Pull Request progress summary

1. Issue Details

The issue we are working on is <u>Cassandra-12760: SELECT JSON "firstName" FROM ... results in </u>{""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:

```
1 [json]
2 -----
3 {"id": "b", "firstName": "Bill"}
```

While the current output is:

```
1 [json]
2 -----
3 {"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"

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 <u>rowToJson</u>. We use a form to record all the methods called when the <u>select json</u> * <u>from user statement</u> is executed, from the beginning to <u>rowToJson</u>.

Here is the list.

4	Α	В	С	D	E
1	java folder	java file	method	line	statement
2	concurrent	SEPWorker	void run()	119	task.run()
	concurrent	AbstractLocalAwareEx	void run()	165	result = callable.call();
3		ecutorService			
4					
5	transport	Message	void channelRead(ctx, request)	630	processRequest(ctx, request);
6	transport	Message	void processRequest(ctx, request)	725	response = request.execute(qstate,
	transport	Message	Response execute(queryState, queryStartNanoTime,	253	response = execute(queryState,
7			traceRequest)		queryStartNanoTime, shouldTrace);
	transport	QueryMessage	Message.Response execute(state, queryStartNanoTime,	108	Message.Response response =
		extends	traceRequest)		ClientState.getCQLQueryHandler().process(query,
8		Message.Request			state, options, getCustomPayload(),
	cq13	QueryProcessor	ResultMessage process(query, state, options,	233	process(query, state, options, queryNanoTime);
			customPayload, queryStartNanoTime) [query: "select json		
9			• from user;"]		
	cq13	QueryProcessor	ResultMessage process(queryString, queryState,	239	CQLStatement prepared =
			options, queryStartNanoTime) [queryString: "select json		getStatement(queryString,);
10			• from user;"]		
			CQLStatement getStatement(queryStr, clientState)	516	CQLStatement.Raw statement =
11					parseStatement(queryStr);
			CQLStatement.Raw parseStatement(queryStr)	546	CQLFragmentParser.parseAnyUnhandled(CqlPars
12		2015			er::query, queryStr);
_		CQLFragmentParser	<r> R parseAnyUnhandled(CQLParserFunction<r></r></r>		
13			parserFunction, input)	500	
14				523	statement.prepare(clientState);
				247	processStatement(prepared, queryState,
15	cq13	QuervProcessor	D	216	options, queryStartNanoTime);
16	cqis	QueryProcessor	ResultMessage processStatemetn(statement,	210	result = statement.execute(queryState, options,
10	statements	SelectStatement	queryState, options, queryStateNanoTime)	250	queryStartNanoTime); execute(??)
17	Statements	Selectstatement	ResultMessage.Rows execute(state, options, queryStartNanoTime)	230	execute(ff)
17	statements	SelectStatement	ResultMessage.Rows execute(???????)	402	msg = processResults(page, options, selectors,
18	Statements	Sciectstatement	Resultiviessage.nows execute(!!!!!!!)	402	nowInSec. userLimit):
10	statements	SelectStatement	ResultMessage.Rows processResults(partitions,	425	ResultSet rset = process(partitions, options,
19	Statements	bereetstatement	options, selectors, nowInSec, userLimit)	423	selectors, nowInSec, userLimit);
,,,	statements	SelectStatement	ResultSet process(partitions, options,	785	ResultSet cqlRows = result.build();
20	Statements	ocicetotateen	selectors.notInSec. userLimit)	,03	nesuriset eqinows = resurisburia(),
	cql3/selection	ResultSetBuilder	ResultSet build()	156	resultSet.addRow(getOutputRow());
22	-4-2/22:22:011		List <bytebuffer> getOutputRow()</bytebuffer>	169	selectors.getOutputRow();
_	cgl3/selection	Selection	Selectors newSelectors(options)	458	rowToJson(current, options.getProtocolVersion(),
23	,				metadata, orderingColumns);
	cql3/selection	Selection	List <bytebuffer> rowToJson(row, protocolVersion,</bytebuffer>		,
24	' '		metadata, orderingColumns)		

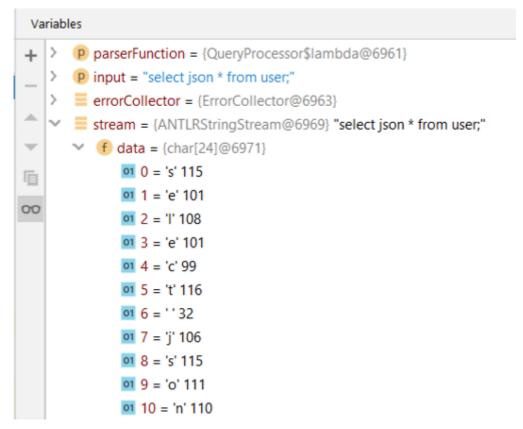
2.3. Parse method for the SelectionStatement

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

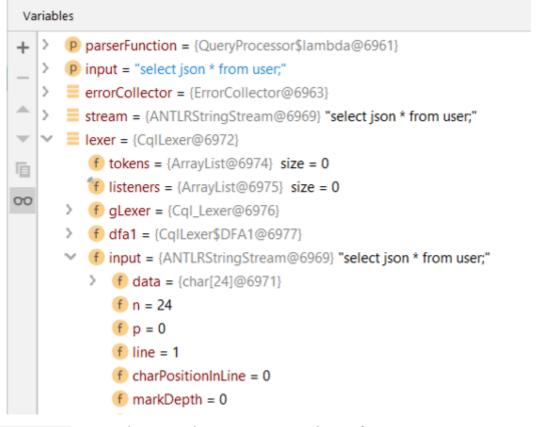
```
1 /**
    * Just call a parser method in {@link CqlParser} - does not do any error
    handling.
 3
    */
    public static <R> R parseAnyUnhandled(CQLParserFunction<R> parserFunction,
    String input) throws RecognitionException {
 5
        // Lexer and parser
 6
        ErrorCollector errorCollector = new ErrorCollector(input);
 7
        CharStream stream = new ANTLRStringStream(input);
 8
        CqlLexer lexer = new CqlLexer(stream);
 9
        lexer.addErrorListener(errorCollector);
10
11
        TokenStream tokenStream = new CommonTokenStream(lexer);
        CqlParser parser = new CqlParser(tokenStream);
12
13
        parser.addErrorListener(errorCollector);
14
15
        // Parse the query string to a statement instance
16
        R r = parserFunction.parse(parser);
17
18
        // 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
19
    here.
        errorCollector.throwFirstSyntaxError();
20
21
22
        return r;
23
    }
```

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
      P parserFunction = {QueryProcessor$lambda@6961}
       p 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;"
佪
          f channel = 0
00
       f 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]"
          f lastMarker = 0
          fp = 0
          f range = -1
```

parser

```
Variables
       errorCollector = {ErrorCollector@6963}
       stream = {ANTLRStringStream@6969} "select json * from user;"
       lexer = {CqlLexer@6972}
      tokenStream = {CommonTokenStream@6983} "select json * from user;"
   parser = {CqlParser@7054}
       f gParser = {Cql_Parser@7069}
佪
          input = {CommonTokenStream@6983} "select json * from user;"
00
             f channel = 0
          f tokenSource = {CqlLexer@6972}
          f 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]"
             f lastMarker = 0
             fp = 0
             f range = -1
         f state = {RecognizerSharedState@7070}
```

r

```
Variables
+ > parserFunction = {QueryProcessor$lambda@6961}
   p input = "select json * from user;"
   > = errorCollector = {ErrorCollector@6963}
  > stream = {ANTLRStringStream@6969} "select json * from user;"
▼ |> |= lexer = {CqlLexer@6972}
   > = tokenStream = {CommonTokenStream@6983} "select json * from user;"
    parser = {CqlParser@7054}
      = r = {SelectStatement$RawStatement@7126} "RawStatement{name=user, selectClause=[], whereClause=, isDistinct=false}"
      f parameters = {SelectStatement$Parameters@7133}
             f orderings = {LinkedHashMap@7140} size = 0
             f groups = {ArrayList@7141} size = 0
             f isDistinct = false
              f) isJson = true
                           {collections$EmptyList@6909} size = 0
          f whereClause = {WhereClause@7134} ""
          f limit = null
          f perPartitionLimit = null
          number | qualifiedName = {QualifiedName@7135} "user"
      bindVariables = {VariableSpecifications@7136} "[]"
```

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.

```
💿 QueryProcessor.java 🗴 🏮 CqlParser.java 🗴 🕲 Cql_Parser.java 🗴 🕲 Selection.java 🗴 🕲 EncryptionUtils.java 🗴 🕲 CQLFragmentParser.java 🗡 📵 File
Generated source files should not be edited. The changes will be lost when sources are regenerated.
                                                                    × ↑ ↓ □ †<sub>II</sub> ¬<sub>II</sub> ⊠<sub>II</sub> | □ ▼  Match Case □ Words □ Regex ?
Q+ isJson
                  // Parser.g:264:1: selectStatement returns [SelectStatement.RawStatement expr] : K_SELECT ( ( K_JSON selectClaus
 952
                 public final SelectStatement RawStatement selectStatement() throws RecognitionException {
 953
 954
                      SelectStatement .RawStatement expr = null;
 955
 956
                      ParserRuleReturnScope sclause =null;
 957
                      QualifiedName cf =null;
 958
                      WhereClause.Builder wclause =null;
 959
 960
                      Term. Raw rows = null;
 961
 962
                               Term. Raw limit = null:
 963
 964
                               Term.Raw perPartitionLimit = null;
 965
                               Map<<u>ColumnMetadata.Raw, Boolean</u>> orderings = new LinkedHashMap<>();
                               List<ColumnMetadata.Raw> groups = new ArrayList<>();
boolean allowFiltering = false;
boolean isJson = false;
 966
 967
 968
 969
 970
                           (///Parser.g:273:5: (K_SELECT ( (K_JSON selectClause )=>K_JSON )? sclause= selectClause K_FROM cf= coi
//Parser.g:273:7: K_SELECT ( (K_JSON selectClause )=>K_JSON )? sclause= selectClause K_FROM cf= colum
 971
 972
 973
 974
                           match(input, K_SELECT, FOLLOW_K_SELECT_in_selectStatement1042); if (state.failed) return expr;
 975
                               Parser.g:275:7: ( ( K_JSON selectClause )=> K_JSON )
 976
                           int alt2=2;
                           alt2 = dfa2.predict(input):
 977
                           switch (alt2) {
 978
                               case 1
                                     // Parser.g:275:9: ( K_JSON selectClause )=> K_JSON
 980
 981
                                    match(input, K JSON, FOLLOW K JSON in selectstatement1068); if (state.failed) return expr;
 982
 983
                                    if ( state.backtracking==0 ) { isJson = true;
 985
 986
 987
 988
```

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 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.