Pull Request progress summary

1. Issue Details

The issue we are working on is Cassandra-12760: SELECT JSON "firstName" FROM ... results in {""firstName"": "Bill"}.

When the user executes the following commands:

```
create table user(id text PRIMARY KEY, "firstName" text);
insert into user(id, "firstName") values ('b', 'Bill');
select json * from user;
```

The expected result is:

```
[json]
----------------------------------------
{"id": "b", "firstName": "Bill"}
```

While the current output is:

```
[json]
----------------------------------------
{"id": "b", "\"firstName\"": "Bill"}
```

The reason is that

> The results of SELECT JSON are designed to be usable in an INSERT JSON statement without any modifications, so all of the same rules about non-text map keys and case-sensitive column names apply.

(from https://www.datastax.com/blog/2015/06/whats-new-cassandra-22-json-support)

The way to solve the issue is from Cesar Agustin Garcia Vazquez:

> We could add a new option like `select plain_json * from user;` which would have this expected behavior.

We love the idea of creating this new `plain_json` option so that people can query information from Cassandra and used it directly in outside projects.

2. Solving the issue

2.1. Locate the "extra double quotes"

First we want to know where the set of double quotes is added to the result. We use the key word **JSON or Json** to search through all the files and stopped at a method `rowToJson`. The following two lines are assumed to be the place we are searching for.

```
    if (!columnName.equals(columnName.toLowerCase(Locale.US)))
        columnName = "\"" + columnName + "\"";
```

But it seems that the extra quotes are added only when the `columnName` is not consist of only lower case letters. To test if it is the case, we did the following tests:

1. Use `firstname` instead of `firstName` as the column name.

```
create table user(id text PRIMARY KEY, "firstname" text);
insert into user(id, "firstname") values ('b', 'Bill');
select json * from user;
```

2. Comment the two lines, built and try the same statement with `firstName` again.

Our experiments show that this is exactly where the quotes are added. But this `rowToJson` method does no evaluation on if the user is expecting a JSON format or not. Its only function is to do the type transfer.

2.2. How to decide if JSON is expected

We want to know which method is the one to decide to call the `rowToJson`. We use a form to record all the methods called when the `select json * from user` statement is executed, from the beginning to `rowToJson`.

Here is the list.

	A	B	C	D	E
1	java folder	java file	method	line	statement
2	concurrent	SEPWorker	void run()	119	task.run()
3	concurrent	AbstractLocalAwareExecutorService	void run()	165	result = callable.call();
4					
5	transport	Message	void channelRead(ctx, request)	630	processRequest(ctx, request);
6	transport	Message	void processRequest(ctx, request)	725	response = request.execute(qstate,
7	transport	Message	Response execute(queryState, queryStartNanoTime, traceRequest)	253	response = execute(queryState, queryStartNanoTime, shouldTrace);
8	transport	QueryMessage extends Message.Request	Message.Response execute(state, queryStartNanoTime, traceRequest)	108	Message.Response response = ClientState.getCQLQueryHandler().process(query, state, options, getCustomPayload(),
9	cql3	QueryProcessor	ResultMessage process(**query,** state, options, customPayload, queryStartNanoTime) **[query: "select json * from user;"]**	233	process(query, state, options, queryNanoTime);
10	cql3	QueryProcessor	ResultMessage process(queryString, queryState, options, queryStartNanoTime) [queryString: "select json * from user;"]	239	CQLStatement prepared = getStatement(queryString, ...);
11			CQLStatement getStatement(queryStr, clientState)	516	CQLStatement.Raw statement = parseStatement(queryStr);
12			CQLStatement.Raw parseStatement(queryStr)	546	CQLFragmentParser.parseAnyUnhandled(CqlParser::query, queryStr);
13		CQLFragmentParser	<R> R parseAnyUnhandled(CQLParserFunction<R> parserFunction, input)		
14				523	statement.prepare(clientState);
15				247	processStatement(prepared, queryState, options, queryStartNanoTime);
16	cql3	QueryProcessor	ResultMessage processStatemetn(statement, queryState, options, queryStateNanoTime)	216	result = statement.execute(queryState, options, queryStartNanoTime);
17	statements	SelectStatement	ResultMessage.Rows execute(state, options, queryStartNanoTime)	250	execute(??)
18	statements	SelectStatement	ResultMessage.Rows execute(????????)	402	msg = processResults(page, options, selectors, nowInSec. userLimit);
19	statements	SelectStatement	ResultMessage.Rows processResults(partitions, options, selectors, nowInSec, userLimit)	425	ResultSet rset = process(partitions, options, selectors, nowInSec, userLimit);
20	statements	SelectStatement	ResultSet process(partitions, options, selectors,notInSec, userLimit)	785	ResultSet cqlRows = result.build();
21	cql3/selection	ResultSetBuilder	ResultSet build()	156	resultSet.addRow(getOutputRow());
22			List<ByteBuffer> getOutputRow()	169	selectors.getOutputRow();
23	cql3/selection	Selection	Selectors newSelectors(options)	458	rowToJson(current, options.getProtocolVersion(), metadata, orderingColumns);
24	cql3/selection	Selection	List<ByteBuffer> rowToJson(row, protocolVersion, metadata, orderingColumns)		

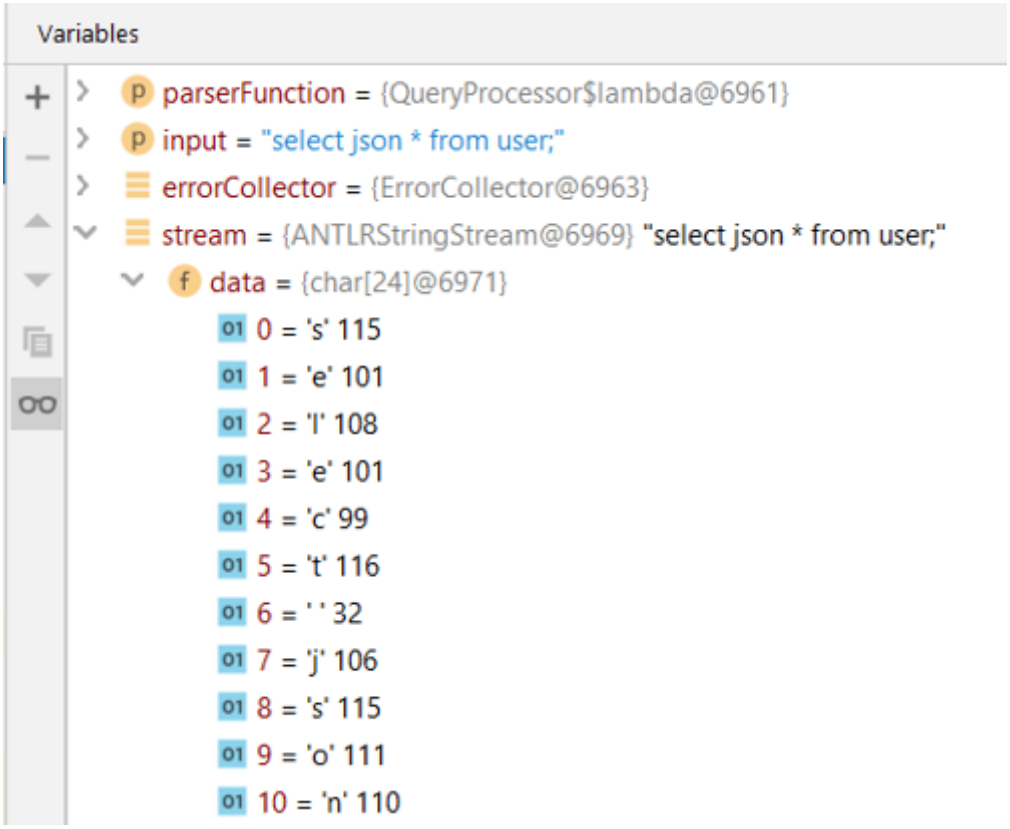
2.3. Parse method for the SelectionStatement

The key method is `parseAnyUnhandled(CQLParserFunction<R> parserFunction, String input)`.

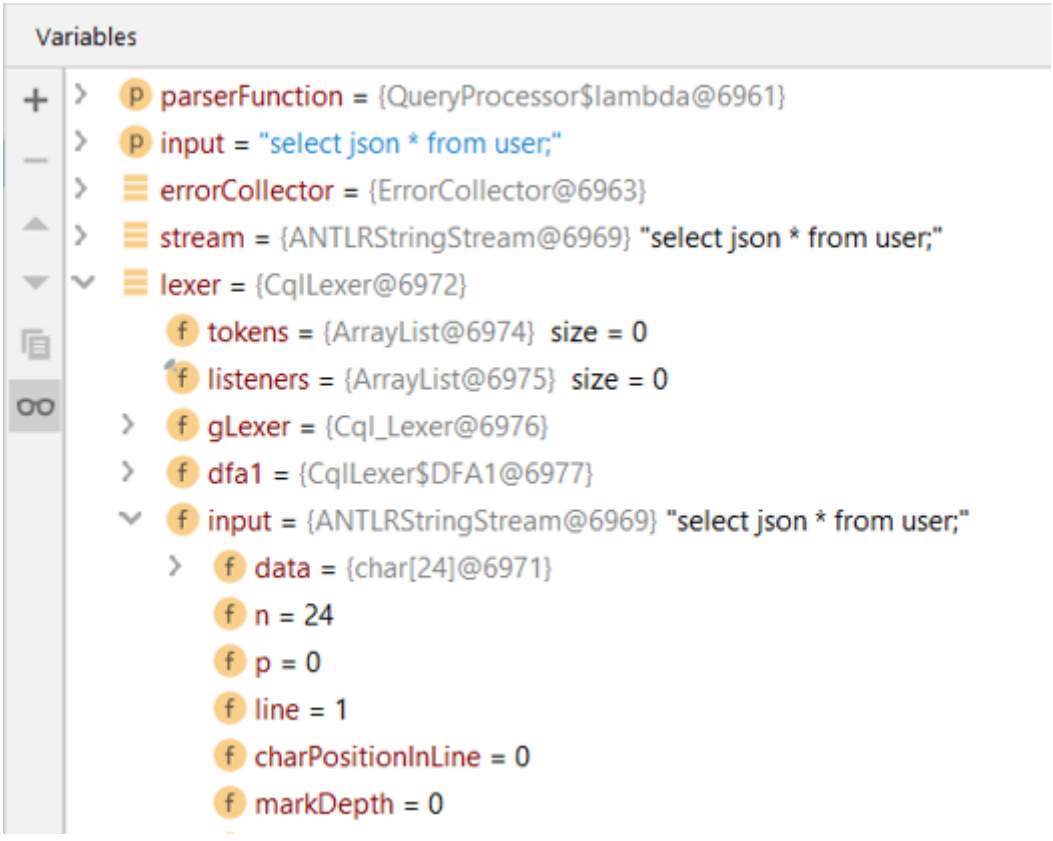
```java
/**
* Just call a parser method in {@link CqlParser} - does not do any error handling.
*/
public static <R> R parseAnyUnhandled(CQLParserFunction<R> parserFunction, String input) throws RecognitionException {
    // Lexer and parser
    ErrorCollector errorCollector = new ErrorCollector(input);
    CharStream stream = new ANTLRStringStream(input);
    CqlLexer lexer = new CqlLexer(stream);
    lexer.addErrorListener(errorCollector);

    TokenStream tokenStream = new CommonTokenStream(lexer);
    CqlParser parser = new CqlParser(tokenStream);
    parser.addErrorListener(errorCollector);

    // Parse the query string to a statement instance
    R r = parserFunction.parse(parser);

    // The errorCollector has queue up any errors that the lexer and parser may have encountered
    // along the way, if necessary, we turn the last error into exceptions here.
    errorCollector.throwFirstSyntaxError();

    return r;
}
```

As the input string is exactly the one we typed: `select json * from user`, we can confidently assume that the JSON evaluation process is done in one of these statements. So we need to check each result to find if it contains the JSON information. The candidates are: `stream`, `lexer`, `tokenStream`, `parser`, and `r`. But we bet on `r` because the comment says "Parse the query string to a statement instance".

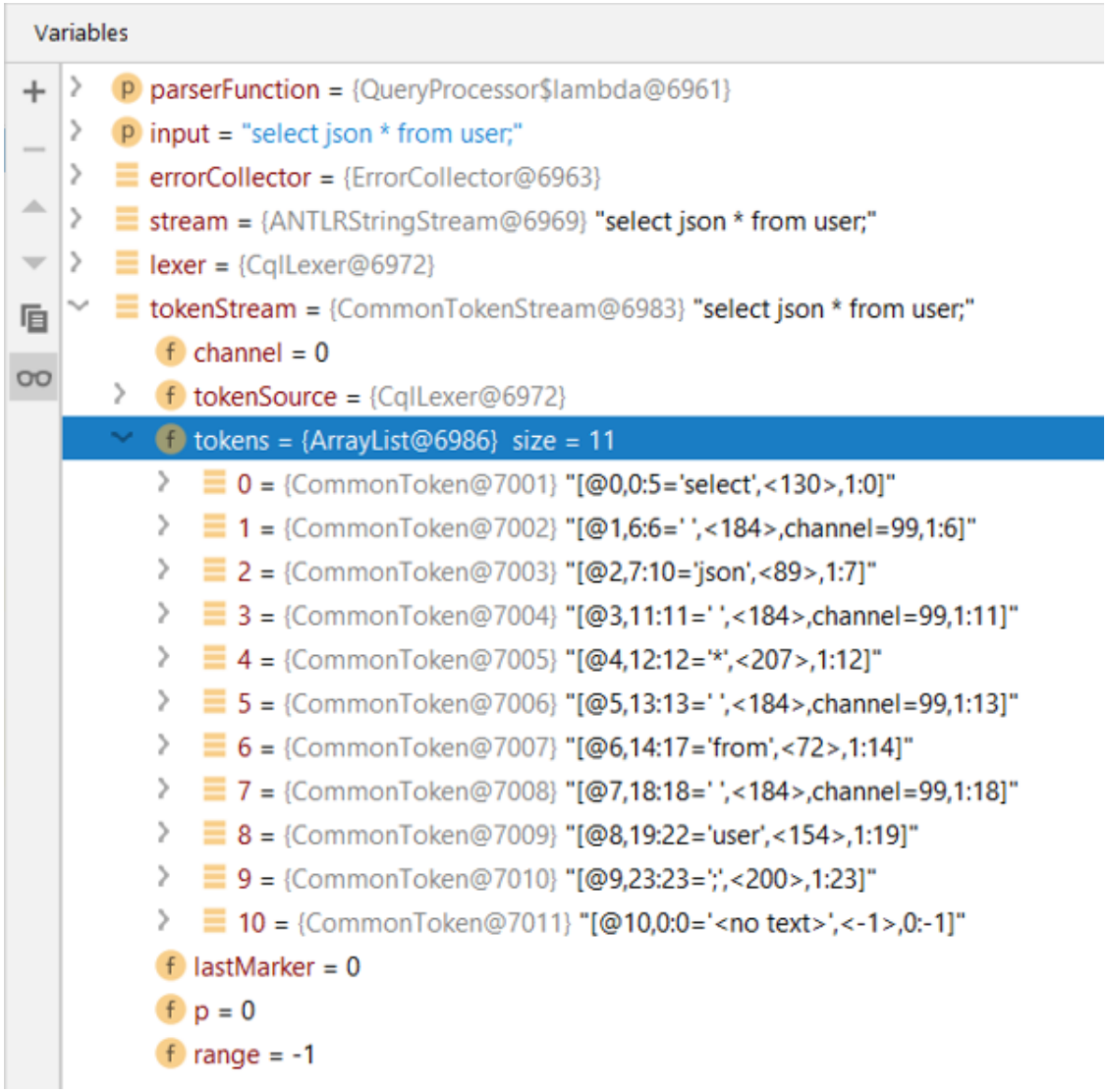
- `stream` changes the string input into a char array.



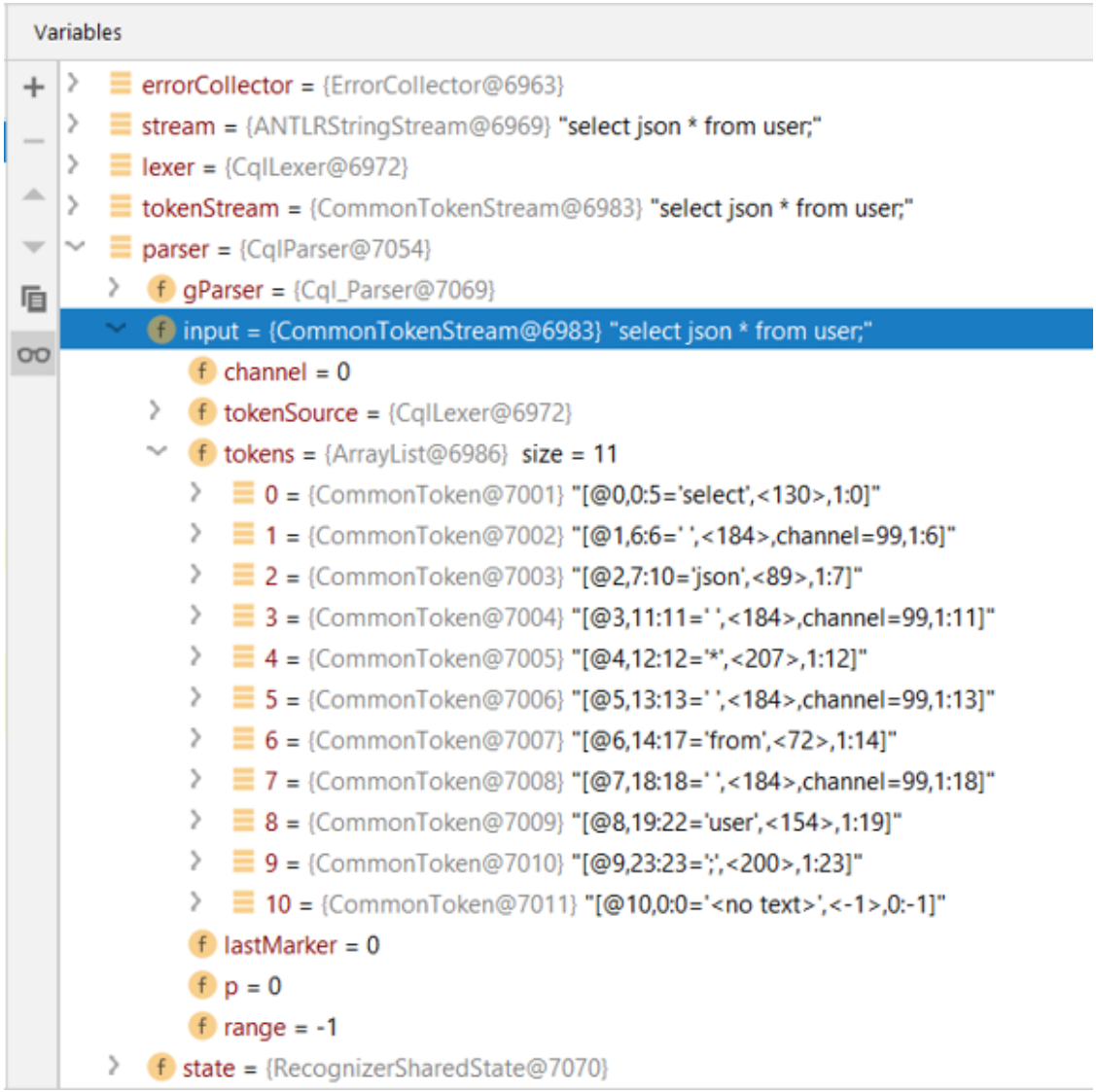
- `lexer`



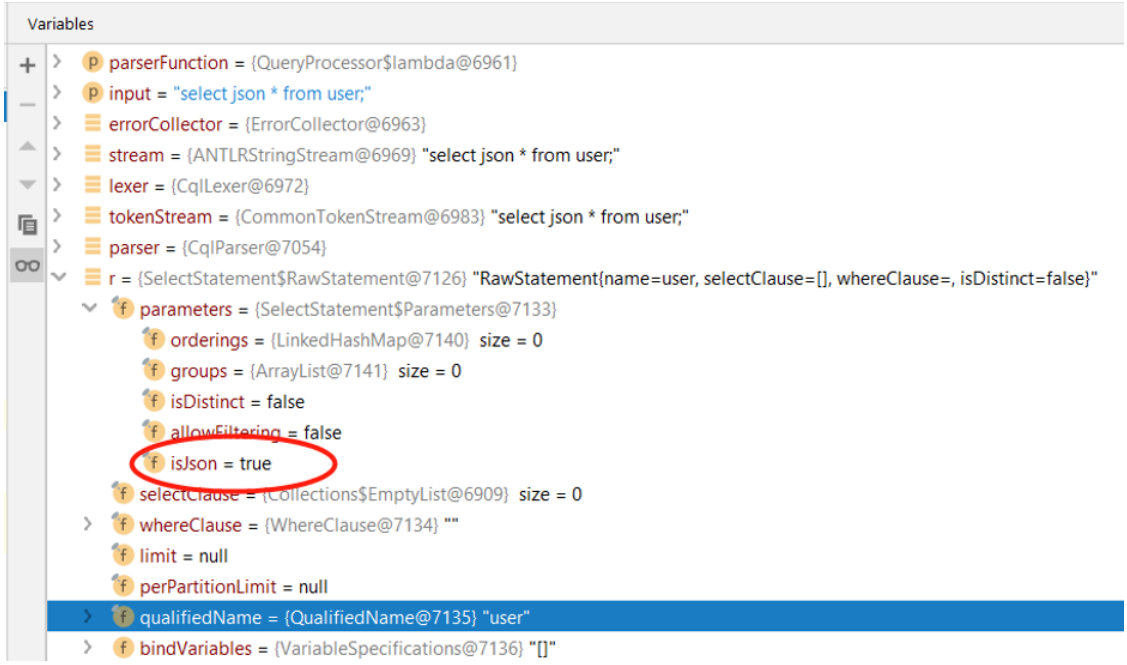
- `tokenStream` groups chars as "select", " ", "json", " ", "*", " ", "from", " ", "user", ";", ""


Variables
parserFunction = {QueryProcessor$lambda@6961}
input = "select json * from user;"
errorCollector = {ErrorCollector@6963}
stream = {ANTLRStringStream@6969} "select json * from user;"
lexer = {CqlLexer@6972}
tokenStream = {CommonTokenStream@6983} "select json * from user;"
channel = 0
tokenSource = {CqlLexer@6972}
tokens = {ArrayList@6986} size = 11
0 = {CommonToken@7001} "[@0,0:5='select',<130>,1:0]"
1 = {CommonToken@7002} "[@1,6:6=' ',<184>,channel=99,1:6]"
2 = {CommonToken@7003} "[@2,7:10='json',<89>,1:7]"
3 = {CommonToken@7004} "[@3,11:11=' ',<184>,channel=99,1:11]"
4 = {CommonToken@7005} "[@4,12:12='*',<207>,1:12]"
5 = {CommonToken@7006} "[@5,13:13=' ',<184>,channel=99,1:13]"
6 = {CommonToken@7007} "[@6,14:17='from',<72>,1:14]"
7 = {CommonToken@7008} "[@7,18:18=' ',<184>,channel=99,1:18]"
8 = {CommonToken@7009} "[@8,19:22='user',<154>,1:19]"
9 = {CommonToken@7010} "[@9,23:23=';',<200>,1:23]"
10 = {CommonToken@7011} "[@10,0:0='<no text>',<-1>,0:-1]"
lastMarker = 0
p = 0
range = -1

- `parser`

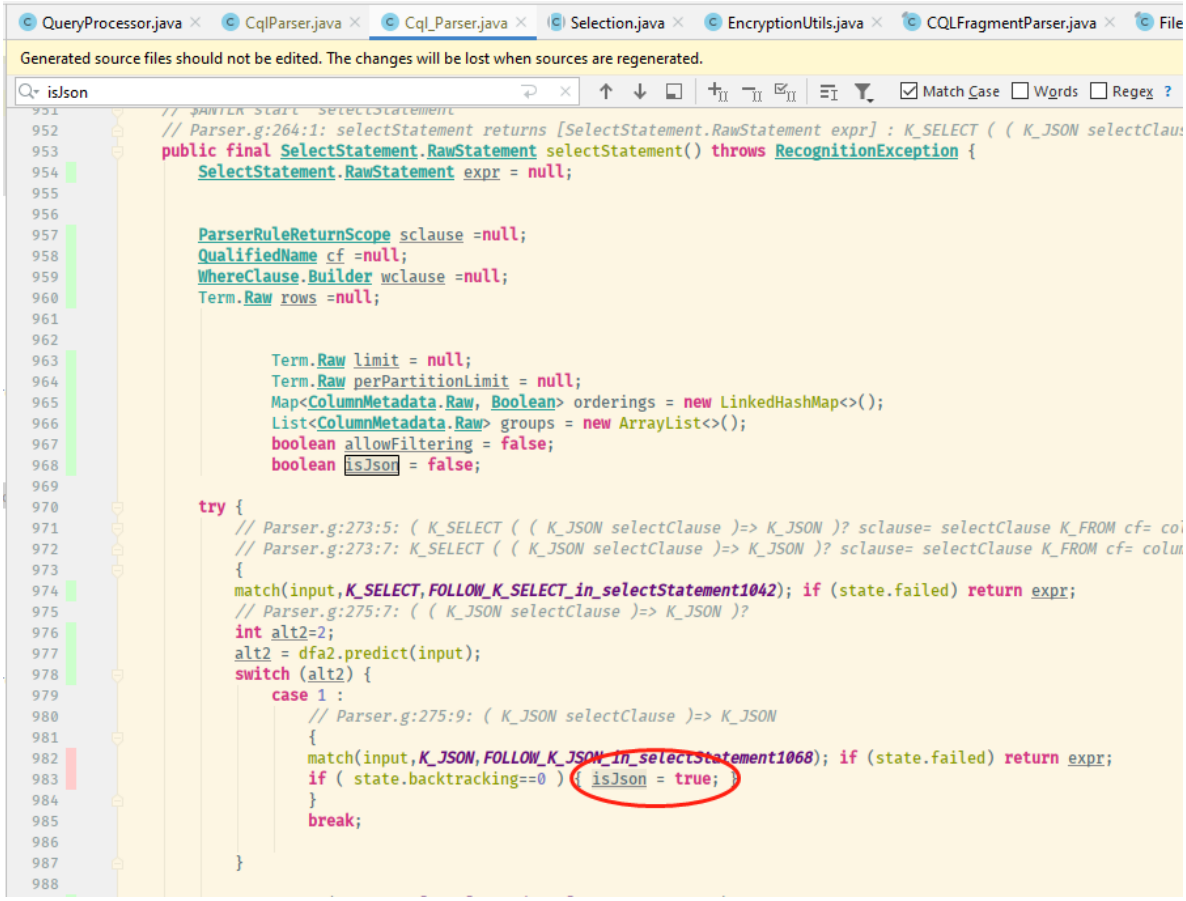


- `r`



So, finally, we find the `isJson` variable coming from the construction of `r`.

Unfortunately, the `R r = parserFunction.parse(parser);` statement leads us to a generated file, which should not be edited.



Here is what we plan to edit. First, the `plain_json` statement should go to the same branch as the `json` feature so we can make it `isJson = true`. Then, we need to create a new variable `isPlainJson` to indicate if the current `isJson` is a `json` or a `plain_json`. This new variable will be added in the two lines mentioned above:

```
if (!isPlainJson && !columnName.equals(columnName.toLowerCase(Locale.US)))
        columnName = "\"" + columnName + "\"";
```

If we are querying a `plain_json`, the extra set of double quotes will not be added, as expected.

3. Problem for now

We are not able to find how this file is generated. So we cannot move on to edit this file and solve the issue. Now, we are communicating with the team to learn more about the source code and to make sure we are on the right track. It seems pretty correct for us, but we are not confident enough when we arrives at a generated file. We are not able to submit a satisfied pull request for now.

4. Next Steps

We are quite interested in solving this problem as this is a new feature that can help many people to use Cassandra data out of the Cassandra ecosystem. We are planning to finish the issue after class.

1. We need to verify with the development team to make sure we are on the right track.
2. We are going to use the help to add a `plain_json` feature in Cassandra.
3. We need more time to write test cases to make sure the `plain_json` feature works well under the current code framework.