SWE 261P - Software Testing and Debugging

Apache Cassandra CQL Statements Test Suite Enhancement

Instructor: James A. Jones

Table of Contents

SWE 261P - Software Testing and Debugging
Project: Apache Cassandra

Cheng-Yi Tang
Master of Software Engineering
University of California, Irvine
chengyit@uci.edu

Jenny Chen
Master of Software Engineering
University of California, Irvine
yingcc7@uci.edu

Cherine Cho
Master of Software Engineering
University of California, Irvine
cherinec@uci.edu

I. INTRODUCTION

Apache Cassandra (C*) is an open-source, highly-scalable, partitioned NoSQL database system designed for distributed data management. It was initially developed at Facebook by Avinash Lakshman and his team in 2008, this innovative database solution has evolved into one of the most powerful distributed database systems available today. Cassandra excels in scenarios requiring high write throughput and where data needs to be distributed across multiple data centers. Its architecture combines Amazon's Dynamo distributed storage and replication techniques and Google's BigTable data and storage engine model. Companies like Apple, Netflix, and Instagram use Cassandra in production for handling massive amounts of data while maintaining high availability and performance.

II. BUILD, ANALYSIS, AND PARTITIONING TESTS

Our goal in the first stage of the project (cassandra-software-testing) is to conduct a comprehensive test analysis of Apache Cassandra, with a focus on implementing partition testing for its key features. We will analyze Cassandra's codebase, which consists of approximately 450,500 lines of code primarily written in Java, document the build and test execution processes, study its existing test frameworks and methodologies, and develop new test cases using systematic partition testing. For our partition testing implementation, we will focus on Cassandra's CQL3 query language test suite, which contains essential test cases for Cassandra's query language functionality. Through this analysis, we aim to demonstrate the effectiveness of partition testing in uncovering potential issues and ensuring the reliability of Cassandra's core functionality.

A. Build

Prerequisites: Java 11 or 17

1. Clone the forked Cassandra repository to local machine:

```
Unset
cd ~/Dev
git clone https://github.com/chengyitang/cassandra-software-testing.git
```

2. Pull OpenJDK image for code execution:
   OpenJDK Docker image is used to provide a consistent Java runtime environment.

Unset
```
docker pull openjdk
```

3. Mount local Cassandra repository to the **/home** directory in container :
   This allows direct access and modification of source code while working in the containerized environment.

Unset
```
cd cassandra-software-testing
docker run -it -v $(pwd):/home/cassandra-software-testing openjdk:11 /bin/bash

# Install dependencies (In container)
apt-get update
apt-get install -y python2.7 ant git
```

B. How to Run Tests?

Unset
```
cd home/cassandra-software-testing

# Test a test file (ex. ../cql3/statements/SelectStatementTest.java)
ant test -Dtest.name=org.apache.cassandra.cql3.statements.SelectStatementTest

# Test a certain method of a file
ant test \
-Dtest.name=org.apache.cassandra.cql3.SelectStatementTest -Dmethod=[methodname]
```

C. Existing Test Cases

Cassandra uses JUnit 4 as its main testing framework. Most testing documents are inside the **test/unit** directory.

(Directory Structure)

```
cassandra-software-testing/
|__test/
|   |__unit/                        # unit tests
|   |   |__org/apache/cassandra/
|   |       |__cql3/                # CQL-related tests
|   |       |__ …
|   |__distributed/                 # distributed tests
…
```

Tests under the **cql3/** directory primarily focus on testing the Cassandra Query Language (CQL) 3.0 functionality, including Syntax Testing, Query Operation Tests, Data Manipulation Tests, Schema Management Tests, Data Type Tests, and so on. These test suites are crucial as CQL3 serves as the primary interface for users to interact with Cassandra, ensuring the reliability and correctness of all query language features from basic CRUD operations to complex schema manipulations.

We focused on the cql3/statements/SelectStatementTest.java class for the reason that 'SELECT' statements directly related to data reading and partitioning, which is suitable for testing different types of partition key and partition boundary. The class has an existing unit test method *testNonsensicalBounds()*, verifying that queries with logically impossible bounds (i.e., nonsensical bounds like 'c > 10 AND c <= 10') correctly return empty result sets. It ensures that the query processor correctly handles logical inconsistencies in 'WHERE' clause conditions. Building upon this foundation, we can expand the test coverage by introducing additional test cases that explore various partition boundaries and edge cases, ensuring robust handling of different partition key scenarios.

D. Systematic Approach: Functional Testing and Partition Testing

Systematic functional testing and partition testing are essential for modern software development for several reasons. Functional testing ensures that software meets its requirements and behaves correctly, while partition testing provides an efficient method to test large input spaces by dividing them into equivalent classes. Rather than testing every possible input, we test representative values from each partition and their boundaries, ensuring comprehensive coverage while minimizing test cases. Together, these systematic approaches enable thorough yet efficient software testing.

E. New Partitioning Tests

testIntegerPartitionBoundaries(){

Partitions:

- Minimum integer value (-2147483648)
- Zero (0)

- Maximum integer value (2147483648)

Representative Values:

- Integer.MIN_VALUE: Tests the lower boundary of integer partition key
- 0: Tests zero case
- Integer.MAX_VALUE: Tests the upper boundary of integer partition key

}

testStringPartitionBoundaries(){

Partitions:

- Empty string ("")
- Single character string ("a")
- Upper-bound character string ("z")

Representative Values:

- "": Tests the minimum boundary case for text partition keys
- "a": Tests a small, valid string as a partition key
- "z": Tests an upper-bound character in ASCII order

}

testNullValuesHandling(){

Partitions:

- Column contains NULL value
- Column contains a non-matching value

Representative Values:

- NULL: Tests how Cassandra handles a NULL value in queries
- 'nonexistent' : Tests a query condition that does not match any stored value

}

testCompositePartitionKeyBounds(){

Partitions:

1. First partition key (k1) - Integer type:
    - Positive integers

        - Negative integers (not currently tested)
        - Zero (not currently tested)
    2. Second partition key (k2) - String type:
        - Single character strings (‘a’, ‘b’)
        - Empty strings (not currently tested)
        - Long strings (not currently tested)
    3. Partition key combinations:
        - Complete key specification (both k1 and k2)

Representative Values:

- k1=1: represents positive integer partition
- k2=’a’: represents simple single-character text values
- (1, ‘a’) and (1, ‘b’): represents different complete composite key combinations

}

testTimestampBoundaries(){

Partitions:

1. Time epochs:
    - Unix epoch start (1970-01-01 00:00:00)
    - Far future (9999-12-31 23:59:59)
    - Current time (not currently tested)
    - Pre-epoch times (not currently tested)
2. Time formats:
    - Standard format (YYYY-MM-DD HH:MM:SS)
    - Different timezone formats (not currently tested)
    - Invalid formats (not currently tested)

Representative Values:

- ‘1970-01-01 00:00:00’: represents Unix epoch start (timestamp lower bound)
- ‘9999-12-31 23:59:59’: represents far future timestamp (timestamp upper bound)

}

testUUIDPartitionKeyBehavior(){

Partitions:

- Randomly generated UUIDs
- Different UUID versions (v1, v4)

Representative Values:

- uuid(): Tests the behavior of a randomly generated UUID as a partition key
- uuid(): Ensures that multiple UUID entries can coexist as partition keys

}

testBooleanPartitionKeyBehavior(){

Partitions:

- true: as a partition key
- false: as a partition key

Representative Values:

- true: Tests the behavior when querying for a "true" partition key
- false: Tests the behavior when querying for a "false" partition key

}

testFloatingPointPartitionBoundaries(){

Partitions:

- Minimum floating-point value (-1.7976931348623157E308)
- Zero (0.0)
- Maximum floating-point value (1.7976931348623157E308)

Representative Values:

- -1.7976931348623157E308: Tests the lower boundary of a floating-point partition key
- 0.0: Tests zero case
- 1.7976931348623157E308: Tests the upper boundary of a floating-point partition key

}

III. FUNCTIONAL TESTING AND FINITE STATE

Finite Model

Finite models are mathematical representations of systems with a limited number of states, variables, or inputs. They are widely used in software testing to ensure the correctness of systems, particularly those with complex logic or multiple states. Finite models allow testers to simplify complex systems by abstracting them into a manageable structure with a fixed number of *states* and *transitions*. This abstraction makes it easier to identify edge cases, ensure comprehensive test coverage, and detect logical inconsistencies early in development. Another key advantage of finite models is their ability to enable exhaustive testing within a controlled scope. Since the number of states and transitions is finite, testers can systematically generate and execute test cases that cover all possible scenarios. Moreover, finite models are useful in identifying deadlocks, unreachable states, and unexpected behaviors within a system. By analyzing state transitions, testers can determine whether the system encounters undefined or unintended states, which may lead to failures or security vulnerabilities. Therefore, by using finite models, we can simplify complex systems, ensure systematic testing within a controlled scope and identify unreachable states to enhance software reliability.

In our implementation for Apache Cassandra's CQL functionality, we applied the *finite state machine* approach to model the query processing workflow. The FSM design encompasses key states such as QueryReceived, ParsingState, ValidationState, ExecutionState, and ResultState, with an ErrorState to handle exceptions. This model effectively captures the lifecycle of a CQL query from reception to result generation, allowing us to test various transition paths and error conditions systematically.

Functionality Described by Finite State Machine

1. QueryReceived State

- Initial state
- System receives a CQL query like SELECT * FROM users
- Transitions to ParsingState

2. ParsingState

- Validates CQL syntax
- Checks for:
    - Required keywords
    - Proper bracket matching
    - Correct syntax structure
- Example error: SELECT FROM users (missing columns)

3. ValidationState

- Checks logical correctness:
    - Table existence
    - Column validity
    - Permission checks
- Example error: querying non-existent table

4. ExecutionState

- Actual query execution
- Data retrieval or modification
- Possible errors:
    - Timeouts
    - Resource constraints

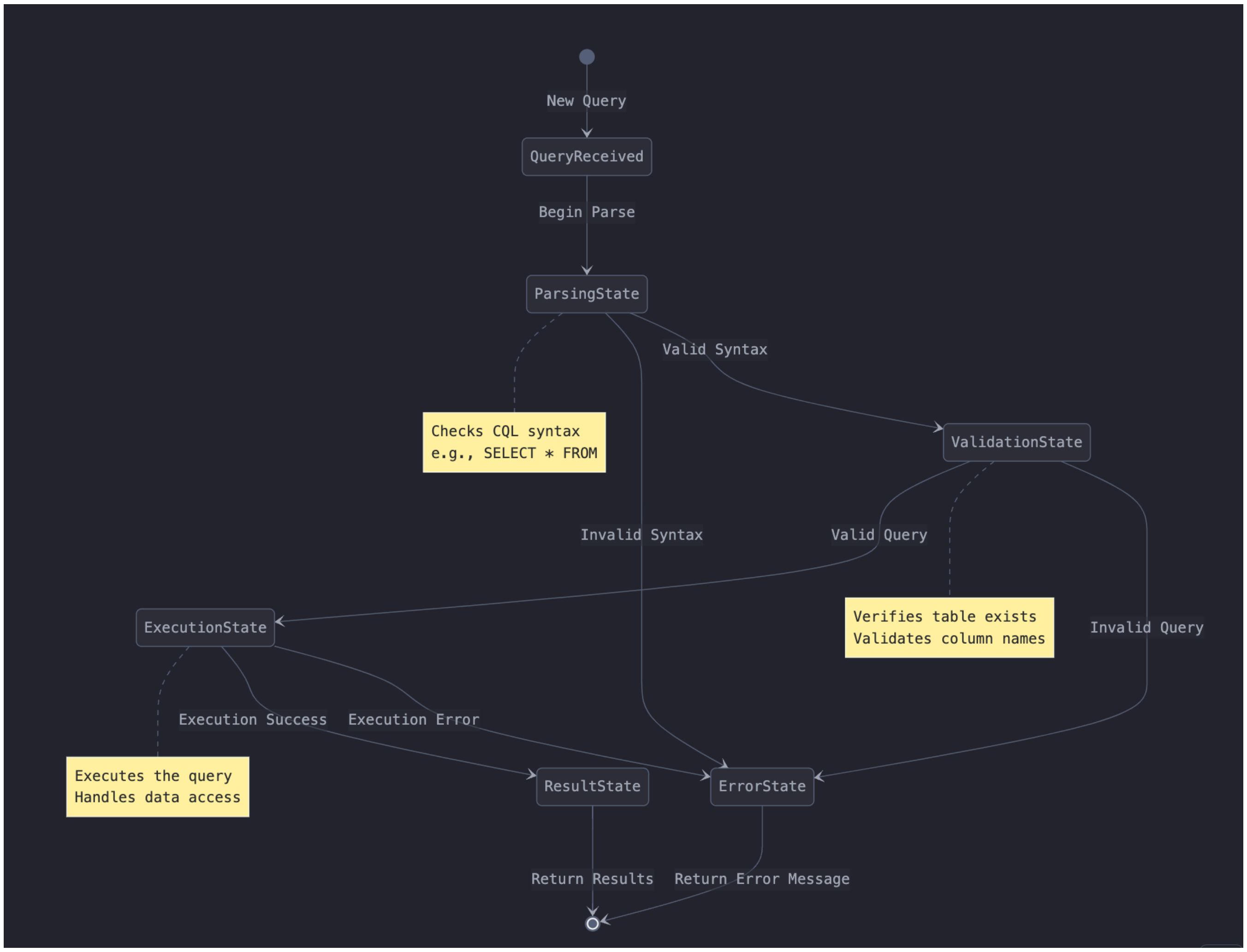
- Runtime errors

5. ResultState

- Successful query completion
- Result set preparation

6. ErrorState

- Handles all error conditions
- Generates appropriate error messages



Pic 1. Finite State Machine (FSM) Flow Graph of CQL Functionality

Test Cases

The following test cases cover all states and major transition paths in the FSM. We add these tests directly to SelectStatementTest.java

1. *testSuccessfulQuery*

State flow: QueryReceived → ParsingState → ValidationState → ExecutionState → ResultState

```java
@Test
public void testSuccessfulQueryFlow()
{
    QueryProcessor.executeOnceInternal("CREATE TABLE ks.fsm_test (id int
PRIMARY KEY, value text)");
    QueryProcessor.executeOnceInternal("INSERT INTO ks.fsm_test (id, value)
VALUES (1, 'test')");

    // Verify the query is parsed and executed correctly
    SelectStatement stmt = parseSelect("SELECT * FROM ks.fsm_test WHERE id =
1");
    Assert.assertNotNull(stmt.makeSlices(QueryOptions.DEFAULT));
}
```

2. *testSyntaxError*

State flow: QueryReceived → ParsingState → ErrorState

```java
@Test
public void testSyntaxError()
{
    try {
        parseSelect("SELECT * test_table");
        Assert.fail("Expected SyntaxException was not thrown");
    } catch (SyntaxException e) {
        // Expected to throw SyntaxError
    }
}
```

3. *testTableValidationError*

State flow: QueryReceived → ParsingState → ValidationState → ErrorState

```java
@Test
public void testTableValidationError() {
      try{
            QueryProcessor.executeOnceInternal("SELECT * FROM
      nonexistent_table");
            Assert.fail("Expected InvalidRequestException was not thrown");
      } catch(InvalidRequestException e){
      }
}
```

4. *testColumnValidationError*

State flow: QueryReceived → ParsingState → ValidationState → ErrorState

```java
@Test
public void testColumnValidationError() {
      try{
            QueryProcessor.executeOnceInternal("CREATE TABLE ks.fsm_test2 (id
      int PRIMARY KEY, value text)");
            QueryProcessor.executeOnceInternal("SELECT nonexistent_column FROM
      ks.fsm_test2");
            Assert.fail("Expected InvalidRequestException was not thrown");
      } catch(InvalidRequestException e){
      }
}
```

5. *testExecutionError*

State flow: QueryReceived → ParsingState → ValidationState → ExecutionState → ErrorState

```java
@Test // New Test: Test Execution Error Handling
```

```
    public void testExecutionError() throws Throwable {
        QueryProcessor.executeOnceInternal("CREATE TABLE ks.fsm_test3 (id int
PRIMARY KEY, value text)");

        try {
            QueryProcessor.executeOnceInternal("SELECT * FROM ks.fsm_test3
WHERE id = 1/0");
            Assert.fail("Expected InvalidRequestException was not thrown");
        } catch (InvalidRequestException e) {
            // Expected exception
        }
    }
```

6. *testComplexQueryValidation*

Purpose: Test complex query

```java
@Test // New Test: Test Complex Query Validation
    public void testComplexQueryValidation() throws Throwable {
        QueryProcessor.executeOnceInternal("CREATE TABLE ks.fsm_test4 (id int
PRIMARY KEY, value text, data int)");
        QueryProcessor.executeOnceInternal("INSERT INTO ks.fsm_test4 (id,
value, data) VALUES (1, 'test', 100)");
        QueryProcessor.executeOnceInternal("INSERT INTO ks.fsm_test4 (id,
value, data) VALUES (2, 'test2', 200)");

        SelectStatement stmt = parseSelect(
            "SELECT id, value, data FROM ks.fsm_test4 WHERE id IN (1, 2) AND
data > 150 ALLOW FILTERING"
        );

        Assert.assertNotNull(stmt);
    }
```

IV. STRUCTURAL TESTING AND COVERAGE TOOL

Structural Testing

Structural testing, also known as white-box testing, is a software testing approach that examines a program's internal structure, logic, and code implementation to ensure correctness and efficiency. Unlike black-box testing, which focuses on input-output behavior, structural testing evaluates how the code executes, covering aspects such as statement, branch, and path coverage. This method is crucial because it helps identify hidden logic errors and dead code that may go unnoticed in functional testing. By ensuring thorough test coverage, structural testing enhances software reliability and performance, ultimately leading to more robust and efficient systems.

Coverage Analysis with JaCoCo

JaCoCo (Java Code Coverage) is the primary code coverage tool used in Apache Cassandra's testing infrastructure. For our analysis of the SelectStatement class, we utilized JaCoCo to measure and improve test coverage.

To run JaCoCo coverage tool for *SelectStatementTest*, execute the following commands:

```
Unset
ant codecoverage
-Dtest.name=org.apache.cassandra.cql3.statements.SelectStatementTest
```

Based on the JaCoCo coverage report for *SelectStatementTest*, we analyzed the coverage metrics and highlight uncovered areas:

Initial Coverage Analysis:

Before implementing new test cases, the JaCoCo report showed the following metrics for *SelectStatementTest*:

• Line coverage: 29%
  322 out of 451 lines are missed

• Branch coverage: 15%
  247 out of 292 branches are missed

• Method coverage: 52%
  27 out of 56 methods are missed

Methods with Poor Coverage (0%)

1. Basic Operations:

  • asCQL(QueryOptions, ClientState)

- authorize(ClientState)
- toString()
- table()
- keyspace()

2. Query Processing:
   - clusteringIndexFilterAsSlices()
   - internalReadForViewDecorator()
   - getRequestedRows(QueryOptions)
   - getFunctions()
3. Filter and Data Operations:
   - rowFilterForInternalCalls()
   - getQueriedColumns()
   - getPartitionKeyBindVariableIndexes()

New Test Cases

Based on the coverage analysis from JaCoCo, we identified several critical areas requiring enhanced test coverage. Our new test suite focuses on five key functional areas of the SelectStatement class. Each test case includes comprehensive assertions to validate the expected behavior and proper error handling. The new test suite is designed to increase code coverage while testing meaningful functionality rather than simply achieving higher coverage numbers.

```java
@Test
    public void testAuthorize() {
        QueryProcessor.executeOnceInternal("CREATE TABLE ks.auth_test (id int PRIMARY KEY, value text)");
        QueryProcessor.executeOnceInternal("INSERT INTO ks.auth_test (id, value) VALUES (1, 'test')");
        SelectStatement stmt = SelectStatementTest.parseSelect("SELECT * FROM ks.auth_test WHERE id = 1");
        try {
            stmt.authorize(ClientState.forInternalCalls());
        } catch (Exception e) {
            fail("authorize() should not throw an exception for a valid ClientState: " + e.getMessage());
        }
    }
```

Java
```java
@Test
    public void testToStringMethod() {
        QueryProcessor.executeOnceInternal("CREATE TABLE ks.to_string_test (id
int PRIMARY KEY, value text)");
        // No row insertion is needed just to check toString() output
        SelectStatement stmt = SelectStatementTest.parseSelect("SELECT * FROM
ks.to_string_test WHERE id = 1");
        String str = stmt.toString();
        assertNotNull("toString() should not return null", str);
        assertTrue("toString() output should contain keyspace",
str.contains("ks"));
        assertTrue("toString() output should contain table name",
str.contains("to_string_test"));
    }
```

Java
```java
@Test
    public void testTableMethod() {
        QueryProcessor.executeOnceInternal("CREATE TABLE ks.table_test (id int
PRIMARY KEY, value text)");
        SelectStatement stmt = SelectStatementTest.parseSelect("SELECT * FROM
ks.table_test WHERE id = 1");
        String tableName = stmt.table();
        assertEquals("table_test", tableName);
    }
```

Java
```java
@Test // Tests if the SelectStatement correctly parses and retrieves the
keyspace from the query.
    public void testKeyspaceMethod() throws Throwable {
        SelectStatement stmt = parseSelect("SELECT * FROM ks.test_table");
        Assert.assertNotNull(stmt);
        Assert.assertEquals("ks", stmt.keyspace());
    }
```

Coverage Improvement

• Line coverage: 60% (from 29%)
182 out of 451 lines are missed

• Branch coverage: 38% (from 15%)
181 out of 292 branches are missed

• Method coverage: 88% (from 52%)
7 out of 56 methods are missed

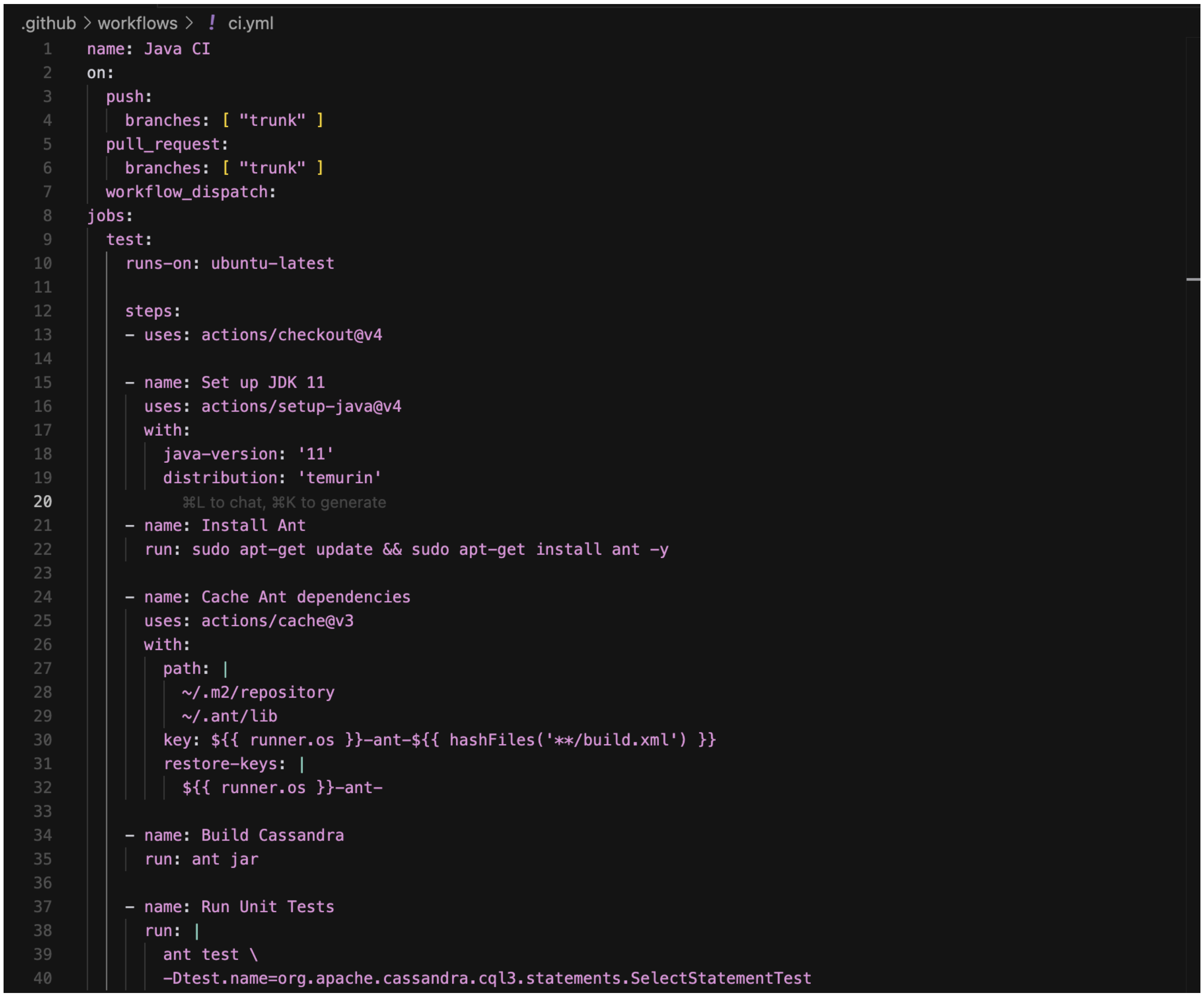
The enhanced test suite provides better confidence in the *SelectStatementTest* class's reliability and correctness across various use cases.

V. CONTINUOUS INTEGRATION

Continuous Integration (CI) is a software development practice where developers frequently integrate their code changes into a shared repository, typically multiple times per day. Each integration triggers an automated build and test process that verifies the changes, enabling early detection of integration issues and bugs. Several robust CI platforms have emerged as industry standards, including Jenkins, CircleCI, TravisCI, and GitHub Actions. Each platform offers unique features while adhering to core CI principles of automation, rapid feedback, and consistent testing.

For our Cassandra test suite enhancement project, we implemented GitHub Actions as our CI platform of choice. The cornerstone of our CI implementation is the *ci.yml* workflow configuration file, which orchestrates an automated testing pipeline specifically designed for our Java-based Cassandra project. This workflow ensures that every code change undergoes rigorous testing before integration, maintaining the reliability and stability of our test suite enhancements. Below are the main components of *ci.yml*:

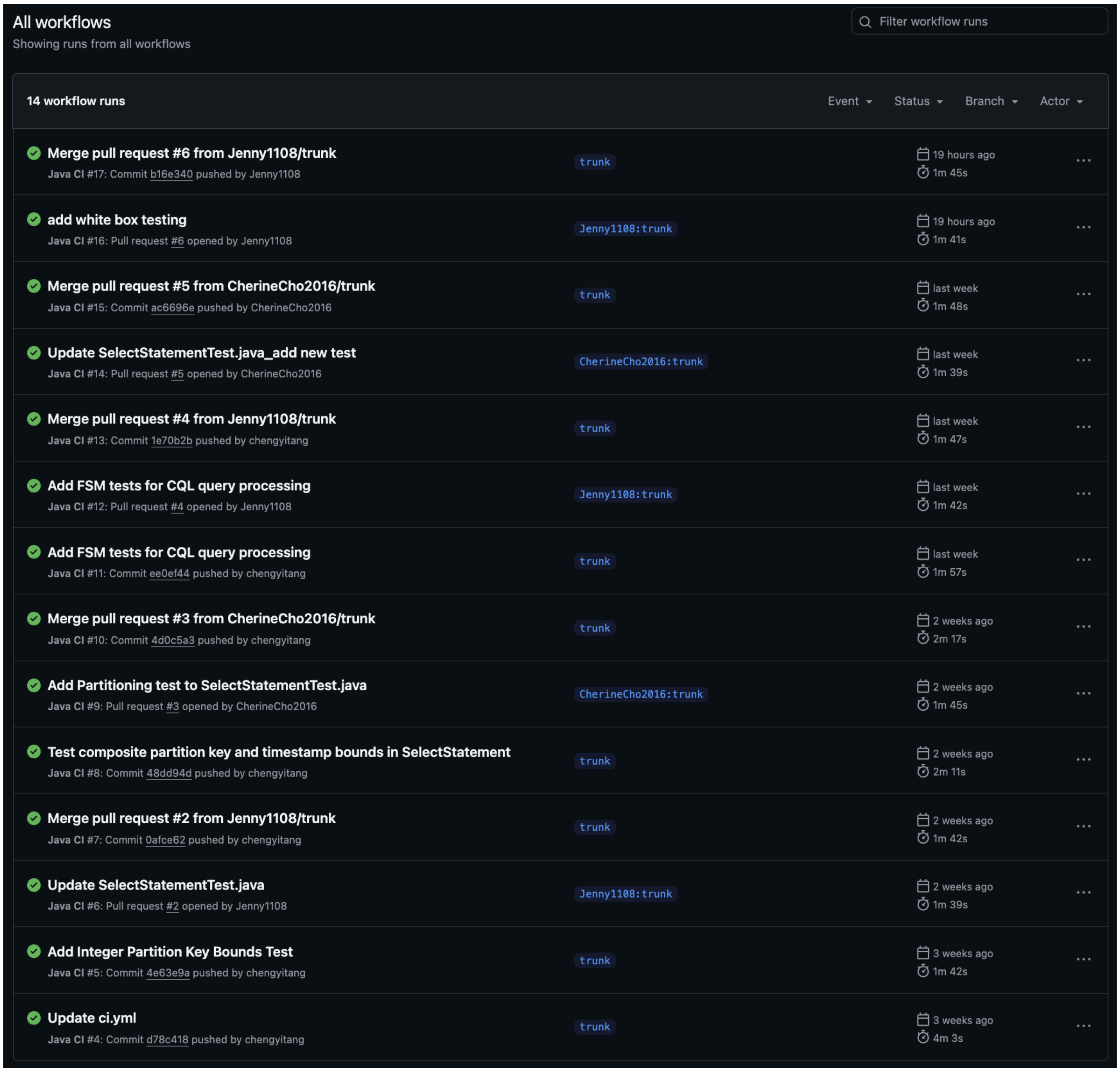
1. Trigger:
    a. Runs on pushes to the "trunk" branch
    b. Runs on pull requests targeting the "trunk" branch
    c. Can be manually triggered via workflow_dispatch
2. Job Configuration
    a. Runs on an Ubuntu latest environment
    b. Contains a series of sequential steps for building and testing
3. Setup Steps
    a. Check out the code repository
    b. Sets up JDK 11 using Temurin distribution
    c. Installs apache Ant build tool
    d. Implements caching for Ant dependencies to speed up subsequent runs
4. Build and Test
    a. Builds Cassandra using ant jar command
    b. Runs a specific unit test: *org.apache.cassandra.cql3.statements.SelectStatementTest*



.github > workflows > ! ci.yml

```yaml
name: Java CI
on:
  push:
    branches: [ "trunk" ]
  pull_request:
    branches: [ "trunk" ]
  workflow_dispatch:
jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4

    - name: Set up JDK 11
      uses: actions/setup-java@v4
      with:
        java-version: '11'
        distribution: 'temurin'
          ⌘L to chat, ⌘K to generate
    - name: Install Ant
      run: sudo apt-get update && sudo apt-get install ant -y

    - name: Cache Ant dependencies
      uses: actions/cache@v3
      with:
        path: |
          ~/.m2/repository
          ~/.ant/lib
        key: ${{ runner.os }}-ant-${{ hashFiles('**/build.xml') }}
        restore-keys: |
          ${{ runner.os }}-ant-

    - name: Build Cassandra
      run: ant jar

    - name: Run Unit Tests
      run: |
        ant test \
        -Dtest.name=org.apache.cassandra.cql3.statements.SelectStatementTest
```

Figure 1:. ci.yml



All workflows
Showing runs from all workflows

Filter workflow runs

14 workflow runs Event Status Branch Actor

Workflow run	Branch	Time	Duration
Merge pull request #6 from Jenny1108/trunk Java CI #17: Commit b16e340 pushed by Jenny1108	trunk	19 hours ago	1m 45s
add white box testing Java CI #16: Pull request #6 opened by Jenny1108	Jenny1108:trunk	19 hours ago	1m 41s
Merge pull request #5 from CherineCho2016/trunk Java CI #15: Commit ac6696e pushed by CherineCho2016	trunk	last week	1m 48s
Update SelectStatementTest.java_add new test Java CI #14: Pull request #5 opened by CherineCho2016	CherineCho2016:trunk	last week	1m 39s
Merge pull request #4 from Jenny1108/trunk Java CI #13: Commit 1e70b2b pushed by chengyitang	trunk	last week	1m 47s
Add FSM tests for CQL query processing Java CI #12: Pull request #4 opened by Jenny1108	Jenny1108:trunk	last week	1m 42s
Add FSM tests for CQL query processing Java CI #11: Commit ee0ef44 pushed by chengyitang	trunk	last week	1m 57s
Merge pull request #3 from CherineCho2016/trunk Java CI #10: Commit 4d0c5a3 pushed by chengyitang	trunk	2 weeks ago	2m 17s
Add Partitioning test to SelectStatementTest.java Java CI #9: Pull request #3 opened by CherineCho2016	CherineCho2016:trunk	2 weeks ago	1m 45s
Test composite partition key and timestamp bounds in SelectStatement Java CI #8: Commit 48dd94d pushed by chengyitang	trunk	2 weeks ago	2m 11s
Merge pull request #2 from Jenny1108/trunk Java CI #7: Commit 0afce62 pushed by chengyitang	trunk	2 weeks ago	1m 42s
Update SelectStatementTest.java Java CI #6: Pull request #2 opened by Jenny1108	Jenny1108:trunk	2 weeks ago	1m 39s
Add Integer Partition Key Bounds Test Java CI #5: Commit 4e63e9a pushed by chengyitang	trunk	3 weeks ago	1m 42s
Update ci.yml Java CI #4: Commit d78c418 pushed by chengyitang	trunk	3 weeks ago	4m 3s

Figure 2: GitHub Actions workflow records

VI. TESTABLE DESIGN AND MOCKING

Testable Design

Aspects and Goals for Testable Design

A testable design is characterized by its modular structure and clear separation of concerns, ensuring that each component has a single, well-defined responsibility that can be independently verified. It emphasizes low coupling and high cohesion, achieved through *dependency injection* so that components receive their dependencies externally rather than hardcoding them, which makes it easier to substitute mocks or stubs during testing. Additionally, a testable design minimizes side effects by avoiding reliance on global state and static methods, thereby isolating functionality and making behavior predictable. Clear interfaces and abstractions are also crucial, as they define explicit contracts that allow for targeted tests without unnecessary interference from other parts of the system.

Problematic Implementation

testNonsensicalBounds() method exhibits several problems that make it difficult to test thoroughly:

```java
@Test
public void testNonsensicalBounds()
{
    QueryProcessor.executeOnceInternal("CREATE TABLE ks.tbl (k int, c int, v int, primary key (k, c))");
    QueryProcessor.executeOnceInternal("INSERT INTO ks.tbl (k, c, v) VALUES (100, 10, 0)");
    Assert.assertEquals(Slices.NONE, parseSelect("SELECT * FROM ks.tbl WHERE k=100 AND c > 10 AND c <= 10").makeSlices(QueryOptions.DEFAULT));
    Assert.assertEquals(Slices.NONE, parseSelect("SELECT * FROM ks.tbl WHERE k=100 AND c < 10 AND c >= 10").makeSlices(QueryOptions.DEFAULT));
    Assert.assertEquals(Slices.NONE, parseSelect("SELECT * FROM ks.tbl WHERE k=100 AND c < 10 AND c > 10").makeSlices(QueryOptions.DEFAULT));
}
```

Issues with this implementation:

1. **Direct Dependency on Database**: The method directly uses database operations, making it impossible to test without a live database.
2. **Multiple Test Cases in One Method**: The method tests three different scenarios without clear separation.

3. **Hard to Test Edge Cases**: There's no way to test what happens when the database isn't available or when queries fail.
4. **No Validation of Query Execution**: It assumes the setup queries executed successfully.

   Implicit Dependencies: The method depends on the behavior of *parseSelect()* and *makeSlices()* but doesn't explicitly declare this.

To improve testability, we did following modifications:

1. Extract the query validation logic into a separate method that doesn't require database operations.
2. Make the validation method explicitly work with the components under test.
3. Allow testing without database operations.

Test Case for the New Testable Code

```java
@Test
public void testBoundsValidation() {
    // Test cases with nonsensical/contradictory bounds
    assertFalse("Greater than AND less than or equal to same value should be
invalid", hasValidBounds("k=100 AND c > 10 AND c <= 10"));

    assertFalse("Less than AND greater than or equal to same value should be
invalid", hasValidBounds("k=100 AND c < 10 AND c >= 10"));

    assertFalse("Less than AND greater than same value should be
invalid",hasValidBounds("k=100 AND c < 10 AND c > 10"));

    // Test cases with valid bounds
    assertTrue("Greater than x AND less than y (where y > x) should be valid",
hasValidBounds("k=100 AND c > 10 AND c < 20"));

    assertTrue("Greater than or equal to x AND less than or equal to x should
be valid (point query)", hasValidBounds("k=100 AND c >= 10 AND c <= 10"));

    assertTrue("Equal to x should be valid", hasValidBounds("k=100 AND c =
10"));

    // Edge cases
    assertTrue("IN clause should be valid", hasValidBounds("k=100 AND c IN (5,
10, 15)"));

    assertFalse("Contradictory equality constraints should be invalid",
hasValidBounds("k=100 AND c = 10 AND c = 20"));
```

```
    // Test with different column types (timestamp)
    assertTrue("Timestamp bounds should follow same rules",
hasValidBounds("k='2023-01-01' AND c > '2023-01-01 10:00:00' AND c <
'2023-01-01 11:00:00'"));

    assertFalse("Contradictory timestamp bounds should be invalid",
hasValidBounds("k='2023-01-01' AND c > '2023-01-01 10:00:00' AND c <
'2023-01-01 09:00:00'"));
}
```

The new approach allows for comprehensive testing of the bounds validation logic without the overhead and potential instability of actual database operations, leading to more reliable, faster, and more focused tests.

Mocking

Mocking is a technique used in unit testing where real objects are replaced with simulated objects which mimic the behavior of the real components. These simulated objects allow developers to isolate the code being tested by removing dependencies on external systems or complex components.

In our Apache Cassandra project, we extend the CQL SelectStatementTest to focus on the QueryProcessor component's behavior by adding a new test class QueryProcessorTest.java under cql3 directory.

Mock Objects:

By using these mock objects, we can verify the behavior of QueryProcessor without relying on actual database connections or query execution.

• QueryProcessor: The main component under test

• ClientState: Simulates client connection state

• ResultMessage.Prepared: Represents prepared statements

Test Scenarios:

• Valid query parsing

• Invalid query handling

• Exception scenarios

Mocking Benefits

Using mocking in our test suite delivers significant advantages by creating an isolated testing environment for QueryProcessor logic. This approach enhances test speed by eliminating database connection and query execution overhead, while simultaneously improving stability by removing dependence on environmental factors. The ability to simulate specific error scenarios gives testers precise control over test conditions, resulting in predictable, deterministic outcomes regardless of when or where tests are run.

Run Test

```
Unset
# Updated in CI workflow
ant test -Dtest.name=org.apache.cassandra.cql3.QueryProcessorTest
```

VII. STATIC ANALYSIS TOOLS

Introduction to Static Analysis Tools

1. What Are Static Analysis Tools?

   Static analysis tools are software programs that examine source code without executing it. These tools help developers identify potential issues such as coding standard violations, security vulnerabilities, and logical errors at an early stage of development.

2. How Do These Tools Help Developers?

   By analyzing the code statically, these tools can detect potential problems before runtime, reducing debugging effort and improving code quality. They assist in enforcing coding standards, maintaining consistency, and catching common programming mistakes such as null pointer dereferences or improper synchronization.

Tools Used for Analysis

For this analysis, we selected two different static analysis tools:

1. Checkstyle

   Checkstyle is a tool designed to ensure compliance with coding standards. It checks for issues related to code formatting, naming conventions, indentation, and other stylistic aspects. This helps in maintaining a uniform codebase that is easier to read and maintain.

2. SpotBugs

   SpotBugs is a tool that detects potential programming errors such as null pointer dereferences, infinite loops, and resource leaks. Unlike Checkstyle, SpotBugs focuses more on logical and runtime-related issues that could lead to software defects.

3. Steps Taken to Run These Tools

To integrate these tools into the project, we add the “.build/pom.xml” file to include their respective Maven plugins. Below are the steps:

Checkstyle:

- Added the Checkstyle plugin to “.build/pom.xml”
- Ran the command:

```
Unset
mvn checkstyle:checkstyle
```

SpotBugs:

- Added the SpotBugs plugin to “.build/pom.xml”
- Ran the command:

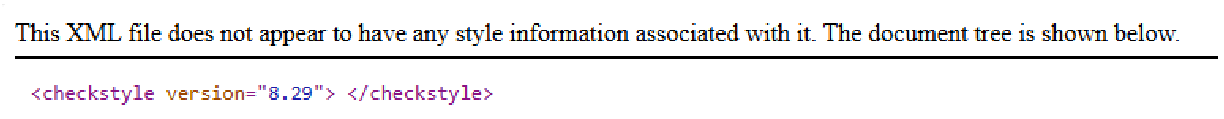
```
Unset
mvn spotbugs:spotbugs
mvn spotbugs:gui
```

Results Analysis and Comparison

1. How Many Warnings Were Found?

Both tools reported no violations or errors:

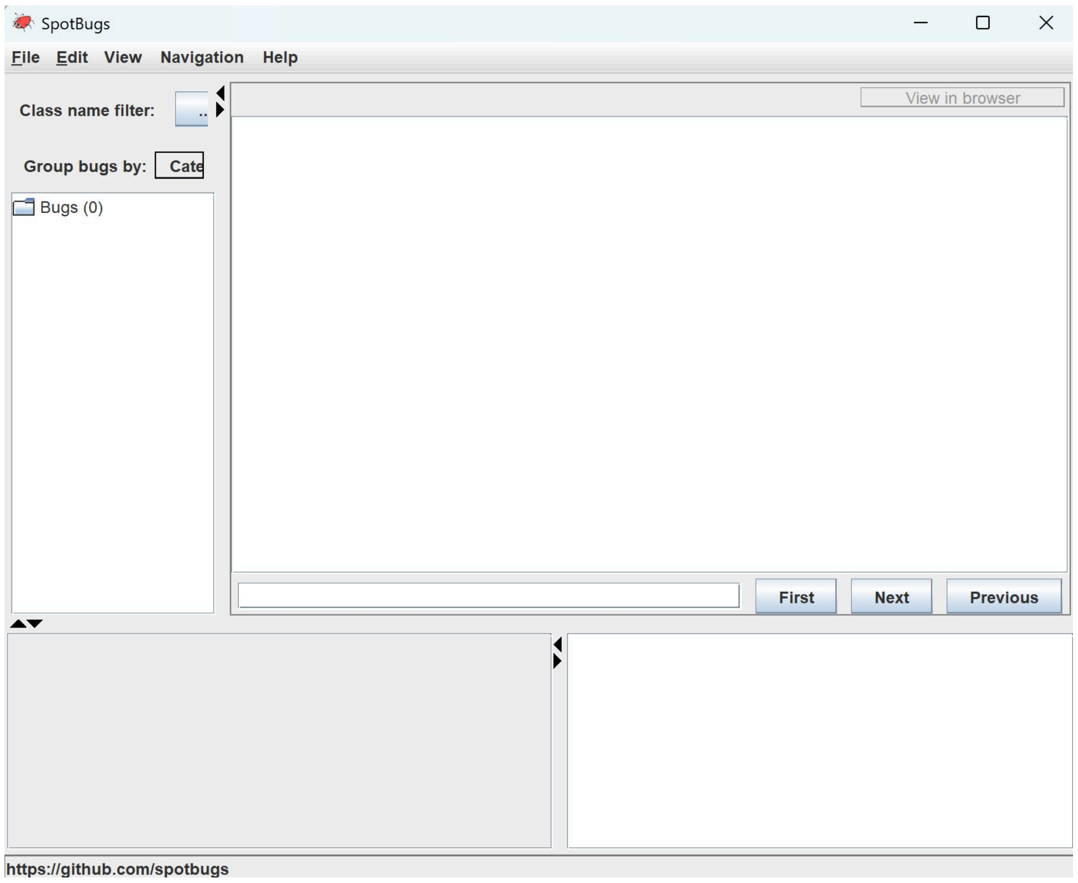
- Checkstyle found no style violations due to an existing configuration that enforces proper coding practices.



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<checkstyle version="8.29"> </checkstyle>
```

- SpotBugs found no potential bugs in the codebase.



2. Important vs. Ignorable Warnings

Since no warnings were reported, there was no need to differentiate between critical and non-critical warnings. However, in a typical scenario, serious issues such as security vulnerabilities or logic errors would require immediate attention, while minor formatting inconsistencies might be less critical.

3. Do the Tools Overlap or Identify Different Issues?

   Checkstyle and SpotBugs serve different purposes:

   - Checkstyle focuses on code style and readability.
   - SpotBugs detects runtime-related issues and logical flaws. In this case, neither tool found any issues, but in a real-world scenario, they could complement each other by highlighting different aspects of code quality.

4. Which Tool Provided More Valuable Information?

   Since both tools returned no findings, their value in this case was limited. However, in general:

   - Checkstyle is useful for enforcing consistency and best practices in a team setting.

- SpotBugs is more valuable for identifying real bugs that could cause runtime failures.

Conclusion

Static analysis tools play a crucial role in software quality assurance. In this experiment, both Checkstyle and SpotBugs found no issues in the given codebase, indicating that it adheres to coding standards and is free from detectable logic errors. These tools are valuable for maintaining high-quality code and preventing potential defects before deployment.