Overview

1. Project Overview

This section provides an overview of the Static Program Analyzer (SPA) architecture and highlights unique Software Engineering (SWE) practices adopted.

1.1 Static Program Analyzer

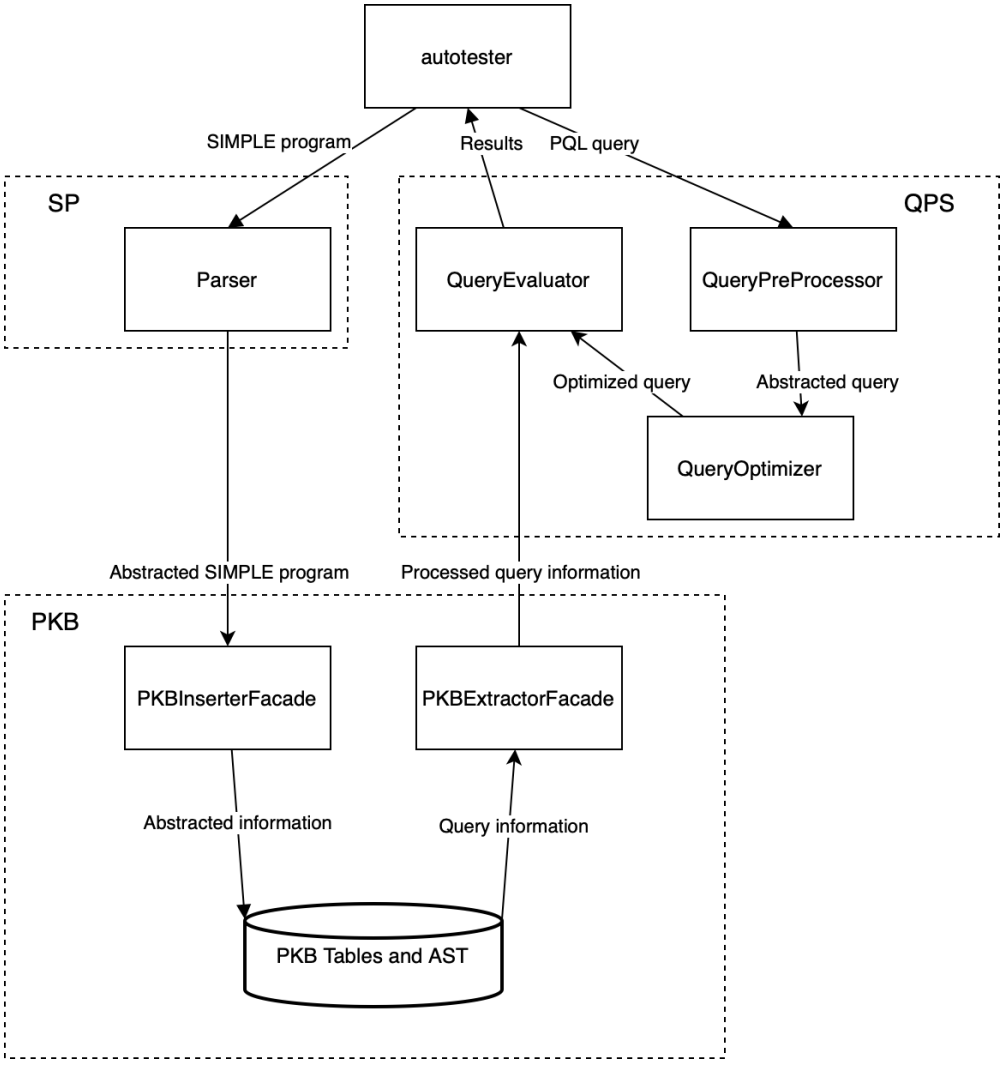

Figure 1.1: Static Program Analyzer Architecture Diagram

The architecture of the Static Program Analyzer (SPA) does not deviate significantly from the version in the lectures. It consists of the Source Processor (SP), Program Knowledge Base (PKB), and the Query Processing Subsystem (QPS). The following subsections describe each SPA sub-component.

1.1.1 Source Processor

The SP parses the input SIMPLE program and inserts design abstractions into the PKB. The SP uses a Recursive Descent Parser to validate and parse the SIMPLE program. The Shunting Yard algorithm is used to convert infix expressions into postfix notation. The algorithm is also required when querying for pattern clauses.

1.1.2 Program Knowledge Base

The PKB processes the information received from SP, validates and stores it, and exposes a set of APIs for the PQL system to perform queries on. The PKB stores information in a set of tables and stores the Abstract Syntax Tree (AST). Two facade classes are used to insert and extract information from this set of tables. They are the PKB Inserter Facade and PKB Extractor Facade respectively. A data structure (BiMap) that provides O(1) access via integer indexes and keys of any type is also implemented for faster queries. Control flow graphs (CFGs) are also stored here.

1.1.3 Query Processing Subsystem

The QPS parses PQL queries and calls the corresponding PKB APIs to retrieve the relevant information. The QPS consists of the Query PreProcessor, Query Optimizer and Query Evaluator. The Query PreProcessor makes use of the table-driven technique to verify if the arguments in the PQL clause are valid, instead of hardcoding the validation logic. The Query Optimizer sorts the Query Nodes to reduce the time taken to perform queries. The Query Evaluator finally takes in the sorted Query Nodes and evaluates them. A Query Result Table data structure is also implemented to store rows of tuples and supports equi-joins and cross products for multi-clause queries. The Query Result Table is used when the Query Evaluator calls the PKB APIs.

1.2 Software Engineering Practices

In this project, Cpplint was used to enforce Google's C++ code style guide (with slight modifications).

GitHub actions were used to support continuous integration. It runs an automated build on our target OS platform (Windows) and performs unit testing, integration testing, system testing, and linting on branch pushes or pull requests.

A logger was also implemented as a singleton object in the SPA codebase, enabling it to be called anywhere in the source code.

Weekly sprints were also conducted and GitHub projects were used to track issues.

Unit, integration, and system testing were conducted extensively for quality assurance. Python scripts were written to automate the creation and execution of source programs and system tests. Load and stress testing were conducted as extensions to fulfill SPA non-functional requirements.

Part 1 – Technical report

2 Source Processor

2.1 Overview

The SP is responsible for the parsing and validation of all incoming SIMPLE source code. It tokenizes the source and validates tokens to ensure that they abide by the grammatical rules specified. Once validation is successful, important source information is then inserted into the PKB via an inserter facade on the fly.

2.2 SP Architecture

To handle design issues related to source evaluation, the SP is decomposed into 4 sub-components:

1. Token (Token, TokenStore)
2. Source Tokenizer (Lexer)
3. Source Parser (Conditional Expression and Normal Parser)
4. Expression Converter

The following architecture diagram showcases the relationships between the sub-components:

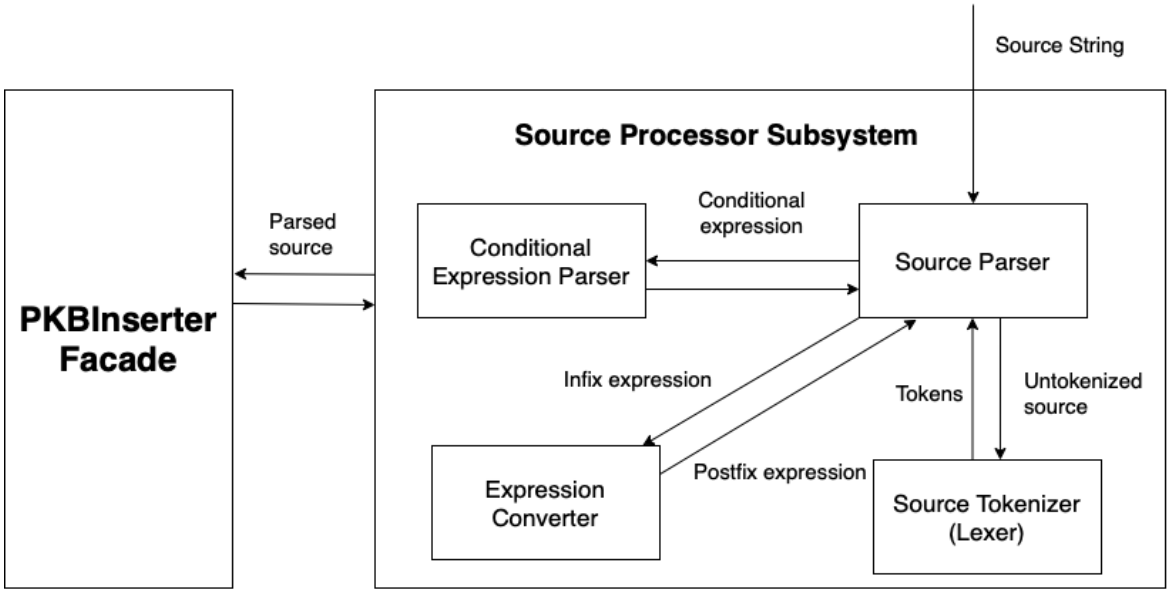

Figure 2.1: SP Architecture Diagram

The decision for the diagram shown in Figure 2.1 is justified in the following subsections.

2.2.1 Design Decision: Aggregation of Conditional and Normal Expression Parsers

Approach 1:
Combining the Parsers for Conditional Expressions and Normal Queries as a single Parser.

Advantage:

- Less code duplication, as conditional expressions are similarly decomposed to factors and terms, just like normal expressions.

Disadvantage:

- Less separation of concerns, as there are significant differences between the grammar syntax for conditional expressions and normal expressions (i.e. parenthesis handling).
- Lower extensibility, as any modifications to the parsing of conditional expressions will require the parsing of normal expressions to be modified as well.

Approach 2:
Splitting the parsing of conditional expressions and normal expressions into two separate Parsers. Conditional expressions will be handled separately.

Advantage:

- Higher separation of concerns as different types of expressions are handled by their respective Parsers in charge.
- More extensible, as changes to the way conditional expressions are parsed can be implemented without affecting the parsing of normal expressions.

Disadvantage:
Greater amount of code duplication, violating the DRY principle. (Do not repeat)

Justification on Final Choice (Approach 2):
Approach 2 was implemented as code duplication is not a significant concern. Conditional expression parsing and normal expression parsing are significantly different and require different data structures to handle. Separation of concerns is more important as it makes testing and future modifications straightforward.

2.2.2 Design Decision: Inserting Objects into the PKB via a Facade or Directly

Approach 1:
Directly inserting the objects into the PKB

Advantage:

- Less testing code due to the absence of the additional facade component.

Disadvantage:

- Higher coupling, since the PKB and the SP will be directly interacting with each other. Also, as the insertion of objects can be highly complex due to the need to

determine AST information, there might be significant pre-processing that is exposed to the SP, which should not be its concern.
Approach 2: Delegating the responsibility of inserting objects into the PKB to a PKB Inserter Facade class, and simply interacting with the facade. **Advantage:** ● Separation of concerns, as unnecessary details of the PKB are hidden from the Source Processor. ● Ease of development as the facade class exposes pre-defined APIs that the SP can call, without having to wait for the PKB to be fully functional. ● Looser coupling, due to an additional level of indirection. **Disadvantage:** ● Larger amount of test code has to be written due to the presence of an additional facade component. This might slow down the development process.
Justification on Final Choice (Approach 2): It would make the most sense to rely on the facade to hide complex validation and pre-computation processes that the PKB goes through in order to construct the AST. In this case, reducing coupling was the primary concern, since future implementations of the SPA might change, and it is critical to be able to make modifications easily. Higher extensibility due to lower coupling was therefore valued over the need for additional testing requirements.

Finally, the functionalities of tokenizing, parsing and conversion are separated into independent sub-components to increase cohesion, since they work on the same issue. This improves the understandability, maintainability and reusability of the components created.

2.3 SP Parsing and Validation

In this section, the design for the parsing and validation of SIMPLE programs will be discussed. The following class diagram provides an overview of the interaction between the SP subcomponents:

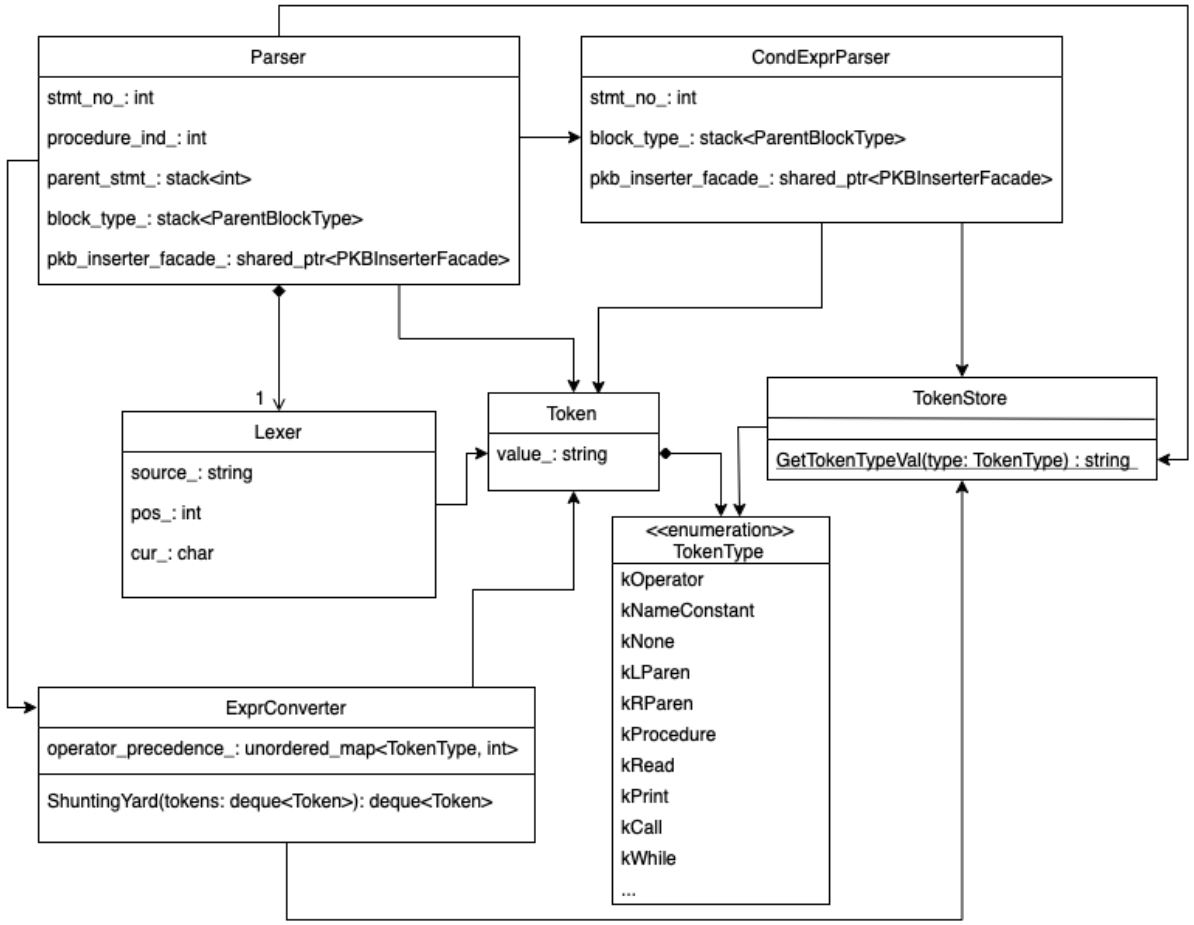

Figure 2.2: SP Parser Class Diagram

Individual components and their descriptions are briefly discussed below. Specific implementation details are outlined in the respective sections of the components.

Component	**Description**
Token	Representation of the final output of the tokenizer. The SIMPLE program will be converted from characters to respective tokens. Each token will carry a type (Token Type), and a value string.
TokenStore	Serves as a general store for different tokens.
TokenType Enum	Stores the different types of tokens that can be present in any source.
Lexer / Tokenizer	Reads through the source string character by character to generate tokens that can be validated and processed by the Parsers.
Parser	Calls the Lexer to retrieve tokens and validates the syntax based on SPA grammar rules. Also adds important object information to the PKB.

CondExprParser	Has similar responsibility as the Parser but applied on conditional expressions only.
ExprConverter	Converts infix expressions to postfix expressions.

Figure 2.3: Responsibilities of each SP Component

2.3.1 Tokens

The SIMPLE program is input from the autotester as a string, and characters are converted to tokens by the tokenizer. The following two approaches were considered.

Approach 1: Parsing SIMPLE program into characters instead of token types

Advantages:

- Code development will be easier since the source string can be directly parsed and validated.
- Less testing, due to the absence of the token classes.

Disadvantages:

- Less abstraction, as the source file is handled directly with characters.
- Less extendible, since there might be new or additional token types added in the future that would require manual modification of functionality, instead of simply adding a new token type enumeration member.
- Less separation of concerns between lexical and grammar validation, as the Parser will have to deal with invalid characters while ensuring proper adherence to grammar rules.

Approach 2: Parsing SIMPLE program into predefined token types

Advantages:

- Responsibilities of lexical validation can be separated from grammar validation.
- Easier development, since the Lexer and Parser can be developed in parallel.

Disadvantages:

- More unit testing is required to ensure the proper functioning of the Lexer and Parser, which might slow down the development process.

Justification on Final Choice (Approach 2):

Approach 2 is chosen because it allows a faster development process, discounting unit tests. More importantly, responsibilities are well-defined between the Lexer and Parser, and tokens are treated as an abstraction of the entire original source. Hence, the Parser does not need to be concerned about lexical errors, and can just focus on ensuring that

the source abides by the SIMPLE grammar rules.

Additionally, each token stores a token type, as well as its associated value. The various token types used are also outlined below:

Keyword Token Types	kProcedure, kRead, kPrint, kCall, kWhile, kIf, kThen, kElse
Operator Token Types	kEquals, kNot, kAnd, kOr, kGreater, kGreaterOrEqual, kLesser, kLesserOrEqual, kEquality, kNotEquals, kAdd, kSubtract, kMult, kDiv, kMod
Names	kName
Constants	kInteger
Brackets	kLParen, kRParen, kLBrac, kRBrac
End of Line, End of File	kEOL, kEOF

Figure 2.4: Token Types Table

While the keyword token types exist, they are used for comparison purposes only. However, keywords such as while, if, else, print are recognized as kName tokens throughout the entire source. The Parser then handles the situation where a keyword token is expected and makes a comparison of its value to the value of the expected keyword token type. It is not possible to parse all keywords as keyword tokens because keywords can also be used as variable names for assignment statements, and it would be challenging to distinguish between variable names and keywords if the Lexer distinguishes them at the tokenizing stage.

To illustrate the tokenization step better, an example source program is presented below, together with the tokens that are generated. Figure 2.5 gives an example SIMPLE program to be tokenized.

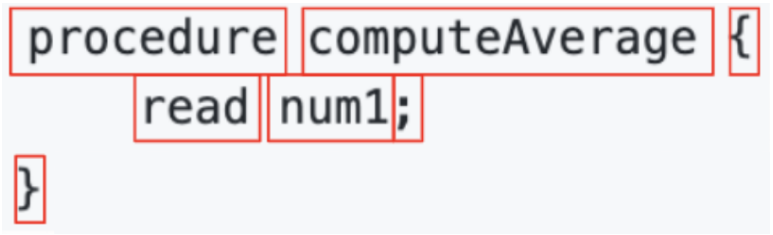

Figure 2.5: Example SIMPLE Program for Tokenization

The following is a tokenized representation of the SIMPLE program.

SIMPLE program string	Token Type	Token Value
procedure	kName	“procedure”
computeAverage	kName	“computeAverage”
{	kLBrac	“{“
read	kName	“read”
num1	kName	“num1”
;	kEOL	“;”
;	kRBrac	“;”

Figure 2.6: Tokens Generated from Example Source

There are a total of 7 tokens that are generated from the example source, and each token is encapsulated by its own unique type and value. These tokens also represent the final output of the lexer.

2.3.2 Lexer and Tokenizer

This subsection describes how tokens are generated. The following is an abridged sequence diagram showing some of the interactions that occur inside of the Lexer. Not all interactions are shown as there are too many alternatives that follow a similar approach.

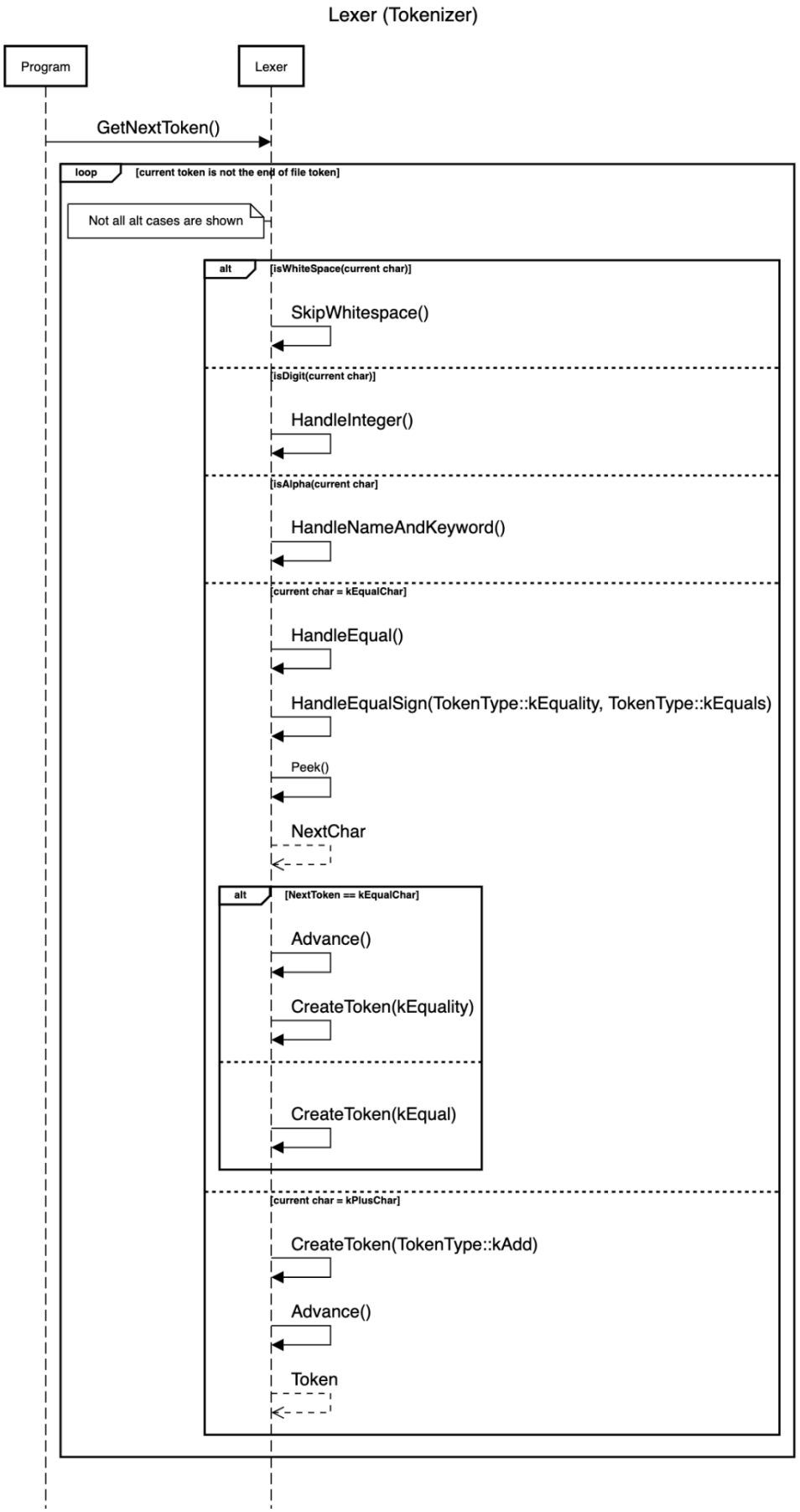

Figure 2.7: SIMPLE Source Tokenization Sequence Diagram

1. The GetNextToken() method of the Lexer is a public method that can be called by the Parser when it intends to retrieve the next token of the source. The Lexer keeps track of the current position with position and current position pointers.

2. When the GetNextToken() method is called, the Lexer defaults handling to different sub-routines depending on the character that is detected in the current position. For simpler characters that do not require any further checks, the Lexer simply calls the CreateToken() function to wrap the character in its respective token type to be returned to the calling Parser.

3. Some characters would require more handling than others. For example, when an equal sign is detected, the HandleEqual() method will be called, which in turn calls HandleEqualSign() to determine if an equals (“=”) token should be created, or the equality (“==”) token. This is where the Peek() function of the Lexer comes in handy.

4. The Peek() function of the Lexer allows the Lexer to be able to determine the next character of the source, without shifting the current position index. If the next character in the source is another equals, then an equality token is returned instead of an equals token.

2.3.3 Parser

The Source Parser makes use of a recursive-descent algorithm. Parsing is also predictive in nature due to the PeekNextToken() functionality that is implemented in the source tokenizer. This enables the ability to uncover the next token without adjusting the current position of the tokenizer, and intelligently choose the next parse method to call. The Parser also makes use of the SIMPLE grammar, and a flowchart is shown below to illustrate the flow of the recursive algorithm.

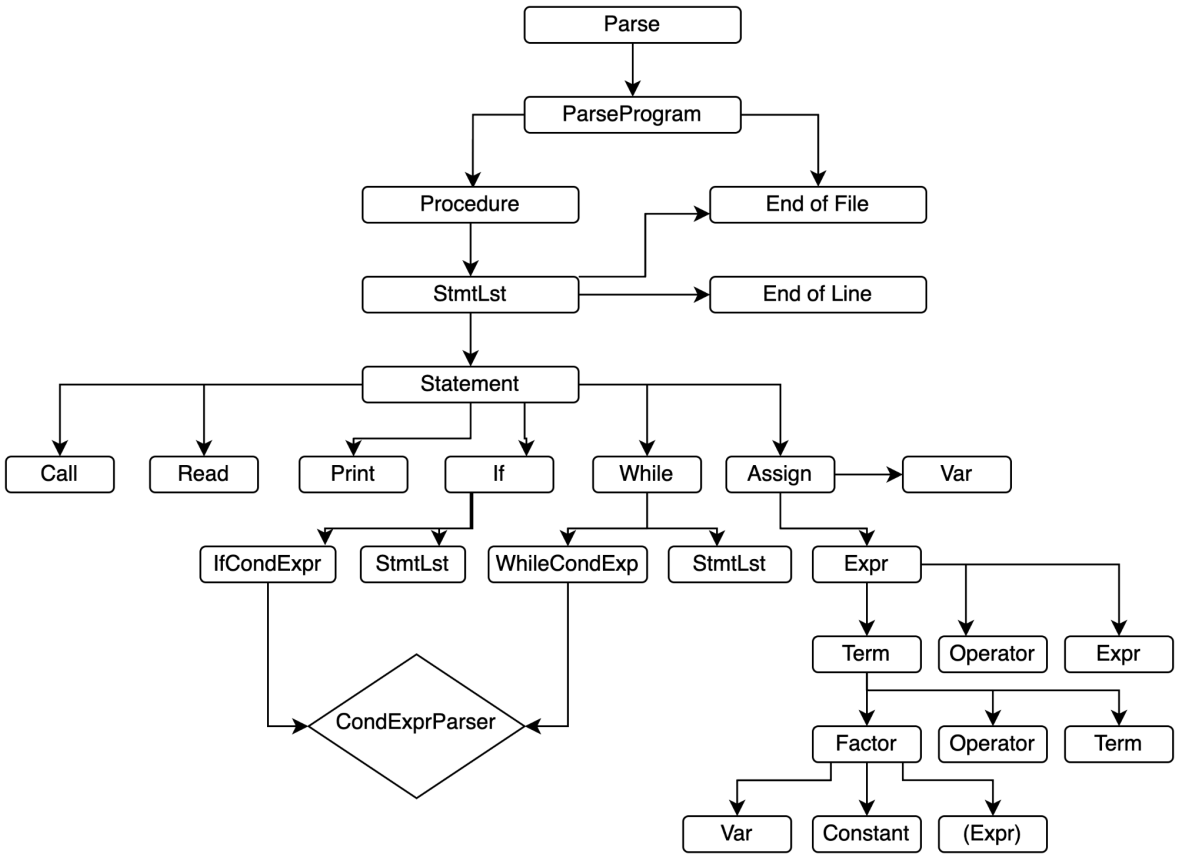

Figure 2.8: SIMPLE Source Recursive Descent Parsing Flowchart

2.3.4 Key APIs: SP Parsing

This section details critical APIs necessary for the functioning of the Source Parser. Recursive parse methods are not included, as they are part of the Recursive Descent algorithm.

TOKEN Eat(TOKEN_TYPE type, BOOL is_keyword, TOKEN_TYPE expected_keyword_type)
Enables the Parser to "consume" the next token as tokenized by the Lexer, and ensure that it is of the expected token type. Otherwise, an exception will be thrown to indicate failed source parsing (due to incompatibility with the SPA grammar rules)

TOKEN EatAny(TOKEN_TYPE_LIST expected_types)
Carries out similar functionality as Eat, but allows multiple different token types to be expected. They can be passed inside of a vector. This makes checking for different operator types much more convenient, instead of using multiple if else blocks. This is especially useful for the evaluation of expressions where different operator types can be expected.

TOKEN_DEQUE EatUntil(TOKEN_TYPE type)
Allows the Parser to consume tokens until a token of the same type as the expected token type is encountered. Returns the whole list of tokens consumed in a deque. This is especially useful for the parsing of conditional expressions, since the entire conditional expression needs to be extracted and passed over to the CondExprParser.

VOID AddExprToPkb(TOKEN_DEQUE expr_tokens)
Inserts important information about expressions into the PKB Inserter Facade class, for processing by the PKB. Expressions are passed in as a deque of tokens, which have already been converted to the postfix form by the expression converter class.

2.3.5 Conditional Expression Parser

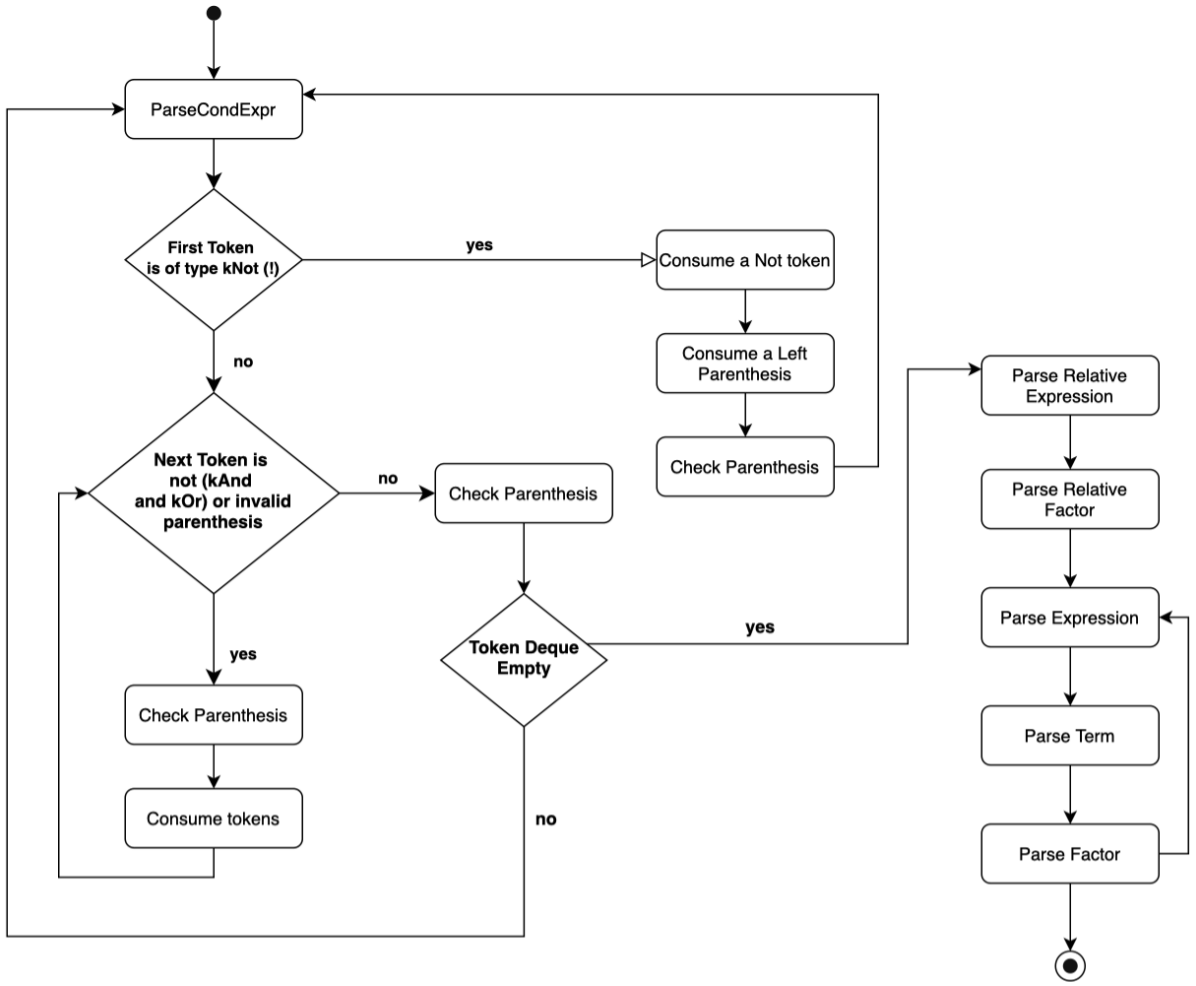

Figure 2.9: Conditional Expression Parser Flow Diagram

A flow diagram is shown above to illustrate how the conditional expression Parser functions. Details are left out in terms of what tokens are consumed. Conditional expressions cannot

be validated in the same manner as normal expressions, since the grammar rules used for both are different. It is also not possible to validate the differences between a conditional expression that starts with a left parenthesis from a normal expression that starts with a left parenthesis. However, conditional expressions only appear in the control blocks of if and while loops, and the responsibility of validating the conditional expressions can be delegated to a Conditional Expression Parser.

The Conditional Expression Parser differentiates between two main types of parsing possibilities:

1. Relative Expressions
2. Recursive Conditional Expressions

If it is a relative expression, then there are no kAnd or kOr tokens and the ParseRelExpr() method is then called to process the relative expression. Otherwise, the Parser will recursively call the ParseCondExpr() method. The idea is to recursively process both the left and right sides separately while ensuring that the parentheses used are valid at the end of parsing. The kAnd or kOr tokens are used to separate both left and right hand sides.

2.3.6 Expression Converter

For pattern expressions, it is important to be able to parse them in postfix form, rather than the infix form that is supplied in the source. Hence, this responsibility is delegated to an Expression Converter class, which makes use of an adapted Shunting Yard implementation to do the conversion. Details will not be included, since it is a relatively well-known algorithm that makes use of a deque and a stack.

The expression converter takes in an unordered map of operator precedences, which allows the converter to recognize the precedences of any operator supplied. This makes the converter extensible for any future modifications, additions of operators, or the changing of precedences. The operator precedence map used for the SPA project is shown below, and this map is passed as a parameter during the converter's initialization.

Key	**Operator Precedence**
Multiplication	2
Division	2
Modulo	2
Addition	1
Subtraction	1

Figure 2.10: Operator Precedence Map

In order for the Expression Converter to be extensible and work with the PQL component as well, an adaptor method GetExprToken() is included to convert existing token types from the token store to types that are specific to only the expression converter. The following types are explicitly recognized by the expression converter:

- kOperator (Operator types)
- kNameConstant (Variable names and constant values)
- kLParen (Left parenthesis)
- kRParen (Right parenthesis)
- kNone (Does not match any token)

2.4 Interaction with PKB Inserter Facade

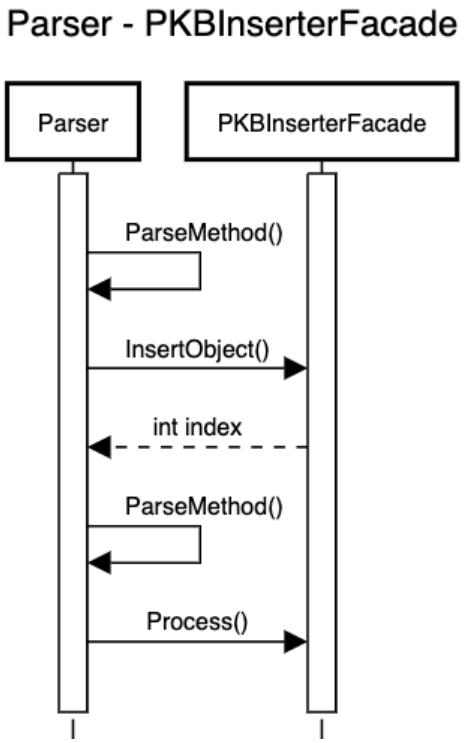

Figure 2.11: Parser-PKB Inserter Facade Interaction Sequence Diagram

The sequence diagram above illustrates the interaction between the Parser and the PKB Inserter Facade. Not all parse methods require the extraction of design information. For example, ParseProgram() does not require any extraction of design information, so the Parser does not call any of the APIs that the facade provides. The facade often returns an index, which uniquely identifies the stored object. This is also used as the statement number / procedure index in the program. An outline is provided below to indicate the parse methods where extraction of important design information takes place.

Parse Method	**Design Information Extracted (API calls to PKB Inserter Facade)**
Parse	Process()

	At the end of parsing, the Parser calls the Process() method of the facade to process the inserted parameters.
ParseProcedure	InsertProcedure() The following parameters are taken in to the above methods: ● Procedure Name: name of procedure to be inserted
ParseRead ParseCall ParsePrint	InsertReadStmt() InsertPrintStmt() InsertCallStmt() The following parameters are taken in to the above methods: ● Name: variable name / called procedure name ● Procedure Index: the procedure index where this statement appears ● Parent Statement: the parent of this statement ● Block type: the type of block where this statement is contained in (i.e. while block, if block, else block) These methods then return a statement index, which can be used to track statement flows throughout parsing.
ParseAssign	InsertAssignStmt() The following parameters are taken in: ● Variable: the left hand side variable name to be assigned a value ● Procedure Index: the procedure index where this statement appears ● Parent Statement: the parent of this statement ● Block type: the type of block where this statement is contained in (i.e. while block, if block, else block)
ParseIf ParseWhile	InsertIfStmt() InsertWhileStmt() The following parameters are taken in: ● Procedure Index: the procedure index where this statement appears ● Parent Statement: the parent of this statement

	• Block type: the type of block where this statement is contained in (i.e. while block, if block, else block)
ParseFactor (CondExprParser)	InsertUsingVar() InsertConstant() The following parameters are taken in: • Name/Value: indicates the name of the variable or the value of the constant • Statement number: the statement number where this variable/constant appears • Statement type: the type of the statement
AddExprToPkb (Parser)	InsertUsingVar() InsertConstant() InsertOperator() The parameters for the InsertConstant() and InsertOperator() are the same as the ones for the conditional expression Parser. The parameters for the InsertOperator() function is outlined below: • Op Value: indicates the operator value (i.e. "+") • Statement number: the statement number where this operator appears

Figure 2.12: Parse methods for extraction of design information

2.5 Design Decision: Semantic Validation of SIMPLE Program

The semantic validation of a SIMPLE program consists of checking for cyclic dependencies between procedures and if all call statements call procedures that exist. The following discusses if this semantic validation should be performed in an intermediary data structure in the SP or in the PKB Inserter Facade.

Approach 1: Semantic validation in the PKB Inserter Facade

Advantages:

- Easier to develop and write code for, due to reduced complexity from having an additional component handle the validation of design information.

Disadvantages:

- Less separation of concerns in the PKB, since validation and data insertion is performed in a single component

Approach 2: Semantic validation in an intermediary data structure in the SP

Advantages:

- Better separation of concerns as the validation of design information is separated from insertion.

Disadvantages:

- More implementation effort as an additional component has to be designed for this validation.
- Slightly more runtime overhead as this data structure has to be instantiated and validation has to be performed on it.

Justification on Final Choice (Approach 1):

The time complexity of both approaches are relatively similar, O(N), but from an implementation perspective, it is much more straightforward to handle insertion of design information together with validation. Furthermore, validation logic is simple and could be easily separated within the insertion logic in the PKB Inserter Facade. Hence, approach 1 was chosen.

The actual semantic validation of the SIMPLE program is discussed in subsection 3.5.4.

3 Program Knowledge Base

In this section, the design and APIs of the PKB will be discussed.

3.1 Overview

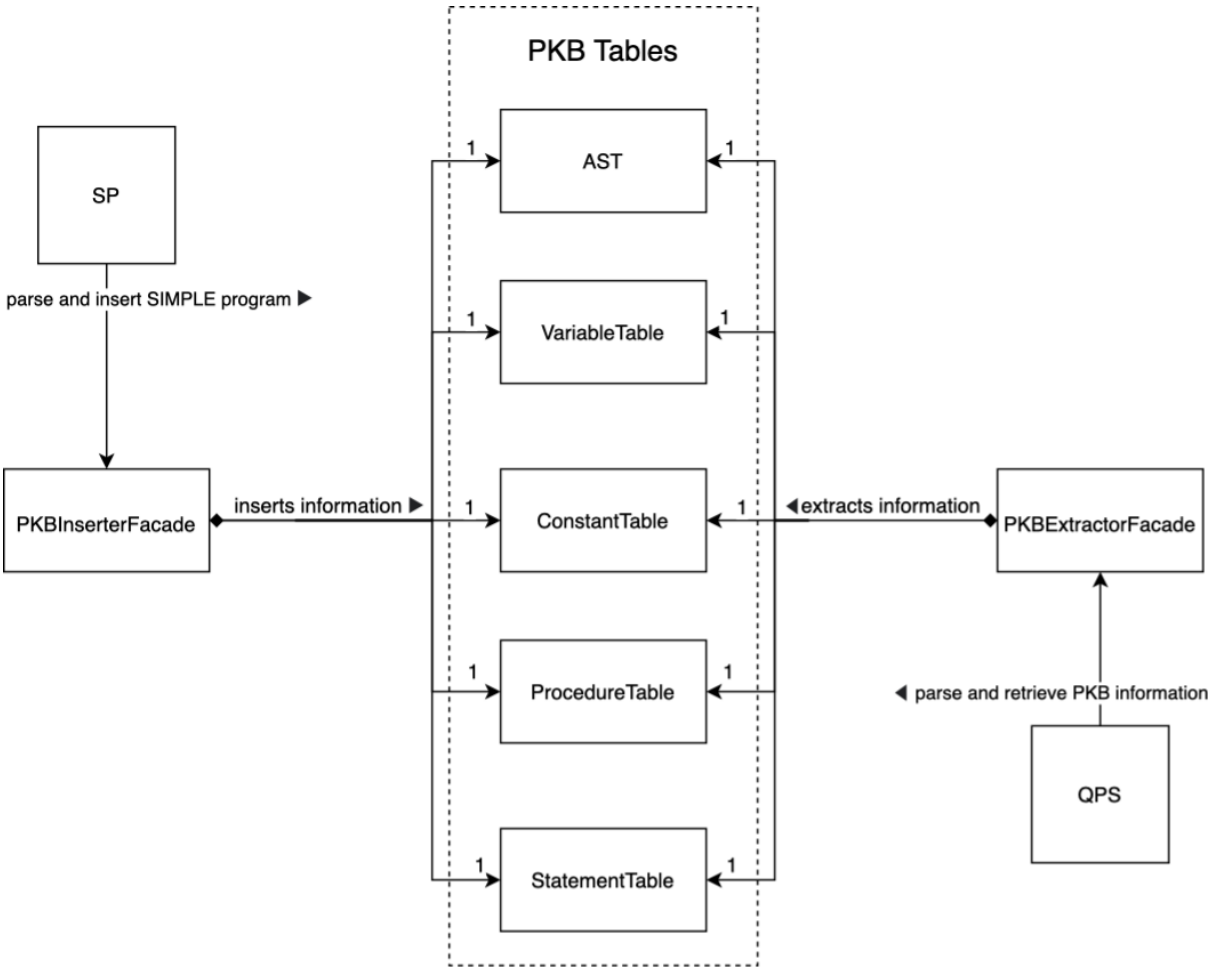

Figure 3.1: PKB Component Diagram

The PKB is made up of the AST and the following tables: Variable Table, Constant Table, Procedure Table, Statement Table. To insert information into the AST and tables, an Inserter Facade class is used. It validates the information to be stored as well and provides a set of APIs to SP. To extract information from the AST and tables, a separate Extractor Facade class is used to provide the APIs for the QPS. The utilization of these facade classes helps increase cohesion and decrease coupling, as the responsibilities of data insertion and extraction are separated.

3.2 AST

This section elaborates on the design of the AST.

3.2.1 AST Design

The AST is implemented as a vector of Stmt objects. The index of this vector corresponds to the statement number of the object, offset by 1.

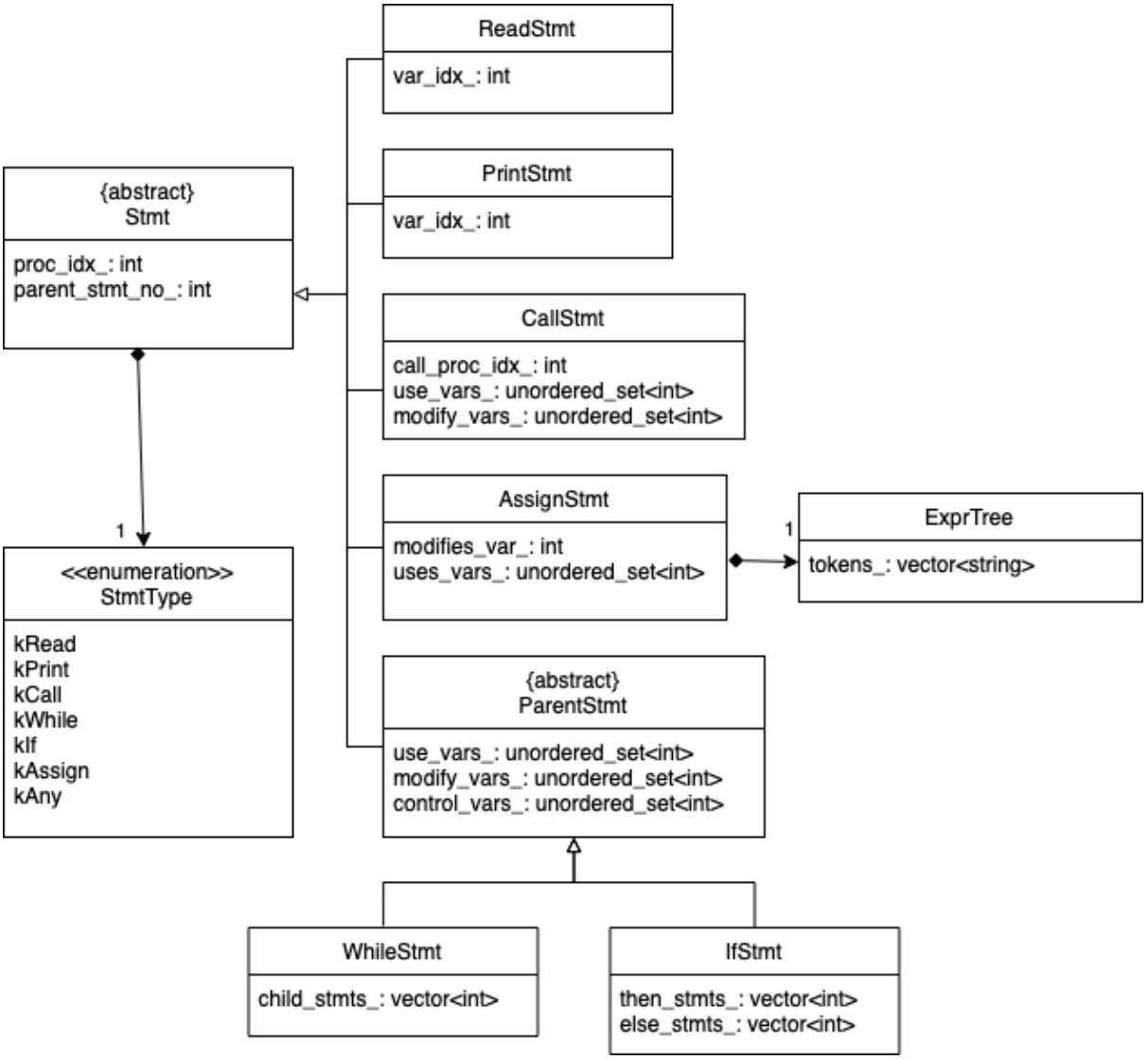

Figure 3.2: Statement Object Class Diagram

The figure above describes how the Stmt objects are implemented. Briefly, each type of statement inherits from the parent abstract class Stmt. Another abstract class, ParentStmt inherits from the main Stmt object class. Only IfStmt and WhileStmt inherit from ParentStmt. A StmtType enumeration is also used to identify the type of each Stmt object. Each Stmt object also stores the index of variables that are being used or modified, to facilitate the evaluation of Uses or Modifies queries.

AssignStmts also consist of an ExprTree attribute, which facilitates in evaluating PQL pattern queries. This ExprTree attribute contains a vector of operators, variables and constants of the assignment statement in postfix order. For example, an assignment statement with right hand side expression: “x + y * 3” is stored as “x”, “y”, “3”, “*”, “+” in the ExprTree.

3.2.2 Design Decision: AST Data Structure

Approach 1: Storing the AST as a vector of Stmt objects

Advantages:

- Easy to implement, as the AST is just a vector.
- Efficient queries in O(1) time, as access to Stmt objects in the AST just requires access to the vector.

Disadvantages:

- May not be extensible when more queries are added (although this is not the case for Advanced SPA). This may occur as the vector does not have a tree representation of the SIMPLE program. It may not be possible to evaluate the expressions of assignment statements or execute while loops.
- AssignStmt stores an ExprTree object as an additional attribute, which incurs additional memory overhead.

Approach 2: Storing the AST as a graph

Advantages:

- Stores most information of the SIMPLE program, which makes it extensible.
- Does not incur additional memory overhead, as expression trees are embedded as part of the AST.

Disadvantages:

- Difficult to implement, as many types of nodes need to be considered
- Queries may take $\Omega(N)$ time, as querying for a statement requires traversing across the entire AST.

Justification on Final Choice (Approach 1):

As runtime efficiency is a requirement, the AST is implemented as a vector of Stmt objects. Furthermore, given time constraints, it was also less feasible to implement the AST as a graph. Should the need to store additional information on the SIMPLE program arise, additional data structures can be used to supplement the vector implementation.

3.3 PKB Tables

This section elaborates on the design of the tables used in the PKB.

3.3.1 PKB Table Design

PKB tables are implemented using a BiMap of table records. Table records refer to a class that encapsulates the data of each row in the table. The BiMap is a data structure with the following properties:

- O(1) element access via integer indexes
- O(1) element access via a key of any type
- Supports ordering via integer indexes

The BiMap is implemented using a vector and a hashmap. The vector stores all the elements in its index order. The hashmap stores keys of any type and maps it to the index of the vector. The following provides an abstract representation of the BiMap:

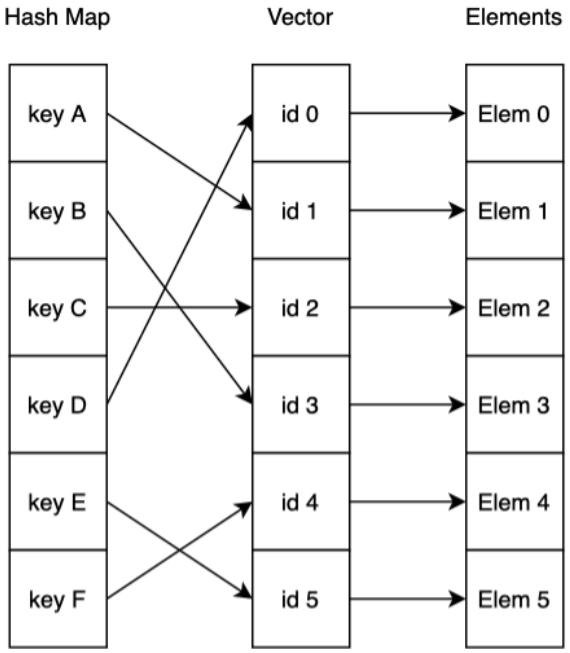

Figure 3.3: BiMap Abstract Representation

3.3.2 Design Decision: Table Data Structure

Approach 1: Implementing tables with the BiMap

Advantages:

- Efficient access time to table records via indexes and keys
- Provides indexing, which reduces memory used in other tables, as the index, rather than the key (typically a string) can be used
- Provides ordering to the tables, allowing tests to be more easily written

Disadvantages:

- The BiMap data structure is implemented with a vector and hashmap, which incurs more memory overhead

Approach 2: Implementing tables as a hashmap of keys of any type to Table records

Advantages:

- Less memory overhead
- Efficient access to table records via keys

Disadvantages:

- Does not provide indexing. As such, other tables have to store the key in their table records. More memory is consumed in the implementation of tables, especially if the keys are big. For example, if Variable Table were to store only its key (a variable name string) other tables such as Procedure Table would have to store this key, instead of an integer index.

Justification on Final Choice (Approach 1):

As keys could become arbitrarily large, such as variable names, it is important that these are not duplicated across tables. Otherwise, the memory consumption would be too high. Hence, using a data structure that provides indexing and efficient access was the most important factor in deciding on the design of the tables.

The following sections describe the design of each table and provide an accompanying example instance. The key of these tables are marked with '[KEY]' as well.

3.3.3 Variable Table

Index	Variable Name [KEY]	Statements using variable	Statements modifying variable	Procedures using variable	Procedures modifying variable
0	"x"	3, 4, 6	7	NONE	0, 1
1	"y"	3, 5	6, 8	NONE	1
2	"counter"	8	NONE	0, 1	NONE

Figure 3.4: Variable Table Example Instance

Variable tables store the following:

- Variable name: string
- Statement that use the variable: unordered set of statement indexes
- Statement that modify the variable: unordered set of statement indexes
- Procedures that use the variable: unordered set of procedure indexes
- Procedures that modify the variable: unordered set of procedure indexes

Unordered sets are used to provide efficient access to Use and Modify queries, when searching for statement numbers or procedures that use or modify a certain variable.

3.3.4 Constant Table

Index	Constant [KEY]
0	1
1	100
2	11

Figure 3.5: Variable Table Example Instance

Constant tables store the following:

- Constant: integer

Constants are stored as integers, as queries such as s.stmt# = constant.value could be queried.

3.3.5 Procedure Table

Index	Proc Name [KEY]	Child Statement Numbers	Used Vars	Modified Vars	Call Procs	Callee Procs	Control Flow Graph
0	“main”	1, 2, 3, 4	0, 1, 2, 3, 4, 5, 6, 7, 8, 10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10	1	NONE	main_cfg
1	“foo”	6, 7, 8	0, 2, 3, 4, 5, 6, 7, 8, 10	2, 3, 4, 7, 8, 9, 10	2	0	foo_cfg
2	“bar”	9, 10, 11, 12, 13	0, 2, 5, 8, 10	2, 3, 4, 7, 10	NONE	1	bar_cfg

Figure 3.6: Procedure Table Example Instance

Procedure tables store the following:

- Procedure name: string
- Direct child statement numbers: vector of statement indexes
- Variables that are used: unordered set of variable indexes
- Variables that are modified: unordered set of variable indexes

- Call procedures: unordered set of procedure indexes
- Callee procedures: unordered set of procedure indexes
- Call Star procedures: unordered set of procedure indexes
- Callee Star procedures: unordered set of procedure indexes
- CFG, which will be discussed in section 3.4

The child statements are stored in ascending order. When querying for Follows or Follows* relations, binary search can be used to search for particular statement indexes directly. Unordered sets are implemented for variables used and modified by the statement to provide efficient access when querying for Uses or Modifies relationships. Call and Callee procedures, as with their star variants, are stored for O(1) Calls and Call* queries. Note that Call Star and Callee Star procedure columns are omitted in Figure 3.5 due to space constraints.

3.3.6 Statement Table

Statement Type	**Statement numbers**
Read	11, 13
Print	6, 14
Assign	1, 4, 7, 8
If	2, 10
While	3
Call	5

Figure 3.7: Statement Table Example Instance

Statement table stores each type of statement type as a vector of statement indexes. This is also the only table in the PKB that does not use a BiMap, as the types of statements are fixed. The statement table does not support any types of queries directly. Instead, it categorizes each statement index into their statement type. As such, when queries that involve certain statement types are executed, there is no need to iterate through all statement indexes and check if they are of the required type. The statement table can be accessed to provide all statement numbers of the required type.

Since the statement numbers retrieved will be iterated over, and that searches on statement indexes will not be performed in the statement table, a vector is used to store the statement indexes.

3.4 CFG

This section discusses the design of the CFG.

3.4.1 CFG Design

The CFG is implemented with five different types of nodes:

- NormalNode (for assign, read, print, call statements)
- IfNode
- WhileNode
- TerminalNode (for the start and end of the CFG)
- DummyNode (to connect the two IfNode branches)

Each node stores the statement number it represents. This is similar to the CFG implementation presented in lectures. The main differences are:

- each statement belongs to each own node,
- branches of if nodes are always connected by DummyNodes,
- TerminalNodes are used to demarcate the start and end of the CFG,
- each node knows about the next and previous node(s)

The additional nodes: TerminalNode and DummyNode, were implemented for easier CFG creation and traversal. Each node also knows about its next and previous nodes to facilitate queries such as Next(5, _) and Next(_, 5). Furthermore, the CFG is indexed with a BiMap to allow for O(1) access to specific required nodes. The following is an example instance of a CFG:

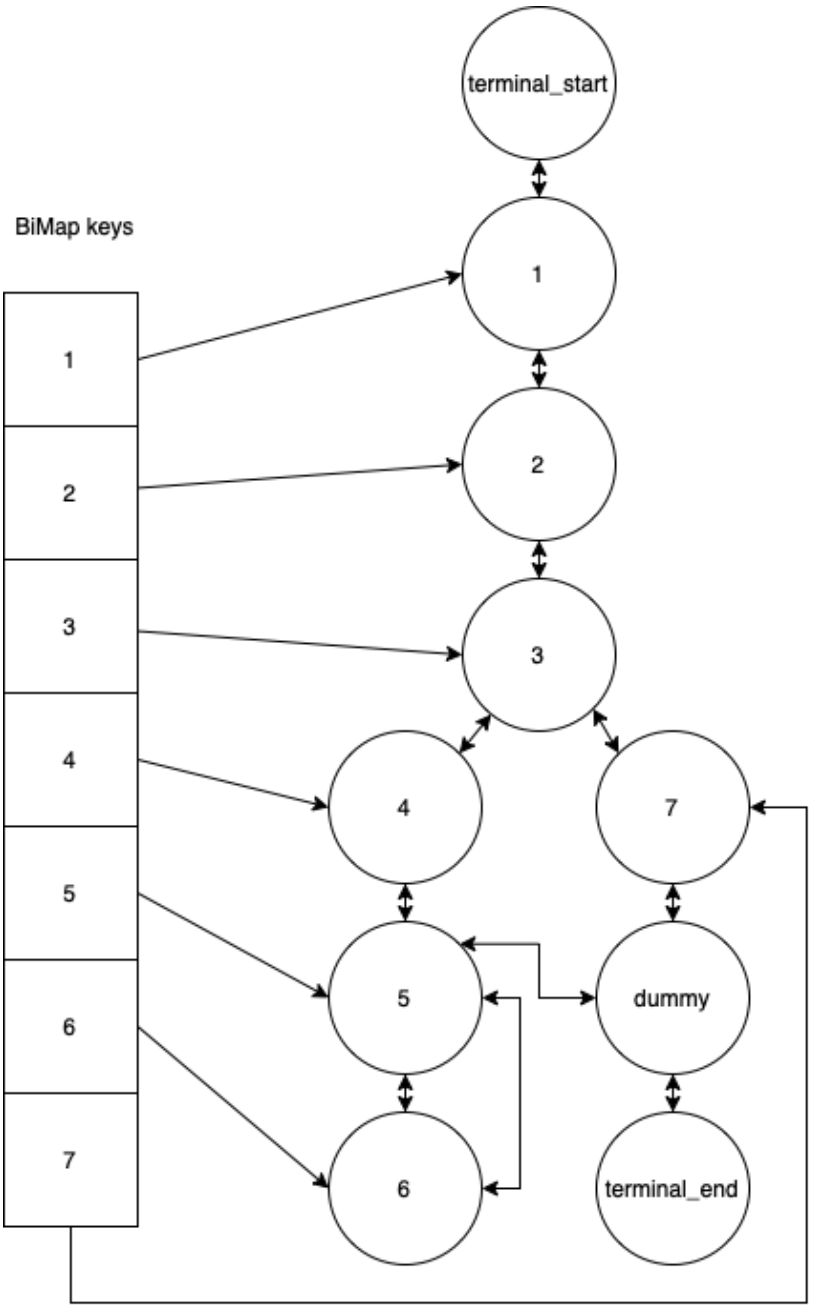

Figure 3.8: CFG Example Instance

Note that Node 3 is an IfNode, Node 5 is a WhileNode. The rest of the nodes are either NormalNodes, TerminalNodes or DummyNodes.

3.4.2 CFG Traversal Strategy

Breadth first search (BFS) was implemented to traverse the CFG, to both get the next and previous nodes. The following activity diagram shows how the next immediate nodes are obtained during an iteration of BFS.

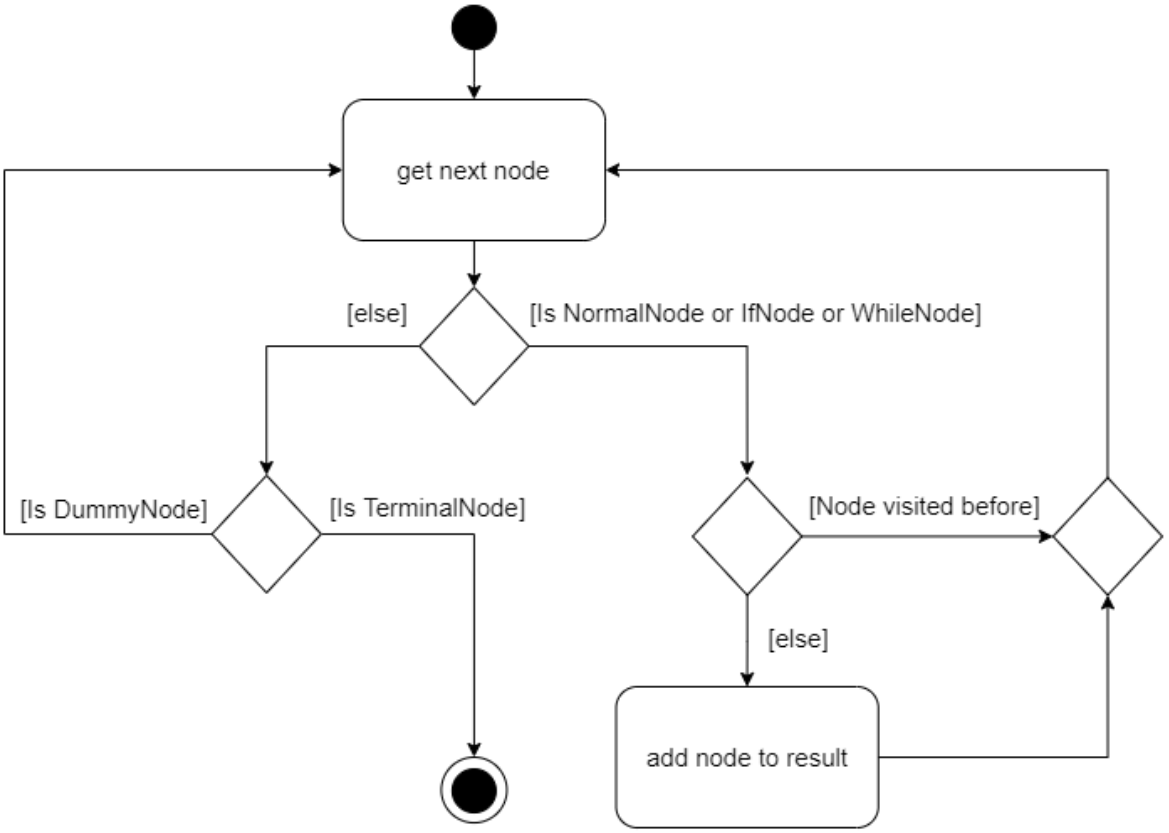

Figure 3.9: CFG BFS Getting Next Nodes

To avoid appending TerminalNodes or DummyNodes to the result, it is necessary to check and handle these two cases in each iteration. Additionally, getting all the next nodes of dummy nodes would allow the algorithm to function correctly. Otherwise, the BFS traversal will always terminate upon reaching a DummyNode.

3.4.3 Design Decision: CFG Traversal

Approach 1: Implementing BFS **Advantages:** ● Less memory consumed, as only nodes in the current level are stored in the queue ● As BFS traverses the CFG level by level, it ensures that all sequential nodes visited can be cached **Disadvantages:** ● Slightly slower run time, since each node in each level is visited
Approach 2: Implementing DFS **Advantages:**

- Slightly faster run time, as once a specific node in the current path is reached, the result can be returned immediately

Disadvantages:

- More memory consumed, as a stack of nodes has to be kept in memory for each branch visited. The longer the SIMPLE program, the bigger the memory used.

Justification on Final Choice (Approach 1):

The assumption that the amount of nesting of if statements in the SIMPLE program is made. If statements are the only case where the CFG increases in width. Hence, it is expected that CFGs would usually be deeper. This means that at each iteration of BFS, the number of nodes stored in the queue will be kept to a minimum. Also, the runtime of BFS would be comparable to that of DFS as the number of nodes in a level would be small.

3.5 PKB Inserter Facade

This section explains how information is inserted into the PKB Inserter Facade. It also discusses its processing and validation logic.

3.5.1 PKB Data Insertion

The PKB Inserter Facade contains the following as attributes:

- pointers to all PKB tables
- pointer to the AST
- call statement cache

The call statement cache stores the procedures that call statements call and validates that all these procedures exist. It is a map of call statement numbers to the procedure name it calls. This is further elaborated on in section 3.5.4.

As mentioned above, the SP passes information regarding procedures and statements into the Inserter Facade and only provides basic information such as statement numbers, variable names, procedure names, etc. The Inserter Facade then inserts the relevant information into the PKB tables and AST. The following figure provides an overview on how these information are stored.

Design Entity	SP parameters	Affected PKB tables and/or AST
Procedure	• procedure name	• procedure table creates a new entry with this procedure name
Constant	• constant name • statement number	• constant table upserts an entry with this constant

	● statement type	● if the statement type is of assignment type, the constant is appended to the ExprTree of the associated AssignStmt
Read Statement	● variable name	● variable table upserts an entry with this variable ● ast creates a new ReadStmt object ● procedure table updates that this variable is being modified
Print Statement	● variable name	● variable table upserts an entry with this variable ● ast creates a new PrintStmt object ● procedure table updates that this variable is being used
Call Statement	● call procedure name	● ast creates a new CallStmt object ● call statement cache is updated to store this call procedure name
While Statement	-	● ast creates a new WhileStmt object
If Statement	-	● ast creates a new IfStmt object
Assign Statement	● variable name	● ast creates a new AssignStmt object ● procedure table indicates that the variable is being modified
Used Variables	● variable name ● statement number ● statement type	● variable table upserts a new entry with this variable ● procedure table updates that this variable is being used ● if statement type is of if or while type, the variable is inserted into the control variables of the associated Stmt ● if statement type is of assignment type, the variable is appended into the ExprTree of the associated AssignStmt
Operators	● operator string ● statement number	● operators only occur in assignment statements and are thus inserted into their ExprTree

Figure 3.10: Storage of Information from SP

For brevity, note that the parent procedure index, parent statement number, and parent block type are omitted from this figure, as the insertion of all statements require these three arguments. The purpose of these information are as follows:

- parent procedure index: statement number is appended to the child procedure statement list of this parent procedure
- parent statement number: statement number is appended to the child statement list of this if or while statement
- parent block type: indicates if the child statement list is to be inserted into a while, if-then or if-else block

Furthermore, note that only variables that are used have their own API for insertion. Variables that are modified are found as a singular entity in read or assignment statements. They can thus be inserted in the read or assignment statement api calls. Also, constants, operators and used variables are inserted in postfix order into the expression tree object of assignment statements via the Shunting Yard algorithm described in section 2.3.6.

The following subsection shows how the SP passes information of a statement into the PKB Inserter Facade and how the Inserter Facade stores it in the PKB tables.

3.5.2 Interaction between SP and PKB

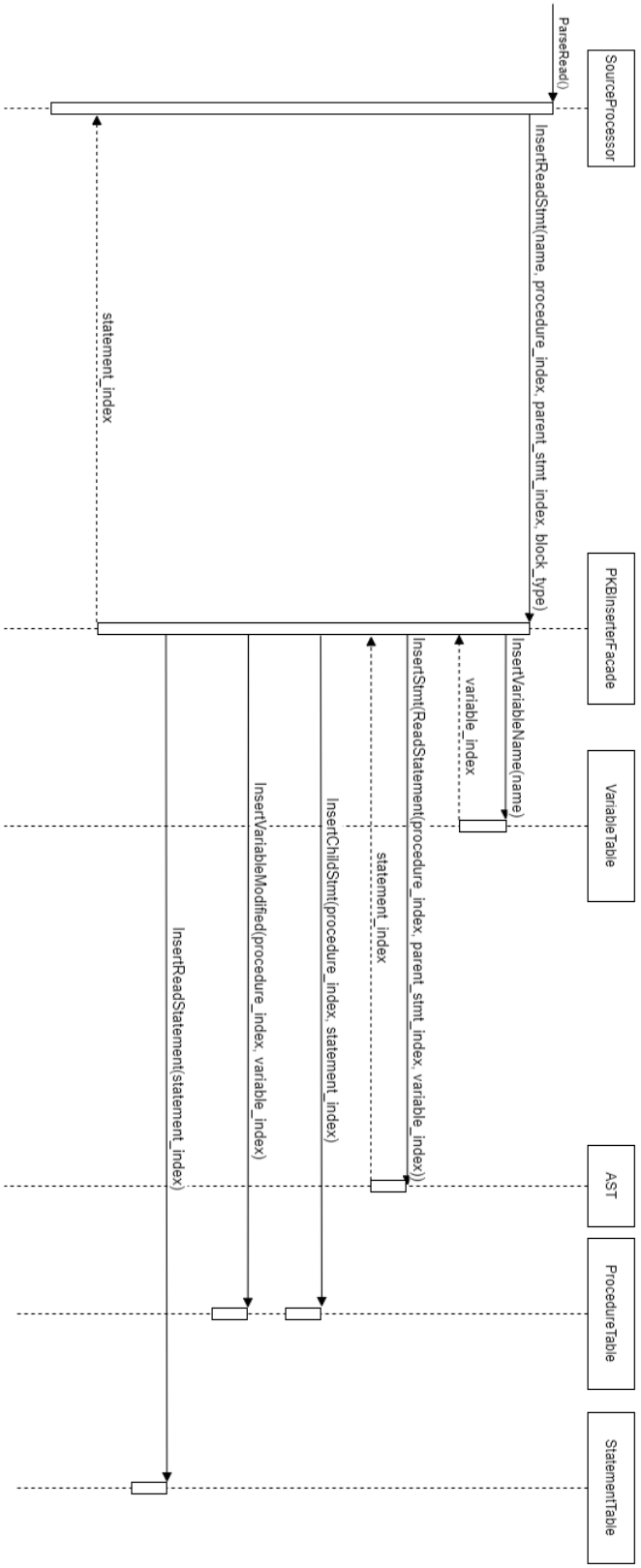

Figure 3.11: SP and PKB Interaction Sequence Diagram

The process starts at the SP. The SP parses a read statement and calls the InsertReadStmt. This is the entry point into the PKB from the SP. All interactions between the SP and the PKB are always done through this inserter facade class. As such, there will never be any interaction between the SP and the tables in the PKB. The PKB Inserter Facade will then call the following methods in subsection 3.5.3 to correctly insert all information of the read statement into the various tables in the PKB.

3.5.3 Key APIs: SP and PKB Interaction

This section details the key APIs brought up in the previous subsection.

Variable Table API

VAR_IDX InsertVariableName(VAR_NAME variable_name)
Inserts a variable record with given variable_name into the variable table and returns its index in the variable table.

AST API

STMT_NO InsertStmt(STATEMENT stmt, PARENT_BLOCK_TYPE type)
Insert statements into the AST and return its index, the statement number.

Procedure Table API

VOID InsertChildStmt(PROC_IDX procedure_index, STMT_NO child_stmt_number)
Inserts the child_stmt_number into the procedure record with given procedure_name in the procedure table.

VOID InsertVariableModified(PROC_IDX procedure_index, VAR_IDX variable_index)
Inserts the given variable_index modified into the procedure record with the given procedure_index in the procedure table.

Statement Table API

VOID InsertReadStatement(STMT_NO statement_number)
Inserts a read statement with the given statement_number.

3.5.4 PKB Insertion Processing and Validation

After all information from the SIMPLE program has been inserted, post processing is executed to perform the following:

1. check for valid procedure calls and cyclic dependencies
2. populate the variables that procedures, if, while, call statements uses or modifies
3. update the variable table with the statements and procedures that modify it
4. create cfgs for each procedure

The following figure details how these steps are performed.

Step	Processing and Validation logic	Rationale
1	● All entries in the call statement cache are checked against the procedure table to see if the procedure exists ● The call statements are updated to reflect the index of the procedure that is being called ● Topological sort is performed on the procedures, starting from procedures without any call statements	● Topological sort is used to determine the order of insertion in step 2 ● Topological sort also detects any cyclic dependencies between procedures. If it is detected, an exception is thrown
2	● Using the topological sorted procedures, each procedure is updated to store the variables its child statements uses or modifies ● Each if, while and call statement within the procedure are also updated to store the variables that its child statement or calling procedure uses or modifies ● The call, calls tar, callee and callee star entries in the procedure table are also updated using this topological sort	● Topological sorted procedures have to be used as procedures that call other procedures will include all the caller procedure's used and modified variables
3	● Each entry in the variable table is updated to reflect the statements and procedures that modify or use it	● The variable table can only be updated at this point after step 2 is completed
4	● CFGs for each procedure are created and inserted into the procedure table ● The AST and direct child statements	

	of the procedure are retrieved for this CFG creation	

Figure 3.12: PKB Insertion Processing and Validation Logic

Note that these processing and validation steps cannot be computed on the fly, as data is inserted into the PKB table statement by statement. For instance, the variables that if statements use are dependent on the statements that are nested within. They can only be updated after all statements have been inserted into the PKB.

3.6 PKB Extractor Facade

This section explains the design of the PKB Extractor Facade. It also describes some of the extraction logic implemented in the class.

3.6.1 PKB Extractor Facade Design

The PKB Extractor Facade consists of the following objects:

1. constant table
2. procedure table
3. statement table
4. variable table
5. AST
6. modifies relation extractor
7. uses relation extractor
8. parent relation extractor
9. follows relation extractor
10. pattern relation extractor
11. calls relation extractor
12. next relation extractor
13. with relation extractor
14. projector

Due to space constraints, an enumeration, rather than a class diagram is used for this description.

First note that the PKB Extractor Facade has access to each PKB table and AST (points 1-5). Also, the logic for the extraction of each relation clause is encapsulated in a relation extractor class (points 6-13). This is to prevent bloating of the actual PKB Extractor Facade class. Finally, the class also consists of a projector object (point 14) to aid with the projection of results from select clauses in PQL queries.

The following subsection describes the algorithm implemented in one instance of the relation extractor.

3.6.2 Interaction between QPS and PKB

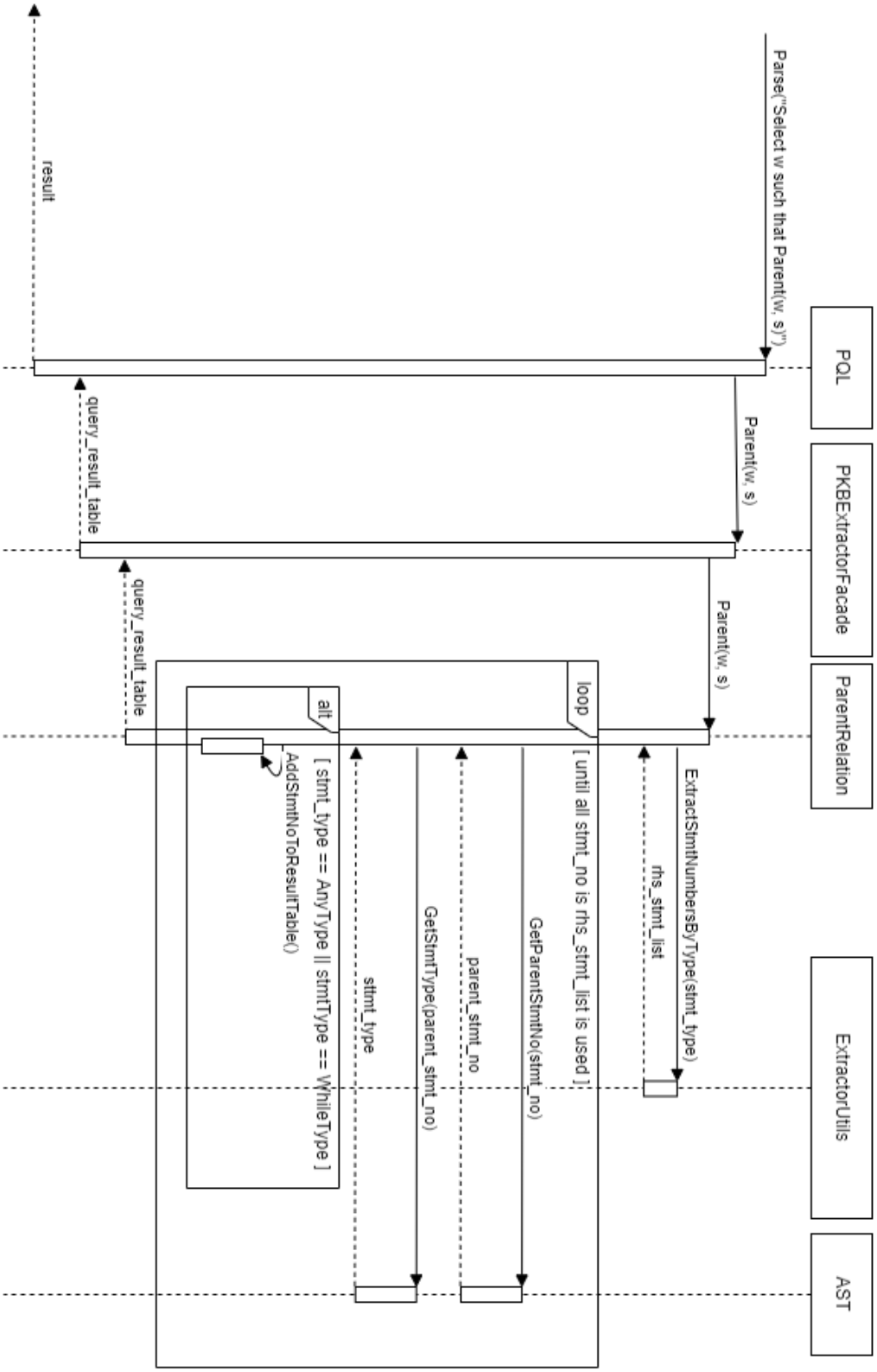

Figure 3.13: QPS and PKB Interaction Sequence Diagram

The relation that is extracted in this example is: “any while statements that is a parent of any statements”. The process starts at the QPS. The QPS parses a query and calls the PKB Extractor Facade to query for the result. Similar to the interaction between SP and PKB, communication from the QPS to the PKB is always done through this extractor facade class.

The PKB Extractor will first call the GetStmtNumbersByType method to extract all the statements of type Any. It will then loop through all of these statements and check if each parent statement matches the while statement type.

The list of APIs that are used in this extraction is displayed in the following subsection.

3.6.3 Key APIs: QPS and PKB Interaction

This section details the key APIs brought up in section 3.4.3.

PKB Extractor Facade API

QUERY_RESULT_TABLE Parent(QUERY_REF lhs_stmt_ref, QUERY_REF rhs_stmt_ref)
Returns a query result table with the computed parent information with the given lhs_stmt_ref and rhs_stmt_ref.

ExtractorUtils API

LIST_OF_STMT_NO GetStmtNumbersByType(STMT_TYPE type)
Gets all the statement numbers of the given statement type.

AST API

STMT_NO GetParentStmtNo(STMT_NO stmt_no)
Returns the parent statement number of the given stmt_no.

STMT_TYPE GetStmtType(STMT_NO stmt_no)
Returns the statement type of the given stmt_no.

3.7 PKB Extraction Logic

This section elaborates on the extraction logic of the various PQL relations. Pseudocode and the time complexity for one API per relation type is shown. It is also assumed that all arguments passed are valid, and the validation checks within each API are omitted for simplicity.

The synonym declarations shown here are assumed throughout the subsections:

- assign a
- if ifs
- procedure p
- variable v
- while w

3.7.1 Uses and Modifies Extraction

Uses and Modifies relationship extraction are similar and only differ by the API call to retrieve the used or modified relation. Hence, only Uses relations are shown here.

PQL relation UsesS: Uses(5, “v”)

Line	Code	PKB Component Referenced
1	initialize set use_v = {}	
2	use_v = variables used in statement 5	AST
3	if “v” in use_v, return true. Else return false.	

Time Complexity: O(1)

PQL relation UsesP: Uses(“p”, “v”)

Line	Code	PKB Component Referenced
1	initialize set use_v = {}	
2	use_v = variables used in procedure “p”	Procedure Table
3	if “v” in use_v, return true. Else return false.	

Time Complexity: O(N), N: Number of variables

3.7.2 Parent Extraction

PQL relation Parent: Parent(3, 4)

Line	Code	PKB Component Referenced
1	parent_stmt_no = parent statement number of 4	AST
2	if parent_stmt_no is 3, return true. Else return false.	

Time Complexity: O(1)

PQL relation ParentT: Parent*(3, 10)

Line	Code	PKB Component Referenced
1	parent_stmt_no = parent statement number of 10	AST
2	while parent_stmt_no is not null:	
3	if parent_stmt_no is 3:	
4	return true	
5	else:	
6	parent_stmt_no = parent statement number of parent_stmt_no	AST
7	end if	
8	end while	
9	return false	

Time Complexity: O(N), N: Number of statements

3.7.3 Follows Extraction

PQL relation Follows: Follows(3, 4)

Line	Code	PKB Component Referenced
1	initialize vector s = []	
2	if statement 3 is if/while statement:	AST
3	s = child statements of statement 3	AST
4	else:	
5	p = parent procedure of statement 3	AST
6	s = child statements of p	Procedure Table
7	end if	
8	binary search for statement 3 in s	
9	if statement 4 is to the immediate right of statement 3 in s, return true. Else return false.	

Time Complexity: O(logN), N: Number of statements

PQL relation FollowsT: Follows*(3, 10)

Line	Code	PKB Component Referenced
1	initialize vector s = []	
2	if statement 3 is if/while statement:	AST
3	s = child statements of statement 3	AST
4	else:	
5	p = parent procedure of statement 3	AST
6	s = child statements of p	Procedure Table
7	end if	
8	binary search for statement 3 in s	
9	binary search for statement 10 in s	
10	if statement 10 appears after statement 3 in s, return true. Else return false.	

Time Complexity: O(logN), N: Number of statements

3.7.4 Assign Pattern Extraction

PQL pattern assign: pattern a (“x”, “y + 1”)

Line	Code	PKB Component Referenced
1	initialize vector result = []	
2	initialize vector a_list = []	
3	initialize set var_set = {}	
4	a_list = all assignment statement numbers	Statement Table
5	var_set = all variables that are modified	Variable Table
6	for a in a_list:	
7	if “x” in var_set and “y + 1” matches the ExprTree	AST

	pattern in AST, append a to result	
8	end for	
9	return result	

Time Complexity: O(M*N), M: Number of tokens in assignment statements, N: Number of statements

As mentioned in section 3.2.1, the ExprTree object contains a vector of tokens from the source program inserted in postfix order. The pattern “y + 1” is also arranged in postfix order by the QPS, which will be discussed in section 4.3.4, before being passed to the PKB for evaluation. To evaluate this pattern, the ExprTree checks if all the elements in its vector match “y + 1” in postfix order, that is “y”, “1”, “+”. To evaluate clauses such as pattern a (“x”, _”y + 1”_), the ExprTree checks if the postfix order of “y + 1” appears as a consecutive sequence within its vector.

3.7.5 If and While Pattern Extraction

PQL pattern if/while: pattern ifs (“x”, _, _) / pattern w (“x”, _)

Line	Code	PKB Component Referenced
1	initialize vector result = []	
2	initialize vector s_list = []	
3	s_list = all if or while statement numbers	Statement Table
4	for s in s_list:	
5	initialize set v_set = {}	
6	v_set = get control variables of s	AST
7	for v in v_set:	
8	if v is “x”, append v to result	Variable Table
9	end for	
10	end for	
11	return result	

Time Complexity: O(M*N), M: Number of variables, N: Number of statements

3.7.6 With Extraction

PQL with clause: with v.varName = p.procName

Line	Code	PKB Component Referenced
1	initialize vector result = []	
2	initialize vector v_list = []	
3	initialize vector p_list = []	
4	v_list = all variable names	Variable Table
5	p_list = all procedure names	Procedure Table
6	for v in v_list:	
7	for p in p_list:	
8	v_name = variable name for v	Variable Table
9	p_name = procedure name for p	Procedure Table
10	if v_name is p_name, insert v in to result	
11	end for	
12	end for	
13	return result	

Time Complexity: O(M*N), M: Number of procedures, N: Number of variables

3.7.7 Calls Extraction

PQL relation Calls: Calls(“procedureAlpha”, “procedureBeta”)

Line	Code	PKB Component Referenced
1	initialize set p_set = {}	
2	p_set = procedures that “procedureAlpha” calls	Procedure Table
3	if “procedureBeta” is in p_set, return true. Else return false.	

Time Complexity: O(1)

PQL relation CallsT: Calls*(“procedureAlpha”, “procedureGamma”)

Line	Code	PKB Component Referenced
1	initialize set p_set = {}	
2	p_set = procedures that “procedureAlpha” calls and transitively calls	Procedure Table
3	if “procedureGamma” is in p_set, return true. Else return false.	

Time Complexity: O(1)

3.7.8 Next Extraction

PQL relation Next: Next(2, 3)

Line	Code	PKB Component Referenced
1	initialize set s_set = {}	
2	Get the CFG corresponding to the procedure that statement 2 resides in	Procedure Table, AST
3	s_set = all next statements of statement 2	CFG
4	if statement 3 is in s_set, return true. Else return false.	

Time Complexity: O(1)

PQL relation NextT: Next*(2, 7)

Line	Code	PKB Component Referenced
1	initialize set s_set = {}	
2	Get the CFG corresponding to the procedure that statement 2 resides in	Procedure Table, AST
3	s_set = all next and transitive next statements of statement 2 (BFS of CFG)	CFG
4	if statement 7 is in s_set, return true. Else return false.	

Time Complexity: O(N), N: Number of statements

3.7.9 Affects Extraction

Note that the following statement numbers 1 and 5, are assumed to be assignment statements.

PQL relation Affects: Affects(1, 5)

Line	Code	PKB Component Referenced
1	v = variable modified by statement 1	AST
2	Get the CFG corresponding to the procedure that statement 1 resides in	Procedure Table, AST
3	BFS on the CFG starting from statement 1 to statement 5. Terminate the BFS when there exists a statement that modifies v.	CFG, AST
4	if BFS reaches statement 5, return true. Else, return false.	

Time Complexity: O(N), N: Number of statements

PQL relation AffectsT: Affects*(1, 5)

Line	Code	PKB Component Referenced
1	initialize queue q = {}	
2	push statement 1 into q	
3	while q not empty:	
4	s = pop element from q	
5	initialize set s_set = {}	
6	v = variable modified by s	AST
7	Get the CFG corresponding to the procedure that s resides in	Procedure Table, AST
8	BFS on the CFG starting from s. Insert any assignment statements that use v into s_set.	CFG, AST
9	if statement 5 in s_set, return true	

10	push elements in s_set to q	
11	end while	
12	return false	

Time Complexity: $O(N^2)$, N: Number of statements

4 Query Processing Subsystem (QPS)

4.1 Overview

The QPS handles all incoming PQL queries from end-users. As such, it will be responsible for tokenizing, validating, and evaluating queries before projecting the relevant results to the autotester.

To better handle the series of design issues related to the evaluation of queries, the query processor subsystem has been decomposed into 4 sub-components:

1. Query Tokenizer
2. Query PreProcessor (parsing)
3. Query Optimizer (optimization/arrangement of Query Nodes)
4. Query Evaluator (evaluation and projection)

The following architecture diagram showcases the relationships between the above sub-components:

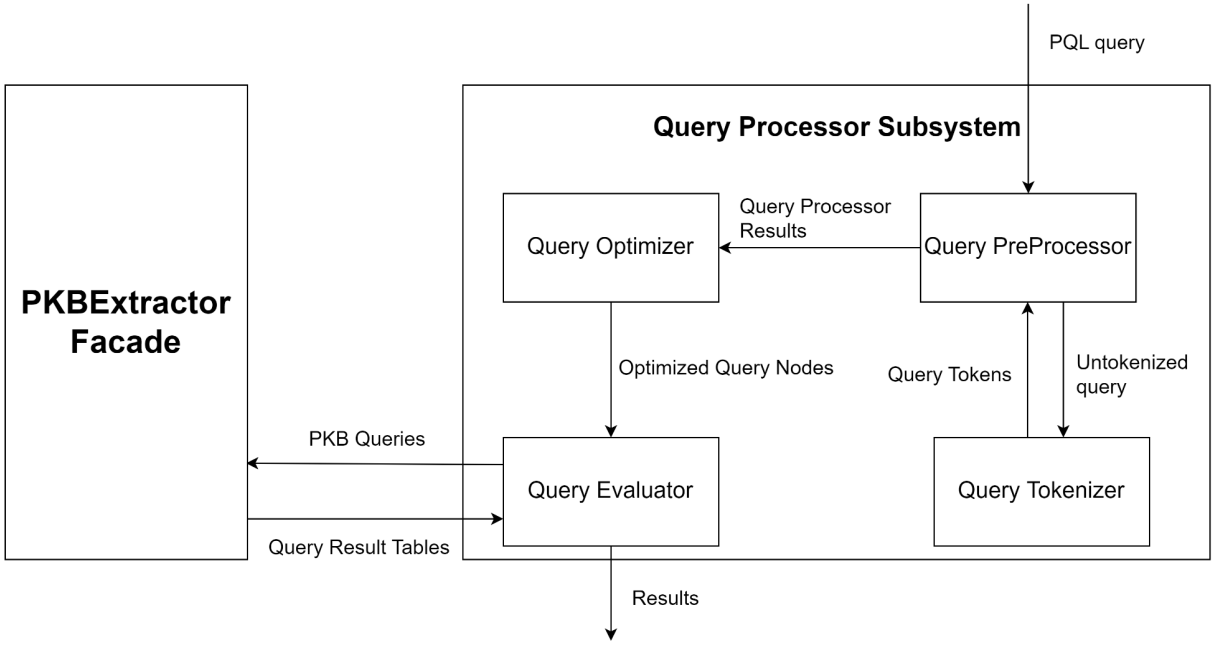

Figure 4.1: QPS Architecture Diagram

The decision for the decomposition shown in Figure 4.1 can be justified by the following 3 points:

Cohesion	Separating the functionalities of tokenizing, parsing, optimizing and evaluation would result in highly focused and strongly related code for each sub-component as they work on the same issue. This improves cohesion and thereby improves the understandability, maintainability, and reusability of the

	components created.
Coupling	In line with the above justification, the improvement in cohesion reduces the impact of coupling as similar functionalities are grouped together. Hence, the degree of dependence between independent components will be reduced. This in turn improves the ease of maintenance, integration, and testing.
Separation of Concerns	By modularizing query processor components, there will be a reduction in functional overlaps and ripple effects when changes are made to different parts of the system.

4.2 Query Nodes

Before elaborating on each QPS subcomponent, a brief overview of Query Nodes will be provided in this section. Query Nodes are implemented to serve as an abstraction for PQL relation clauses.

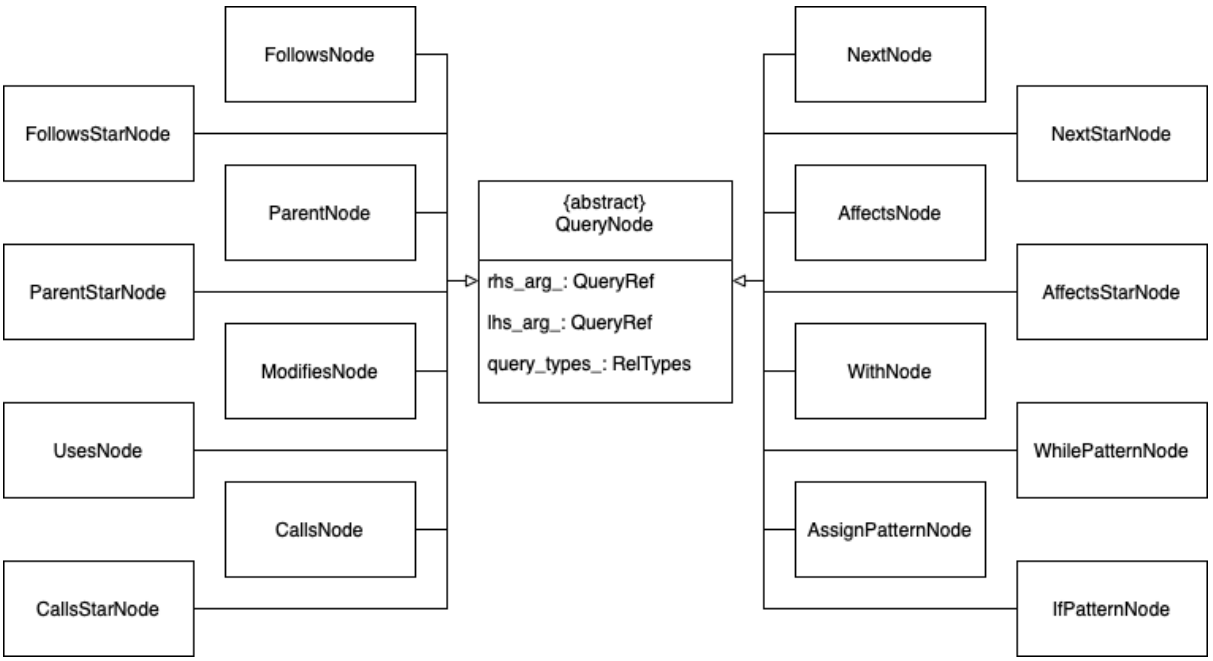

Figure 4.2: Query Node Class Diagram

Query Nodes store information for all clauses in PQL queries. Each clause type has its own concrete Query Node implementation, which is inherited from the abstract Query Node class. Left and right parameters (lhs_arg_ and rhs_arg_) are stored as Query Ref objects in the Query Node, which denote the statement or entity references found in relation clauses. For example, in Follows(s3, 4), both s3 and statement number 4 are stored as Query Ref

objects. The relationship type is also stored in the Query Node (query_type_) class as an enum. This is to facilitate the sorting of Query Nodes in the Query Optimizer, which is explained in section 4.4.

4.3 Query PreProcessor and Query Tokenizer

4.3.1 Overview

In this section, the design for the Query PreProcessor and Query Tokenizer sub-components will be discussed in tandem.

The class diagram below will be referred to in the following sections and gives an overview of the relationship between the PreProcessor and Tokenizer subcomponents.

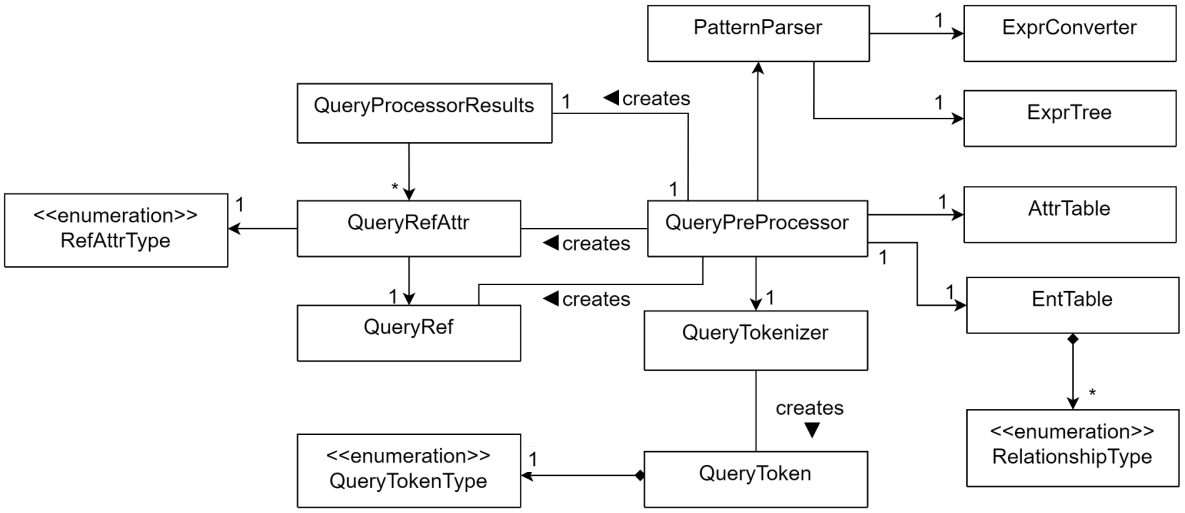

Figure 4.3: Query PreProcessor Class Diagram

As detailed in Figure 4.3, every Query PreProcessor object will be associated with only 1 EntTable and Query Tokenizer reference, both of which are critical for the parsing process. The Query Tokenizer will also be responsible for creating Query Token objects encapsulating the type (defined by the enum class Query TokenType) and the value of tokens generated from the query input. The EntTable will be responsible for storing relationship to invalid argument type mappings (eg. entRef/stmtRef). Finally, the Query PreProcessor will parse the Query Tokens obtained into Query Ref or Query Ref Attribute (for select and with clauses) objects which will subsequently be stored in a Query ProcessorResults object.

4.3.2 Usage Scenario

The following is an example usage scenario of how a query is tokenized and validated by the Query PreProcessor:

1. A Query PreProcessor object will always be initialized together with a Query Tokenizer and EntTable object. The EntTable object will be populated with the relevant relationship-argument type mappings during initialization.

2. The QueryPreProcessor::Parse() method is then called when there is an incoming query.

3. Thereafter, the QueryTokenizer::GetNextToken() method will be called to fetch the Query Tokens.

4. The Query Tokenizer object is responsible for tokenizing and encapsulating words in the provided query into Query Token objects which would, in turn, be identified by a type as defined by the Query Token Type enum.

5. The Query PreProcessor will then call the relevant Parse methods to validate the syntax of the Query Tokens against the grammar rules provided in the specifications.

6. With synonym attribute types introduced in the latest iteration, Select clause synonyms that are declared without an explicit type will invoke the AttrTable::GetDefaultAttrType() method to aid in the creation of a Query Ref Attr object.

7. In the validation process, the EntTable::ValidateArgTypes() or EntTable::ValidatePatternTypes() method may be called to check if the arguments provided in the relationship or pattern clauses are of valid types.

8. Next, the PreProcessor creates the relevant Query Ref object.

9. These Query Ref objects are then used to create Query Node objects to encapsulate the relevant information for relationship and pattern clauses.

10. These Query Node objects are then added to the nodes vector in the Query ProcessorResults object

11. After the Query PreProcessor completes parsing all clauses, it returns the Query ProcessorResults object.

This process is shown in the following sequence diagram Figure 4.4:

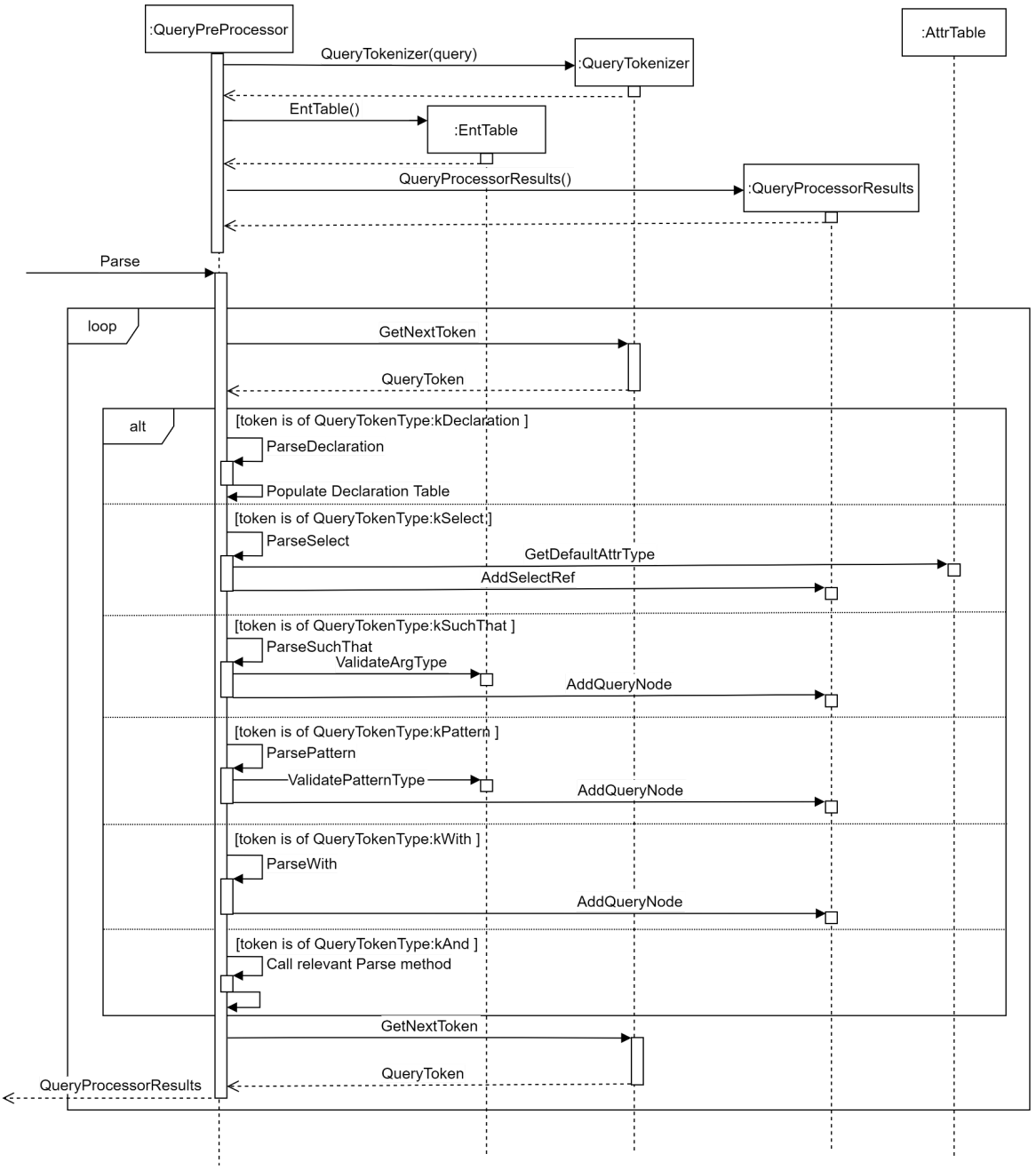

Figure 4.4: PQL Query Parsing Sequence Diagram

4.3.3 Key APIs: QPS Parsing

The following table details critical APIs encountered in the above sequence diagram:

Query PreProcessor APIs

QUERY_PROCESSOR_RESULTS Parse()
Returns a Query PreProcessor Results object populated with parsed query information to facilitate query optimization and evaluation.

VOID Init()
Initializes the entity table with records of all relationship and pattern types with their respective accepted argument types in a key-value pair mapping.

VOID ParseDeclaration()
Validates and parses the detected query declaration according to grammar rules. If the detected syntax is valid, the design-entity and synonym key-value pair will be accepted. Else, an exception will be thrown.

VOID ParseSelect()
Validates and parses the detected Select clause and its corresponding declaration* according to the grammar rule. If the detected syntax is valid, the Query Processor Results object will be populated with the declaration parsed. Else, an exception will be thrown.

VOID ParseSuchThat()
Validates and parses the detected relationship clause and its arguments according to the grammar rule. If the detected syntax is valid, the Query Processor Results object will be populated with the relationship details. Else, an exception will be thrown.

VOID ParsePattern()
Validates and parses the detected pattern clause and its arguments according to grammar rules. If the detected syntax is valid, the Query Processor Results object will be populated with the pattern details. Else, an exception will be thrown.

VOID ParseWith()
Validates and parses the detected With clause and its arguments according to the grammar rule. If the detected syntax is valid (the corresponding attribute matches the synonym type), a Query Node will be created with the With clause details and added to the Query Processor Result object. Else, an exception will be thrown.

VOID ParseAnd()
Validates and parses the detected And clause according to grammar rule. If the detected

syntax is valid, the relevant relationship or pattern parsing method will be called.

EntTable API

VOID AddEntry(RELATIONSHIP_TYPE type, ENTITY_TABLE_ENTRY entry)
Adds a relationship enum type and its corresponding argument types stored in an ENTITY_TABLE_ENTRY as a key-value pair to the Entity Table.

VOID ValidateArgTypes(STRING rs_type, QUERY_REF_TYPE arg1, QUERY_REF_TYPE arg2)
Checks if both arguments are allowed for the relationship defined in the string provided. Else, an exception will be thrown.

VOID ValidatePatternTypes(STRING pattern_name, QUERY_REF_TYPE arg1)
Checks if the argument provided for the pattern's first argument is allowed. Else, an exception will be thrown.

AttrTable API

ATTR_TYPE GetDefaultAttrType(QUERY_REF_TYPE arg1)
Returns the default attribute type associated with the Query Ref Type supplied. For instance, if a "constant" Query Ref Type is supplied, the "value" Attr Type will be returned.

Query Tokenizer API

QUERY_TOKEN GetNextToken()
Tokenizes the next token encountered in the query provided during initialization. Returns a Query Token object that encapsulates the type and value of the token. This method will also ignore all preceding whitespace before the token is detected.

4.3.4 Pattern Parsing

As assign pattern clauses accept both raw and partial expressions as arguments, the QueryPreProcessor::ParsePattern method, when invoked, will call upon the PatternParser::ParseRawExpr method to parse expressions in the same logical flow as outlined in section 2.3.5.

The parsed expression, now in the form of in-fix ordered Query Tokens, will have to be converted to a post-fix order to facilitate subsequent queries to the PKB during the

evaluation stage in the Query Evaluator. To facilitate this conversion, the Expression Converter, as detailed in section 2.3.6, will be used.

However, as the API of the Expression Converter only accepts Token objects, the parsed Query Tokens will have to be converted to a suitable Token representation via the TokenAdapter::ToToken method.

4.3.5 Design Decision: Relationship Clause Argument Validation

Approach 1: Table-Driven Design using EntTable

The table-driven design relies on the creation of key-value pairs to validate relationship and pattern arguments. Relation types are stored as keys while invalid argument types are stored as values.

The Entity Table (EntTable) can be looked up to check if the provided argument type is valid. The design of the EntTable has also been augmented to account only for syntactic mappings. Hence, a mapping of invalid argument types based on grammar rules is adopted without accounting for semantic validity. Refer to Figure 4.4 below for an example instance of the Entity Table.

Advantages:

- Improved flexibility to design model changes. Since each relationship and its invalid argument types are stored as key-value pairs, more relationships or types can be added easily. This improves the ease of further extension.
- Accessing the list of argument types for each relationship/pattern clause can be done in O(1) time.

Disadvantages:

- Increased space complexity. Since a table has to be initialized to store the relevant argument type mappings for each relationship and pattern clause, there would be an increase in memory consumption.
- However, as the number of relationship and argument types are limited, the memory overhead incurred is minimal.

Approach 2: Use If-Else conditional logic for validating each relationship type

Advantages:

- There is no additional memory used.

Disadvantages:

- Hardcoding will be prevalent as this implementation requires the implementation of available relationship types and their accompanying argument requirements.

This increases the complexity of making an extension to the program. Furthermore, the overall complexity of the PreProcessor will also increase if the number of combinations of types increases.
Justification on Final Choice (Approach 1): As approach 1 supports better extensibility for future iterations with a minimal trade-off in memory complexity, it was chosen as the method to validate queries.

Relation	**Argument 1**	**Argument 2**
Parent	kRawVar | kRawProc **(these are invalid stmtRef types)**	kRawVar | kRawProc
Parent*	kRawVar | kRawProc	kRawVar | kRawProc
Follows	kRawVar | kRawProc	kRawVar | kRawProc
Follows*	kRawVar | kRawProc	kRawVar | kRawProc
Uses	Empty list **(this implies that both stmtRef and entRef types are accepted)**	kStmtNumber **(this is an invalid entRef type)**
Modifies	Empty list	kStmtNumber
Next	kRawVar | kRawProc	kRawVar | kRawProc
Next*	kRawVar | kRawProc	kRawVar | kRawProc
Affects	kRawVar | kRawProc	kRawVar | kRawProc
Affects*	kRawVar | kRawProc	kRawVar | kRawProc
Calls	kStmtNumber	kStmtNumber
Calls*	kStmtNumber	kStmtNumber
Pattern	kStmtNumber	Empty list

Figure 4.5: EntTable Example Instance

As observed in Figure 4.5, invalid instead of valid types are stored in the EntTable. Hence, the validation methods check if the provided arguments match any of the entries found in the EntTable for the respective relation required. If found, an exception will be thrown.

4.3.6 Design Decision: Tokenizing and Parsing Incoming Queries

Approach 1: Separating tokenizing and parsing responsibilities into different components **Advantages:** ● Better cohesion within each component and coupling is reduced as well ● Adherence to the Single Responsibility Principle ● Open to extension **Disadvantages:** ● May introduce bottlenecks as the development of the PreProcessor can only start after the development of the tokenizer has been completed.
Approach 2: Combining both tokenizing and parsing into 1 processor component **Advantages:** ● Ease of implementation as functionalities can be written together in tandem under the same class **Disadvantages:** ● Increased coupling, which possibly creates a God class
Justification on Final Choice (Approach 1): Approach 1 is ultimately chosen due to its better adherence to software engineering principles. Moreover, as both sub-components can be developed by the same person, the issue of the bottleneck would be resolved.

4.4 Query Optimizer

4.4.1 Overview

The Query Optimizer is responsible for optimizing the processing of queries that are parsed by the Query PreProcessor. This is useful for multi-clause queries, which would each need to be evaluated in order to arrive at the final output. The order of evaluation is a key factor in determining the time needed to process queries, and the Query Optimizer is responsible for sorting the clauses into the most efficient order.

To handle optimization, the Query Optimizer needs to be able to retrieve parsed query abstractions from the Query PreProcessor, and proceed to generate output that can be processed by the Query Evaluator. Two structs are used for this process: QueryProcessorResults and QueryOptimizerResults. As mentioned in section 4.3, the QueryProcessorResults struct is generated by the Query PreProcessor component, which is then passed to the Query Optimizer in the initialization stage. The Query Optimizer sorts the

nodes and creates the QueryOptimizerResults struct, which is then passed to the Query Evaluator component for evaluation.

The following figure outlines key attributes of the Query Optimizer.

Attributes	Purpose
clause_no	Gives an index numbering to the different query clauses
group_no	Gives a group numbering to the different groups created
processed_query_nodes	Stores Query Nodes that contain synonyms found in the select_ref or are connected to it.
select_string_set	Stores all select synonyms in string form in a set for easier comparison subsequently
raw_nodes	Stores nodes that only contain a raw integer or raw variable or procedure name. Respectively, these refer to PQL Integers and PQL Identifiers (wrapped in double quotation marks)
node_index	Maps the index number of clauses to the actual Query Node itself
adj_list	Adjacency list to group Query Nodes
group_synonyms	Maps clause groups to their respective synonym lists. This helps to sort groups based on whether their synonyms are present in the select clause.
res	Stores the QueryOptimizerResults struct to be output.

Figure 4.6: Key Components of Query Optimizer

The following subsections describe the process in which the Query Nodes are sorted and grouped. The following PQL query will be considered: stmt s1, s2, s3, s4, s5, s6; Select <s1, s5> such that Parent(s1, s2) and Follows(s3, s4) and Modifies(s5, “v”) and Uses(s6, “v”) and Next(s5, 10) and Affects(s2, s3) and Follows(11, 12).

4.4.2 Grouping of Query Nodes

The relations are first separated into groups with connected synonyms and raw nodes (relations that either yield true or false results):

- **Raw nodes:** Follows(11, 12)
- **Group 1:** Parent(s1, s2), Affects(s2, s3), Follows(s3, s4)
- **Group 2:** Modifies(s5, “v”), Next(s5, 10)

- **Group 3:** Uses(s6, “v”)

This grouping is done using the adjacency list (adj_list) attribute in the Query Optimizer. Edges between nodes are formed when both share at least one synonym. These are stored in the adjacency list. Depth first search is then performed over the adjacency list to separate these nodes into connected groups.

Thereafter, the groups are then iterated over. Groups that contain synonyms in the select synonyms are differentiated from groups that do not:

- **Raw nodes:** Follows(11, 12)
- **Non-select nodes group 1:** Uses(s6, “v”)
- **Select nodes group 1:** Parent(s1, s2), Affects(s2, s3), Follows(s3, s4)
- **Select nodes group 2:** Modifies(s5, “v”), Next(s5, 10)

4.4.3 Optimization Heuristics

The actual optimization of each group is described in this subsection. Each group, containing both non-select and select clauses, are optimized. As the same optimization algorithm is used on all groups, only select nodes group 1 will be explained: Parent(s1, s2), Affects(s2, s3), Follows(s3, s4).

Firstly, groups are sorted based on the restrictiveness of the clauses. The following shows the estimated restrictiveness of each clause from most restrictive to least restrictive:

- With
- Pattern (assign, if and while)
- Calls
- Calls*
- Parent
- Follows
- Modifies
- Uses
- Affects
- Next
- Follows*
- Parent*
- Affects*
- Next*

Based on this restrictiveness order, the nodes will be sorted to Parent(s1, s2), Follows(s3, s4), Affects(s2, s3). This sorting is done to ensure that the combination of results per clause is conducted over the smallest possible result set.

Next, the nodes are sorted again to ensure that the incoming node to be evaluated contains at least one synonym that appears in all the nodes that occur prior to it. The nodes will then be sorted to: Parent(s1, s2), Affects(s2, s3), Follows(s3, s4). This sorting is to ensure that the

combination of query results will be executed with an equi-join algorithm, rather than a cross product algorithm, which is slower. Equi-joins are executed when the two results to be combined share at least one common synonym. Cross products are executed when the results to be combined do not share any common synonyms. This is further elaborated in section 4.6. Furthermore, this round of sorting minimizes the number of swaps required, in order to retain the restrictiveness order as much as possible.

4.4.4 Design Decision: Optimization Heuristic

Approach 1: Pre-evaluation Optimization

Optimization is conducted prior to evaluation, as described in section 4.4.3. The restrictiveness of each clause is estimated and the Query Nodes are sorted such that equi-joins can be performed.

Advantages:

- The evaluation of each query clause incurs O(N) time, as equi-joins take as long

Disadvantages:

- The estimation of query restrictiveness may not be accurate
- Greater optimization runtime overhead as two rounds of sorting are required

Approach 2: Post-evaluation Optimization

Optimization is conducted after evaluation. The results of each Query Node are stored in memory, then sorted based on its size. The results are sorted in ascending order and then combined.

Advantages:

- Query results with large differences in size can be evaluated faster. This is because query results smaller in size can be combined first. Future combination of these results will then result in less iterations, as opposed to combining large results right at the start of the evaluation.

Disadvantages:

- Combination of query results may incur cross products, as the results are sorted based on size, not if synonyms are connected
- More memory may be used as each result has to be stored in memory and sorted, before they can be combined

Justification on Final Choice (Approach 1):

Approach 1 is chosen. It was found that the bottleneck of the overall SPA system stems from the cross product of results. Approach 1 reduces this bottleneck the most and is thus chosen as the optimization heuristic.

4.5 Query Evaluator

4.5.1 Overview

After the optimization of Query Nodes, the nodes in the QueryOptimizerResults will be extracted and stored in a newly instantiated Query Evaluator object. This Query Evaluator object then evaluates this abstract and optimized representation of the PQL query. The following is a class diagram of the Query Evaluator:

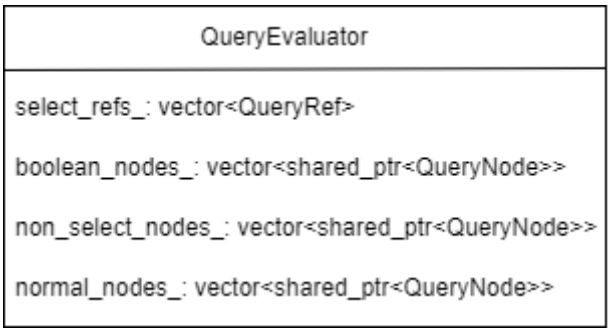

Figure 4.7: Query Evaluator Class Diagram

Briefly, the responsibilities of each attribute is highlighted as follows:

- select_refs_: stores the list of PQL synonyms to be projected on
- boolean_nodes_: stores the list of Query Nodes that contain only PQL Integers or Identifiers
- non_select_nodes_: stores the group of Query Nodes that do not contain any synonyms in select_refs_ or are not connected to any other Query Nodes that share synonyms in select_refs_
- normal_nodes_: stores the group of Query Nodes that contain synonyms in select_refs_ and its connected nodes.

To evaluate queries, the Query Evaluator traverses across all these Query Nodes. Each Query Node implements evaluate methods to retrieve information from the PKB. These information are stored as Query Result Tables, discussed further in section 4.6, and are subsequently joined and projected on to yield the desired result within the Query Evaluator.

4.5.2 Interaction with PKB

This section provides an example of how a generic Query Node retrieves data using the relevant PKB API:

1. Based on the left and right arguments, the Query Node will call a GetCombination method to receive its query combination.
2. The node will then call the PKB API with the appropriate parameter, depending on the type of query combination.
3. The data will be retrieved via the PKB API and returned to the Query Node .

To better illustrate this process, an example for FollowsNode is shown in this sequence diagram:

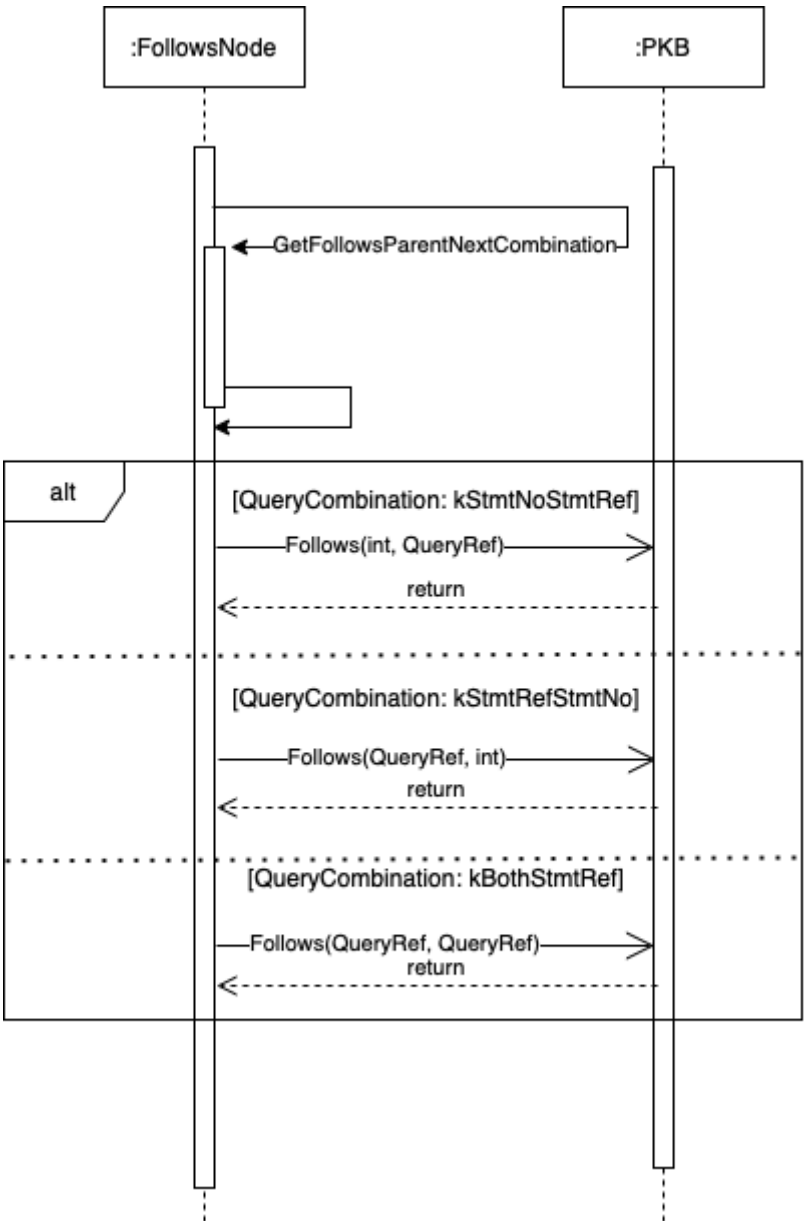

Figure 4.8: PKB Follows API Call Sequence Diagram

4.5.3 Query Evaluator Evaluation Order

This section discusses how queries are evaluated within the Query Evaluator. The following activity diagram provides an overview on how this is performed.

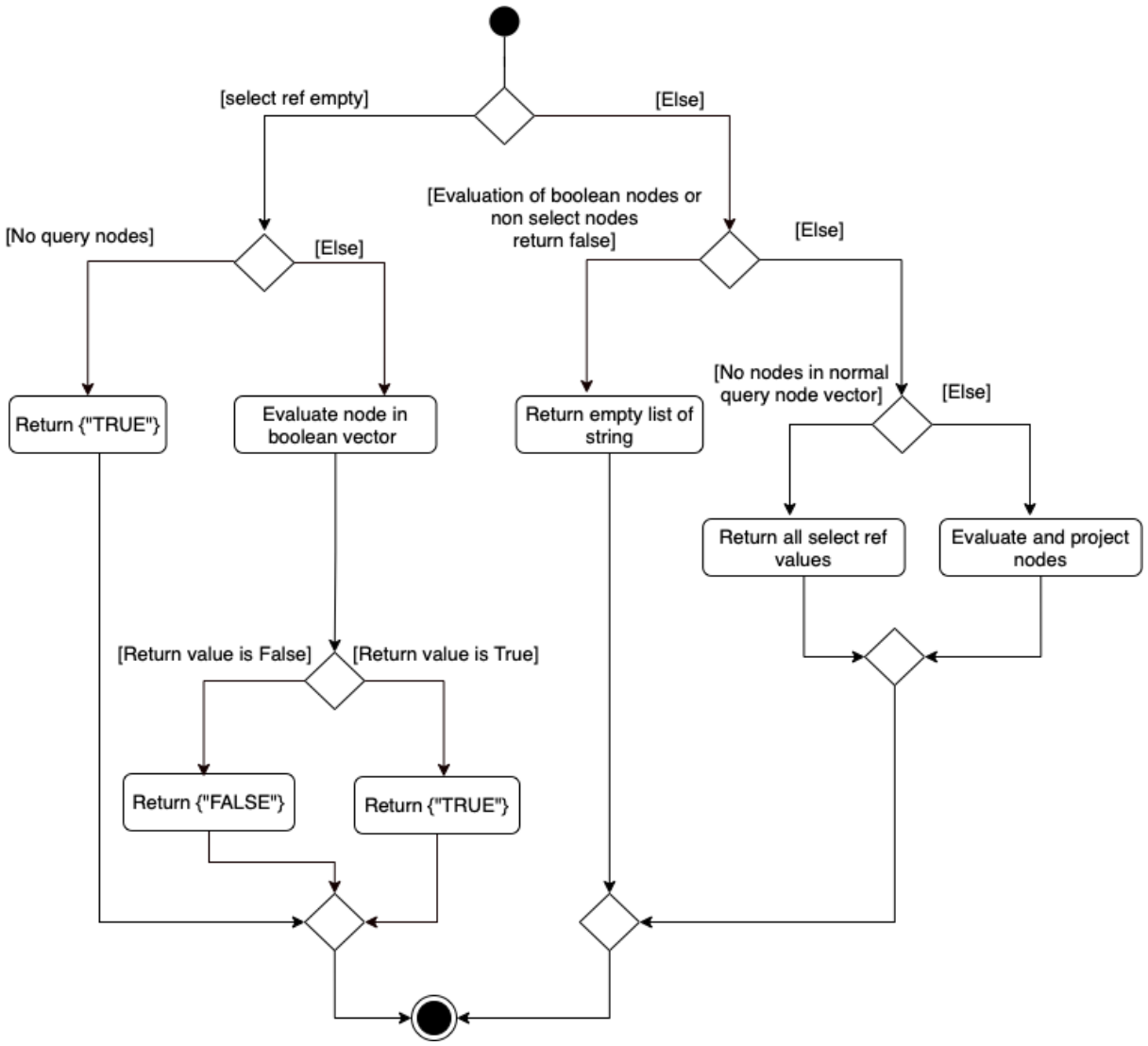

Figure 4.9: Query Evaluator Evaluation Activity Diagram

As seen in this activity diagram, the select refs attribute is first checked to determine if it is empty. If it is empty, a BOOLEAN select query is assumed, otherwise, the standard synonym selection is assumed. Next, regardless of selection paths, the boolean nodes and non select nodes are evaluated. These nodes, if evaluated to false, will return “FALSE” or an empty list, depending on the type of select query. If it evaluates to true, “TRUE” will be returned for BOOLEAN selects while the normal nodes will be evaluated next for synonym selects. After evaluation, the tables obtained from each Query Node will be combined and projected.

4.6 Query Result Table

Query Result Tables are data structures that store the results of evaluate methods from the Query Nodes. It supports the projection of selected synonyms in PQL queries. It also supports equi-joins and cross products between tables. This section discusses the design of Query Result Tables.

4.6.1 Overview

Query Result Tables are implemented to resemble csv (comma separated values) files with columns and rows. A vector of vectors is used to represent the list of rows, while a hashmap is used to map column names to the indexes within the row. For instance, consider an abstract representation of a table as shown below:

“x”	**“y”**
10	100
20	200
30	300

Figure 4.10: Query Result Table Abstract Representation

The following diagram illustrates the actual implementation representation of the Query Result Table.

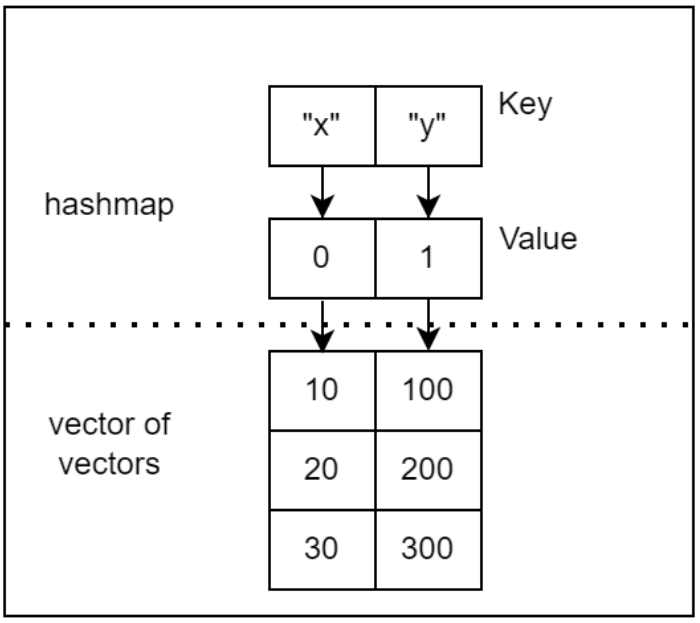

Figure 4.11: Query Result Table Actual Representation

4.6.2 Design Decision: Row or Column Major Tables

Approach 1: Row Major Tables

A row major implementation refers to storing the table as a list of rows, rather than a list of columns.

Advantages:

- Easier to implement as the size of the rows can be enforced easily. Additionally, equi-joins require the access of rows, rather than columns. This makes the

implementation of equi-joins easier as well.

Disadvantages:

- Slower projection of results with runtime O(N), as each row in the table has to be iterated over to aggregate the relevant columns.

Approach 2: Column Major Tables

A column major implementation refers to storing the table as a list of columns.

Advantages:

- Faster projection of results with runtime O(1), as the columns just need to be selected and copied from the table

Disadvantages:

- Difficult to implement because additional checks have to be implemented to ensure that the columns are of equal length
- Slightly more runtime overhead when iterating over the columns for table joins. This is because caching exploits spatial locality and adjacent columns (rather than rows) are more likely to be cached in column major tables. Since equi-joins involve the access of rows, a column major implementation may incur slightly more runtime overhead.

Justification on Final Choice (Approach 1):

Approach 1 was ultimately chosen due to its ease of implementation. Since the aggregation of Query Result Tables is critical in determining the accuracy of PQL queries, it is less risky to implement row major tables. Furthermore, projection of results from the Query Result Table is only done once per PQL query. Hence, the difference in projection runtime would be minimal as well.

4.6.3 Design Decision: Table Join Algorithms

Equi-joins and cross products between tables have to be supported for the aggregation of data in the Query Nodes. Equi-joins are used whenever tables share connected synonyms. Otherwise, cross products are used. The following compares two algorithms that performs equi-joins between two tables:

Approach 1: Hash Joins

Hash join involves the partitioning of one of the two tables into buckets via a hash function h. After partitioning, h is applied to each row of the other unpartitioned table. If the newly hashed row can be found in any partitioned bucket, this row is combined and output into the result table.

Advantages:

- Faster runtime of O(N)

Disadvantages:

- Additional space complexity of O(N) is required to store the partition
- Extension to support cross products is inefficient, as all rows on both tables have to be iterated over
- Additional runtime overhead of partitioning, especially when the number of rows to be joint is small

Approach 2: Loop joins

Loop joins require the iteration over both tables. This is to check if each row in a table is equivalent to any other rows in the other table. Only output the row if it appears in both tables.

Advantages:

- Easily extended to support cross products
- No additional space complexity needed
- No runtime overhead of partitioning

Disadvantages:

- Slower runtime of $O(N^2)$

Justification on Final Choice (Combination of Approach 1 and Approach 2):

A combination of Approach 1 and 2 is used in the final implementation. Time constraints are a priority in the SPA requirements, and the overhead of joining tables using an $O(N^2)$ algorithm is significant. Hence, hash join has to be implemented. Cross product incurs an overhead as well, and is therefore extended from the loop join algorithm rather than the hash join algorithm. Furthermore, through optimization and testing, it was discovered that performing loop joins on tables with 1 row was more efficient than hash joins. Thus, the table join method was implemented to contain a conditional statement that uses

- cross product: when tables do not share any common columns
- loop join: when either tables contain 1 row only
- hash join: for all other cases

5 Testing

In this section, the technical aspects of testing of the project will be discussed.

Four types of testing will be discussed: unit testing, integration testing, system testing, and load testing.

5.1 Unit Testing

Unit tests isolate and test individual modules independently. This reduces the complexity and time taken for debugging found since bugs can be narrowed down to modular units under test.

Furthermore, this increases the ease of integration tests as emphasis can be placed on testing API interactions between components/modules in higher-level testing as individual functions have already been tested on a modular scale with unit testing.

Two sample unit test cases for PKB and Query Processor are provided to showcase unit testing adopted in the project.

For the PKB, the validation of successful insertion and extraction of data from the PKB tables will be demonstrated.

For the Query Processor, unit tests for the decomposed subcomponents, namely the Query Tokenizer and Query PreProcessor, will be demonstrated. For the former, successful tokenization of the provided query input will be demonstrated while for the latter, successful consumption of the query input will be demonstrated.

5.1.1 PKB Variable Table Insertion Test

```
32  TEST_CASE("Test variable statement insertion for uses relationship",
33            "[pkb-variable-table]") {
34    VariableTable vt{};
35    PopulateVariableTable(vt);
36    SECTION("success") {
37      REQUIRE_NOTHROW(vt.InsertStmtUsingVariable(0, 1));
38      REQUIRE_NOTHROW(vt.InsertStmtUsingVariable(0, 1));
39      REQUIRE_NOTHROW(vt.InsertStmtUsingVariable(1, 2));
40      REQUIRE_NOTHROW(vt.InsertStmtUsingVariable(2, 3));
41    }
42    SECTION("invalid statement number failure") {
43      REQUIRE_THROWS_WITH(vt.InsertStmtUsingVariable(0, 0),
44                          Contains("Statement invalid: '0'"));
45      REQUIRE_THROWS_WITH(vt.InsertStmtUsingVariable(0, -1),
46                          Contains("Statement invalid: '-1'"));
47    }
48    SECTION("index out-of-bounds failure") {
49      REQUIRE_THROWS(vt.InsertStmtUsingVariable(-1, 1));
50      REQUIRE_THROWS(vt.InsertStmtUsingVariable(3, 1));
51    }
52  }
```

Figure 5.1: Variable Table Insertion Unit Test

Link to test code.

The unit test above checks if the **InsertStmtUsingVariable(int stmt_number)** method can correctly insert statement numbers that use a variable, into the corresponding variable record entry. It tests the following parameters:

- Valid statement numbers and variable indexes
- Invalid statement numbers, such as integers less than or equal to 0
- Invalid variable indexes, such as indexes that exceed the size of the table, or are negative

Note that the retrieval of variable information is tested in another function.

5.1.2 PKB Extractor Facade Modifies Test

```
302     SECTION("Test IsModifies(stmt_no, var_name)") {
303       REQUIRE(modifies.IsModifies(1, "x") == false);
304       REQUIRE(modifies.IsModifies(1, "y") == false);
305       REQUIRE(modifies.IsModifies(1, "z") == true);
306
307       REQUIRE(modifies.IsModifies(2, "x") == true);
308       REQUIRE(modifies.IsModifies(2, "y") == false);
309       REQUIRE(modifies.IsModifies(2, "z") == false);
310
311       REQUIRE(modifies.IsModifies(3, "x") == true);
312       REQUIRE(modifies.IsModifies(3, "y") == false);
313       REQUIRE(modifies.IsModifies(3, "z") == false);
314
315       REQUIRE(modifies.IsModifies(4, "x") == true);
316       REQUIRE(modifies.IsModifies(4, "y") == false);
317       REQUIRE(modifies.IsModifies(4, "z") == false);
318
319       REQUIRE(modifies.IsModifies(5, "x") == false);
320       REQUIRE(modifies.IsModifies(5, "y") == false);
321       REQUIRE(modifies.IsModifies(5, "z") == false);
```

Figure 5.2: PKB Extractor Modifies Relation Unit Test

Link to test code.

This unit test checks if the **Modifies(int stmt_no, string var_name)** method in the PKB can extract 'Modifies' information accurately. It tests the following parameters:

- Valid statement numbers that modify existing variables
- Valid statement numbers that do not modify existing variables
- Negative statement numbers, that are not accepted
- Variables that do not exist in the (line 348)

As many 'SECTIONS' were used, the remaining portion of the unit test is not included in this example.

5.1.3 QPS Query Tokenizer Test

```
7   bool IsTokenEq(QueryTokenizer& t, QueryTokenType type, std::string val) {
8     QueryToken tok{t.GetNextToken()};
9     QueryToken expected{QueryToken{type, std::move(val)}};
10    return tok == expected;
11  }
```

Figure 5.3: Query Tokenizer Token Equality Method

```
153 TEST_CASE("Test query token equality 6", "[query-preprocessor-tokenizer]") {
154   QueryTokenizer qt{
155       QueryTokenizer("procedure p1,p2,p; read r; assign a; call c;Select a "
156                      "such that Parent(12,a)")};
157   REQUIRE(IsTokenEq(qt, QueryTokenType::kDeclaration, "procedure"));
158   REQUIRE(IsTokenEq(qt, QueryTokenType::kAttribute, "p1"));
159   REQUIRE(IsTokenEq(qt, QueryTokenType::kComma, ","));
160   REQUIRE(IsTokenEq(qt, QueryTokenType::kAttribute, "p2"));
161   REQUIRE(IsTokenEq(qt, QueryTokenType::kComma, ","));
162   REQUIRE(IsTokenEq(qt, QueryTokenType::kAttribute, "p"));
163   REQUIRE(IsTokenEq(qt, QueryTokenType::kSemicolon, ";"));
164   REQUIRE(IsTokenEq(qt, QueryTokenType::kDeclaration, "read"));
165   REQUIRE(IsTokenEq(qt, QueryTokenType::kAttribute, "r"));
166   REQUIRE(IsTokenEq(qt, QueryTokenType::kSemicolon, ";"));
167   REQUIRE(IsTokenEq(qt, QueryTokenType::kDeclaration, "assign"));
168   REQUIRE(IsTokenEq(qt, QueryTokenType::kAttribute, "a"));
169   REQUIRE(IsTokenEq(qt, QueryTokenType::kSemicolon, ";"));
170   REQUIRE(IsTokenEq(qt, QueryTokenType::kDeclaration, "call"));
171   REQUIRE(IsTokenEq(qt, QueryTokenType::kAttribute, "c"));
172   REQUIRE(IsTokenEq(qt, QueryTokenType::kSemicolon, ";"));
173   REQUIRE(IsTokenEq(qt, QueryTokenType::kSelect, "Select"));
174   REQUIRE(IsTokenEq(qt, QueryTokenType::kAttribute, "a"));
175   REQUIRE(IsTokenEq(qt, QueryTokenType::kSuchthat, "such that"));
176   REQUIRE(IsTokenEq(qt, QueryTokenType::kRelationship, "Parent"));
177   REQUIRE(IsTokenEq(qt, QueryTokenType::kBracketL, "("));
178   REQUIRE(IsTokenEq(qt, QueryTokenType::kInteger, "12"));
179   REQUIRE(IsTokenEq(qt, QueryTokenType::kComma, ","));
180   REQUIRE(IsTokenEq(qt, QueryTokenType::kAttribute, "a"));
181   REQUIRE(IsTokenEq(qt, QueryTokenType::kBracketR, ")"));
182 }
```

Figure 5.4: Query Tokenizer GetNextToken Unit Test

Link to test code.

The unit test in Figure 5.4 assesses and validates if the GetNextToken() method can successfully tokenize the query input into an appropriate Query Token object with a valid type (enum) and value (raw string representation). Additionally, it can also be observed in Figure 5.3 that the token equality comparison has been abstracted to a test method, IsTokenEq() for code reusability.

The following parameters are also assessed in this test:

- Valid identification of declaration tokens (which uses keywords like procedure)
- Valid identification of symbol tokens such as brackets and semicolons
- Valid identification of select and relationship tokens such as Parent/Follows
- Valid categorization of undefined token values as a possible synonym name tagged by the kAttribute enum

Invalid inputs are tested separately in a negative test case.

5.1.4 QPS Query PreProcessor Test

```
59  TEST_CASE("Test consume synonym method 1", "[query-preprocessor-parser]") {
60    QueryPreProcessor parser{CreateQueryPreProcessor(
61        "assign stmt read print call while if variable procedure constant")};
62    REQUIRE(parser.ConsumeSynonym() == "assign");
63    REQUIRE(parser.ConsumeSynonym() == "stmt");
64    REQUIRE(parser.ConsumeSynonym() == "read");
65    REQUIRE(parser.ConsumeSynonym() == "print");
66    REQUIRE(parser.ConsumeSynonym() == "call");
67    REQUIRE(parser.ConsumeSynonym() == "while");
68    REQUIRE(parser.ConsumeSynonym() == "if");
69    REQUIRE(parser.ConsumeSynonym() == "variable");
70    REQUIRE(parser.ConsumeSynonym() == "procedure");
71    REQUIRE(parser.ConsumeSynonym() == "constant");
72  }
73
74  TEST_CASE("Test consume synonym method 2", "[query-preprocessor-parser]") {
75    QueryPreProcessor parser{
76        CreateQueryPreProcessor("Follows Parent Modifies Uses")};
77    REQUIRE(parser.ConsumeSynonym() == "Follows");
78    REQUIRE(parser.ConsumeSynonym() == "Parent");
79    REQUIRE(parser.ConsumeSynonym() == "Modifies");
80    REQUIRE(parser.ConsumeSynonym() == "Uses");
81  }
82
83  TEST_CASE("Test consume synonym method 3", "[query-preprocessor-parser]") {
84    QueryPreProcessor parser{CreateQueryPreProcessor("Select such that Pattern")};
85    REQUIRE(parser.ConsumeSynonym() == "Select");
86    REQUIRE(parser.ConsumeSynonym() == "such that");
87    REQUIRE(parser.ConsumeSynonym() == "Pattern");
88  }
```

Figure 5.5: Query PreProcessor Consume Synonym Unit Test

Link to test code.

The unit test in Figure 5.5 assesses and validates if the **ConsumeSynonym()** method can successfully consume a valid **Query Token** object suitable for a synonym name despite bearing different token types as seen in the previous sample. This allows the handling of situations where keywords are used as synonym names. For instance, assign Select;

The following parameters are assessed in this test:

- Valid consumption of declaration type tokens for synonym names
- Valid consumption of relationship type tokens for synonym names

- Valid consumption of other keyword type tokens (ie Select, pattern) for synonym names

5.1.5 Unit Test Statistics

The figure below shows the line coverage (%) of unit tests across code folders.

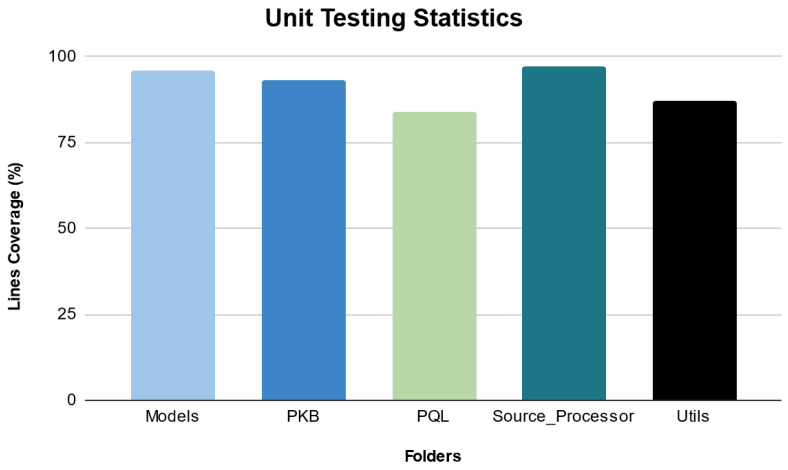

Figure 5.6: Unit Tests Line Coverage

Unit testing efforts were consistent across the Models, PKB, PQL, and Source_Processor folders. Each of these folders had at least 75% of unit test line coverage. Models, PKB and Source_Processor folders were rigorously tested. Each of these folders had at least 90% of line coverage.

5.2 Integration Testing

In this section, integration testing for each of the major components in SPA will be discussed. The integration tests aim to discover bugs as modular components interact with one another.

5.2.1 SP to PKB Integration Testing

The SP is responsible for calling respective APIs in the PKB Inserter Facade in order to extract key design information into the PKB. In order to test the integration between the SP and the PKB Inserter Facade, a sample SIMPLE program is implemented:

```
 procedure Example {
1  x = 2;
2  z = 3;
3  i = 5;
4  while (i!=0) {
5    x = x - 1;
6    if (x==1) then {
7      z = x + 1; }
    else {
8      y = z + x; }
9     z = z + x + i;
10    call q;
11    i = i - 1; }
12  call p; }

procedure p {
13  if (x<0) then {
14    while (i>0) {
15      x = z * (3 + 2) + y;
16     call q;
17      i = i - 1; }
18   x = x + 1;
19    z = x + z; }
  else {
20    z = 1; }
21  z = z + x + i; }

procedure q {
22  if (x==1) then {
23    z = x + 1; }
  else {
24    x = z + x; }
25  print y;
26  read h; }
```

Figure 5.7: Test SIMPLE Program

This SIMPLE program is parsed with the SP and inserted into the PKB. To test if the information has been inserted correctly, getters are called on the PKB tables to ensure that the required information exists and is accurate.

The following is a snippet of the SP and PKB integration test.

```
71  TEST_CASE("Test pkb insertion and processing", "[integration-sp-pkb]") {
72    std::shared_ptr<ConstantTable> c{new ConstantTable()};
73    std::shared_ptr<VariableTable> v{new VariableTable()};
74    std::shared_ptr<ProcedureTable> p{new ProcedureTable()};
75    std::shared_ptr<StatementTable> s{new StatementTable()};
76    std::shared_ptr<AST> ast{new AST()};
77    std::shared_ptr<PKBInserterFacade> pkb{
78        new PKBInserterFacade(c, v, p, s, ast)};
79    ReadFile(pkb, "integration_source_1.txt");
80
81    REQUIRE(p->GetProcedureName(0) == "Example");
82    REQUIRE(p->GetProcedureChildStmts(0) == std::vector<int>{1, 2, 3, 4, 12});
83    REQUIRE(p->GetProcedureVariablesModified(0) ==
84            std::unordered_set<int>{0, 1, 2, 3, 4});
85    REQUIRE(p->GetProcedureVariablesUsed(0) ==
86            std::unordered_set<int>{0, 1, 2, 3});
87    REQUIRE(p->GetCallProcs(0) == std::unordered_set<int>{1, 2});
88    REQUIRE(p->GetCalleeProcs(0).empty());
89    REQUIRE(p->GetCallStarProcs(0) == std::unordered_set<int>{1, 2});
90    REQUIRE(p->GetCalleeStarProcs(0).empty());
91
92    REQUIRE(p->GetProcedureName(1) == "p");
93    REQUIRE(p->GetProcedureVariablesModified(1) ==
94            std::unordered_set<int>{0, 1, 2, 4});
95    REQUIRE(p->GetProcedureVariablesUsed(1) ==
96            std::unordered_set<int>{0, 1, 2, 3});
97    REQUIRE(p->GetProcedureChildStmts(1) == std::vector<int>{13, 21});
98    REQUIRE(p->GetCallProcs(1) == std::unordered_set<int>{2});
99    REQUIRE(p->GetCallStarProcs(1) == std::unordered_set<int>{2});
100   REQUIRE(p->GetCalleeProcs(1) == std::unordered_set<int>{0});
101   REQUIRE(p->GetCalleeStarProcs(1) == std::unordered_set<int>{0});
```

Figure 5.8: SP and PKB Integration Test

Link to test code.

The methods used in this test are outlined in the following table:

Purpose	**Integration Test Method Calls**
Making sure that procedure information is successfully	**GetProcedureName**(procedure index) - Returns name of procedure at index **GetProcedureChildStmts**(procedure index) - Verifies child statements of procedure at index

inserted into the PKB	**GetProcedureVariablesModified**(procedure index) - Ensures that all modified variables are tagged under the correct procedure index **GetProcedureVariablesUsed**(procedure index) - Ensures that all used variables are tagged under the correct procedure index
The successful insertion of Variable and constant information	**GetConstants**() - Ensures that all constants are inserted **GetVariableName**(variable index) - Ensures that variable names are successfully inserted at index **GetModifyingStatemenGetReadStmt(26)ts**(variable index) **GetUsingStatements**(variable index) - Ensures that variables are modifying and using the correct indexes
The successful insertion of call, print, read, print statement information	**GetCallStatements**() **GetIfStatements**() **GetReadStatements**() **GetPrintStatements**() **GetWhileStatements**() **GetAssignStatements**() - Ensures that the statement numbers of different statement types are inserted correctly **GetAssignStmt**(statement number) **GetIfStmt**(statement number) **GetWhileStmt**(statement number) **GetPrintStmt**(statement number) **GetReadStmt**(statement number) **GetCallStmt**(statement number) - Ensures that the statement numbers of different statement types are inserted correctly

Figure 5.9: SP and PKB Integration Test Methods

5.2.2 QPS to PKB Integration Testing

```
8   TEST_CASE("Test Follows* clause", "[integration-pql-pkb]") {
9     std::shared_ptr<TestConstantTable> c{new TestConstantTable()};
10    std::shared_ptr<TestVariableTable> v{new TestVariableTable()};
11    std::shared_ptr<TestProcedureTable> p{new TestProcedureTable()};
12    std::shared_ptr<TestStatementTable> s{new TestStatementTable()};
13    std::shared_ptr<TestAST> ast{new TestAST()};
14    std::shared_ptr<PKBExtractorFacade> pkb{
15        new PKBExtractorFacade(c, v, p, s, ast)};
16    PkbPopulate::PopulateSingleProcPkb(c, v, p, s, ast);
17
18    SECTION("Test Follows* single clause - (stmt_no, stmt_no)") {
19      std::list<std::string> actual_result;
20      std::list<std::string> expected_result;
21
22      expected_result = {"1",  "10", "11", "12", "13", "14", "15", "16", "17",
23                         "18", "19", "2",  "20", "21", "22", "23", "24", "25",
24                         "26", "3",  "4",  "5",  "6",  "7",  "8",  "9"};
25      actual_result =
26          TestQPS::Evaluate("stmt s1; Select s1 such that Follows*(1, 2)", pkb);
27      expected_result.sort();
28      actual_result.sort();
29      REQUIRE(actual_result == expected_result);
30
31      expected_result = {};
32      actual_result =
33          TestQPS::Evaluate("stmt s1; Select s1 such that Follows*(s1, 1)", pkb);
34      REQUIRE(actual_result == expected_result);
35    }
```

Figure 5.10: QPS and PKB Integration Follows* Test

Link to test code.

All aspects of the QPS are tested during the integration testing from PQL to PKB. To test the system, a PQL query is first provided to the Query PreProcessor which creates a Query Optimizer. After invoking the optimizer's optimize method, it will return a QueryOptimizerResult struct containing sorted Query Nodes in their respective groups. This struct is then passed to the Query Evaluator, which will traverse the Query Nodes in the different vectors to invoke the evaluation method on individual Query Nodes. The Query Nodes will then use an API call to acquire the result from the PKB. The Query Evaluator

process evaluates all returned results before returning a final output. The final output is checked to ensure that the Query Evaluator returns a correct result.

For example, the table below illustrates the QPS to PKB integration testing test plan with different clauses and parameters.

Declaration: stmt s, s1, s2; while w; if ifs; assign a, a1,a2;

Clauses that are tested	**Parameter Permutation**
Select clause	1. Single results a. variables, print, while, if, read, assign, constants and statement numbers 2. Tuples a. <variables, if> , <stmt, read>,<if procedure> b. <while, procedure>, <while, variable> 3. Boolean a. General i. Select Boolean with no such that or pattern clause b. Select Boolean for queries with return a result c. Select Boolean for queries with does not return a result d. Queries used are mainly from Uses, Modifies, Follow and Pattern clauses

Such that **Follows/Follows*and** **Parent/Parent***	1. Both Stmt ref - synonyms a. Some examples but not limited to: i. Follows(s1, s2) & Parent(s1,s2) ii. Follows(a, s1) & Parent(a, s1) 2. Stmt No and StmtRef- synonym a. Some examples but not limited to: i. Follows(2, s1) & Parent(4,s) ii. Follows(3, w) & Parent(13,w) 3. Stmt Ref - synonym and Stmt No a. Some examples but not limited to: i. Follows(s1, 26) & Parent(ifs, 8) ii. Follows(s1,11) & Parent(w,5) 4. Both Stmt No a. Some examples but not limited to: i. Follows(1,2) & Parent(4,5) ii. Follows(9,10) & Parent(13,15) 5. Wildcard a. Some examples but not limited to: i. Follows (a ,_) & Parent(w, _)
Such that **Uses** **and Modifies**	1. Stmt Ref - synonyms and Variable a. Uses(s, v) & Modifies (s,v) 2. Stmt Ref and “Ident” a. Uses (21, “z”) & Modifies(s, “i”) i. “i” and “z” are variables declared in source program 3. Stmt No and Variable a. Uses(5, v) & Modifies(2, v) 4. Procedure a. Uses(procedure name, procedure name) & Modifies (procedure name, procedure name) 5. WildCard a. Uses (s, _) & Modifies(w, _) b. Uses(p, _) & Modifies(p, _)
Such that **Calls and Calls***	1. Both Procedures a. procedure p, q; Select p such that Calls(p, q) 2. Procedure and “Ident” a. procedure q; Select q such that Calls(q, "p") i. “p” is a procedure defined in source program 3. Procedure and “Ident”

	a. procedure q; Select q such that Calls(“p”, q) i. “p” is a procedure defined in source program 4. Wildcard a. Calls(_ , _) & calls(_ , procedure name)
Such that **Next and Next***	1. Both Stmt Ref a. Next(ifs, a) & Next(a1, a2) 2. Stmt Ref and Stmt No a. Next(ifs, 7) & Next(print, a) 3. Stmt No and Stmt Ref a. Next(6, s) & Next(12, if) 4. Both Stmt No a. Next(4, 5) & Next(20, 21)
Such that **Affects and Affects***	1. Both Stmt Ref a. Affects(a,a) & Affects(a,s) 2. Stmt Ref and Stmt No a. Affects(a,9) & Affects(s,7) 3. Stmt No and Stmt Ref a. Affects(4, s) and Affects(2, a) 4. Both Stmt No a. Affects(2, 13) & Affects(11, 11) 5. Wildcard a. Affects(_, _) and Affects(a, _) and Affects(s, _)
With	Some examples but not limited to: 1. constant c; assign a; Select c with c.value = a.stmt# 2. variable v; Select v with v.varName = "x” 3. procedure p; Select p with "proc" = p.procName
Assign Pattern Clause	1. Partial Pattern Matching a. assign a, z; Select a pattern a ("z", _"z + x"_) 2. Full Pattern Matching a. assign a; Select a pattern a (“y", "z + i") b. assign a; Select a pattern a ("y", "x + z") 3. Wildcard a. assign a;Select a pattern a (_, _"z "_)

If and While Pattern Clauses	Some examples but not limited to: 1. if ifs; variable v; Select ifs pattern ifs (v, _, _) 2. if ifs; stmt s; Select s pattern ifs ("z", _ , _) 3. while w; stmt s; Select s pattern w ("x" , _) 4. while w; variable v; Select v pattern w (v, _)
Multi clauses	Combination of such that and pattern clause: 1. Follows/* and Pattern clauses 2. Parent/* and Pattern clauses 3. Modifies Pattern clauses 4. Uses and Pattern clauses

Figure 5.11: QPS and PKB Integration Test Plan

5.2.3 Integration Test Statistics

The figure below shows the line coverage (%) of integration tests across code folders.

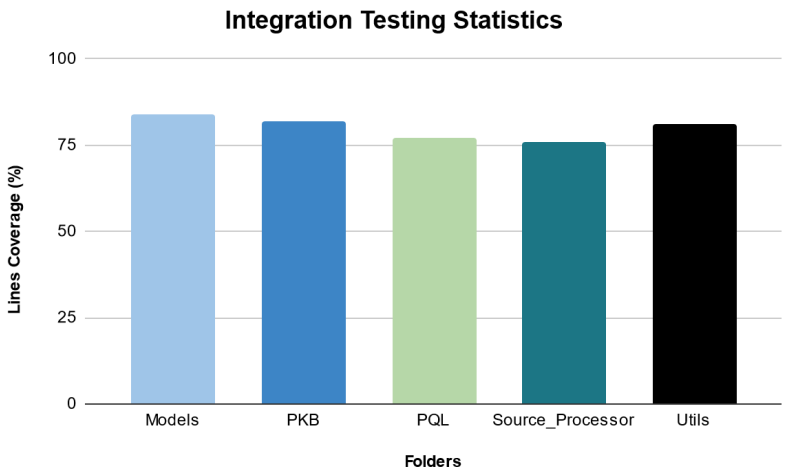

Figure 5.12: Integration Tests Line Coverage

Integration testing efforts were consistent across the Models, PKB, PQL, Source_Processor and Utils folders. Each of these folders had at least 70% of line coverage. As the API calls between SPA subcomponents were the main focus of integration tests, not all lines of code are covered in integration tests. Hence, the line coverage for integration tests is lower than that for unit tests.

5.3 System Testing

Unit and integration testing ensures that the internal components of the SPA behave as intended. However, there might be a potential mismatch between the specified external and internal behavior of the SPA. Thus, it is necessary to test the SPA against external specifications through system testing.

5.3.1 System Testing Approach

Since end-users of the SPA place a large emphasis on functional correctness (as reflected by how the SPA is graded primarily on program correctness), a rational test strategy would be to cover all parameter permutations of possible queries. However, such an approach will result in too large of a system test set. As such, system testing is done by choosing pairwise combinations of parameters to attain a reasonable test coverage.

5.3.2 System Test Design

The system test cases are designed using a top-down approach to test in a structured and organized manner. Firstly, test objectives are defined. Then, pairwise combinations of query arguments are considered to fulfill the test objectives. Based on the test objective and combinations of query inputs, an appropriate source program is decided upon to construct to test the query inputs. Lastly, the output of the source program will be determined based on the query inputs.

For example, the table below illustrates the utilization of a top-down approach to plan for the Parent Suite test cases.

1. Test Objective	**2. Parameter Permutations**	**3. Sample Clause Combinations (Pairwise)**
Test that possible permutations of Parent parameters return correct result stmt s, s1, s2; while w; if ifs; constant c; assign a;	Parent(<synonym>, <synonym>)	Parent(s1, s2), Parent(a, s), Parent(w, w)
	Parent(<synonym>, _)	Parent(ifs, _), Parent(c, _), Parent(a, _)
	Parent(<synonym>, <INTEGER>)	Parent(s, 4), Parent(w, 5), Parent(ifs, 6)
	Parent(_ , <synonym>)	Parent(_, a), Parent(_, c), Parent(_, s)
	Parent(_, _)	Parent(_, _)
	Parent(_ , <INTEGER>)	Parent(_, 3), Parent(_, 4)
	Parent(<INTEGER>, _)	Parent(2, _), Parent(_, 3)
	Parent(<INTEGER>, <synonym>)	Parent(3, s), Parent(4, w), Parent(5, ifs)
	Parent(<INTEGER>, <INTEGER>)	Parent(3,2), Parent(3, 3), Parent(3,4)

Figure 5.13: Parent Suite Top-Down Approach Illustration

The clause combinations are then pairwise combined with the Select synonym to generate query combinations. From there, specific query combinations are shortlisted for testing based on three major considerations.

Firstly, query combinations that have non-trivial resultant queries are shortlisted so that the test output is meaningful. For example, the query “stmt s1, s2; Select s1 such that Parent(s1, s2)” is non-trivial because it selects statements s1 that are parents of statement s2. However, the query “constant c; stmt s1, s2; Select c such that Parent(s1, s2)” is trivial because Parent(s1, s2) returns a boolean value which either returns all or no constants. As such, the second query example does not provide any valuable insights into the workings of the Parent relationship.

Secondly, queries belonging to different equivalence partitions from non-trivial queries are shortlisted. Queries belonging to different equivalence partitions are likely to be processed by the SPA in different manners. By selecting queries from different equivalence partitions, test effectiveness is improved. However, in cases where it is uncertain if two particular non-trivial queries belong to the same equivalence partition, both queries are shortlisted for comprehensive test coverage.

Lastly, test cases which handle boundary values of equivalence partitions for parameter permutations such as Parent(<INTEGER>, <INTEGER>) are added. This is because bugs often result from incorrect handling of equivalence partition boundaries.

5.3.3 System Test Organization

The system tests are organized into test collections and test suites. A test collection consists of multiple test suites. Each test suite has a source and query file. For example, the Iteration 1 test collection contains the Parent test suite. The Parent test suite has a source text file and a queries text file that specifically tests the Parent relationship behaviour.

5.3.4 System Test Sample

Here, two sample test cases for the SIMPLE source code and PQL query are provided to illustrate test plan entries.

SIMPLE Source Code

Test Suite-No	Test Objective	Input(Autotester Format)	Expected Output	Pass/Fail
Source-01	Test that valid source program can be parsed to generate Uses/Modifies /Pattern	Single Assignment procedure sourceSuite { z = ((1% (2) % a / (b -C) +3 /d)) % 4+E /5* F - (g + 6 + 7 - 8) * (	1 (The value 1 is returned by SPA when query is parsed and evaluated)	Pass

	abstractions	(h)) + 9 + i % 10 % (1) ; } 1 - Source-01 stmt s; Select s 1 5000		

Source-02	Test that valid source program can be parsed to generate Follows(*)/ Parent(*) abstractions	Single If-Else
		
		procedure sourceSuite {
		if((1 > a)&&((2 - 3 + ((B))) + (4) > 5))then {c = 3;} else { c = 4;}
		}
		
		2 - Source-02
		stmt s;
		Select s
		1, 2, 3
		5000
			1, 2, 3
			
			(The values 1, 2 and 3 are returned by SPA when query is parsed and evaluated)
				Pass

Figure 5.14: Source Suite Sample Tests

PQL Query

Test Suite-No	Test Objective	Input (Autotester Format)	Expected Output	Pass/Fail
Parent-02	Test that possible permutations of Parent parameters return correct result **Permutation - Parent(<synonym>, <synonym>)**	2 - Parent-02 stmt s1, s2; Select s1 such that Parent(s2, s1) 4, 5, 6, 7 5000	4, 5, 6, 7	Pass

Parent-45	Test that possible permutations of Parent parameters return correct result **Permutation - Parent(<INTEGER>, <synonym>)**	45 - Parent-45 if ifs; Select ifs such that Parent(3, ifs) 5 5000	5	Pass

```
procedure parentSuite {
1.  var1 = 1;
2.  var2 = 2;
3.  while (var1 != 1) {
4.        var3 = 3;
5.        if (var2 != 2) then {
6.          var4 = var2;
          }
          else {
7.          var4 = var2;
          }
   }
 }
```

Figure 5.15: Parent Suite Sample Tests

5.3.5 System Test Statistics

The two figures below show the breakdown of tests across test suites.

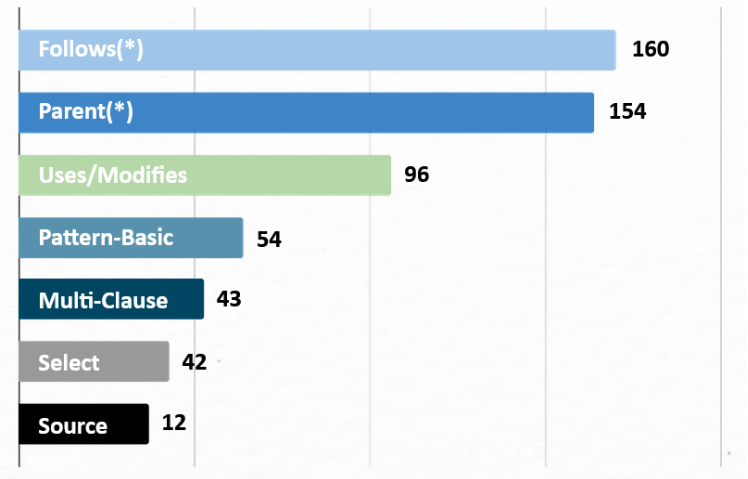

Figure 5.16: Iteration 1 Test Cases Breakdown

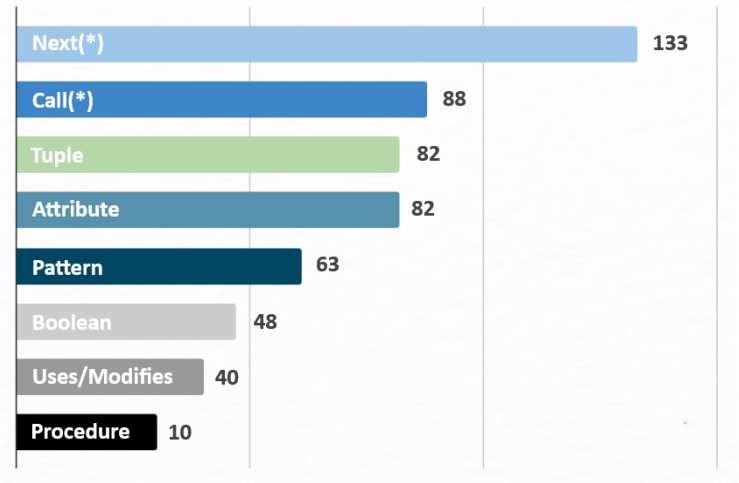

Figure 5.17: Iteration 2 Test Cases Breakdown

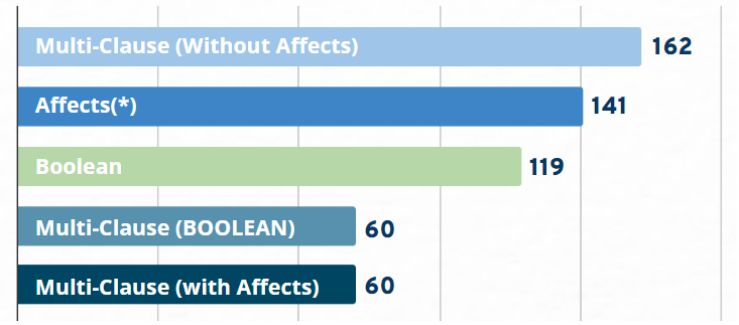

Figure 5.18: Iteration 3 Test Cases Breakdown

The Iteration 1 test collection has 10 test suites and has 561 test cases. The Iteration 2 test collection has 11 test suites and 546 test cases. The Iteration 3 test collection has 4 test suites and 542 test cases. In iteration 3, 119 boolean tests were also added to the Boolean suite in iteration 2 to account for many complex edge cases.

Test cases are mostly balanced across test suites for a holistic test coverage. Note that suites indicated with an asterisk (*) consist of two separate test suites. For example, Parent(*) consists of the Parent and Parent* test suites.

5.4 Load Testing

To attain an even more comprehensive test coverage, load tests were conducted on SPA. While the primary focus of systematic testing is to ensure external functional behavior correctness, the focus of load testing is to ensure that external non-functional requirements on system performance are fulfilled. Specifically, load testing aims to provide guarantees on the ability of our SPA to handle specified program loads.

5.4.1 Load Test Design

Since the SPA is graded primarily on program correctness, a rational strategy for load testing will be to ensure that the program does not crash unexpectedly when the SPA is evaluating significant program loads. Load requirements for test cases are identified based on the CS3203 SPA Correctness Grading Specifications. The grading specifications are as follows:

- Maximum number of lines of source program code (Iteration 3 evaluation) - 500 lines
- Maximum number of queries (Iteration 1-3 evaluation) - 500 lines
- Query answers must be returned within 5000ms
- Maximum size for INTEGER - Within reasonable assumptions
- Maximum length of NAME - Within reasonable assumptions
- Maximum level of nesting in source program - Within reasonable assumption.

From there, load requirements are identified from the grading specifications.

Program Load Requirements:

- SPA should be able to parse 1000 lines of a single text file source program code (no nesting)
- SPA should be able to parse a source program code with the maximum number of nesting allowed by 500 lines of code
- SPA should return answers must be returned within 5000ms
- Source program should be able to handle INTEGER with a maximum size of 2,147,483,647 (Typical maximum positive value for 32-bit signed binary integer in computing)
- Source program should be able to handle NAME with a maximum length of 50 characters (Typical string length is around 20)

5.4.2 Load Test Objectives

Based on the above load requirements, the following test objectives to fulfill are defined:

Test Suite	Test Objective	Input	Pass/Fail
Assignment	Test that SPA is able to parse 100/500/1000 lines of assignment statements in source program **Clear condition - Parsing time <= 3/5/10 seconds**	100/500/1000 lines of assignment statements in Source program + 1 statement query Statement query: stmt s; Select s	Pass
Nested	Test that SPA is able to parse a source	100/500 layers of nested Source program + 1 statement query	Pass

	program with 100/500 layers of nesting **Clear condition - Parsing time <= 5/70 seconds**	Statement query: stmt s; Select s	

Figure 5.19: Load Test Samples

5.5 Stress Testing

In Iteration 3, stress testing was conducted to identify speed bottlenecks for optimization. Stress testing was used in conjunction with optimization heuristics designed by the team. Assuming that certain optimization heuristics improved SPA speed efficiency, stress testing was used as a proof of concept to validate the improvement in bottleneck performance.

5.5.1 Stress Test Design

The following suite below is defined for stress testing. The team created a complicated source program of approximately 600 lines and added computationally heavy clauses such as Affects(*), Follows(*) and Next(*) to push the SPA to beyond the point of failure. The metric for failure was the time limit of 5000 milliseconds. The figure below shows the stress test suite design.

Test Suite	Test Objective	Input	Result (Windows OS)
Stress	Test the maximum number of clauses that can be handled by the SPA from Affects(*), Calls(*), Follows(*) and Next(*) **Clear condition - Each query executes within 5000ms**	Approximately 600 lines of complicated Source program + multi-clause queries of various query lengths. Clauses used include Affects(*), Calls(*), Follows(*), and Next(*) clauses.	SPA is able to handle: 1. At most two computationally heavy clauses when selecting BOOLEANs, 2. At most one computationally heavy clause tuple selection queries 3. At most two computationally heavy clause when selecting single synonyms

Figure 5.20: Stress Test Design

5.5.2 Stress Test Samples

The figure below shows sample inputs in the stress test suite.

Test Suite-No	Query	Result

		(Windows OS)
Stress-13	stmt s1, s2; Select BOOLEAN such that Affects*(s1, s2) and Next*(s1, s2)	Pass
Stress-14	stmt s1, s2; Select BOOLEAN such that Affects*(s1, s2) and Next*(s1, s2) and Follows*(s1, s2)	Timeout
Stress-25	stmt s1, s2; Select <s1, s2> such that Affects*(s1, s2)	Pass
Stress-27	stmt s1, s2; Select <s1, s2> such that Affects*(s1, s2) and Next*(s1, s2)	Timeout
Stress-32	stmt s1, s2; Select s1 such that Affects*(s1, s2) and Next*(s1, s2)	Pass
Stress-33	stmt s1, s2; Select s1 such that Affects*(s1, s2) and Next*(s1, s2) and Follows*(s1, s2)	Timeout

Figure 5.21: Stress Suite Sample Tests

5.5.3 Stress Test Results

Next(*) clause evaluation and boolean selection optimizations were made post-stress testing, improving boolean and single synonym selection speed efficiency.

However, assuming a fixed number of clauses, clauses involving tuple selection were slower relative to its boolean or single synonym selection counterparts as cross product had to be performed for tuple selection. Hence, it was observed from the stress test results that the SPA could handle lesser computationally heavy clause queries with respect to tuple selections.

Through the joint usage of optimization heuristics and stress testing for optimization validation, the team managed to improve the bottleneck performance of the SPA to two computationally heavy clause queries on average on the Windows Operating System.

5.6 Test Strategy

In this section, various test strategies adopted for SPA testing will be discussed. This includes the defect management lifecycle, automation techniques, and planning of testing activities.

5.6.1 Defect Management Lifecycle

The defect management lifecycle is an integral aspect of quality assurance which provides a framework in identifying, troubleshooting, and ensuring proper closure of defects found

during testing. For the project, a simplified defect management lifecycle with five defect states is adopted. This is sufficient for managing defects in a six-member team. Bugs are resolved within 40 hours of identification. The diagram below shows the happy path of our defect management workflow.

Figure 5.22: Defect Management Workflow Happy Path

The first state of the lifecycle is the New state where bugs are identified in the SPA. Upon identification, the team member handling Quality Assurance (QA) will notify the rest of the team.

In the Assigned State, the bug is diagnosed upon further investigations. A Github Issue is created and assigned to the relevant team member in charge of the component causing the bug. The assignee of the issue has to resolve the bug within 24 hours.

Upon bug resolution, QA will retest within 12 hours in the Retest state before verifying that the bug has been successfully resolved in the Verified/Closed state.

5.6.2 Automation Techniques

Apart from adopting a defect management workflow for proper defect closure, Python scripts were used for system and load test automation to reduce manual work.

Automated Query Generation

For system testing, a script (query_generator.py) was used to generate autotester readable test queries written in the system test plan. Incremental serial numbers associated with the test queries were dynamically generated by the test script. This allowed for easier test set extension as new test cases can be slotted into an existing test suite easily without reordering serial numbers allocated to each test case.

Automated Local System Test Runs

In addition, a script (test_runner.py) was written to pull a built autotester from the build folder, run all test suites, and automatically validate XMLs generated for test failures.

Automated Remote System Test Runs

System tests are introduced into continuous integration on GitHub for automated execution of system tests upon code pushes. This is done by running test_runner.py remotely. test_runner.py pulls the autotester.exe from the build folder, runs test cases in the test folder, and validates XML output for any test failures. The figure below shows a successful

remote test run. The last line of code in the figure, line 6310, indicates successful XML validation as the test run reported: “All Iteration 1 xml passed”.

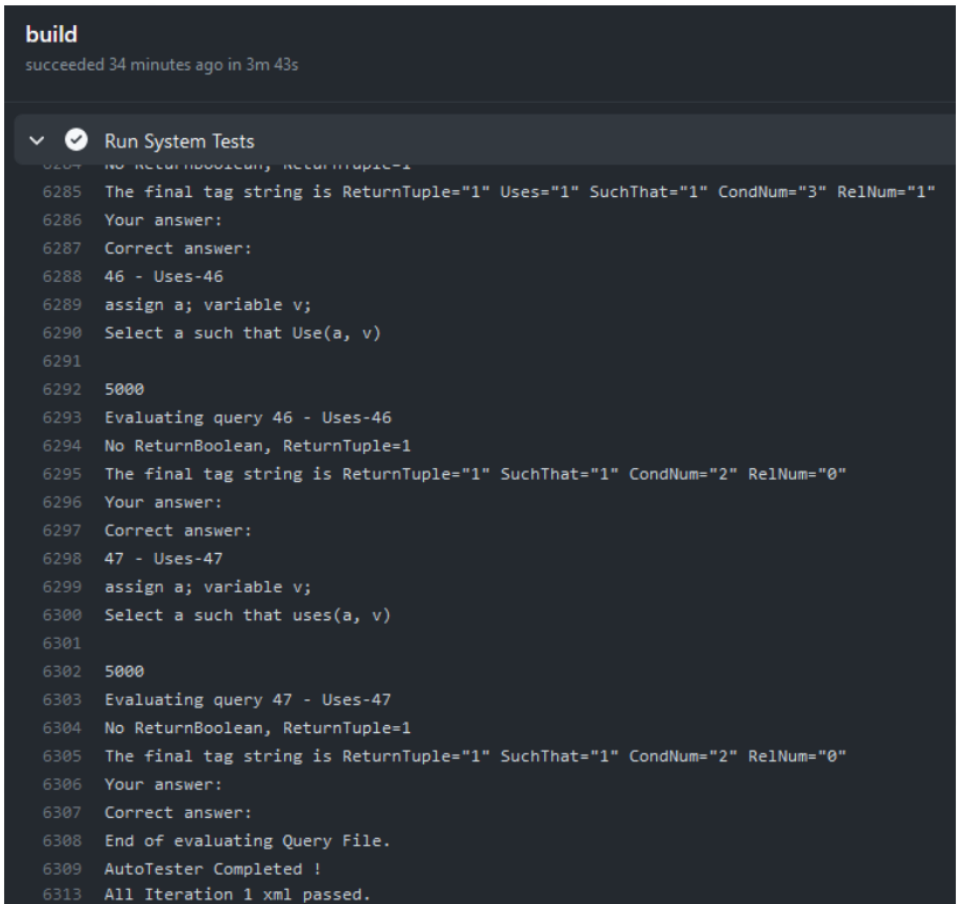

Figure 5.23: System Tests Continuous Integration

Automated Source Program Generation

For load testing, the generation of complex SIMPLE source programs was also automated using the script source_generator.py. source_generator.py is a script which recursively generates syntactically valid source programs. The figure below showcases the complexity of a source program used for load testing.

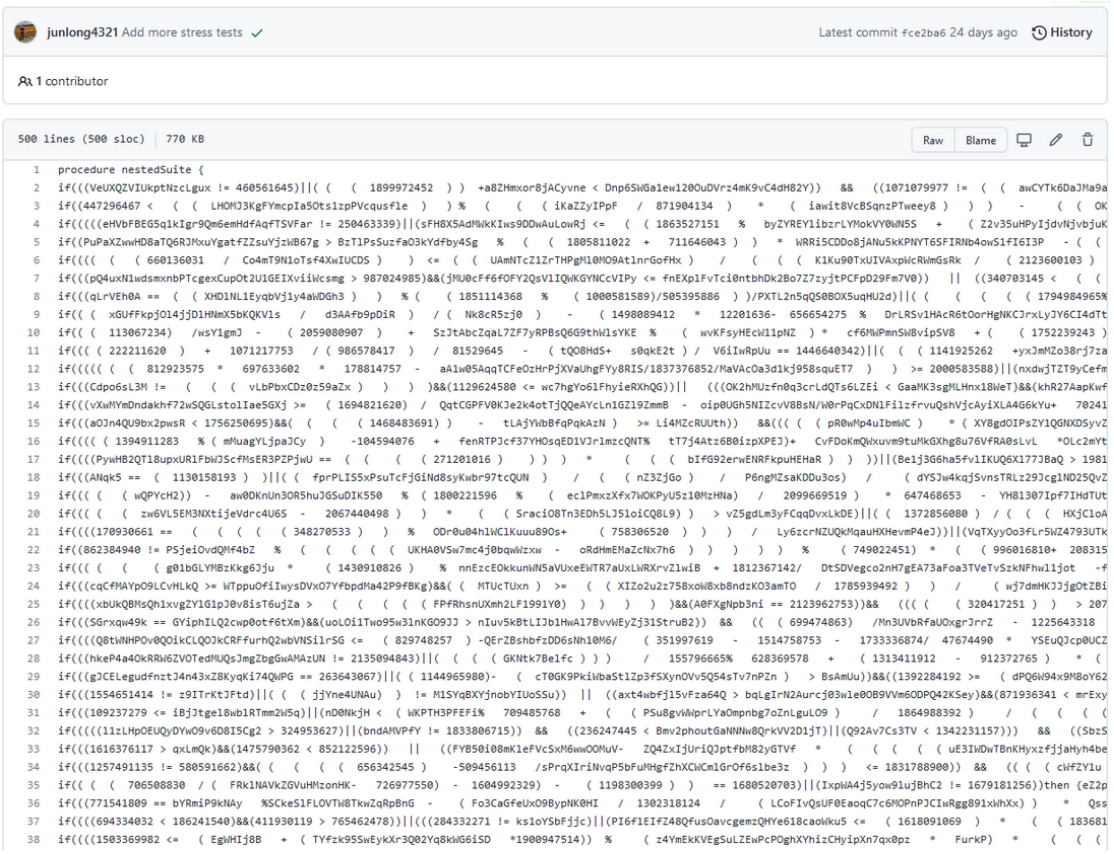

Figure 5.24: Complex Source Program (Load Testing)

5.6.3 Test Plan - Iteration 1

The Gantt chart below illustrates actual testing activities for SPA Iteration 1.

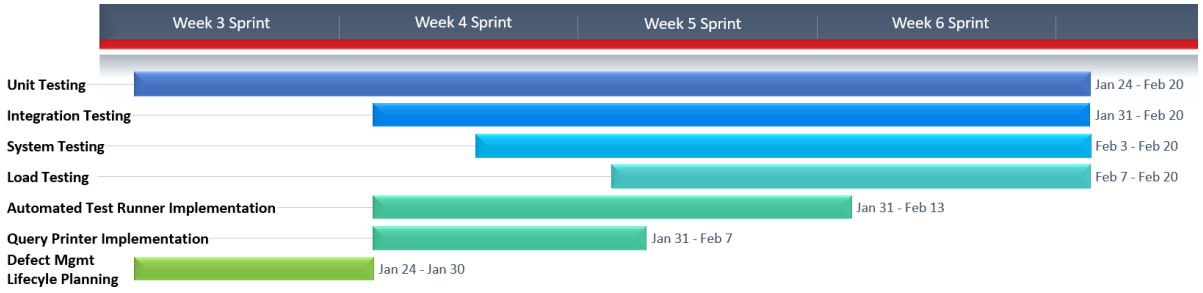

Figure 5.25: Test Plan Gantt Chart (Iteration 1)

Unit testing was conducted across all sprints in Iteration 1 to ensure that modular units in the code base functioned as intended. This is crucial because higher-level classes are dependent on lower-level modular units.

On the other hand, integration testing was started early in the iteration as it is harder to locate test failure causes as the code base becomes larger.

Defect management lifecycle was planned and discussed among team members early in the development cycle to familiarize everyone with proper defect closure.

Unfortunately, system and load testing were delayed as the team was occupied with basic SPA implementation in Iteration 1. Delaying system testing was a costly mistake as major refactors had to be done to the codebase. This will be further elaborated in reflection section 7.2.2.

5.6.4 Test Plan - Iteration 2

The Gantt chart below illustrates actual testing activities for SPA Iteration 2.

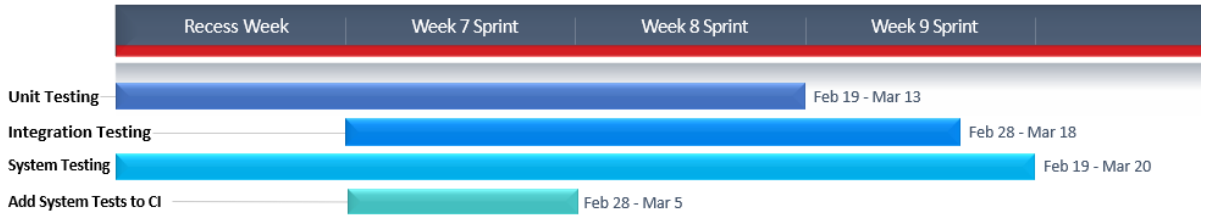

Figure 5.26: Test Plan Gantt Chart (Iteration 2)

A learning point from Iteration 1 was to avoid system testing delays as it caused major refactors to the codebase. Thus, the team started conducting system tests early in the iteration. This arrangement worked well as developers were informed of important edge cases early in the iteration by the tester. As such, developers were able to thoroughly consider design and implementation details before beginning on implementation. This prevented major code refactors to the codebase.

On the other hand, integration testing began in Week 8 as the team was in the process of implementing the remaining subset of basic SPA during Recess Week and Week 7.

A potential improvement will be to pay more attention to non-functional testing in Iteration 3. This is because SPA efficiency is graded in Iteration 3. Yet, non-functional testing was not conducted in Iteration 2. As such, the team will be ramping up non-functional testing efforts to improve SPA speed and reduce memory usage in Iteration 3.

5.6.5 Test Plan - Iteration 3

The Gantt chart below illustrates actual testing activities for SPA Iteration 3.

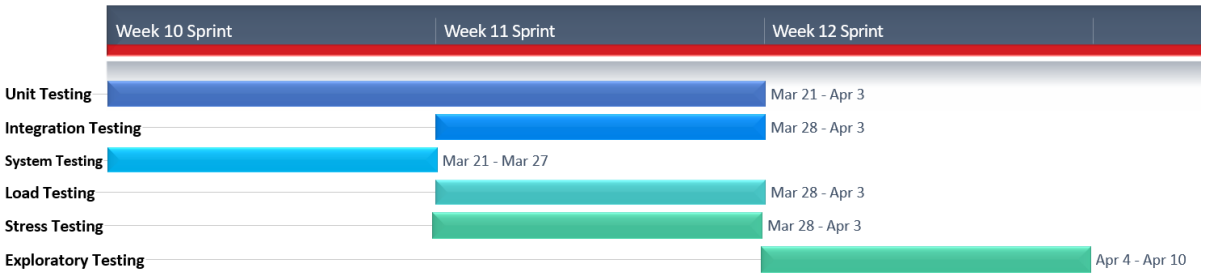

Figure 5.27: Test Plan Gantt Chart (Iteration 3)

In Iteration 3, system testing was completed in Week 10. Load tests were refined and stress tests were added in Week 11 to improve non-functional testing efforts. Stress tests included Next(*) and Affects(*) to ensure that the SPA could handle non-precomputed load, since precomputation of the above mentioned clauses were not allowed.

In Week 12, a reactive testing technique called exploratory testing was adopted. Exploratory testing was suitable as the tester already has prior experience working on the SPA system tests. Exploratory testing was done by running queries found in CS3203 Assignment 1 and Assignment 2 and diagnosing test failures. After test failures were diagnosed, new test cases were devised on-the-fly based on the tester's intuition to detect bugs. This technique allowed the team to quickly identify edge case bugs leading up to the Iteration 3 submission.

Part 2 – Project management

6 Project Management

6.1 Management Tools and Workflow

This section describes the management tools and workflows that the team uses to manage the SPA project.

6.1.1 Weekly Sprints

Weekly sprint meetings are held on Sunday mornings. The meeting begins with a retrospective meeting where issues progress for the current sprint is reviewed. Following which, issues for the next sprint are identified and estimated time commitments are made for the sprint goals. Time is also set aside to discuss implementation details.

6.1.2 GitHub Issues

Tasks are created and tracked using GitHub Issues.

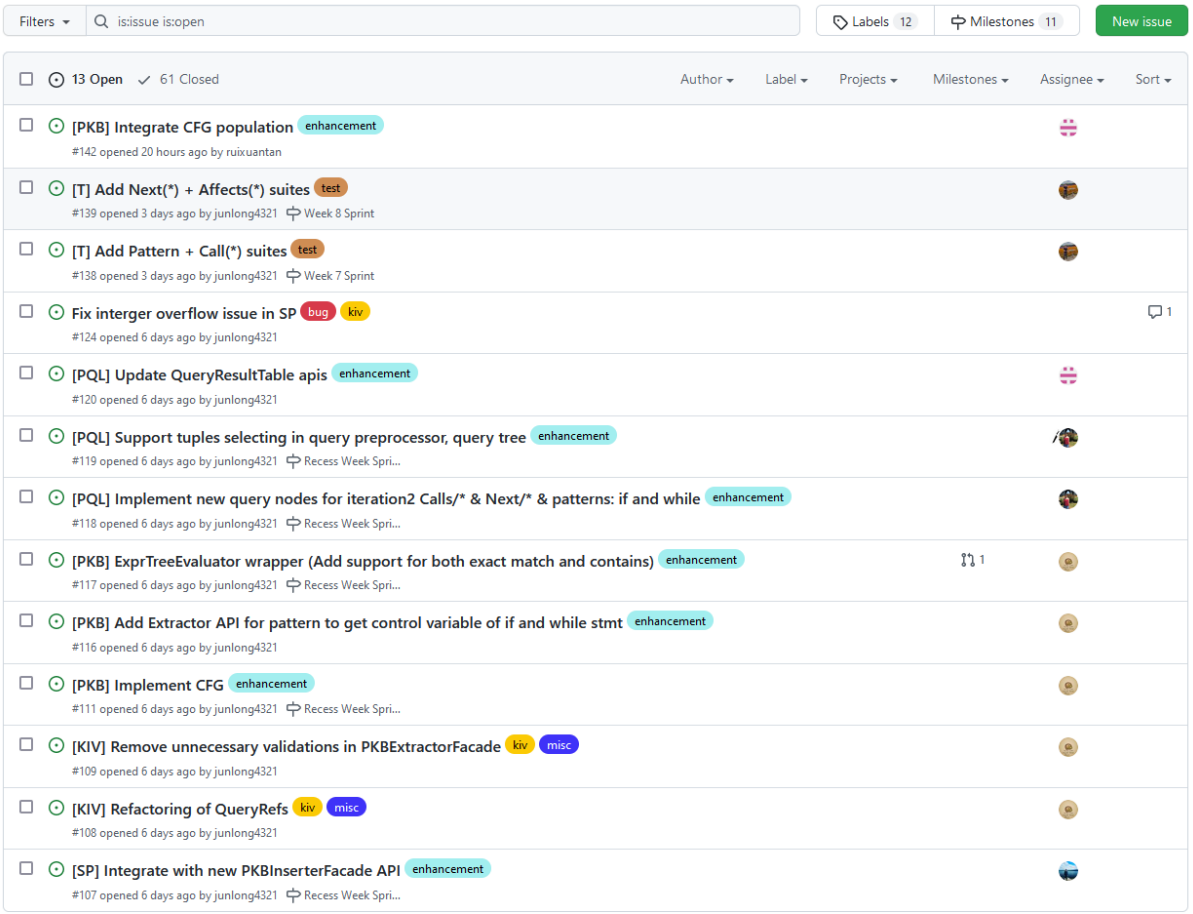

Figure 6.1: Team26 GitHub Issues

An issue is created when a new task arises. Tasks can be coding-related, such as SPA implementation and bug fixes, or non-coding related, such as discussion of implementation design.

Tasks are further categorized with the following tags:

1. G: General
2. KIV: Keep in view
3. PKB: Program Knowledge Base
4. PQL: Query Processor
5. SP: Source Processor
6. T: Testing

Tags allow for greater issue visibility by segmenting tasks into predefined categories.

Each issue is assigned to a team member to emphasize on task ownership.

6.1.3 GitHub Projects

GitHub Projects is used in conjunction with GitHub issues as a visualization tool to optimize workflow. The project board provides the team with a high level overview on the progress of each member and of the current sprint.

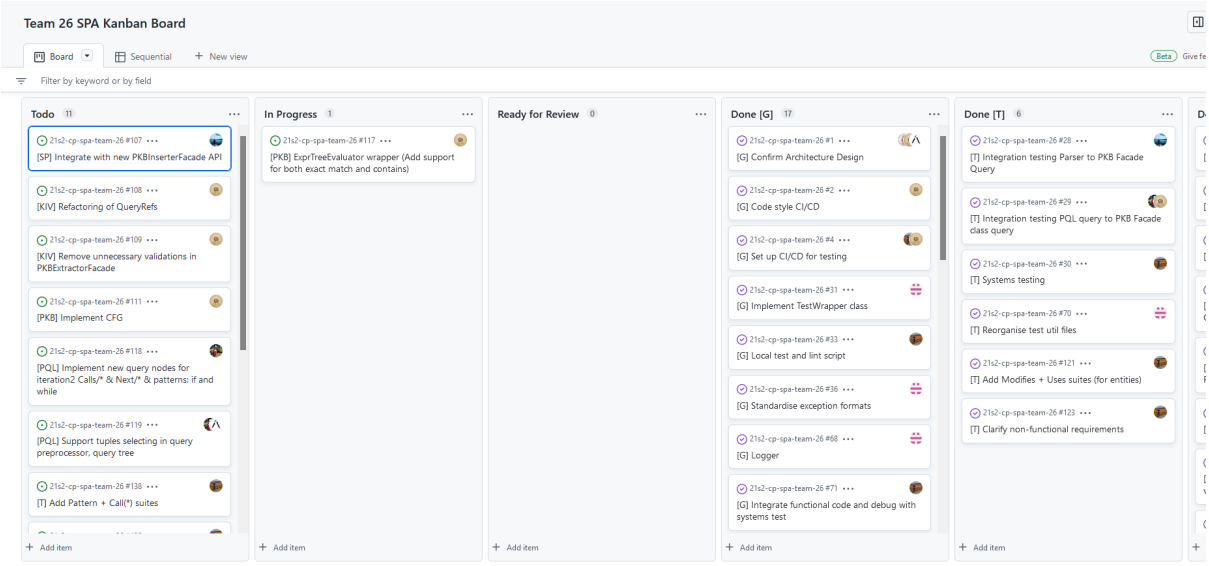

Figure 6.2: Team26 GitHub Project Management Board

The board is split into mainly 4 sections: Todo, In Progress, Ready for Review, Done Issues that are to be done in the current sprint are added to the Todo section. If a component is currently being worked on, it will be moved to the In Progress section. Once implementation is completed, the task is moved to the Ready for Review section. The task is moved into the Done section when all reviews are done and the code is merged. Each team member is in charge of managing their own tasks on the board.

6.1.4 Github Actions for Continuous Integration

The team uses GitHub actions for Continuous Integration (CI), automating code integrations from multiple contributors into a single branch.

CI consists of two workflows:

1. Build project
2. Style check

Both of these workflows run upon any branch pushes or pull requests.

Build project builds the project and runs unit, integration and system tests. Build or test failures will lead to a rejection by GitHub. Upon failure, merging of the pull request is not allowed.

The style check workflow uses cpplint, a static code style checker for C++ to ensure consistent code style. It uses Google C++ code conventions. Ensuring consistent style improves the readability of the code base, allowing for ease of future modifications.

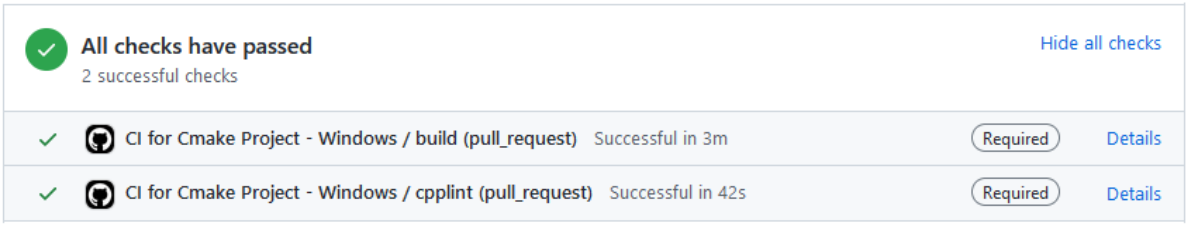

Figure 6.3: GitHub CI All Checks Pass

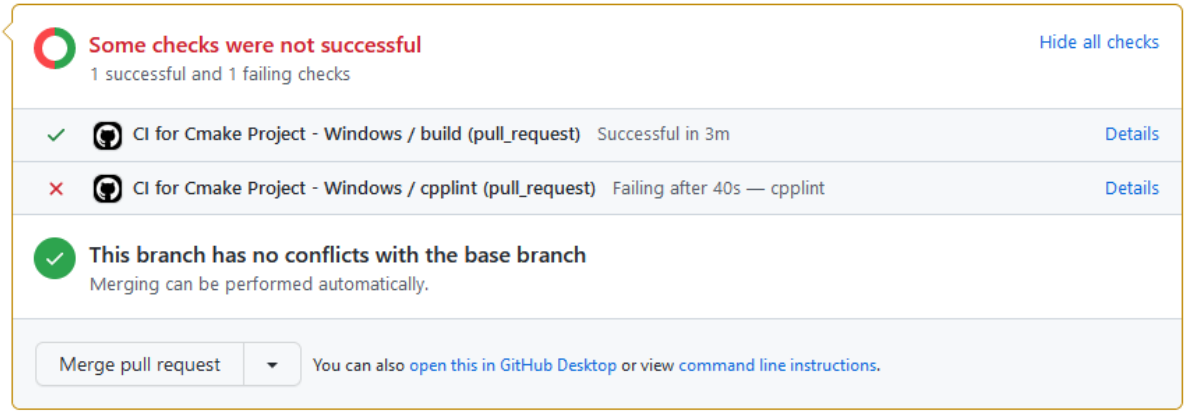

Figure 6.4: Team26 GitHub CI One Check Fail

6.2 Project Plan

In this section, Gantt charts will be used as a visual aid to show tasks distribution and schedule among team members.

6.2.1 Iteration 1 Project Plan

The Gantt chart below showcases major activities accomplished in Iteration 1 on a weekly basis.

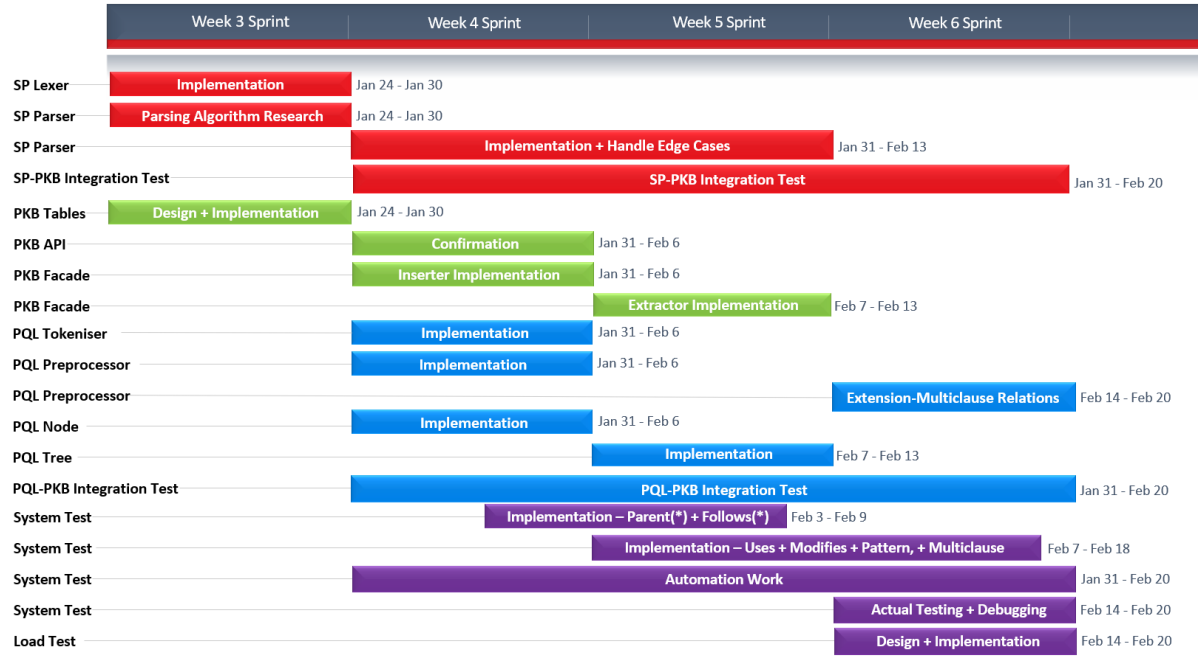

Figure 6.5: Project Plan Gantt Chart (Iteration 1)

As the focus of Iteration 1 is to build the SPA ground-up, emphasis was placed on research, design and concurrent implementation of components between Weeks 3 and 5.

System and load testing was done between Weeks 4 and 6 as team members were occupied with SPA implementation in the earlier weeks. For Iterations 2 and 3, the team will delegate a member to specifically focus on QA. The implementation of system tests will start early in the Iteration to prevent major code refactors, a costly mistake made in Iteration 1 which is elaborated under Reflection section 7.2.2

6.2.2 Iteration 2 Project Plan

The Gantt chart below showcases major activities accomplished in Iteration 2 on a weekly basis.

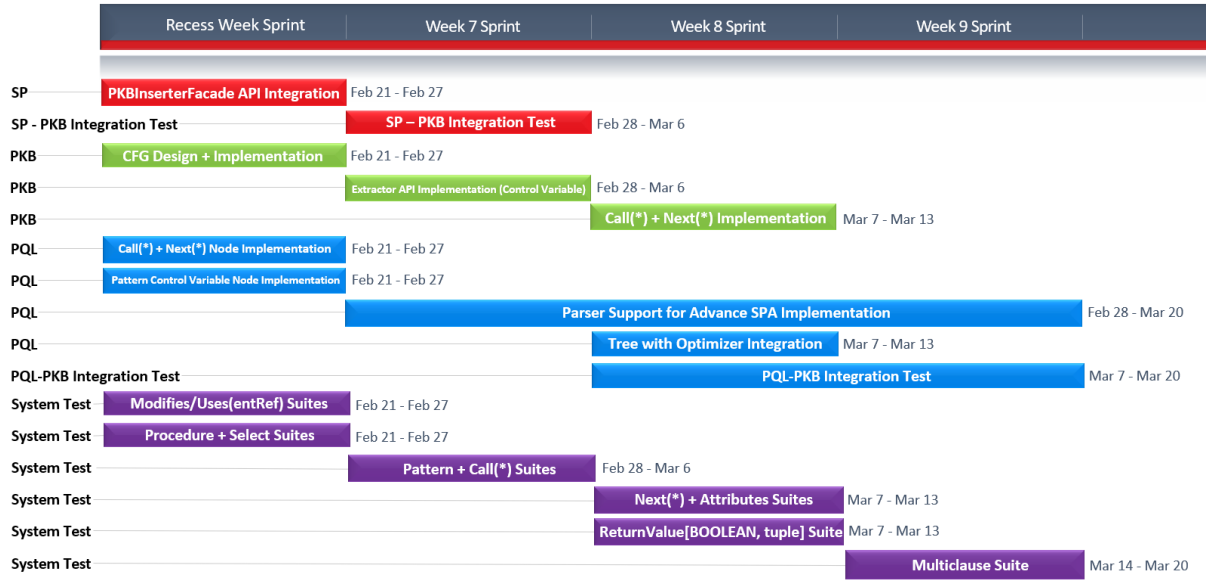

Figure 6.6: Project Plan Gantt Chart (Iteration 2)

As most deadlines could be met on time with the Iteration 1 plan, the team followed a similar plan for Iteration 2.

The focus for the first half of Iteration 2 was on completing basic SPA implementation and beginning on advanced SPA implementation. Similar to Iteration 1, integration tests were carried out immediately once individual components were ready for integration. The main difference between Iterations 1 and 2 is that consistent system testing was conducted throughout Iteration 2.

By following the above plan, requirements for Advanced SPA are fully implemented apart from Affects(*), and attribute tuple selection, which will be implemented in Iteration 3.

6.2.3 Iteration 3 Project Plan

The Gantt chart below showcases major activities accomplished in Iteration 3 on a weekly basis.

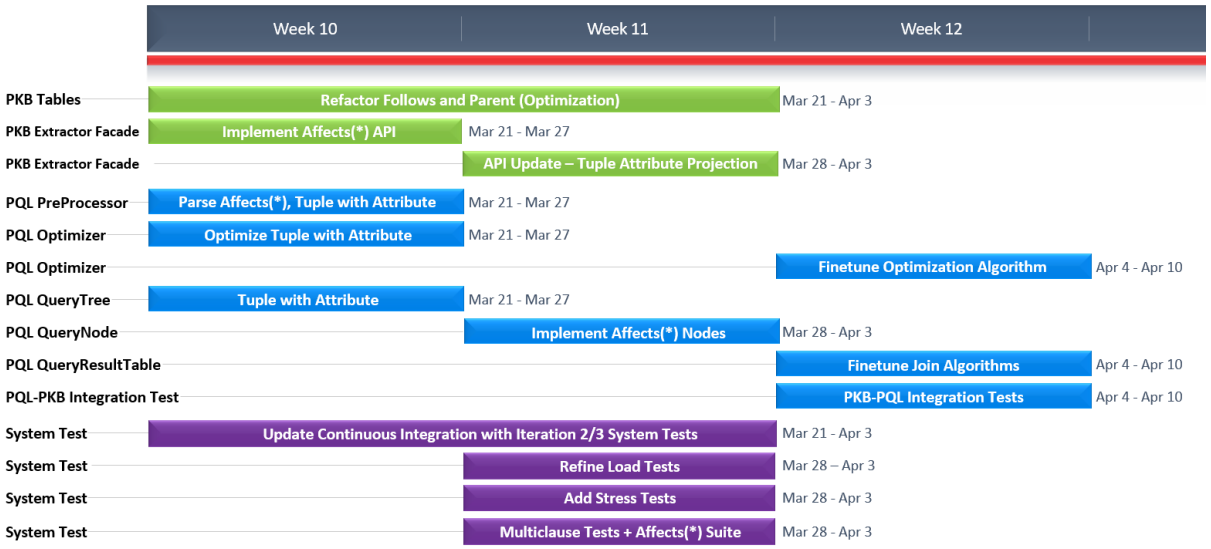

Figure 6.7: Project Plan Gantt Chart (Iteration 3)

In Iteration 3, the focus for PKB and PQL was to implement the remaining advanced SPA requirements of Affects(*), tuple support for attribute projection, and further optimization of algorithms used.

On the other hand, system testing focused on Affects(*) and multiclause queries suite implementation. Specifically, two multiclause suites were implemented. The first multiclause suite was implemented to test multiclause boolean selection while the second multiclause suite was implemented to test multiclause synonym selection. Both multiclause suites included Affects(*) clauses to test the newly implemented Affects(*) rigorously.

In summary, as a result of consistent project planning from Iteration 1 to 3, the team was able to complete both basic and advanced SPA requirements and perform rigorous testing on time.

Part 3 – Conclusion

7 Reflection

This section provides reflections on some of the insights that were gleaned from working on the project. For each iteration, two effective practices that were adopted by the team and two areas that could be improved upon are elaborated.

7.1 Effective Practices - Iteration 1

7.1.1 Consistent Clarifications

Throughout Iteration 1, whenever an ambiguity arose from project specification, our team sought immediate clarification with our teaching assistant and Professors on Microsoft Teams.

For example, although it was stated in the CS3203 Project Wiki FAQ that reasonable assumptions can be made regarding maximum size for INTEGER, we were unsure of what basis these assumptions can be made under. As such, we clarified on Microsoft Teams if we could use the value 2,147,483,647, the maximum positive value for 32-bit signed binary integer in computing as a reasonable assumption.

Adopting a mindset of making consistent clarification saved us a lot of time from potential code changes, which could have arisen if we had assumed and wrongly interpreted the project requirements.

7.1.2 Weekly Timetables

In addition, apart from utilizing Github as a platform to manage our project, the team made an effort to draft weekly timetables where we indicated which days of the week we will be working on our task.

For example, the timetable below shows each team member's assigned task for Week 7 Sprint.

Date	**PKB**	**PQL**	**T**
28/02/2022			Implement Next(*) Suite [Jun Long]
03/03/2022	Next/Calls/Pattern APIs Implementation [Jun Wei]		
04/03/2022		Integrate with Next/Calls/Pattern API [Si Ting]	

		Refactor Query Evaluator [Si Ting]	
05/03/2022	CFG Insert Methods Implementation [Rui Xuan]	Implement Full Expression + Next/Calls Support [Ming Lim] Implement Query Optimizer [Jia Dong]	
06/03/2022		Integrate Query Evaluator with Query Optimizer [Si Ting]	Implement Affects(*) Suite [Jun Long]
Code Demonstration (07/03/2022)			

Figure 7.1: Weekly Timetable

This kept everyone accountable for their assigned tasks. As such, we were able to meet most deadlines that we set.

7.2 Potential Improvements - Iteration 1

7.2.1 Insufficient Component Design Planning

While designing the various components of SPA, we did not come up with a detailed plan of what was to be implemented. As such, we had to refactor a component, the Query Evaluator, halfway through Iteration 1.

The design and implementation details of the Query Evaluator were only briefly mentioned. We only agreed that the Query Evaluator was to store Query Nodes, and would traverse through them to evaluate the PQL query. Other details such as the data structure used, the specific implementation of the nodes (polymorphism), and the traversal of the nodes were not finalized.

This caused implementation problems as there was a mismatch between the expected and actual PKB APIs that were designed and called. Consequently, our project schedule got delayed as we had to refactor the Query Evaluator and PKB APIs.

For subsequent iterations, we will come up with a concrete plan of the design before working on implementation. This can be done by sketching out UML diagrams to communicate the design details. We will also use Google Docs to finalize implementation details in writing. For Iteration 2, this will apply to the design of the Query Optimizer, update of the Query Evaluator, Calls/* data structure, and CFG.

7.2.2 Delay of System Testing

Due to tight time constraints with basic SPA implementation, we only appointed a member to focus solely on QA in Week 5. As a result, we did not have the aid of system test cases to consider possible edge cases thoroughly while implementing components.

This issue surfaced in the interaction between the PQL and PKB components. Edge cases such as wildcards, passing raw variables such as Uses(s, "x") were neglected. To solve this issue, we had to assign wildcards a special numeric value for each 'stmtRef' and 'entRef' variant and create new PKB APIs to accept raw variable strings as parameters. This eventually led to more difficulty in debugging, which led to another refactor. We would have noticed these edge cases if we had begun writing system test cases early in the iteration.

For Iterations 2 and 3, we will prioritize the development of system tests. Our teammate responsible for QA will inform the respective component teams of important edge cases that need to be considered.

7.3 Effective Practices - Iteration 2

7.3.1 Thorough Design and Planning of SPA Components

In Iteration 2, we made an effort to thoroughly plan out the design of all SPA components before working on the code implementation. This plan was done for the CFG and evaluation of with-node clauses. We wrote the details of these designs on word documents and shared them with all team members on the cloud.

In the design of the CFG for example, the API details and graph traversal algorithms were finalized before implementation began. As a result, most edge cases were covered on the first implementation attempt. Although some minor bugs still surfaced, there was no need to perform an entire refactor of the code.

This practice also allowed those who were responsible for the components affected to be on the same page with respect to implementation details. For example, in the design document of with-node clauses, information such as edge cases, evaluation strategy, and parsing were written. This allowed those who were in charge of the PKB Extractor Facade, Query PreProcessor, and Query Node components to have easy reference to the APIs and data structures that were used. As such, there were no implementation delays due to miscommunication.

We will continue with this practice in Iteration 3, especially for the final stages of optimization as well as the querying of Affects and Affects* clauses.

7.3.2 Early Start of System Tests Implementation

As mentioned in section 7.2.2, the implementation of system tests was delayed in Iteration 1. At the start of Iteration 2, we assigned a team member to be in charge of system tests and began implementation. This helped us in discovering edge cases such as wildcards in if/while pattern queries. For example, in "if ifs; Select ifs pattern ifs(_, _, _)", statement numbers of conditionals without any variables should not be returned. Another edge case discovered

was having ‘BOOLEAN’ as synonyms. For example, in “stmt BOOLEAN; Select BOOLEAN”, all statement numbers should be returned instead of TRUE/FALSE.

As such, unlike in Iteration 1, there were no major refactors that had to be done to account for these edge cases. The team member handling QA notified everyone of these edge cases whenever they were discovered. Thus, everyone was able to account for these edge cases while implementing their components. For Iteration 3, we would continue with this practice of implementing system tests early.

7.4 Potential Improvements - Iteration 2

7.4.1 Delay of System Testing CI

System tests were not integrated into GitHub CI upon Iteration 1 completion. As a result, some refactoring work performed in Iteration 2 caused some bugs to surface towards the deadline for Demo 2 in Week 8. As such, we wrote a Python script to integrate the system tests into our GitHub CI.

Regression testing is important, and we should continuously update the CI to include the most updated batch of system tests. Right before the demo for Iteration 2, we updated the GitHub CI to include system tests. This way, we also ensure that any modifications made to the code in Iteration 3 will not create more bugs.

7.4.2 Neglect of Complete Basic SPA Requirements

We neglected the update of Uses and Modifies relationships for call statements in Iteration 2. This occurred because call statements were not completely implemented during Iteration 1 and we did not notice this discrepancy. It was only brought to our attention when additional system tests were written that coincidentally included call statement queries for Uses and Modifies. Although the fix was fairly straightforward, it is important to prevent this from happening in the future.

At the start of Iteration 3, our team will set aside time to go through the requirements for Advanced SPA again. We will create GitHub issues for requirement cases not implemented so that we can pay attention to them on our GitHub project board. Potential requirements that we may miss out on are Affects corner cases, and selection of tuples with attribute names.

7.5 Effective Practices - Iteration 3

7.5.1 Efficient Optimization Workflow

We adopted an efficient workflow to optimize SPA. As recommended by the teaching team, we first tried to identify bottlenecks in our system, before proceeding to implement the optimizations. To identify bottlenecks within the system, our logger was used to print out timings of each critical phase of query evaluation, such as retrieving information from the PKB, combination of results, etc. Group members using Visual Studio could also use its debugging tools to identify these bottlenecks. Afterwhich, we identified the cause of these bottlenecks and proceeded with the actual optimization.

For example, when conducting load tests, we realized that queries containing two connected clauses with at least one wild card were timing out. One such query was “Select BOOLEAN such that Next*(s1, _) and Affects*(300, s1)”. Upon further investigation, we found out that the equi-join of query result tables was taking a long time, because there were many duplicated entries in “Next*(s1, _)”. We proceeded to eliminate these duplicates before performing the equi-joins and reduced the time taken for such queries.

7.5.2 Investing Effort into CI Implementation

We made a significant effort in implementing GitHub CI to run regression tests, linting and to check for any build warnings. For example, system tests for iteration 2 were immediately mounted onto our CI right after its submission. These helped to eliminate many potential bugs when implementing new features. The automation of linting and checking of build warnings also maintained code quality and reduced build issues across different development OS platforms. Furthermore, manual labour in running system tests, checking and resolving code quality issues were reduced.

7.6 Potential Improvements - Iteration 3

7.6.1 Insufficient Unit Testing

Our unit tests were not extensive enough to cover all edge cases within SPA. These caused bugs to surface toward the end of iteration 3. This issue occurred mainly in the Query PreProcessor, when distinguishing between semantic and syntactic errors of synonyms. Edge cases we did not cover in our unit tests were duplicated synonyms and undeclared synonyms. As such, these bugs only surfaced while conducting exploratory tests and were fixed thereafter.

This lack of coverage in our unit tests occurred as a result of not having a full understanding of Select BOOLEAN clauses. Such clauses should return false when a semantic error is detected and nothing when a syntax error is detected. We neglected this edge case as we focused more on the code coverage metric when writing unit tests. It is thus important that all specified requirements of the system need to be considered when writing unit tests as well.

7.6.2 Not Writing Optimal Code

Code that was not optimal was written in the previous implementation. These optimizations were obvious and could have been implemented right at the start. As such, this incurred more effort in the optimization of SPA. There were more areas of the code to identify for optimization and more optimization code had to be written as well.

For example, in the evaluation of BOOLEAN PQL queries, cross products are never required between disconnected groups of nodes. This is because cross products never yield an empty result and we do not need the specific statement numbers, variable or procedure names to project on. We just need to simply check if the result is empty and output true or false. However, when implementing this BOOLEAN evaluation, cross product was included which caused stress tests to fail as we conducted optimization of SPA. More time was spent

determining where to optimize consequently. We could have kept optimization in mind as we were designing the strategy for BOOLEAN evaluations and ensure that our algorithm does not incur unnecessary overhead.

Appendix

8 Extension Proposal

This section proposes a new feature to be extended in the SPA.

8.1 Extension Definition

“if pattern” queries will be extended to search for statements that are in the ‘then’ or ‘else’ blocks of if statements.

The PQL ‘if’ grammar rule will be modified to:

```
if : syn-if '(' entRef ',' stmtRef ',' stmtRef ')'
```

Similar to the definition in advanced SPA, entRef would still refer to the control variable in the if statement. The stmtRef in the middle refers to statements that appear only as a direct child in the ‘then’ block of the if statement. The stmtRef on the right side refers to statements that appear only as a direct child in the ‘else’ block of the if statement. Specifically, both of these statement references, stmtRef, have to satisfy Parent(ifs, stmtRef). Examples of these queries, along with their evaluated results will be shown in the following:

Statement Number	SIMPLE Source Program
-	`procedure printMultipleOfSix {`
1	`  read x;`
2	`  if (x % 2 == 0) then {`
3	`    read y;`
4	`    while (y % 3 != 0) {`
5	`      read y; } }`
-	`  else {`
6	`    y = 6; }`
7	`  z = x * y;`
8	`  print z; }`

Figure 8.1: Sample Extension SIMPLE Program

With reference to the SIMPLE program displayed in Figure 8.1, the following queries will have the results:

stmt s; variable v; if ifs; read r;	
Query	Results
Select ifs pattern ifs (v, 3, 6)	2
Select BOOLEAN pattern ifs (_, 5, s)	FALSE
Select s pattern ifs (“x”, r, _)	1, 2, 3, 4, 5, 6, 7, 8
Select <s, ifs> pattern ifs (“x”, _, s) such that Modifies (s, “y”)	6 2

Figure 8.2: Sample Extension “if pattern” Queries

8.2 Changes Required

This subsection details the changes required for each component in SPA.

8.2.1 SP

No changes are required for the SP. The current implementation of SP already parses and inserts statements within if statements into the PKB, according to its occurrence in the ‘then’ or ‘else’ block.

8.2.2 PKB

The PKB Extractor Facade will need to extend its pre-existing Pattern API to take in four instead of two arguments. The four arguments should include (in order):

- if synonym,
- control variable synonym,
- then block statement reference,
- else block statement reference

A struct may be used to aggregate these arguments together. The following is an example of one such API:

PKB Extractor Facade API

QUERY_RESULT_TABLE Pattern(IF_NODE_ARGS args)
Returns a query result table for “if pattern” queries, based on the arguments listed in args.

The actual extraction algorithm will be elaborated on in section 8.3.

8.2.3 QPS

The Query PreProcessor will need to extend its parsing to take in four arguments for “if pattern” clauses. Specifically, its ParsePattern method needs to be extended to parse these 4 arguments.

The EntTable component also needs to be updated to allow construction and validation of four arguments. It should only allow synonyms and statement numbers as input in the second and third argument. String identifiers or entity references are syntactically incorrect and an error will be thrown if they are detected.

The Query Optimizer will also need to update its optimization algorithm.“if pattern”s will now take a longer time for queries, as four arguments need to be processed. Hence, the optimization algorithm can be tweaked to deprioritize “if pattern” clauses toward the end of query evaluations.

Finally, the “if pattern” Query Node will need to be extended as well to take in four arguments for “if pattern” clauses. The relevant APIs provided by the PKB Extractor Facade would then need to be integrated. No change is required for the Query Evaluator.

8.3 Implementation Details

The PKB Extractor Facade would need to search for if statements that satisfy the arguments in the “if pattern” clause. The algorithm would first fetch all if statements from the Statement Table. It would then iterate through these if statements and perform the checks shown in the following activity diagram:

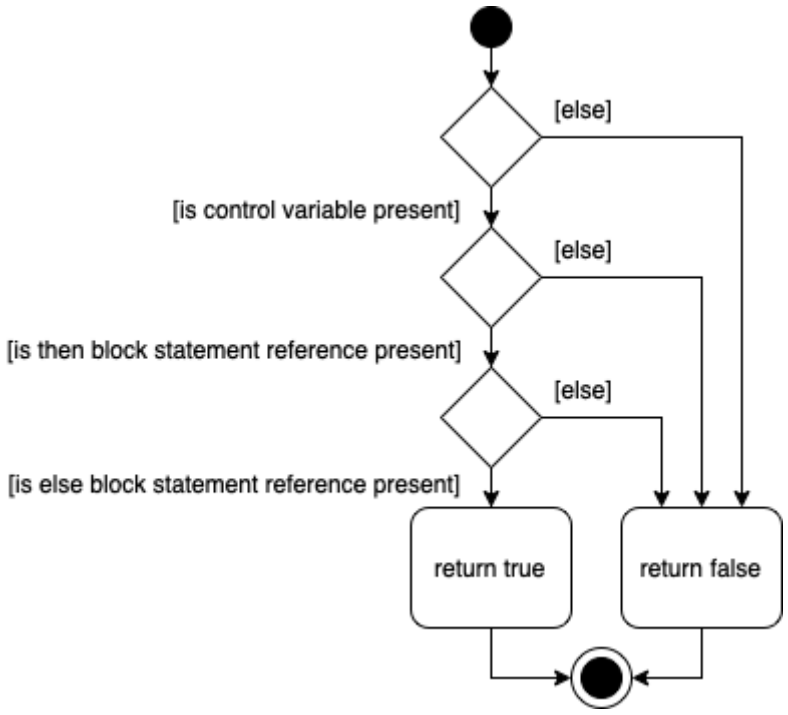

Figure 8.3: Extended “if pattern” Clause Check

The if statement number will only be added to the results if the workflow in the activity diagram returns true.

Additionally, to optimize the search for statement reference, note that the then and else block statement numbers are stored as vectors in sorted ascending order. Hence, if the statement reference were a statement number, binary search can be used. If it were a statement synonym, the list of statement numbers have to be iterated through.

8.4 Possible Challenges

There are no foreseeable implementation difficulties as most of the information for “if pattern” clauses are already stored in the PKB. Only logic for parsing, validation and extraction of the “if pattern” clauses is required. However, the coordination of the implementation needs to be managed. The new PKB Extractor Facade API should first be implemented, as it does not depend on any other components. Then, the “if pattern” Query Node will need to be updated, followed by the Query PreProcessor. The Query Optimizer may be updated last, as the order of evaluation of nodes only affects the speed of queries, not its correctness.

With regards to testing, current unit tests and integration tests will be augmented. More test cases pertaining to this “if pattern” extension must be added as well. Using pairwise testing, the following combinations need to be implemented:

entRef arg	**stmtRef arg**	**stmtRef arg**
variable reference	stmt statement reference	stmt statement reference
raw variable reference	if statement reference	if statement reference
-	statement number	statement number

Figure 8.4: Extended “if pattern” test cases

A total of at least 2*3*3 = 18 additional test cases will thus need to be implemented. Note that no modifications to pre-existing “if pattern” clauses are required. This is because wildcards are still syntactically and semantically valid in this extension.

Additionally, more test cases need to be added to the multi-clause system test suite. This is because the “if pattern” clause may take in multiple synonyms, which may affect the joining of results with other clauses. To ensure that all edge cases are covered, system tests should be written first, before the actual implementation of the functional code.

8.5 Benefits to SPA

With this extension, users of SPA would be able to search for if statements or statements that are directly nested within if statements. This is beneficial because if-else branches are two separate execution paths that may hold different logic. With reference to the program in

Figure 8.1 for example, suppose that variable “y” is erroneously assigned a value other than 6 in statement 6, which causes the program to print out numbers that are not multiples of 6. The user can just enter this PQL query to search for the cause of error:

```
stmt s; if ifs;

Select s pattern ifs (_, _, s) such that Modifies(s, “y”)
```

The user would then be able to view all statements that modify variable “y” in the else block. Without this pattern clause, the user could still query for a Parent relation, but he would have to search through these results to determine if it belongs in the then or else block.

Furthermore, by limiting these queries to only the direct child statement numbers, the user would not need to search for nested statements in the results. However, the search for nested statements can still be achieved. Consider the program in Figure 8.1. To retrieve the statement number 5, this PQL query can be used:

```
stmt s1, s2; if ifs;

Select s2 pattern ifs (_, s1, _) such that Parent*(s1, s2)
```

End of Report

This page is intentionally left blank and marks the last page of the report.