a(synonym, unrestricted match) • synonym does not overlap with select synonym • synonym overlaps with select synonym 2. pattern a(synonym, partial match) • synonym does not overlap with select synonym

• synonym overlaps with select synonym 3. (synonym, “IDENT”) • synonym does not overlap with select synonym • synonym overlaps with select synonym 4. (_, synonym) - INVALID 5. (_, _) - INVALID 6. (_, “IDENT”) - INVALID 7. (INTEGER, synonym) • synonym does not overlap with select synonym • synonym overlaps with select synonym 8. (INTEGER, _) 9. (INTEGER, “IDENT”) 10. ("IDENT”, synonym) • synonym does not overlap with select synonym • synonym overlaps with select synonym 11. (“IDENT, _) 12. (“IDENT”, “IDENT”)	• synonym overlaps with select synonym 3. pattern a(_, unrestricted match) 4. pattern a(_, partial match) 5. pattern a(“IDENT”, unrestricted match) 6. pattern a(“IDENT”, partial match)
Pattern clause (While)	**Pattern clause (If)**
1. pattern w(synonym, _) • synonym does not overlap with select synonym • synonym overlaps with select synonym 2. pattern w(_, _) 3. pattern w(“IDENT”, _)	1. pattern i(synonym, _, _) • synonym does not overlap with select synonym • synonym overlaps with select synonym 2. pattern i(_, _, _) 3. pattern i(“IDENT”, _, _)
With clause	
1. with attrRef (value / stmt#) = INTEGER 2. with attrRef (procName / varName) = “IDENT” • synonym of attrRef overlaps with select synonym • synonym of attrRef do not overlap with select synonym	

3. `with synonym1 = synonym2` e.g. `with statement = assign` • both do not overlap with select synonym • synonym1 overlaps with select synonym • synonym2 overlaps with select synonym 4. swapping of both arguments produce the same result 5. `with "IDENT" = "IDENT"` (same "IDENT") 6. `with INTEGER = INTEGER` (same INTEGER) 7. `with attrRef = attrRef` (same attrRef) 8. with synonym = synonym (same synonym) (Items 5 to 8 are trivially true. Check that the timing to evaluate this should be very small which shows optimisation works.)	

Synonym overlap (Iteration 1: such that and pattern clauses)
1. no synonym overlapping among select synonym, such that clause and pattern clause 2. synonym overlapping between select synonym and such that clause (but not pattern clause) 3. synonym overlapping between select synonym and pattern clause (but not such that clause) 4. synonym overlapping between such that clause and pattern clause (but not select synonym) 5. synonym overlapping among select synonym, such that clause and pattern clause
Synonym overlap
1. no synonym overlapping between any synonyms 2. some synonym overlapping 3. all synonyms overlapping i.e. all synonyms in the select synonym, such-that, pattern and with clauses are the same
Multiple combinations
1. no "and" keyword 2. "and" keyword used
Whitespace variant

1. minimal whitespace 2. multiple whitespace

10.2. Files used in System Test

Test suite	Source file	Queries file
1	01_single_procedure_source	01_single_procedure_queries
2	02_nested_while_if_source	02_nested_while_if_queries
3	03_wia_source 03_wia_minimal_source	03_wia_queries
4	04_attrRef_bool_tuple_source	04_attrRef_bool_tuple_queries
5	05_multiple_queries_source	05_multiple_queries_queries
6	06_stress_source	06_stress_queries
7	07_08_affects_extension_source	07_affects_queries
8	07_08_affects_extension_source	08_extension_queries
9	09_stess_source_a 09_stess_source_b	09_stess_queries_a 09_stess_queries_b

Test suite 8 is used to test the implementation of the extension.

10.3. List of test cases for Parser-PKB Integration Test

Test Cases	Other test cases they are part of
singleLine	multiProc1
basic	multiProc2
while	
whileTop	
whileMiddle	
whileBottom	multiProc1
ifElse	
ifElseTop	
ifElseMiddle	
ifElseBottom	multiProc2

whileIfElse	
whileIfElseTop	
whileIfElseMiddle	
whileIfElseBottom	multiProc3
ifElseWhile	
ifElseWhileTop	
ifElseWhileMiddle	
ifElseWhileBottom	multiProc2
nestedWhileWhile	
nestedWhileWhileTop	
nestedWhileWhileMiddle	
nestedWhileWhileBottom	multiProc3
nestedWhileIfElse	
nestedWhileIfElseTop	
nestedWhileIfElseMiddle	
nestedWhileIfElseBottom	multiProc3
nestedIfIfElse	
nestedIfIfElseTop	
nestedIfIfElseMiddle	
nestedIfIfElseBottom	
nestedIfElseIf	multiProc1
nestedIfElseIfTop	
nestedIfElseIfMiddle	
nestedIfElseIfBottom	multiProc2
nestedIfWhileElse	
nestedIfWhileElseTop	
nestedIfWhileElseMiddle	
nestedIfWhileElseBottom	multiProc1
nestedifElseWhile	
nestedifElseWhileTop	
nestedifElseWhileMiddle	
nestedifElseWhileBottom	multiProc3
complicatedtest1	
complicatedtest2	
complicatedtest3	

11. Appendix C: Extension

11.1. Iteration 2 Extension

Nest and **Nest***

Description

Our group proposes to extend the program design abstractions by adding a new advanced relationship called Nest.

Nest is a design abstraction that accounts for the nesting-depth level difference between 2 statements in the same procedure

In regard to the Source Language SIMPLE, we formally define the nesting-depth level of a statement as the number of times the statement is nested in an If Statement or While Statement in a procedure.

With reference to the sample SIMPLE program in Section 1.1, the nesting-depth level of statement 5 is 0 while the nesting-depth level of statement 9 is 2.

Definition

Nest is an overloaded relationship that takes in 2 different parameters.

For any statements `s1` and `s2` and an integer `k`, `k >= 0`:

? `Nest ( 1, k)` holds if the nesting level of `s1` is exactly `k`

1. `Nest (s1, s2, k)` holds if the nesting level of `s1` and the nesting level of `s2` are exactly `k` levels apart
2. `Nest* (s1, s2, k)` holds if the nesting level of `s1` and the nesting level of `s2` are exactly `k` levels apart and `Parent*(s1, s2)` is true.

Utility

Deep level of nesting can cause various problems in the code. The first issue is that excessive nesting makes logic hard to follow. For long procedures with deep levels of If-Else or While nesting can cause logical workflow to become blurred and thus rendering the code unreadable and untraceable.

This would in turn cause problems when refactoring or debugging the code. A small change in a statement can cause ripple effects and alter the logic path of a particular deep branch that was once functioning properly.

The conventional way of keeping track of nesting-depth level is by looking at the indentation level of a particular statement. However, for long and deeply nested procedures, tracking this by eye becomes unfeasible and also prone to human error.

Thus, by having the design abstraction Nest Relationship, these problems can be solved and make it easier for programmers of the SIMPLE source.

Implementation

To correctly extract the nesting levels of different statements in the SIMPLE source, these will be the steps taken by the Design Extractor:

- When traversing the graph, keep an integer reference of the current nesting-depth level. Let us call this integer `nestingDepth`. `nestingDepth` will be set to 0 at the start of traversal.
- When we enter a container statement (If-Else or While node), we increment `nestingDepth` by `1`. When we exit a container statement, we decrement `nestingDepth` by `1`.
- For every statement, when we add the statement type and statement number to the `PKB`, we also call a separate `PKB` method called `addDepth(s1, k)` where `s1` is the statement number and `k` is `nestingDepth`.
- When we identify a procedure node, we reset `nestingDepth` to `0`.

To store the information required for the `Next` relationship, these are the changes we will make to the `PKB`:

- We would create a new static storage class in the PKB called `DepthStorage` and it would contain a `depthTable` of type `map<STMT_NUM, DEPTH_LEVEL>`
- Below is a table of the abstract API we would implement

API for Source Processor	Description
BOOLEAN addDepth (**STMT_NUM** stmtNum, **DEPTH_LEVEL** depth)	Stores a statement into the `PKB` with the nesting depth level it is at. Returns `TRUE` if the statement number -> nesting-depth level is added successfully or returns `FALSE` otherwise.

To parse and validate the new types of queries the PQL can receive, these will be the additional implementation we would add:

1. Add `NEST` and `NEST_T` (to represent `Nest*`) to `SUCH_THAT_REL_TYPE`
2. Map the relationship types to a vector of sets of argument types, where each set validates the argument type for each argument
    a. `Nest(stmtRef, nestLevel)`: `NEST` is mapped to a vector of two sets of argument types. The first set contains `SYNONYM`, `UNDERSCORE` and `INTEGER`. The second set contains `INTEGER`.
    b. `Nest(stmtRef, stmtRef, nestLevel)`: `NEST` is also mapped to a vector of three sets of argument types. The first set contains `SYNONYM`, `UNDERSCORE` and `INTEGER`. The second set contains `SYNONYM`, `UNDERSCORE` and `INTEGER`. The third set contains `INTEGER`.
    c. `Nest*(stmtRef, stmtRef, nestLevel)`: `NEST_T` is mapped to a vector of three sets of argument types. The first set contains `SYNONYM`, `UNDERSCORE` and

INTEGER. The second set contains SYNONYM, UNDERSCORE and INTEGER. The third set contains INTEGER.

3. Map the relationship types to a vector of sets of synonym types, where each set validates the synonym type for each argument if the argument is a synonym
    a. Nest(stmtRef, nestLevel): NEST is mapped to a vector of one set of synonym types.
    b. Nest(stmtRef, stmtRef, nestLevel): NEST is also mapped to a vector of two sets of synonym types.
    c. Nest*(stmtRef, stmtRef, nestLevel): NEST_T is mapped to a vector of two sets of synonym types.
    d. Each set of synonym types contains STATEMENT, READ, PRINT, CALL, WHILE, IF, ASSIGN and PROG_LINE.

For Nest, there are two possible numbers of arguments (2 or 3). After extracting the arguments from this Nest clause, check the argument list length to see which vector should be used to validate the argument types (and synonym types if applicable).

For evaluation of suchThatClause of the relationship type Nest, we can simply define a new RELATIONSHIP_TYPE NEST/NEST_STAR and evaluate using the SuchThatClause. However, since the clause now contains three parameters, perhaps it would be more ideal to create a subclass of SuchThatClauseEvaluator instead.

API for Query Processing Subsystem	Description
STMT_NUMS getNestedOf (**DESIGN_ENTITY** NestedType, **DEPTH_LEVEL** depth)	Returns the set of Nested statement number for Nest (s1, depth)
STMT_NUMS getNesterOf (**DESIGN_ENTITY** NesterType, **STMT_NUM** NestedNum, **DEPTH_LEVEL** depth)	Returns the set of Nester statement number for Nest (s1, NestedNum, depth)
STMT_NUMS getNestedof (**STMT_TYPE** NestedType, **STMT_NUM** NesterNum, D**EPTH_LEVEL** depth)	Returns the set of Nested statement numbers for Nest (NesterNum, s1, depth)
VALUE_PAIRS getNesterNestedOf (**DESIGN_ENTITY** NesterStmtType, **DESIGN_ENTITY** NestedStmtType, **DEPTH_LEVEL** depth)	Returns the set of pairs of Nester and Nested statement numbers s1, s2 of statement types for Nest (s1, s2, depth)
STMT_NUMS getNesterStarOf (**DESIGN_ENTITY** NesterType, **STMT_NUM** NestedNum, **DEPTH_LEVEL** depth)	Returns the set of Nester statement number for Nest* (s1, NestedNum, depth)
STMT_NUMS getNestedStarof (**STMT_TYPE** NestedType,	Returns the set of Nested statement numbers for Nest* (NesterNum, s1, depth)

STMT_NUM NesterNum, **DEPTH_LEVEL** depth)	
VALUE_PAIRS getNesterNestedOf (**DESIGN_ENTITY** NesterStmtType, **DESIGN_ENTITY** NestedStmtType, **DEPTH_LEVEL** depth)	Returns the set of pairs of Nester and Nested statement numbers s1, s2 of statement types for Nest* (s1, s2, depth)

PQL Queries with Nest(s1, k), Nest (s1, s2, k), Nest* (s1, s2, k)

These are extensions to the PQL grammar that would be made with the addition of this new relationship.

Lexical Tokens: **nestLevel** : INTEGER
Grammar Rules: relRef : ModifiesP | ModifiesS | UsesP | UsesS | Calls | CallsT | Parent | ParentT | Follows | FollowsT | Next | NextT | Affects | AffectsT | **Nest | NestT** **Nest : 'Nest' '(' stmtRef ',' nestLevel ')'** **Nest : 'Nest '(' stmtRef ',' stmtRef ',' nestLevel ')'** **NestT : 'Nest*' '(' stmtRef ',' stmtRef ',' nestLevel ')'**