NUS
National University of Singapore

School of Computing

CS3203 Software Engineering Project

AY23/24 Semester 2

Project Report – System Overview

Team 11

Team Members	Student No.	Email
AMADEUS ARISTO WINARTO	A0221733N	e0559265@u.nus.edu
BERNARDUS KRISHNA	A0196717N	e0389203@u.nus.edu
CHAN CHOON YONG	A0222743L	e0560275@u.nus.edu
CHEN HSIAO TING	A0222182R	e0559714@u.nus.edu
NG JUN WEI, TIMOTHY	A0217635A	e0543671@u.nus.edu
SIMON JULIAN LAUW	A0196678A	e0389164@u.nus.edu

Consultation Hours: Tuesday, 1 PM - 2 PM

Tutor: Sumanth

Table of Content

1. System Architecture

The Simple Program Analyzer (SPA) consists of 3 main components - Source Processor (SP), Program Knowledge Base (PKB), and Query Processing Subsystem (QPS). The following Fig 1 shows the high-level architecture diagram. The dashed lines indicate the interactions between the major components, including external users.

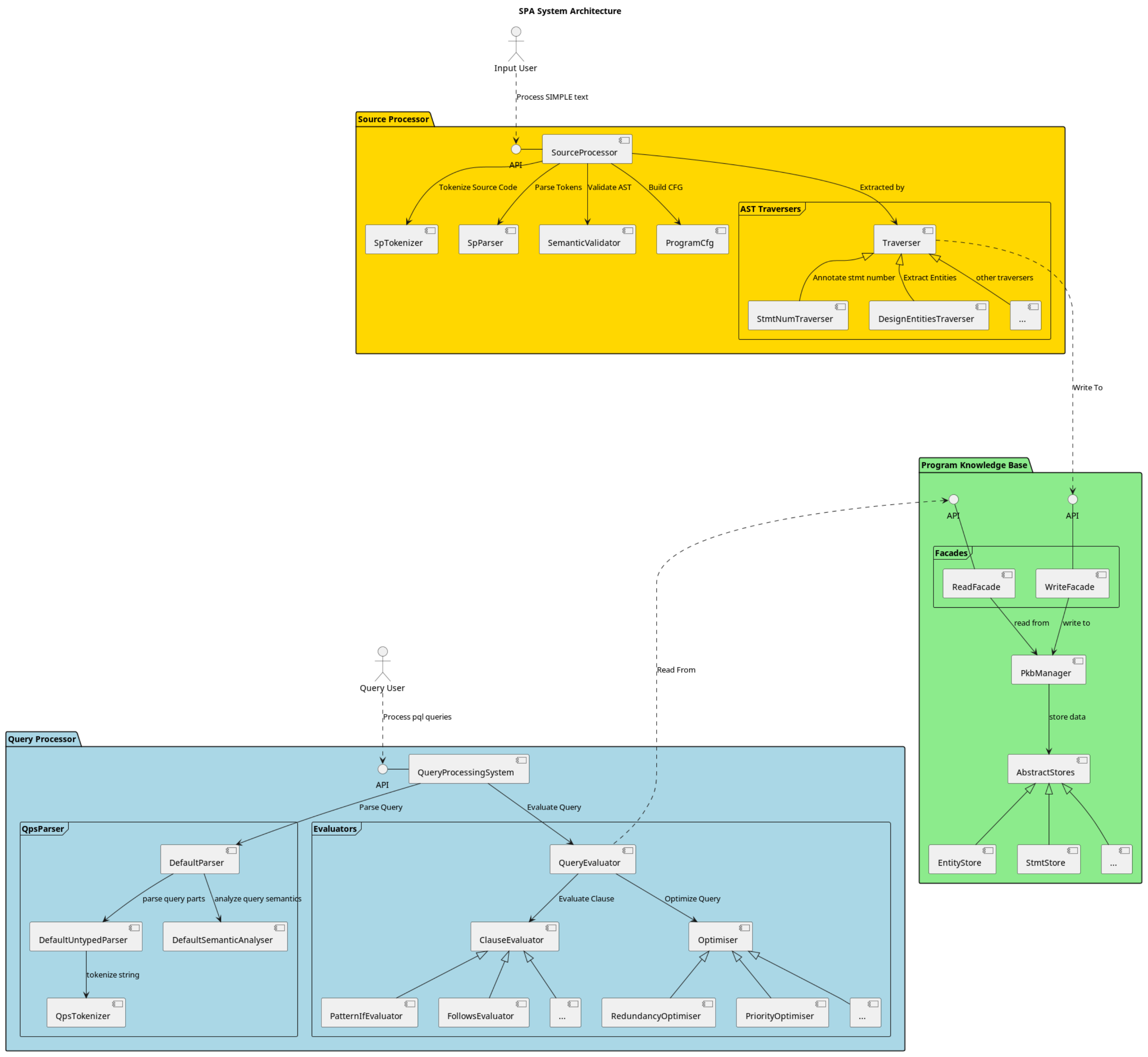


Fig 1. System Architecture Diagram of SPA

Moreover, higher-resolution diagrams can be found at https://nus-cs3203.github.io/documentation/.

2. Component Architecture and Design

2.1 SP

2.1.1. Class Diagram of SP

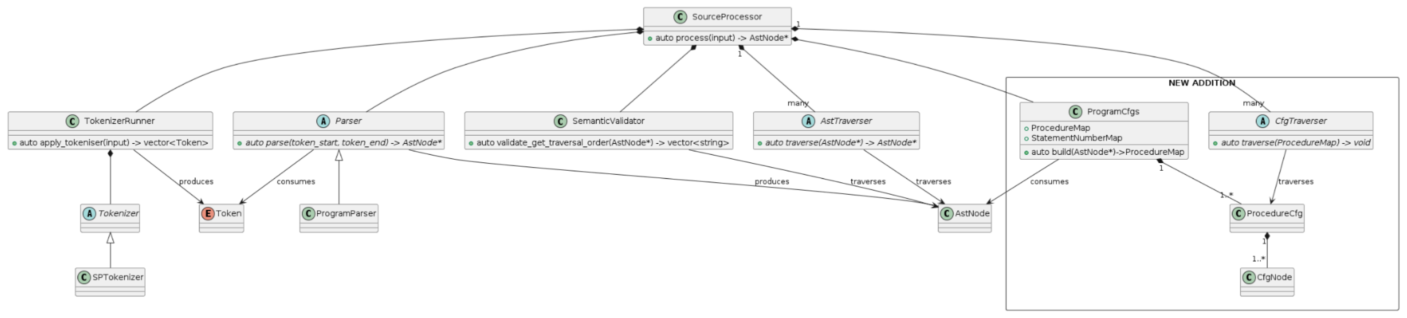

SourceProcessor
auto process(input) -> AstNode*
TokenizerRunner
auto apply_tokeniser(input) -> vector<Token>
Parser
auto parse(token_start, token_end) -> AstNode*
SemanticValidator
auto validate_get_traversal_order(AstNode*) -> vector<string>
AstTraverser
auto traverse(AstNode*) -> AstNode*
NEW ADDITION
ProgramCfgs
ProcedureMap
StatementNumberMap
auto build(AstNode*)->ProcedureMap
CfgTraverser
auto traverse(ProcedureMap) -> void
Tokenizer
Token
ProgramParser
AstNode
ProcedureCfg
CfgNode
SPTokenizer
produces
consumes
produces
traverses
traverses
consumes
traverses
1
many
1
many
1
1..*
1
1..*

Fig 2. High-Level Class Diagram for SP.

Three CFG classes caters to each abstraction level:

1. `CfgNode` encapsulates functionally similar & consecutive Statement Number(s) of a Procedure.
2. `ProcedureCfg` encapsulates the CFG of a Procedure. `ProcedureCfg` includes `CfgNodes`.
3. `ProgramCfg` encapsulates the CFGs of all Procedure in the SIMPLE program. `ProgramCfg` includes `ProcedureCfgs`.

This promotes Maintainability by categorizing relevant CFG functions according to the appropriate abstraction levels.

Moreover, while `AstTraverser` traverses the AST to extract Design Abstractions, `CfgTraverser` traverses the `ProgramCfg` to extract Design Abstractions.

2.1.2. Sequence Diagram for CFG Traversal

Fig 3. shows an example of the data flow when the SP traverses the AST to build CFG for each procedure in the SIMPLE program.

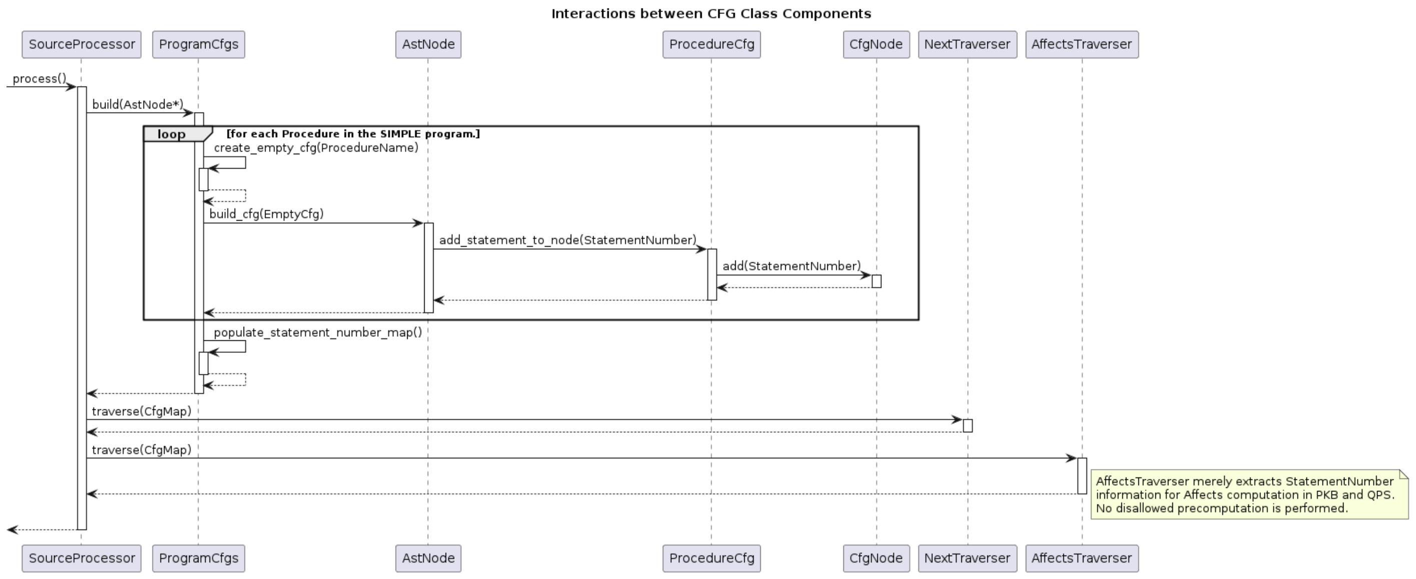

Interactions between CFG Class Components
SourceProcessor
ProgramCfgs
AstNode
ProcedureCfg
CfgNode
NextTraverser
AffectsTraverser
process()
build(AstNode*)
loop
[for each Procedure in the SIMPLE program.]
create_empty_cfg(ProcedureName)
build_cfg(EmptyCfg)
add_statement_to_node(StatementNumber)
add(StatementNumber)
populate_statement_number_map()
traverse(CfgMap)
traverse(CfgMap)
AffectsTraverser merely extracts StatementNumber information for Affects computation in PKB and QPS. No disallowed precomputation is performed.

Fig 3. Sequence Diagram for building CFGs using AST, and traversing CFGs for Design Abstractions.

Using the Strategy Pattern, the different algorithms to build the CFG are parked under each respective `AstNode`. For example, a `While AstNode` and a `Read AstNode` will build the CFG differently. This promotes Extensibility as any new `AstNode` class can be easily implemented.

2.1.3. Design Decisions for CFG Traversal

Decision 1: Implementation of the `CfgNode`'s Edge Representation

Alternatives	Ease of implementation	Extensibility (To extract new CFG-related information)	Time Complexity
Store OutNeighbours of a `CfgNode` in an Adjacency List	**High**. Adjacency List (using C++ Unordered Map) is a rudimentary data structure. The Adjacency List maps `CfgNode` to its OutNeighbours.	**Medium**. Extended features are parked in `ProgramCfg`, leaving `CfgNode` untouched (Open-Closed Principle). To facilitate CFG Traversal, an auxiliary Statement Number Map could be derived from the Adjacency List using C++ STL operations.	Retrieve any `CfgNode's` OutNeighbours [for `NextTraversal`]: **O(1)** with an auxiliary Statement Number Map. Retrieve all Statement Numbers in a CFG [for `AffectsTraversal`]: **O(N)**, N is the number of `CfgNodes`. Moreover, optimised C++ STL operations can be employed.
Store OutNeighbours of a `CfgNode` in the `CfgNode` itself (Directed Graph)	**Medium**. `CfgNode` has OutNeighbours as a class attribute. This employs Object Oriented Programming concepts.	**Medium**. New features are compartmentalized within `CfgNode`, limiting the impact of change (Single Responsibility Principle).	Retrieve any `CfgNode's` OutNeighbours [for `NextTraversal`]: **O(N+E)** using DFS/BFS search algorithm. N is the number of `CfgNodes` and E is the number of OutNeighbours. Retrieve all Statement Numbers in a CFG [for `AffectsTraversal`]: **O(N+E)** using DFS/BFS search algorithm.

Decision and Justification

We chose the first option. Extensibility and Maintainability are the main priority to accommodate feature and optimization modifications in Milestones 2 and 3. Since the Adjacency List design stores the OutNeighbours in the `ProcedureCfg`, while the Directed Graph design stores the OutNeighbours in the `CfgNode`, both options are similarly extensible to new changes, where the changes will be reflected in the `ProcedureCfg` and the `CfgNode` respectively.

However, the Adjacency List design enables greater optimization. An Adjacency List allows us to use optimized C++ STL operations and to create auxiliary data structures to pre-process frequent CFG traversal queries. For example, an auxiliary Statement Number Map that takes O(N) space eliminates the need to run costly Graph Searches for each CFG traversal. Since program efficiency is an important criterion, we chose the Adjacency List design which is more extensible to greater optimization modifications.

Decision 2: How should we organize the new CFG Traverser algorithms?

Alternatives	Extensibility	Maintainability	Ease of Implementation	Adherence to DIP
Single CFGTraverser Class with giant if-else or switch-case conditionals	**Low.** Extensions must be written with care to ensure accurate code branching.	**Low.** Tight coupling between algorithms in one class.	**High.** Minimizes the complexity of inheritance and polymorphism.	**Low.** The SP runner uses this large class with different algorithms.
Strategy Pattern to vary CFG Traverser algorithms to run	**High.** A new algorithm inherits from a common interface, with common methods.	**Moderate.** Easy to independently modify algorithms, while respecting the inheritance structure.	**Moderate.** Careful design of inheritance structure to be planned.	**High.** Each algorithm is an interface and the SP runner uses the appropriate interface.

Decision and Justification

We chose the Strategy Pattern design where the main SP runner takes in various CFG Traversers through independent `CfgTraverser` classes (e.g. `NextTraverser`, `AffectsTraverser`). Extensibility has the highest priority in our SPA, because SP should be flexible to support new Design Abstractions and Optimisation Modifications.

Decision 3: How should we organize CFG-related logic?

Alternatives	Maintainability	Ease of Implementation	Adherence to SRP
Single CFG class that incorporates all logic to construct the CFG.	**Low.** Tight coupling between logic to build CFG across different abstraction levels.	**High.** Minimizes the complexity of inheritance and polymorphism.	**Low.** Distinct logic to build CFG across different abstraction levels is accumulated in a single large class.
Few CFG classes segment the CFG logic by abstraction levels.	**Moderate.** Easy to independently modify logic to build CFG at each abstraction level, while respecting the inheritance structure.	**Moderate.** Careful design of inheritance structure to be planned.	**High.** Logic to build CFG at the Statement-level, Procedure-level, Program-level are separated, reducing coupling.

Decision and Justification

We chose to segment the CFG logic by abstraction levels. `CfgNode` builds the CFG while parsing Statements, `ProcedureCfg` builds the CFG while parsing Procedures and `ProgramCfgs` consolidates all CFGs across Procedures. This design encourages the Separation of Concerns, increasing Maintainability, and Ease of Optimization at each abstraction level.

2.1.4. Design Principles and Patterns

On top of the Design Principles and Patterns mentioned in the Milestone 1 report, the following section highlights the Design Principles and Patterns we employed in Milestones 2 and 3.

Design Principles:

- Single Responsibility Principle (SRP)

`AstTraversers` and `CfgTraversers` have been distinctly separated because both Traversers are responsible for traversing through different data structures.

- Open Closed Principle (OCP)

The OutNeighbours of a `CfgNode` is stored outside `CfgNode`, requiring no modification of `CfgNode`. Optimisation modifications can be made without affecting the underlying implementation of `CfgNode.`

- Don't Repeat Yourself (DRY)

Building the CFG requires parsing each `AstNode` into the corresponding `CfgNode`. Instead of implementing numerous classes to contain the algorithm to parse each `AstNode`, each `AstNode` is extended with the corresponding algorithm. This decision minimizes code repetition.

Design Patterns:

- Strategy Pattern

Similar to SP's Parser and Tokeniser, we encapsulate the algorithm to parse each `AstNode` into the corresponding `CfgNode` within the `AstNode` itself. This enhances extensibility because new `AstNode` can be easily implemented, and maintainability because the Strategy Pattern allows the code to build the CFG to be succinct by generalizing a family of algorithms.

2.2 PKB

2.2.1. Class Diagram of PKB

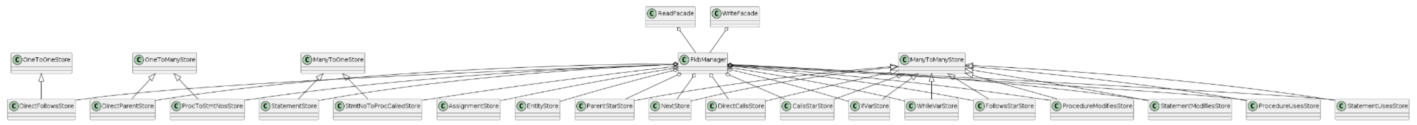


Fig 4. High-Level Class Diagram for PKB (For a higher-resolution version, click here)

2.2.2. Sequence Diagram for Data Storage (e.g. Parent)

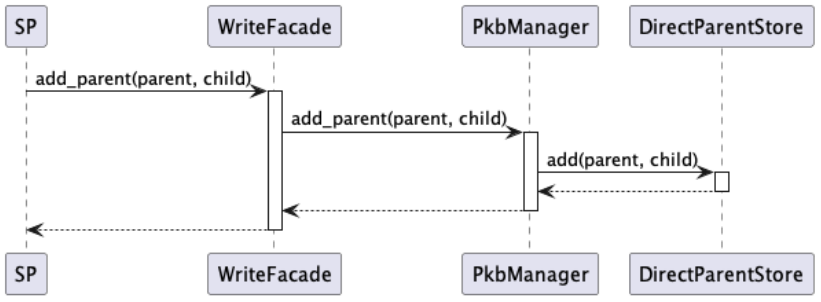


Fig 5. Sequence Diagram for adding Parent Relationship via the PKB WriteFacade

To populate the transitive relationship in Follows and Parent relationships (Follows* and Parent*), we implemented the `finalise_pkb()` method call, which SP can call after populating all the required Follows and Parent relationships. Refer to Appendix A.1 for the sequence diagram for populating the transitive relationship.

2.2.3. Sequence Diagram for Data Retrieval (e.g. Next)

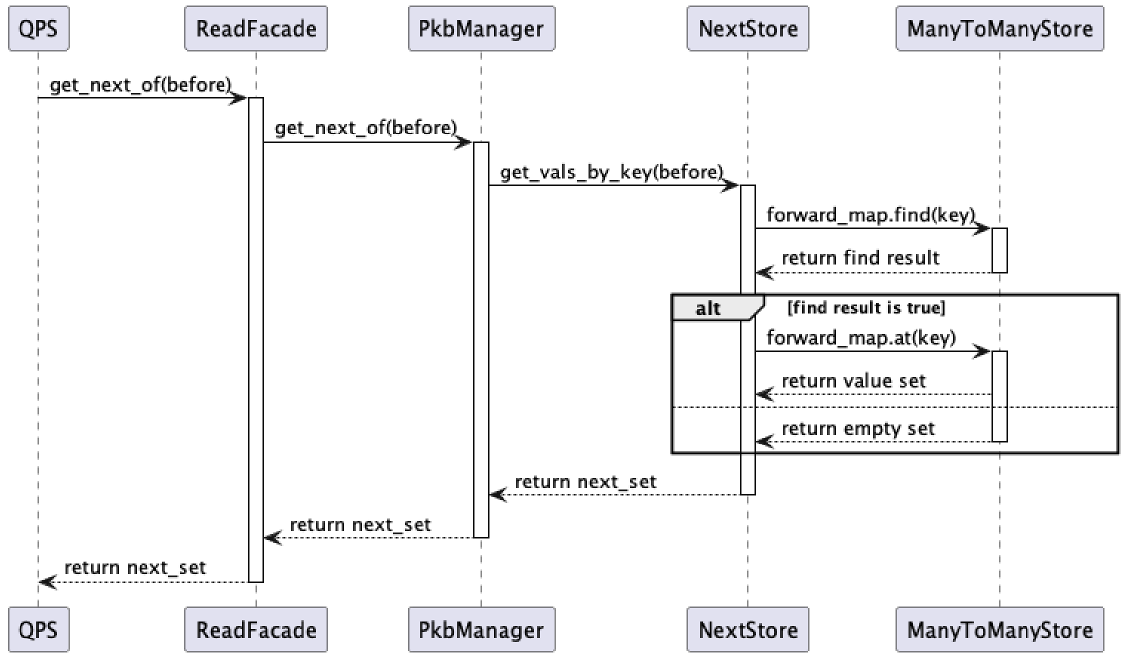


Fig 6. Sequence Diagram for retrieving Next relationship via the PKB ReadFacade

2.2.4. Design Decisions

Decision 1: Facade Classes or Direct Interactions

Alternatives	Extensibility	Maintainability	Ease of Implementation	Adherence to SRP
Exposing entire PKB and all its stores methods	**High**. Direct access allows for quick modifications and additions	**Low**. Complexity and interdependencies between stores can lead to a fragile system	**High**. Easy to implement as only one change is needed in the respective store	**Low**. Every store is responsible for both updating PKB and interacting with SP or QPS
Exposing PKB APIs only through WriteFacade and ReadFacade	**Moderate**. Any new requirements would require changes in both the stores and the Facade classes	**High**. Facades hide the underlying complexity of PKB and minimise unintended access to the stores, reducing unintended side effects	**Moderate**. Changes in the API of the stores would also require changes in Facade classes	**High**. Adhere to SRP as WriteFacade and ReadFacade are only responsible for writing and reading respectively

Decision and Justification

We chose to implement Facade classes, option 2. Instead of having each store directly interact with SP and QPS, we have the `WriteFacade` class which is solely responsible for all interactions between PKB and SP, and the `ReadFacade` class which is solely responsible for all interactions between PKB and QPS. This protects PKB from being exposed to external classes and prevents unauthorized or unintended modifications to the PKB's internal structures.

Decision 2: Individual Stores or Abstract Stores

There were three alternatives considered:

- Implement individual store separately (e.g. `FollowsStore` , `ParentStore`, `UsesStore`, etc. do not share any parent class, and their class members do not share any parent class as well)
- Implement abstract stores and use them for encapsulation (e.g. `UsesStore` has two `ManyToMany` stores, one for `ProcedureUses` and one for `StatementUses` relation)
- Implement abstract stores and use them for inheritance (e.g. both `ProcedureUsesStore` and `StatementUsesStore` inherit from `ManyToManyStore`)

The following lists our considerations when deciding which alternative to choose.

Alternatives	Extensibility	Maintainability	Ease of Implementation	Adherence to DRY
1) Implement individual store separately	**Low**. Every new requirement will need similar changes across all stores that have the same behaviors	**Low**. Every change in requirement will need changes across all stores that have the same behaviors	**Low**. Many repetitions of codes for all similar stores	**Low**. Every relationship store will share very similar codes.
2) Implement abstract stores and use them for encapsulation	**Moderate**. When extending, changes are needed in the respective stores	**Moderate**. There is a possibility of the need to modify the encapsulating class on top of the abstract stores.	**Moderate**. Need to create all store classes with methods that call the abstract stores APIs, a lot of repeated code	**Moderate**. The encapsulating classes will still have similar codes due to similar functionalities.
3) Implement abstract stores and use them for inheritance	**High**. When extending, only the abstract stores need to be changed	**High**. No specific relationship stores need to be maintained, only abstract stores	**Moderate**. High-level abstraction initially. However, very easy to implement later	**High**. Each relationship store does not share a common code as everything is inherited from the abstract stores.

Decision and Justification

We decided to use option 3:

- All relationship store classes are implemented from an abstract store directly. These relationship store classes do not have to define their own APIs, they can use the abstract store APIs directly.
- When extending or modifying, only the abstract store classes need to be changed. This improves extensibility, and maintainability and reduces lots of code repetition.

Decision 3: Calculating transitive relationship (Parent* and Follows*) after every addition or at the very end

Alternatives	Time Efficiency	Ease of Implementation
Calculate respective transitive relationship after every addition	**Low**. Calculate transitive relationship after every add operation can be costly	**Moderate**. There is extra logic needed depending on the order of insertion of the corresponding direct relation

Calculate all transitive relationships only once after all PKB write operations	**High**. Calculate transitive relationship only once, after all write operations	**High**. Only one API needed to populate all transitive relationships. Order of insertion does not matter as we have a guarantee that every direct relation has been inserted

Decision and Justification

We chose option 2:

- Only one calculation is required at the end of all write operations.
- The order of insertion of the corresponding direct relation does not matter. Calculation of transitive relationships after every add operation is removed, significantly improving time efficiency.

2.2.5. Design Principles and Patterns

Design Principles

- Single Responsibility Principle (SRP)

  Each class is responsible for a single role. E.g. next_store is responsible for storing the next relationships, and affects_evaluator is responsible for calculating affect relationships on-the-fly.

- Open-closed Principle (OCP)

  Each store inherits from one of the abstract store superclasses, e.g. next_store implements `ManyToManyStore`, which allows for the extension of new requirements on top of the superclass without modifying it.

- Interface Segregation Principle (ISP)

  Separate Facade classes for both reading and writing. This means that SP does not have to depend on unnecessary interfaces for reading, and QPS does not have to depend on unnecessary interfaces for writing.

- Don't repeat yourself (DRY)

Inheritance of relationship stores from abstract store superclasses help to reduce repeated code significantly. This helps to ensure DRY since we now do not have to repeat the same code logic in every store that behaves similarly.

Design Patterns

- Facade Pattern

  The Facade Pattern for PKB simplifies the interface for SP and QPS, providing a unified and high-level access point to the various stores and relationships. It enhances modularity and reduces the complexity of interactions, making the system easier to use, understand, and maintain.

2.3 QPS

2.3.1. Class Diagram of QPS

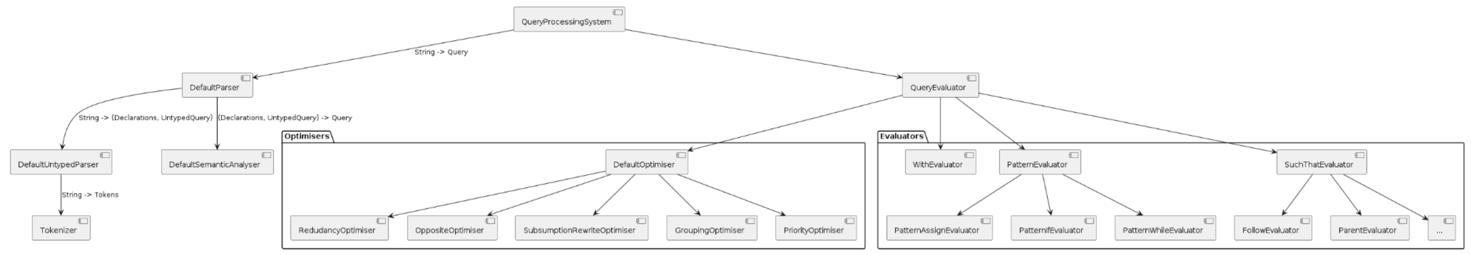


Fig 7: Architecture Diagram of QPS. (Refer to this for a higher-quality image)

2.3.2. Sequence Diagram for Next* Computation

Fig 8. shows an example of the data flow when retrieving the Next* relationship.

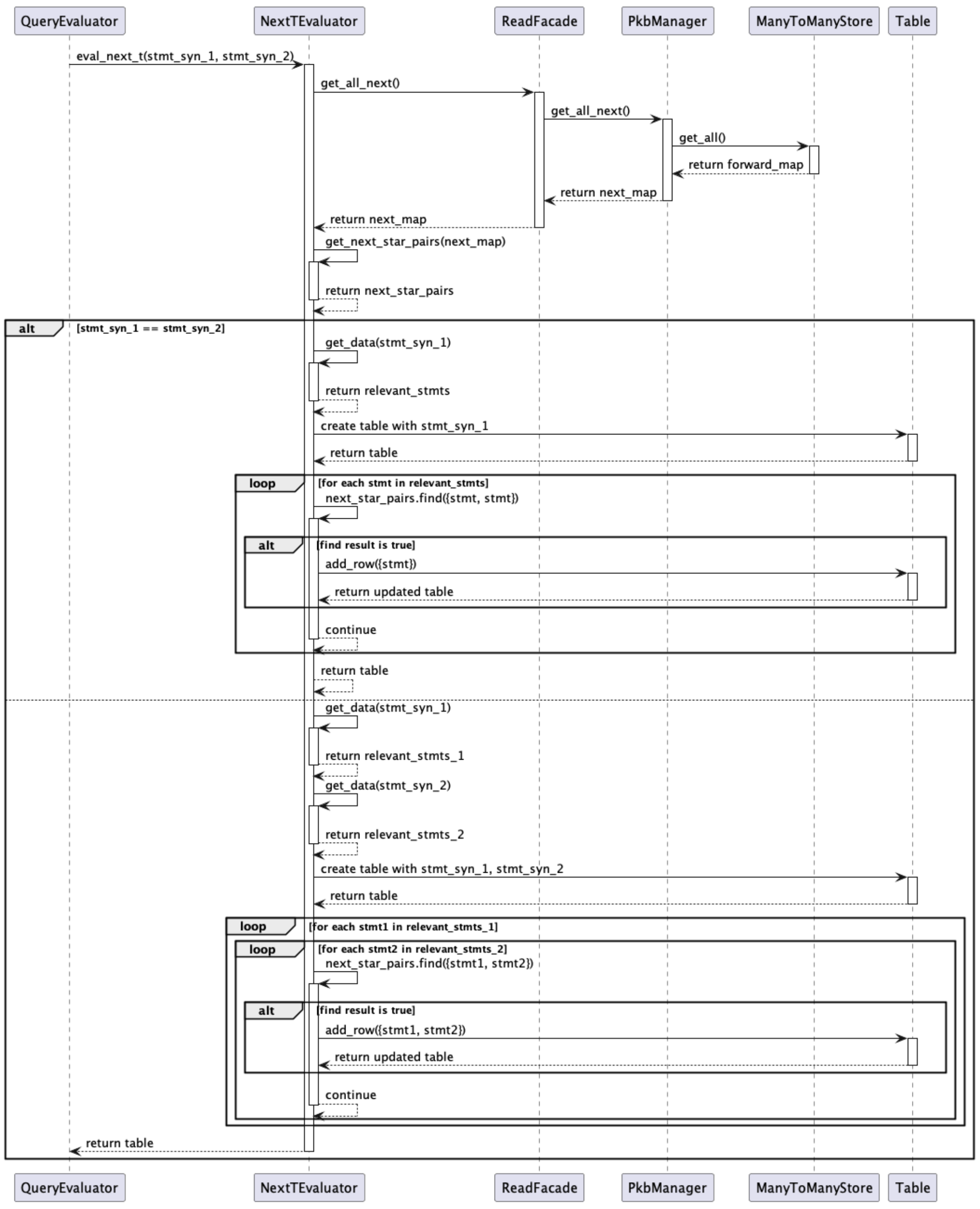


Fig 8. Sequence Diagram for calculating Next* Relationship (For a higher-resolution version, click here)

2.3.3. Sequence Diagram for Affects Computation

Fig 9. shows an example of the data flow when retrieving the Affects relationship.

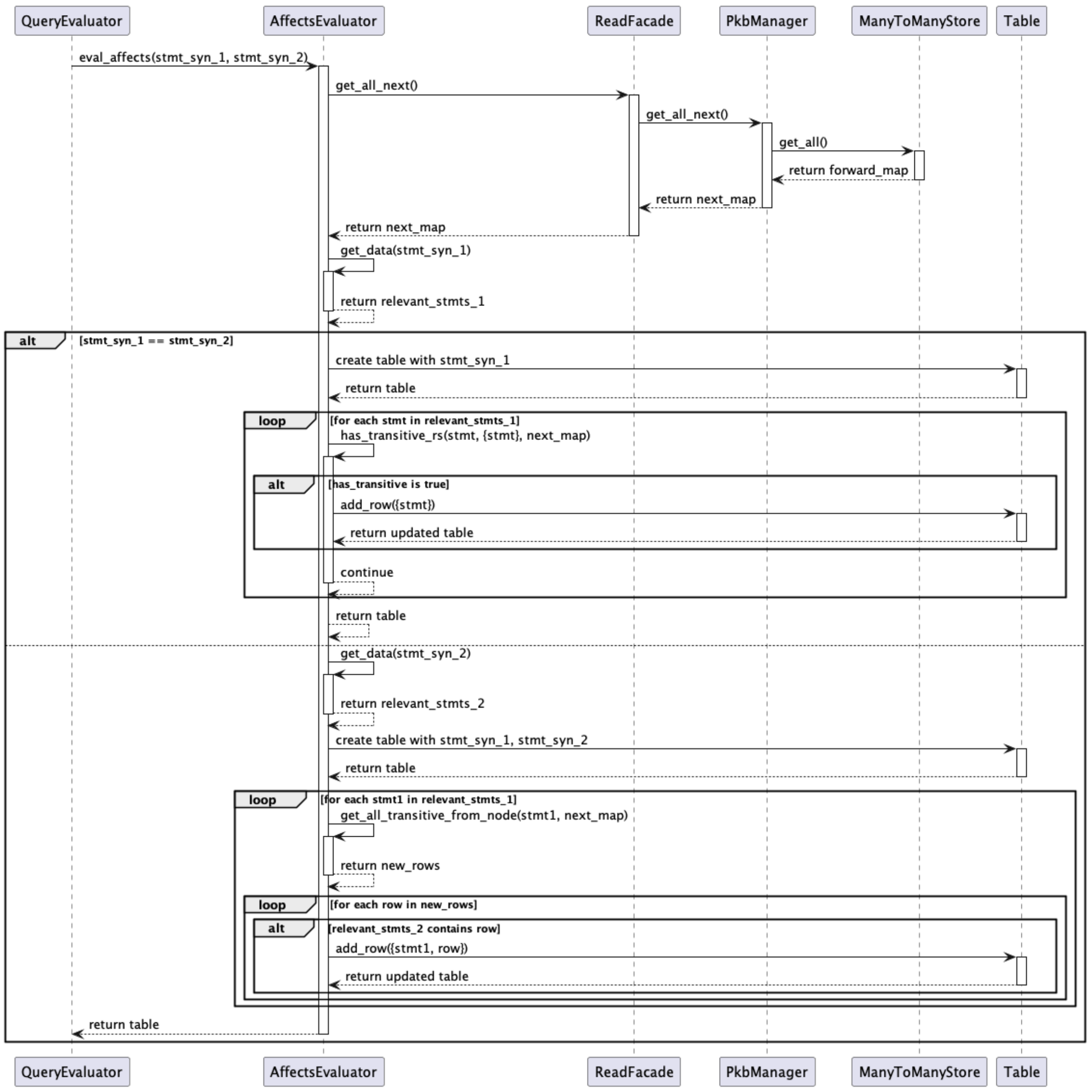


Fig 9. Sequence Diagram for calculating Affects relationship (For a higher-resolution version, click here)

2.3.4. Query Optimisations

An `Optimiser` takes a `Query` as input and returns a `std::vector<Query>` as output. The evaluator owns a `DefaultOptimiser` which owns 5 other optimisers, as shown in Fig 7.

- `RedundancyOptimiser`: remove clauses that are exactly the same
- `OppositeOptimiser`: early exit if there exists 2 clauses that are exact opposite of one another
- `SubsumptionRewriteOptimiser`: remove clauses based on the *subsumption rule,* to be defined below.
- `GroupingOptimiser`: group clauses based on *connectivity*, to be defined below.
- `PriorityOptimiser`: prioritise clauses using certain rules, to be defined below.

Subsumption Rule

Idea:

If the resulting table from the evaluating clause A is a subset of clause B's result table, then evaluating B is redundant.

Subsumption Rule for Relationships

- Relationship A *subsumes* relationship B if and only if type(A) subsumes type(B) and args(A) subsumes args(B)
- Relationship type P subsumes relationship type Q if and only if P == Q or P is the direct version of Q
- Examples:
    - `Follows` subsumes `FollowsT`
    - `Follows` subsumes `Follows`
    - `Follows` does not subsume `Parent`
    - `Follows` does not subsume `ParentT`
- Relationship argument X subsumes Relationship argument Y if and only if at least one of these are true:
    - X == Y; or
    - Y is a wildcard

Subsumption Rule for Clauses

Clause T *subsumes* Clause S if and only if at least one of these are satisfied:

- Case 1:
    - T is a positive clause
    - S is a positive clause
    - T has a relationship A
    - S has a relationship B
    - A subsumes B
- Case 2:
    - T is a negative clause
    - S is a negative clause
    - T has a relationship A
    - S has a relationship B
    - B subsumes A

If clause A subsumes clause B, then B can be discarded safely without needing to be evaluated.

Examples:

1. Follows(s1, s2) and Follows(s1, _):
    - s2 subsumes _ --> Follows(s1, s2) subsumes Follows(s1, _) --> Follows(s1, _) can be discarded
2. Follows(s1, s2) and Follows*(s1, s2)
    - Follows(s1, s2) subsumes Follows*(s1, s2) --> Follows*(s1, s2) can be discarded
3. Follows(s1, s2) and Not Follows(s1, s2):
    - The subsumption rule does not apply since one is positive and the other one is negative
4. NotFollows(s1, _) and NotFollows(s1, s2):
    - Follows(s1, s2) subsumes Follows(s1, _) --> NotFollows(s1, s2) subsumes NotFollows(s1, _) --> NotFollows(s1, _) can be discarded

The `SubsumptionRewriteOptimiser` thus removes clauses that are subsumed by another clause.

Connectivity and GroupingOptimiser

- Two clauses are *connected* if they share at least one synonym.
- The `GroupingOptimiser` returns queries whose clauses are connected within the same query but not connected between different queries.
- Connectivity is deduced via Depth-First Search (DFS).

- Queries are ordered according to the number of synonyms selected, reducing the size of the intermediate table.

Clauses within the same group are ordered such that the following property applies:

- If clause X is connected to some other clause Y,
  - then either X is the first clause in the ordering, or
  - there exists 1 clause ahead of X in the ordering such that the clause is connected to X.
- Thus, the following clause ordering is *impossible*:
  - Follows(1, s1), Follows(1, s2), Follows(s1, s2), since
  - Follows(1, s2) is connected to Follows(s1, s2) but it is neither the first in the ordering and there does not exist a clause in front of it that is connected to Follows(s1, s2).

Enforcing this minimizes the number of cross-products required when joining intermediate tables.

PriorityOptimiser

- The `PriorityOptimiser` prioritises positive clauses over negative clauses, as positive clauses are easier to evaluate than negative clauses.
- Future implementation of eagerly evaluating negative clauses is possible to mitigate large intermediate tables.

2.3.5. Design Decisions

Decision 1: Next*(stmt_syn_1, stmt_syn_2) Optimization

Algorithm A:

1. Get all possible statements from both synonyms
2. Run DFS to check whether there is a path

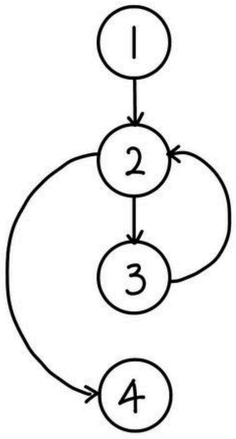
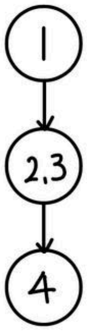
Algorithm B:

1. Group the statements into Strongly Connected Components (SCC).
2. For all nodes within an SCC, they can reach one another and thus we can populate the next* results with exhaustive permutations
3. For nodes between SCC, if there is a path from SCC 1 to SCC 2 in the DAG (Directed Acyclic Graph), Next*(a, b) where a is an element inside SCC 1 and b is an element inside SCC 2 is always true.

Alternatives	Ease of implementation	Time Efficiency	Space Complexity
Algorithm A	**High**. Easy to scan through data, check conditions, and add relevant data to the table	**Low, O($n^{2(V+E)}$)**, where V is the number of vertices (statements) and E is the number of edges (next relationships)	**Low, O(V^2)**. In the worst case, every vertex could be connected to every other vertex, leading to V^2 pairs
Algorithm B	**Low**. Requires complex algorithms such as Tarjan's Algorithm (for SCC), Topological Sorting (for DAG) and Permutations	**High, O(V^3)**, where V is the number of vertices (statements)	**Low, O(V^2)**. In the worst case, every vertex could be connected to every other vertex, leading to V^2 pairs

Refer to the table below for a simple example of option 2.

SIMPLE Source	Our optimisation Strategy

a = 0;
while (a < 5) {
   a = a + 1;
}
print a;

	Strongly Connected Components (SCC): {2, 3}
	
	Directed Acyclic Graph (DAG) formed from SCCs:
	Nodes: {1}, {2, 3}, {4}
	Edges: {1} → {2, 3} and {2, 3} → {4}
	
	Next Relationships:
	1 → 2
	2 → {3, 4}
	3 → 2
	
	Next* Relationships:
	3 → {2, 4}
	2 → {3, 4}
	1 → {2, 3, 4}

Decision and Justification

We chose option 2 as it has a superior time efficiency compared to option 1 despite having more complex implementations. Such significant improvement in time efficiency is highly valued especially because Next* relationships are computed on-the-fly.

Decision 2: Encapsulation of behaviors or Separation of behaviors for entities

Alternatives	Extensibility (Addition of types)	Extensibility (Addition of behaviors)
Encapsulation of behaviors via Inheritance	**High**. Adding new types is a matter of extending the base class, provided the type has similar behaviors.	**Low**. Inheritance Hierarchy likely requires modification when adding new behaviors or types that require new behaviors.
Separation of	**Low**. The Visitor interface needs to be	**Low**. A new Visitor interface likely needs to be

behaviors via Visitor Pattern	modified, as well as all concrete visitors.	implemented, and all Visitables must be modified to be able to accept the new Visitor
Separation of behaviors via `std::variant`	**Moderate**. Adding new types is a matter of creating a new class, and adding to the variant.	**High**. No Visitable nor Visitor interfaces need to be modified. The new concrete visitor can be created at the point of usage at any time.

Decision and Justification

We chose to separate behavior from data for relationships such as `Follows` and `Pattern`, using `std::variant` for easier addition of types whose behavior may differ from currently known types, e.g. a relationship with 3 members.

Decision 3: Evaluation of negated clauses within or outside individual clause evaluators

Alternatives	Ease of Implementation	Extensibility	Efficiency
Evaluate negated (not) clauses within individual clause evaluator objects.	**Low**. A new evaluation function will have to be written for each negated relationship for each possible parameter combination.	**Low**. Future clause evaluators will require twice the number of evaluation functions since both negated and non-negated have to be implemented.	**Low**. For each clause evaluation, either a subtraction of the non-negated terms from the full domain is necessary (requiring cross-product), or all possible results will have to be checked through and evaluated.
Evaluate negated clauses as non-negated clauses, subtract resultant tables when combining results	**High**. A table subtraction operation needs to be written. An additional boolean to indicate negation is added to clauses.	**High**. No additional changes need to be taken into account for any future clause types.	**Moderate**. Table subtraction may require some cross-product operations, in cases where the two tables involved share no common headers.

Decision and Justification

We chose the second design for its greater extensibility, efficiency and ease of implementation.

2.3.6. Design Principles and Patterns

Design principles

- Don't Repeat Yourself (DRY)
    - Much of the logic in Next* and Affect calculations are based on the existence of transitive relationships
    - We extract the similar logic into one common function has_transitive_rs.
    - For Affects computation, we added conditions checking (e.g. starting node needs to be an assignment, ending nodes need to be an assignment, use a variable that was modified in starting node, must be in the same procedure, etc).
    - DRY is adhered to since we do not have to repeat the same code logic in every similar calculation.

2.4 Cross-Component Design Considerations

2.4.1. Saving the CFG class by SP in PKB or using existing stores for Next* & Affects

Alternatives	Ease of implementation	Coupling	Space Complexity
Storing the whole CFG class by SP inside the PKB	Low. To make use of the CFG class requires knowledge of the class structure. To enable O(1) adjacent node computation, a map from the statement number to the node pointer is also needed.	High. Storing the whole CFG class inside PKB introduces coupling between the components. Any changes in CFG class internal representation might require code changes on the PKB side.	High. Storing the whole CFG would mean that there will be redundant storage as some of the information inside CFG is already found inside other stores.
Using existing stores (e.g. Next store) and add new stores for any needed additional information	High. Leverages existing relationships in PKB. Adding new information is easy as new stores creation is a one-liner and SP already has the information required when building their CFG.	Low. The two systems remain independent as the stores are maintained by PKB.	Low. Requires only the existing Next store and minor additional stores (e.g. procedure to statement numbers map). No redundant information is stored.

Decision and Justification

We chose option 2 as it is easy to implement, introduces no coupling between SP and PKB, and does not store redundant data inside the PKB.

3. Use of AI code generators

GitHub Copilot is leveraged heavily for automated system test case generation. A few sample queries are hand-crafted, and the GitHub Copilot will generate more permutations based on those samples. If the generated test cases are of low quality, developers' intervention such as writing more samples or supplying more context through the examples are needed.

```
// ai-gen start(copilot, 1, e)
// prompt: used copilot - Additional comments: generate far far more test cases similar to
the ones given
if ifs; Select ifs pattern ifs(_,_,_)
if ifs; while w;variable v; Select <w, v> pattern w(v,_)
if ifs; while w;variable v; Select <w, ifs> pattern w(v,_)
if ifs; while w;variable v; Select <ifs, w, v> pattern ifs(v,_,_)
if ifs; while w;variable v; Select <ifs.varName, w> pattern ifs(v,_,_)
if ifs; while w;variable v; Select <w, v.varName> pattern w(v,_)
if ifs; while w; Select ifs pattern ifs("b",_,_)
if ifs; while w; Select ifs pattern ifs("fizz",_,_)
if ifs; while w; Select w pattern w("x",_)
if ifs; while w; Select w pattern w(_,_)
// ai-gen end
```

Fig 10. Samples of System Test Case Generated by GitHub Copilot

4. Test Strategy

4.1. General Testing Strategy

Our team systematically tests the SPA system by adhering to the following testing workflow:

1. Test each component individually using extensive unit testing.
2. Test the interaction between 2 major components using integration testing.
3. Test the overall system using correctness system testing.
4. Test the performance of the entire SPA system using stress testing.

All the unit and integration tests are hand-crafted while most of the system testing, including stress testing, is generated with the help of AI. However, some of the test cases are still hand-crafted to catch edge cases that are often missed by AI.

4.1.1. Unit Testing

Each developer who created a component would also write the unit test cases for that specific component. We adopt a **white box** testing approach where we test and verify all paths of the code.

4.1.2. Integration Testing

A pair of developers from SP + PKB or QPS + PKB will develop tests for interaction between those two components, and finally, all the developers contribute to a single end-to-end test.

4.1.3. Correctness System Testing

The correctness of the system is checked through **black-box** testing. There are four main categories of correctness system testing:

- **Invalid Syntax and Semantics Tests** - Ensuring correct handling of invalid syntax and semantics in SIMPLE sources and queries.
- **Single Clause Tests** - Verifying the successful execution of single clause queries such as Next, Follow, etc. This covers parsing, storage, retrieval, and evaluation.
- **Multiple Clauses Tests** - Ensuring different combinations of multiple clauses queries are successfully and correctly working.
- **Edge Case Tests -** Verify the robustness and correctness of the system under extreme cases. For example, keywords such as Next and Call are used for synonym identity, exoteric characters such as vertical tab and carriage returns in the SIMPLE program, etc.

4.1.4. Performance System Testing

Complex sources and queries containing more than 300 lines and more than 4 computationally intensive clauses respectively, such as Next* and Affects are tested to ensure that complex queries can be executed correctly under the 5-second time constraint by default.

4.2. Test Statistics

The table below shows the breakdown of the test cases we implemented.

Test	Type	Number of Assertions / Test Cases
Unit Test	SP	339 assertions in 18 test cases
	PKB	713 assertions in 23 test cases
	QPS	1846 assertions in 66 test cases

	Common	143 assertions in 17 test cases
	Total	**3041 assertions in 124 Test Cases**
Integration Test	SP-PKB	200 assertions in 14 test cases
	PKB-QPS	58 assertions in 3 test cases
	SP-PKB-QPS	22 assertions in 1 test case
	Total	**280 assertions in 18 test cases**
Correctness System Test	Invalid Queries (Syntax/Semantic)	214 test cases
	Single Clause Queries	602 test cases
	Multi Clause Queries	43 test cases
	Edge Cases Queries	34 test cases
	Total	**893 test cases**
Performance System Test	**Total**	**94 test cases**

4.3. Automation Approach (Excluding AI-Generated Test Cases)

There are a few important automation for our testing workflows:

4.3.1. Python and Bash Scripts

Python and Bash scripts have been developed to run various system tests using autotester and summarize the results in CLI quickly.

```
$ python3 run_autotester.py Tests11/Milestone2/invalid_tests
[.../invalid_semantics_simple_recursive_source] Pass all system testing (1/1)
[.../invalid_semantics_simple_source] Pass all system testing (1/1)
[.../invalid_syntax_queries_source] Pass all system testing (52/52)
[.../invalid_boolean_queries_source] Pass all system testing (21/21)
[.../invalid_tuple_queries_source] Pass all system testing (56/56)
[.../invalid_pattern_queries_source] Pass all system testing (15/15)
[.../invalid_syntax_simple_source] Pass all system testing (1/1)
[.../invalid_semantics_queries_source] Pass all system testing (67/67)
Total passing test cases: 214/214
All tests passed!
```

Fig 11. Python Script Output of Autotester Wrapper Script

Furthermore, we also develop a Python script to generate system test cases automatically from a query file text template. Using this method, we just need to check whether the generated answers are correct.

```
while w; variable v; Select <w, v> pattern not w(v, _)
while w; variable v; Select w pattern not w(v, _)
while w; variable v; Select v pattern not w(v, _)
while w; Select w pattern not w(_, _)
```

Fig 12. Content of Raw Queries Text File

```
1 - generated TC 1
while w; variable v; Select <w, v> pattern not w(v, _)

6 z, 6 y, 6 x, 6 v
5000
2 - generated TC 2
while w; variable v; Select w pattern not w(v, _)

6
5000
3 - generated TC 3
while w; variable v; Select v pattern not w(v, _)

z, y, x, v
5000
4 - generated TC 4
while w; Select w pattern not w(_, _)


5000
```

Fig 13. Generated System Test Case

4.3.2. Continuous Integration (CI) using GitHub Actions

For every push and open Pull Request (PR), GitHub Actions will run a few workflows to automatically build, test, and verify code changes.

4.4. Bug Handling Approach

Bugs are first listed in the Notion. During stand-ups, we will try to identify the root problem of each bug, create GitHub issues, and assign them to the relevant developers. Upon fixing the bugs, more unit and system test cases should be added to cover those bugs, and regression testing should be performed afterward to ensure that the rest of the system is doing fine.

In case of emergency, all developers will be notified of the bug through Telegram, and discussions are quickly held to best solve the problem.

5. Reflection

CS3203 has been a challenging yet invaluable experience that provided hands-on practice with a full software engineering life cycle. This experience emphasized essential skills in the tech industry such as architecture design, implementation, testing, agile methodologies, and project management.

Initially, a major challenge for many of us was our limited familiarity with C++. Despite this, through consistent effort and support from more experienced team members, we were able to progressively enhance our proficiency.

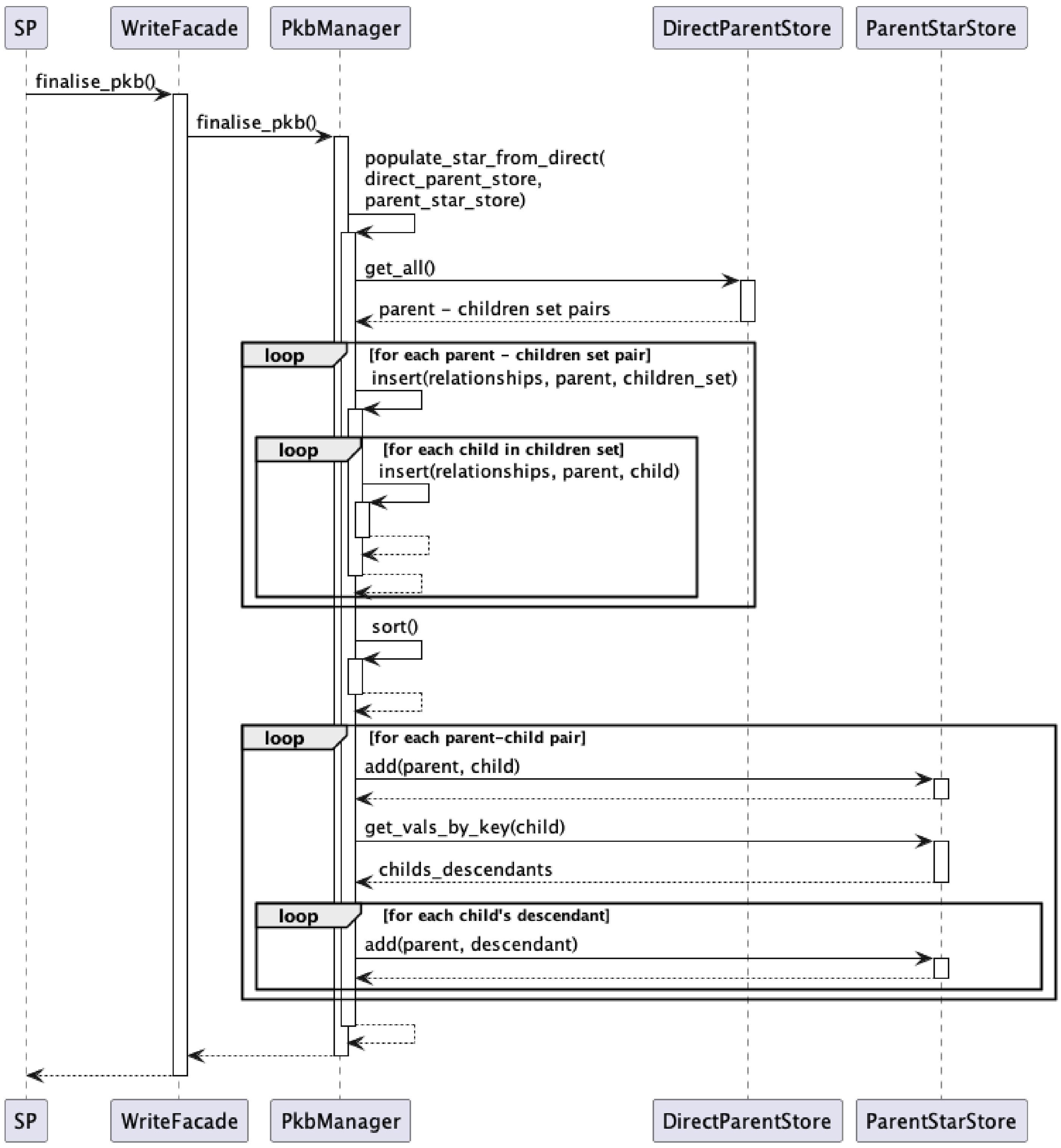
In terms of team communication, the initial lack of familiarity among team members limited our interactions, which led to challenges such as inconsistent knowledge and awareness of each other's progress and intentions. After identifying this issue during our first sprint review, we implemented strategies to improve communication. These included frequent interactions between members working on the same components to align designs, and regular updates in our group chat after major implementations. The improved communication facilitated smoother development and more effective responsibility delegation.

Regarding project management, we held two stand-ups per week, each lasting less than an hour, with meeting minutes recorded on Notion. Adhering to agile methodologies and utilizing practices like scrum, stand-ups, sprints, backlogs, and retrospectives allowed us to adapt quickly to changes, meet deadlines, and continuously improve. These practices are vital for any technical role in the future.

Overall, despite the heavy workload, we are pleased with the learning opportunities, new skills, and friendships gained through this journey. We believe that this blend of soft and hard skills will greatly benefit us as effective team players in our future careers.

Appendix A: Sequence Diagrams for PKB

1. Sequence Diagram for populating transitive relationship of the ParentStarStore


SP
WriteFacade
PkbManager
DirectParentStore
ParentStarStore
finalise_pkb()
finalise_pkb()
populate_star_from_direct(
direct_parent_store,
parent_star_store)
get_all()
parent – children set pairs
loop
[for each parent – children set pair]
insert(relationships, parent, children_set)
loop
[for each child in children set]
insert(relationships, parent, child)
sort()
loop
[for each parent-child pair]
add(parent, child)
get_vals_by_key(child)
childs_descendants
loop
[for each child's descendant]
add(parent, descendant)
SP
WriteFacade
PkbManager
DirectParentStore
ParentStarStore