CSCE 608
Database Systems
Dr. Jianer Chen

Project 2 Tiny-SQL Interpreter

Robert Quan
Xien Thomas
Vishakh Shukla

INTRODUCTION:

In this project, we constructed and implemented a SQL interpreter called Tiny-SQL, that can run SQL queries from a user input or a test file that has a list of queries to be executed in order. The project was written in Java and included the StorageManager library for LINUX. Given a predefined Tiny-SQL grammar from class, we designed the queries to be activated and correctly executed based on this unique grammar. One of the challenges presented in this project was correctly executing the sub queries that may be presented in the SQL statements. The queries are also sent to an output text file (sql_output.txt) ment for that specific test run and block IOs. These are the outputs from the test or the individual user statements and it contains a nice formatted output of the relation we are working with and the total time is recorded and converted into seconds for evaluation purposes.

The general structure of the project was to implement in the following:

- An **Interface** that enables the user to interact with the interpreter. The user has an option to run individual Tiny-SQL statements or read from a text file.
- A **Parser** that accepts the input Tiny-SQL query and converts it into a parse tree.
- A **Logical Query Plan Generator** which converts the parse tree into a logical query plan. We wrote a tree data structure to contain the parsed logical plan and optimized the selection operations in the tree to decrease the necessary disk reads.
- A **Physical Query Plan Generator** which converted the optimized logical query plans into executable physical query plans. For the Physical optimizations, we have implemented the One Pass algorithms (for projection, selection, product, cartesian join, duplicate elimination, and sorting) and the Two Pass Sort Merge Join algorithms (natural join, duplicate elimination, and sorting).
- A set of **Subroutines** that implemented a variety of data query operations that used the released library StorageManager to simulate the computer disks and memory.

An option to use a predefined parser was allowed, such was LEX, YACC, or JavaCC, but we decided to write our own parser to better understand with the grammar and execution of the queries and their subroutines. There are 9 Java classes, since the program was written in an object oriented structure, that are used in the parsing, optimizing, execution of subqueries, and return of the input queries. The Main.java file contains the interface code, which allows custom user query input, and creates the Core object which is used in distributing the parsed and optimized queries to the Storage Manager library. The Storage Manager library is used to simulates a fictitious disk containing 100 tracks of unlimited length and a small fictitious memory of 10 data blocks. This Storage Manager has a speed of 50M memory and 300M relation, where the memory capacity is 10 blocks and each relation holds 60 tuples.

In the end, we completed the project by using a collaborative system based on Cloud 9 online IDE with a LINUX environment with the Java compiler and a repository storage on GitHub.

CHANGES TO LIBRARY:

We made a few changes to the Storage Manager library that were most for visual aesthetic. One of the biggest changes was to change the tabulation size on the column header and the tabulation size of the entries when the relations were printed out. This gave the output a nicer visualization of the returned queries.

SOFTWARE ARCHITECTURE:

The project is divided into three main Java classes and six other subclasses. These main classes include the following:

- **Main:** The user interface and main file exists here and is used to create the Core object which executes the input queries from either the Test file or the User input.
- **Parser**: The Core will send the queries to the Parser where the Parser will correctly optimize and include the subqueries to be parsed. This object imitates the Logical Query Plan generator.
- **Core**: After the Parse has optimized and returned the logical query plan, the core is in charge of executing the Physical query plan generator. The Core has also optimized the natural join operators by using the Two-Pass Multiway Merge-Sort Algorithm.

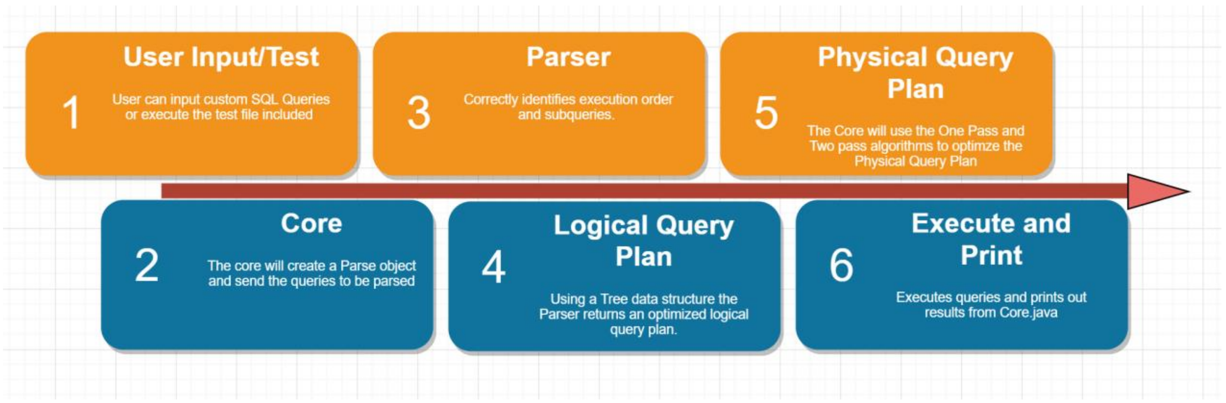
DATA STRUCTURES:

The data structures that were used in this project were implemented as **Heap.java (**which is used for the execution order of the physical parse tree), **TreeNode.java (**which is the object that contains the logical query plan and more information) **,** and **SubTreeNode.java (**which is the actual data structure that holds the parsed logical query plan)**.**

ALGORITHMS:

The Algorithms that were implemented between the Parse and Core to optimize the logical and physical query plans were: The One-pass Selection operation, The Two-Pass Multiway Merge-Sort for the natural joins where we implement heap structures.

The other files that are helper files for the whole program include Argument.java, Builder.java, Tuple_with_position.java. These are classes which aid in the execution and implementation of our program. All of these files implement in some way, the library files from the StorageManager class which was given to use for the project. It is in Core where most of the Storage Manager objects are manipulated and created.


User Input/Test
1
User can input custom SQL Queries or execute the test file included
Core
2
The core will create a Parse object and send the queries to be parsed
Parser
3
Correctly identifies execution order and subqueries.
Logical Query Plan
4
Using a Tree data structure the Parser returns an optimized logical query plan.
Physical Query Plan
5
The Core will use the One Pass and Two pass algorithms to optimze the Physical Query Plan
Execute and Print
6
Executes queries and prints out results from Core.java

PARSING THE QUERIES:

In this section, we parse each query line by line by a given string. Our class called “Parser” is used to parse each query and set flags based on the context of the query. Our main class “core” uses parser objects to know which type of code to execute and also what type of data should be used in a query execution. The parser is the bond that keeps the program running smoothly. Usually, if anything wrong with the format of the query the parser will catch the cases. Parser became the most important class when it came to bugs in the test cases; sometimes it would not be possible to understand the flow while debugging without the class. To order to fully describe the parser and understand to write it, we had to deconstruct a query and think how would you describe a query logically.

Queries from the test cases are parsed line by line and word by word. We initially try to match the first keyword in a query in a series of if statements. Out of the possible outcomes, you could either “Create Table”, “Insert into”, “Select”, “Delete”, or “Drop”. Each if statement begins to find these 5 options by comparing the first element in the string. We used “equalsIgnoreCase” method for this comparison. Once one of the options are picked then extra information is given; for example, when a query starts with the word “create” then the program would display “creating table,” onto the terminal. The same logic goes for the other options and all of them show that they completed an action at the end of the if statement.

The second level of the parse could be much different from each of the 5 initial options since they are all different directions. When creating a table you could check if the user had entered the word “Table” and gave the table a name to later on append this name to a list of table names, but most importantly you need to copy the attribute names and their data type. When creating the attributes, essentially one list is being defined by types; if an attribute “score” is declared, then it will be labeled a type “int”.

When inserting values into the table, first the program need check for the right format and then append the values in order of their attributes. We implemented a substring system where we catch important keys, like “value”, and grab everything in parentheses after it then parses everything again. To this point, the string we created is a comma splice value, so we grab each value separated by a comma and append the value to a list; this is similar for what we did when creating a table. Using "create table" and "insert into", help us realize how important the parentheses and commas were. Without the punctuations, it would be a lot harder to parse the tables.

The last three options are “Drop”, “Delete”, and “Select”. Dropping a table only requires for the table name to be appended where deleting a table requires the table to delete from a particular where clause. Deleting where scores equal to NaN (Not a Number) is one example where the parse will know the condition of the where clause and if their attribute is equal to NaN. Selecting a table is the most crucial part of the parser since that many things can derive from the select option. After a select statement the query can continue with four different clauses: From, Order, Where, and Distinct. Each clause has a different outcome depending on the query. In the implementation, flags were raised if different clauses were in the query.

By exhausting the different outcomes of a query we were able to make a class that was able to parse as much information from the queries to be later be used in the “core” of our project. The parser should only extract enough data for the core class to use for their implementation and for optimization.

DISCUSSION ON OPTIMIZATION TECHNIQUES:

- **Logical Query Plan Optimization:** For this optimization, we decided to implement the one pass algorithms for projection, selection, product, join, duplicate elimination, and sorting. This was done by researching algorithms in the book, and pushing the selection statement down to the relationship before any join would be conducted to optimize the disk IOs.

- **Physical Query Plan Optimization:** By implementing the two-pass algorithms for natural join, duplicate elimination, and sorting, we notice around a 13% - 20% decrease in Disk IOs and in the time taken for the SELECT query to activate. The Two-Pass Merge Sort algorithm from the class was implemented for the Natural Join operator.

The Cross join operator was implemented, but there was no good algorithm for large relations that was optimized. This can be seen in the test file given by the professor where the Cross-Join will be the longest operation due to its recursive creation of large amounts of tuples.

VALIDATING CORRECTNESS THROUGH TESTING:

We were supplied a set of data and queries that was meant to test the correctness of the interpreter. The data included 11 tables and as much as 60 tuples in one table. After studying the correctness of the Grammar for the TinySQL, we noticed there were queries that were not in accordance with the grammar. For these queries, our interpreter will be able to return a relation but not in accordance with statements outside the TinySQL grammar. For all other legal statements, our interpreter returns each relation with precision and correctness.

EX: **GOOD**: SELECT DISTINCT * FROM course
BAD: SELECT * FROM courses WHERE id NOT 3
The “NOT” keyword was not included in the TinySQL grammar, thus not implemented in the project.
Operations that are valid in our Interpreter:
CREATE TABLE, DROP TABLE, INSERT, DELETE, SELECT, WHERE, AND, OR, (,),+, <. >,=, ORDER.

EXPERIMENTAL RESULTS:

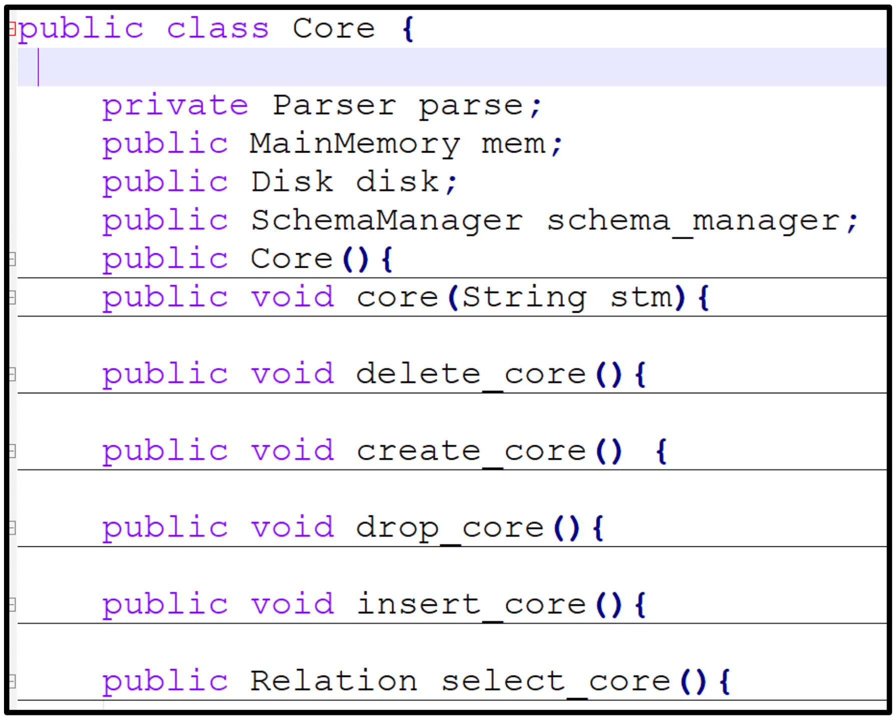
The Storage Manager library included a function, disk.getDiskIOs, which returned the disk count each time we executed a query. Based on this number we decided to run a few tests that included inputting 200 tuples from one of the members project 1. In Table 1, there is a list of different tuple execution statements that were recorded when testing the different algorithms and the disk IOs. The Non-Optimized are the statements when run with the One Pass algorithms and the Optimized are the Two Pass algorithms.

Number of Tuples	Execution Type	Optimized Time executed	Optimized Block IOs	Non-Optimized Block IOs + Time
20	SELECT * FROM course	5 sec	79 ops	92 ops, 7 sec
60	SELECT * FROM course	16 sec	235 ops	430 ops, 27 sec
100	SELECT * FROM course	27 sec	399 ops	511 ops, 36 sec
20	SELECT DISTINCT grade FROM course	11 sec	163 ops	194 ops, 14 sec
60	SELECT * FROM r, s WHERE r.a = s.a AND r.b = s.b	13 sec	184 ops	201 ops, 15 sec
100	SELECT * FROM r, s, t WHERE r.a = s.a AND r.b = s.b AND s.c = t.c	39 sec	723 ops	892 ops, 46 sec

For the 200 Tuples experiment from Project One, we used one group members Project 1 tuples that he used for his Web interface. The results are the following:

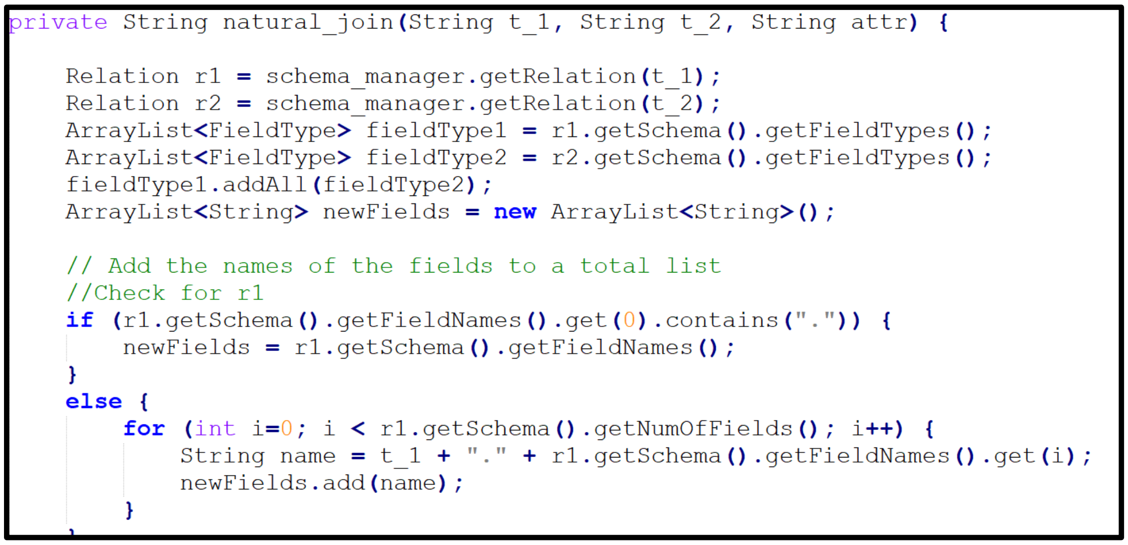
Number of Tuples	Execution Type	Time executed	Block IOs	Non-Optimized Block IOs + Time
200	SELECT * FROM mytable	41 sec	579 ops	712 ops, 57 sec
200	WHERE productName = “kilo”	47 sec	677 ops	745 ops, 61 sec
200	SELECT DISTINCT * FROM mytable	88 sec	1243 ops	1422 ops, 102 sec
200	... WHERE productName = “pluto” AND productPrice = 19	47 sec	677 ops	689 ops, 52 sec
200	... WHERE productPrice > 200	49 sec	697 ops	723 ops, 55 sec

IMAGES:



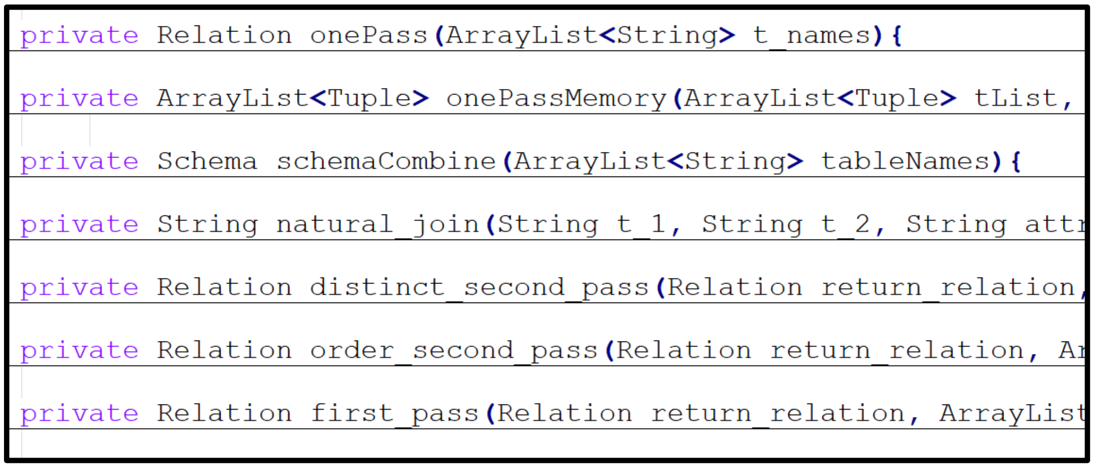
```
public class Core {

    private Parser parse;
    public MainMemory mem;
    public Disk disk;
    public SchemaManager schema_manager;
    public Core(){
    public void core(String stm){

    public void delete_core(){

    public void create_core()  {

    public void drop_core(){

    public void insert_core(){

    public Relation select_core(){
```

Fig 1, Core class functions and variables.



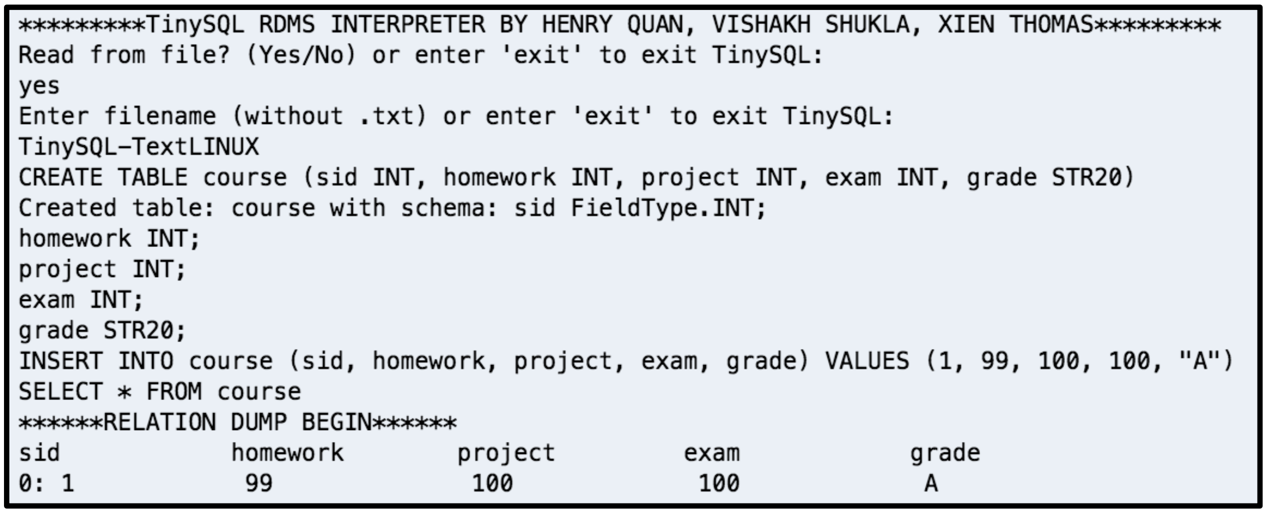
```
private String natural_join(String t_1, String t_2, String attr) {

    Relation r1 = schema_manager.getRelation(t_1);
    Relation r2 = schema_manager.getRelation(t_2);
    ArrayList<FieldType> fieldType1 = r1.getSchema().getFieldTypes();
    ArrayList<FieldType> fieldType2 = r2.getSchema().getFieldTypes();
    fieldType1.addAll(fieldType2);
    ArrayList<String> newFields = new ArrayList<String>();

    // Add the names of the fields to a total list
    //Check for r1
    if (r1.getSchema().getFieldNames().get(0).contains(".")) {
        newFields = r1.getSchema().getFieldNames();
    }
    else {
        for (int i=0; i < r1.getSchema().getNumOfFields(); i++) {
            String name = t_1 + "." + r1.getSchema().getFieldNames().get(i);
            newFields.add(name);
        }
```

Fig 2, Natural Join Optimized for two relations.



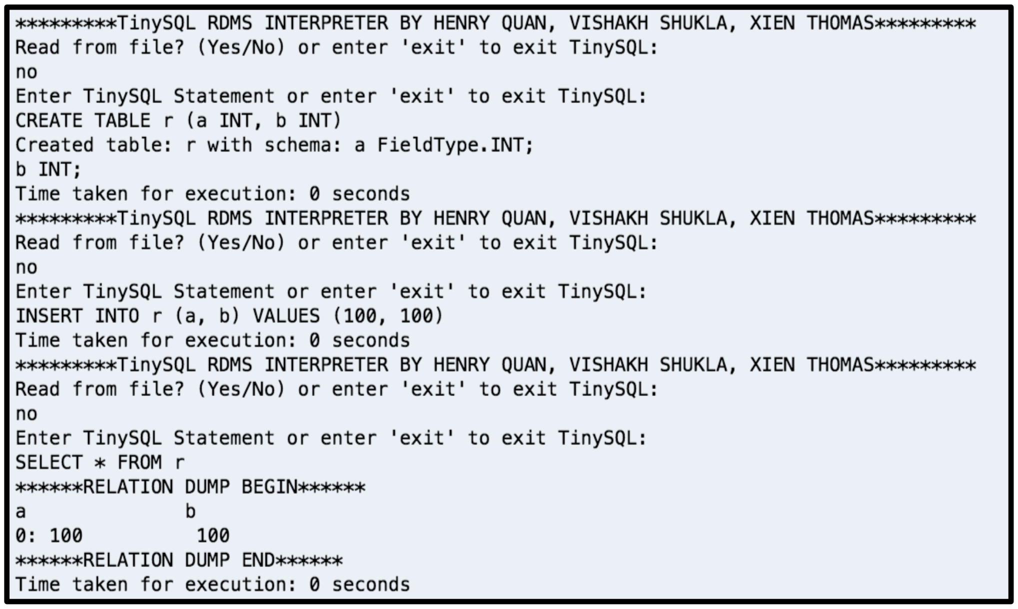
```
private Relation onePass(ArrayList<String> t_names){
private ArrayList<Tuple> onePassMemory(ArrayList<Tuple> tList,
private Schema schemaCombine(ArrayList<String> tableNames){
private String natural_join(String t_1, String t_2, String attr
private Relation distinct_second_pass(Relation return_relation,
private Relation order_second_pass(Relation return_relation, Ar
private Relation first_pass(Relation return_relation, ArrayList
```

Fig 3, List of One pass and Two pass merge algorithm optimization



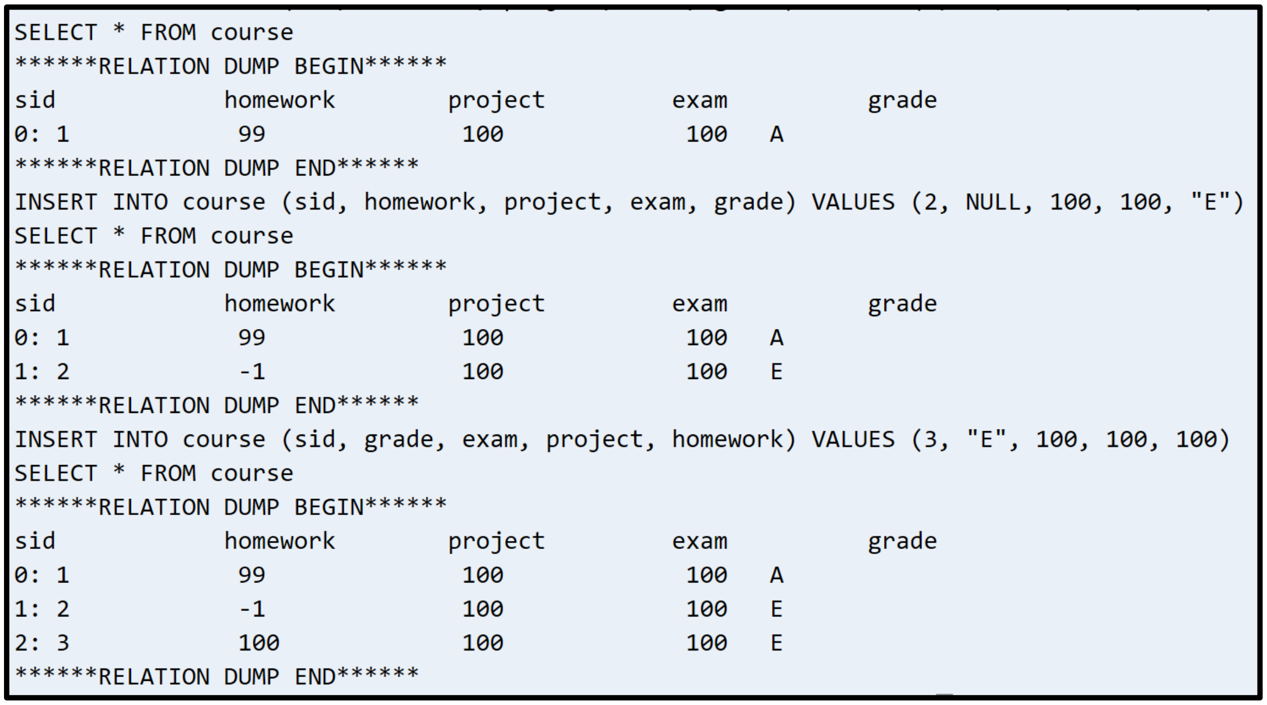
```
*********TinySQL RDMS INTERPRETER BY HENRY QUAN, VISHAKH SHUKLA, XIEN THOMAS*********
Read from file? (Yes/No) or enter 'exit' to exit TinySQL:
yes
Enter filename (without .txt) or enter 'exit' to exit TinySQL:
TinySQL-TextLINUX
CREATE TABLE course (sid INT, homework INT, project INT, exam INT, grade STR20)
Created table: course with schema: sid FieldType.INT;
homework INT;
project INT;
exam INT;
grade STR20;
INSERT INTO course (sid, homework, project, exam, grade) VALUES (1, 99, 100, 100, "A")
SELECT * FROM course
******RELATION DUMP BEGIN******
sid             homework         project          exam             grade
0: 1             99               100              100              A
```

Fig 4, User Interface with Read From File Selected



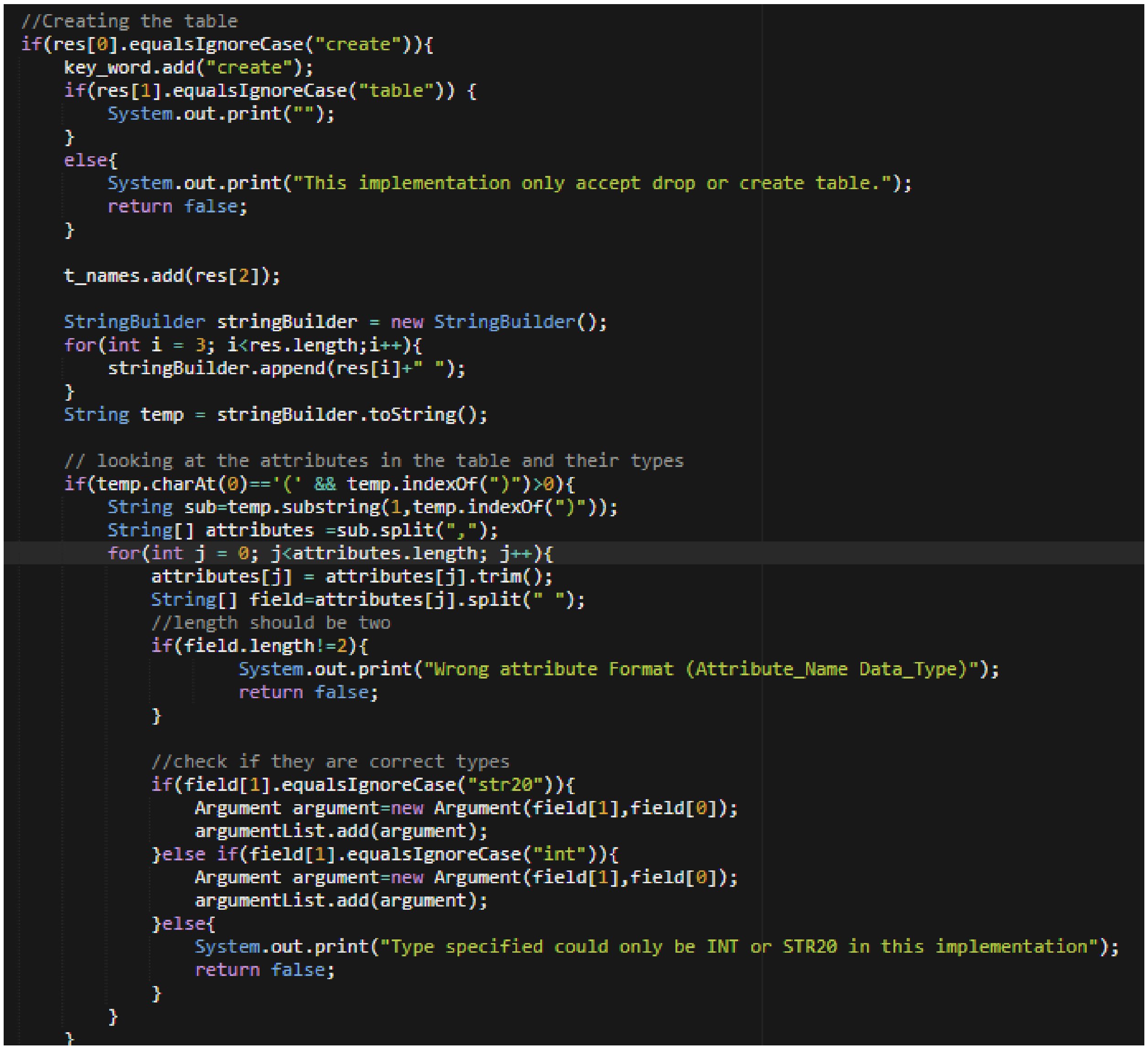
```
*********TinySQL RDMS INTERPRETER BY HENRY QUAN, VISHAKH SHUKLA, XIEN THOMAS*********
Read from file? (Yes/No) or enter 'exit' to exit TinySQL:
no
Enter TinySQL Statement or enter 'exit' to exit TinySQL:
CREATE TABLE r (a INT, b INT)
Created table: r with schema: a FieldType.INT;
b INT;
Time taken for execution: 0 seconds
*********TinySQL RDMS INTERPRETER BY HENRY QUAN, VISHAKH SHUKLA, XIEN THOMAS*********
Read from file? (Yes/No) or enter 'exit' to exit TinySQL:
no
Enter TinySQL Statement or enter 'exit' to exit TinySQL:
INSERT INTO r (a, b) VALUES (100, 100)
Time taken for execution: 0 seconds
*********TinySQL RDMS INTERPRETER BY HENRY QUAN, VISHAKH SHUKLA, XIEN THOMAS*********
Read from file? (Yes/No) or enter 'exit' to exit TinySQL:
no
Enter TinySQL Statement or enter 'exit' to exit TinySQL:
SELECT * FROM r
******RELATION DUMP BEGIN******
a               b
0: 100           100
******RELATION DUMP END******
Time taken for execution: 0 seconds
```

Fig 5, User Interface with Individual Tiny-SQL Statements



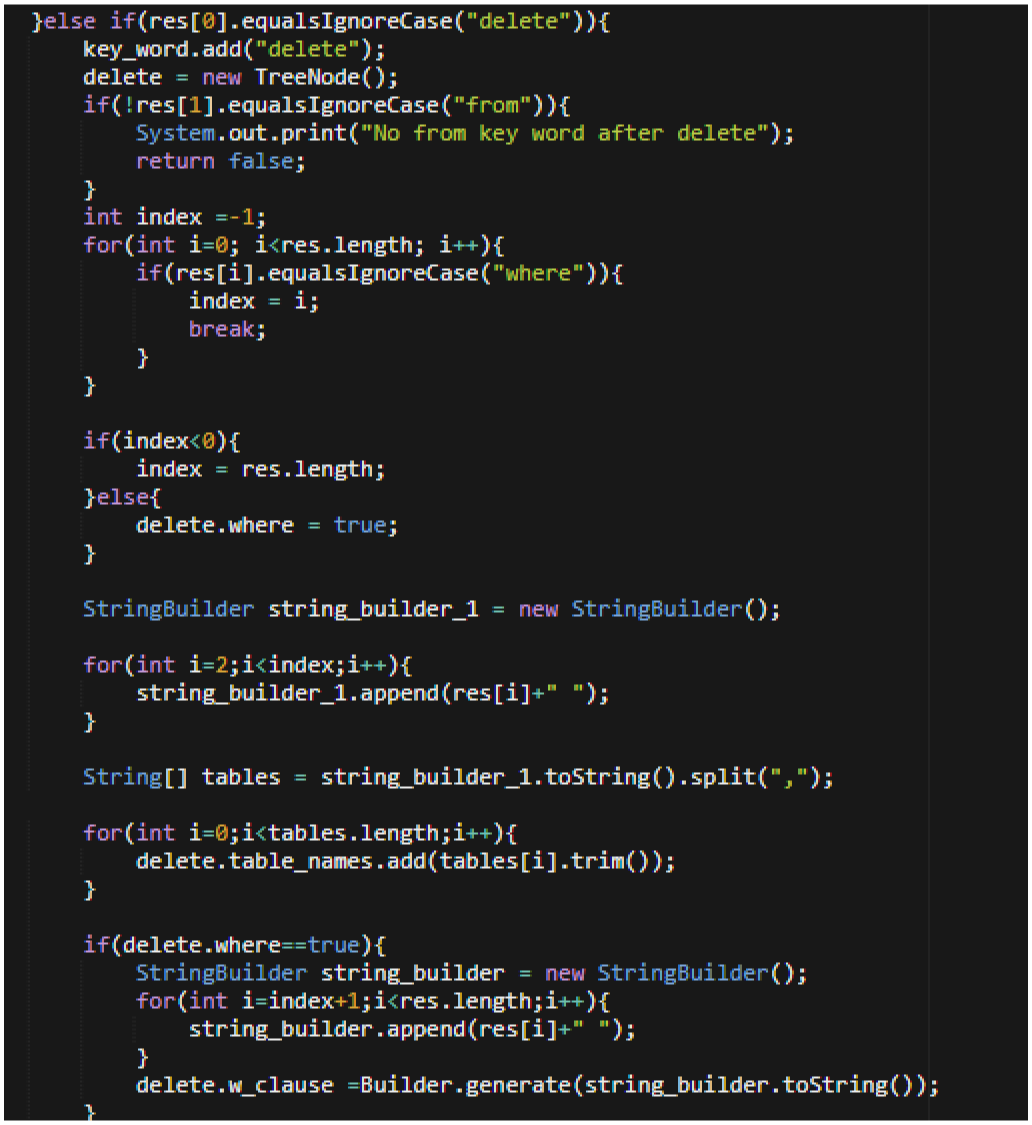
```
SELECT * FROM course
******RELATION DUMP BEGIN******
sid              homework         project          exam             grade
0: 1              99               100              100    A
******RELATION DUMP END******
INSERT INTO course (sid, homework, project, exam, grade) VALUES (2, NULL, 100, 100, "E")
SELECT * FROM course
******RELATION DUMP BEGIN******
sid              homework         project          exam             grade
0: 1              99               100              100    A
1: 2              -1               100              100    E
******RELATION DUMP END******
INSERT INTO course (sid, grade, exam, project, homework) VALUES (3, "E", 100, 100, 100)
SELECT * FROM course
******RELATION DUMP BEGIN******
sid              homework         project          exam             grade
0: 1              99               100              100    A
1: 2              -1               100              100    E
2: 3              100              100              100    E
******RELATION DUMP END******
```

Fig 6, Standard SELECT output from SQL commands, other commands not shown.



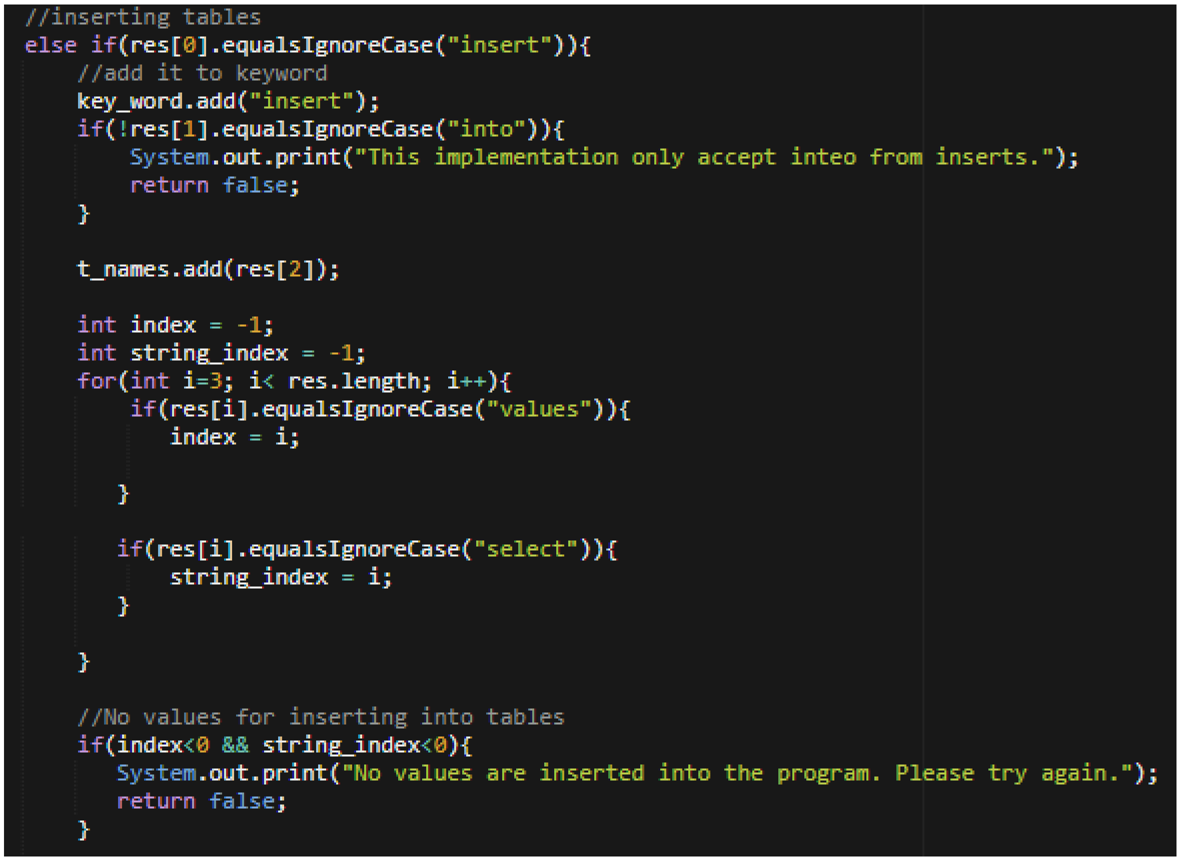
```java
//Creating the table
if(res[0].equalsIgnoreCase("create")){
    key_word.add("create");
    if(res[1].equalsIgnoreCase("table")) {
        System.out.print("");
    }
    else{
        System.out.print("This implementation only accept drop or create table.");
        return false;
    }

    t_names.add(res[2]);

    StringBuilder stringBuilder = new StringBuilder();
    for(int i = 3; i<res.length;i++){
        stringBuilder.append(res[i]+" ");
    }
    String temp = stringBuilder.toString();

    // looking at the attributes in the table and their types
    if(temp.charAt(0)=='(' && temp.indexOf(")")>0){
        String sub=temp.substring(1,temp.indexOf(")"));
        String[] attributes =sub.split(",");
        for(int j = 0; j<attributes.length; j++){
            attributes[j] = attributes[j].trim();
            String[] field=attributes[j].split(" ");
            //length should be two
            if(field.length!=2){
                    System.out.print("Wrong attribute Format (Attribute_Name Data_Type)");
                    return false;
            }

            //check if they are correct types
            if(field[1].equalsIgnoreCase("str20")){
                Argument argument=new Argument(field[1],field[0]);
                argumentList.add(argument);
            }else if(field[1].equalsIgnoreCase("int")){
                Argument argument=new Argument(field[1],field[0]);
                argumentList.add(argument);
            }else{
                System.out.print("Type specified could only be INT or STR20 in this implementation");
                return false;
            }
        }
    }
```

Fig 7, Create table parser in the parser class.



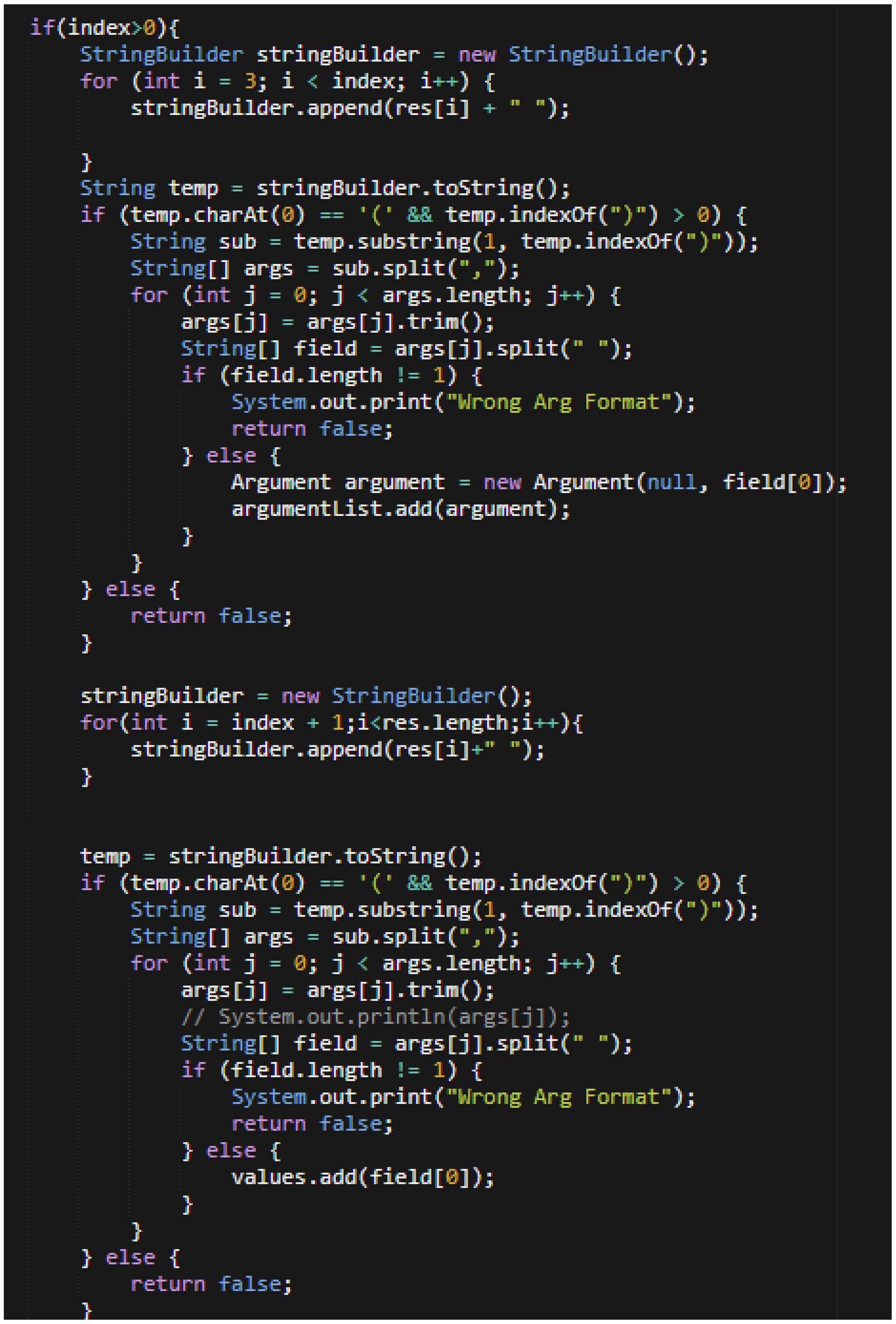
```
}else if(res[0].equalsIgnoreCase("delete")){
    key_word.add("delete");
    delete = new TreeNode();
    if(!res[1].equalsIgnoreCase("from")){
        System.out.print("No from key word after delete");
        return false;
    }
    int index =-1;
    for(int i=0; i<res.length; i++){
        if(res[i].equalsIgnoreCase("where")){
            index = i;
            break;
        }
    }

    if(index<0){
        index = res.length;
    }else{
        delete.where = true;
    }

    StringBuilder string_builder_1 = new StringBuilder();

    for(int i=2;i<index;i++){
        string_builder_1.append(res[i]+" ");
    }

    String[] tables = string_builder_1.toString().split(",");

    for(int i=0;i<tables.length;i++){
        delete.table_names.add(tables[i].trim());
    }

    if(delete.where==true){
        StringBuilder string_builder = new StringBuilder();
        for(int i=index+1;i<res.length;i++){
            string_builder.append(res[i]+" ");
        }
        delete.w_clause =Builder.generate(string_builder.toString());
    }
```

Fig 8, Delete parser from the parser class.



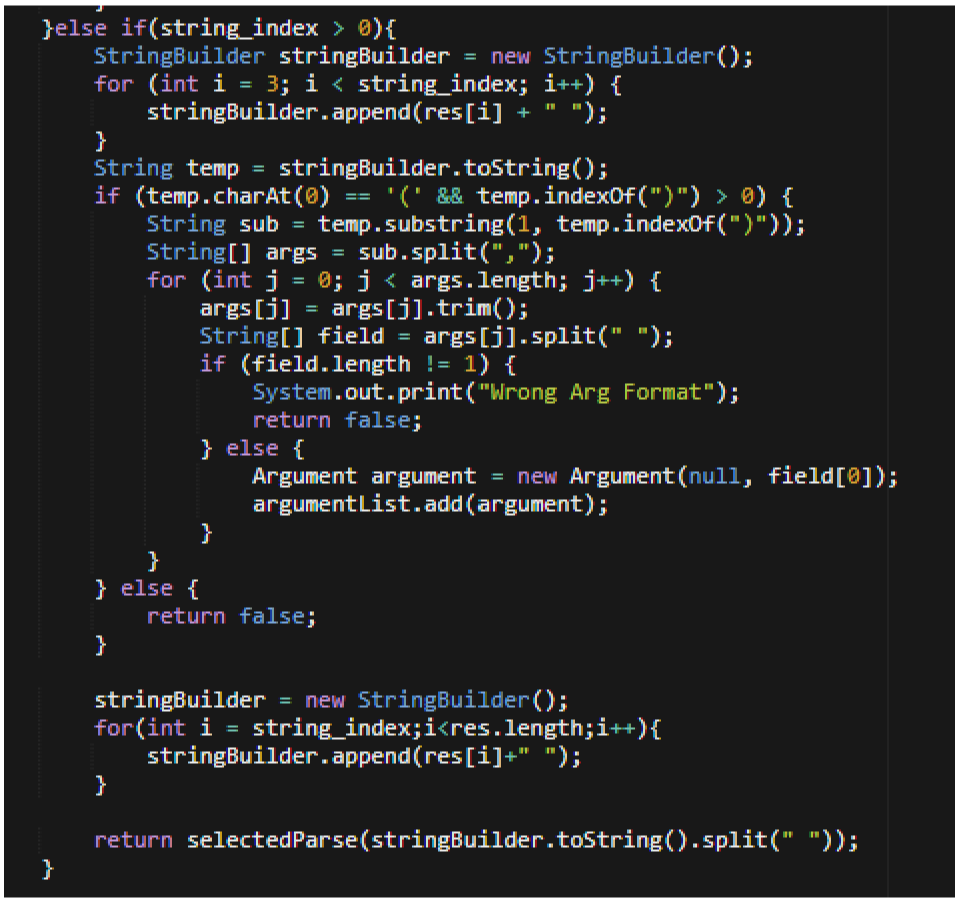
```java
//inserting tables
else if(res[0].equalsIgnoreCase("insert")){
    //add it to keyword
    key_word.add("insert");
    if(!res[1].equalsIgnoreCase("into")){
        System.out.print("This implementation only accept inteo from inserts.");
        return false;
    }

    t_names.add(res[2]);

    int index = -1;
    int string_index = -1;
    for(int i=3; i< res.length; i++){
        if(res[i].equalsIgnoreCase("values")){
           index = i;

       }

       if(res[i].equalsIgnoreCase("select")){
           string_index = i;
       }

    }

    //No values for inserting into tables
    if(index<0 && string_index<0){
       System.out.print("No values are inserted into the program. Please try again.");
       return false;
    }
```

Fig 9, Insert parser (part 1)



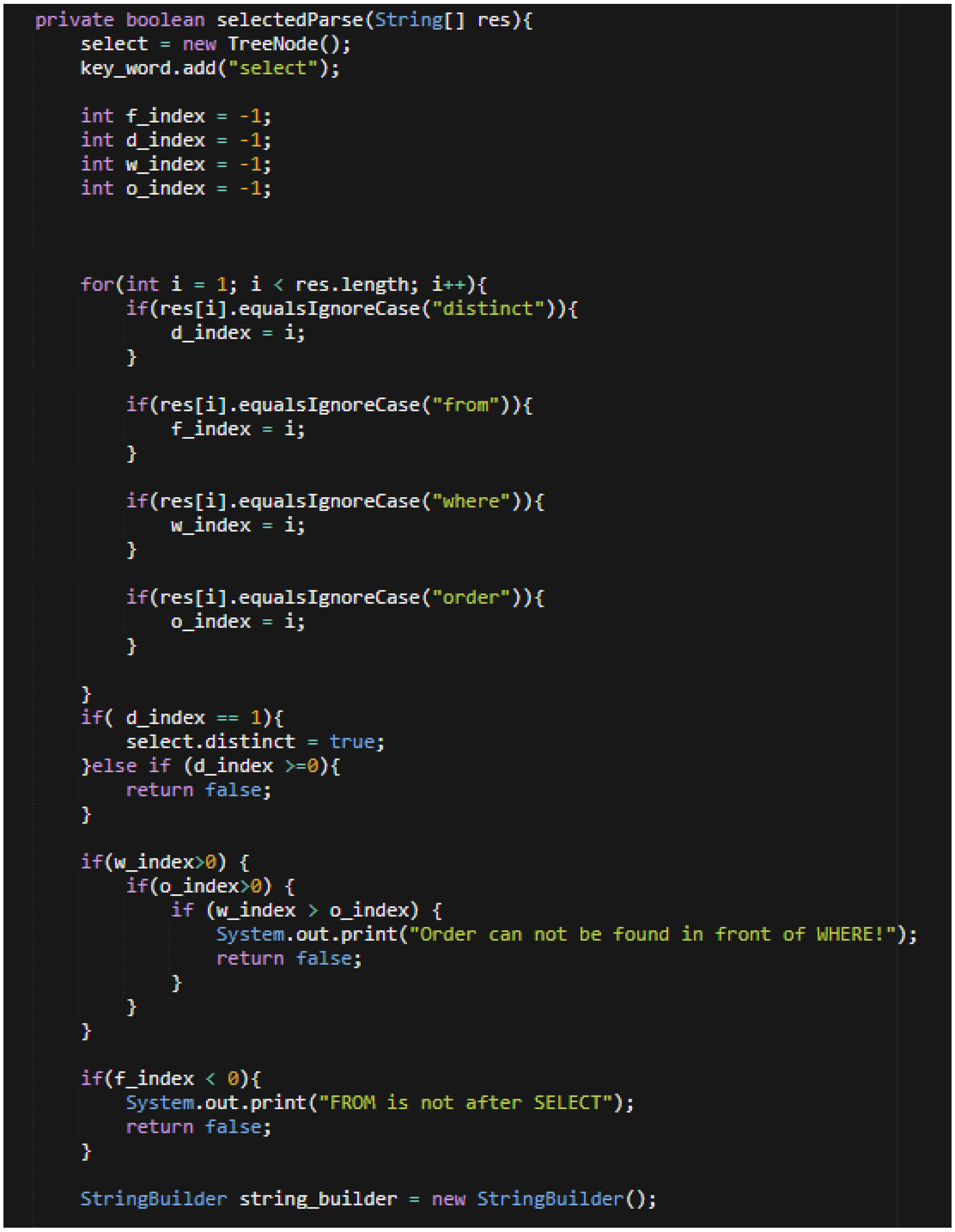
```
if(index>0){
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 3; i < index; i++) {
        stringBuilder.append(res[i] + " ");

    }
    String temp = stringBuilder.toString();
    if (temp.charAt(0) == '(' && temp.indexOf(")") > 0) {
        String sub = temp.substring(1, temp.indexOf(")"));
        String[] args = sub.split(",");
        for (int j = 0; j < args.length; j++) {
            args[j] = args[j].trim();
            String[] field = args[j].split(" ");
            if (field.length != 1) {
                System.out.print("Wrong Arg Format");
                return false;
            } else {
                Argument argument = new Argument(null, field[0]);
                argumentList.add(argument);
            }
        }
    } else {
        return false;
    }

    stringBuilder = new StringBuilder();
    for(int i = index + 1;i<res.length;i++){
        stringBuilder.append(res[i]+" ");
    }


    temp = stringBuilder.toString();
    if (temp.charAt(0) == '(' && temp.indexOf(")") > 0) {
        String sub = temp.substring(1, temp.indexOf(")"));
        String[] args = sub.split(",");
        for (int j = 0; j < args.length; j++) {
            args[j] = args[j].trim();
            // System.out.println(args[j]);
            String[] field = args[j].split(" ");
            if (field.length != 1) {
                System.out.print("Wrong Arg Format");
                return false;
            } else {
                values.add(field[0]);
            }
        }
    } else {
        return false;
    }
```

Fig 10, Insert parse (part 2)



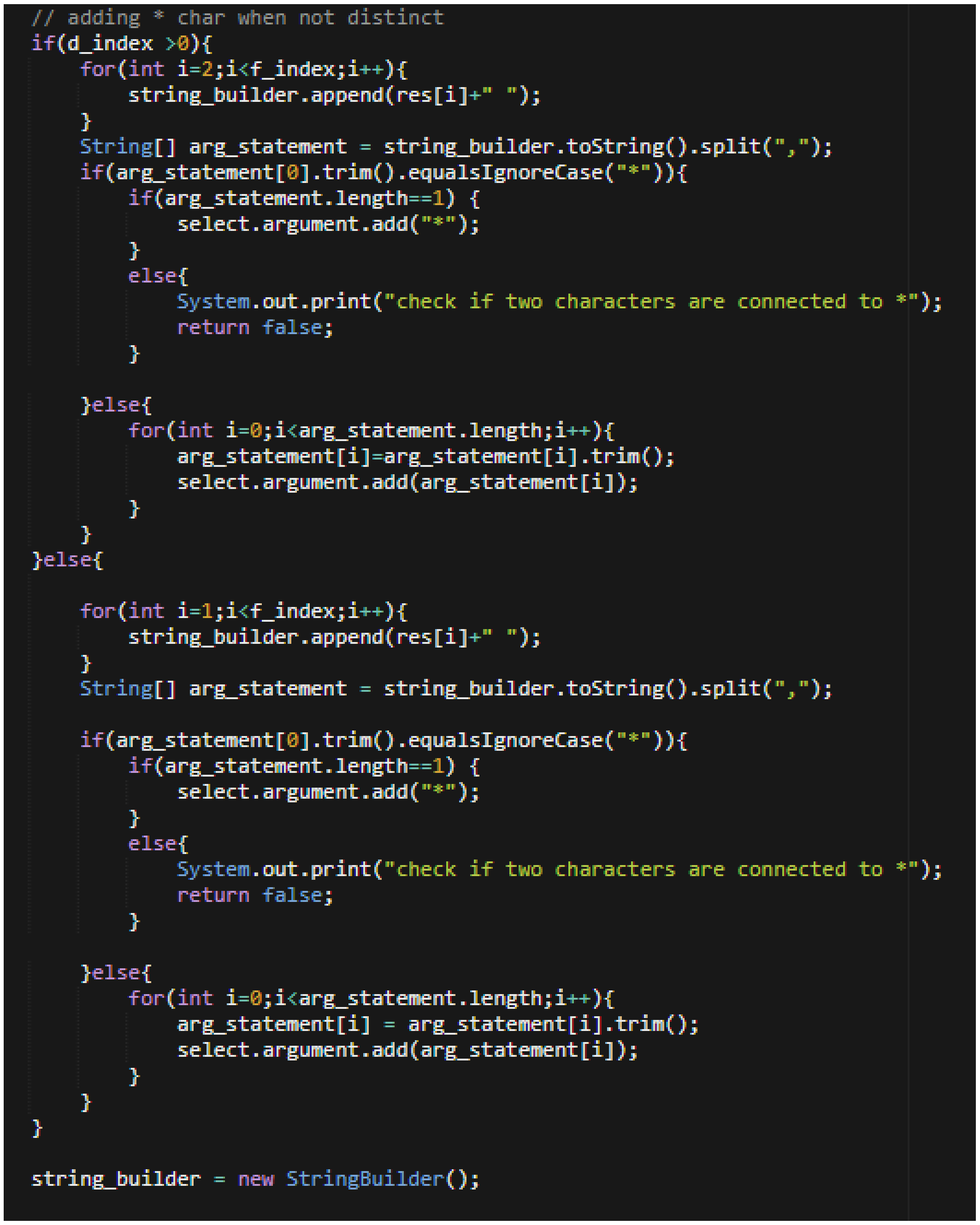
```
}else if(string_index > 0){
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 3; i < string_index; i++) {
        stringBuilder.append(res[i] + " ");
    }
    String temp = stringBuilder.toString();
    if (temp.charAt(0) == '(' && temp.indexOf(")") > 0) {
        String sub = temp.substring(1, temp.indexOf(")"));
        String[] args = sub.split(",");
        for (int j = 0; j < args.length; j++) {
            args[j] = args[j].trim();
            String[] field = args[j].split(" ");
            if (field.length != 1) {
                System.out.print("Wrong Arg Format");
                return false;
            } else {
                Argument argument = new Argument(null, field[0]);
                argumentList.add(argument);
            }
        }
    } else {
        return false;
    }

    stringBuilder = new StringBuilder();
    for(int i = string_index;i<res.length;i++){
        stringBuilder.append(res[i]+" ");
    }

    return selectedParse(stringBuilder.toString().split(" "));
}
```

Fig 11, Insert parse (part 3)



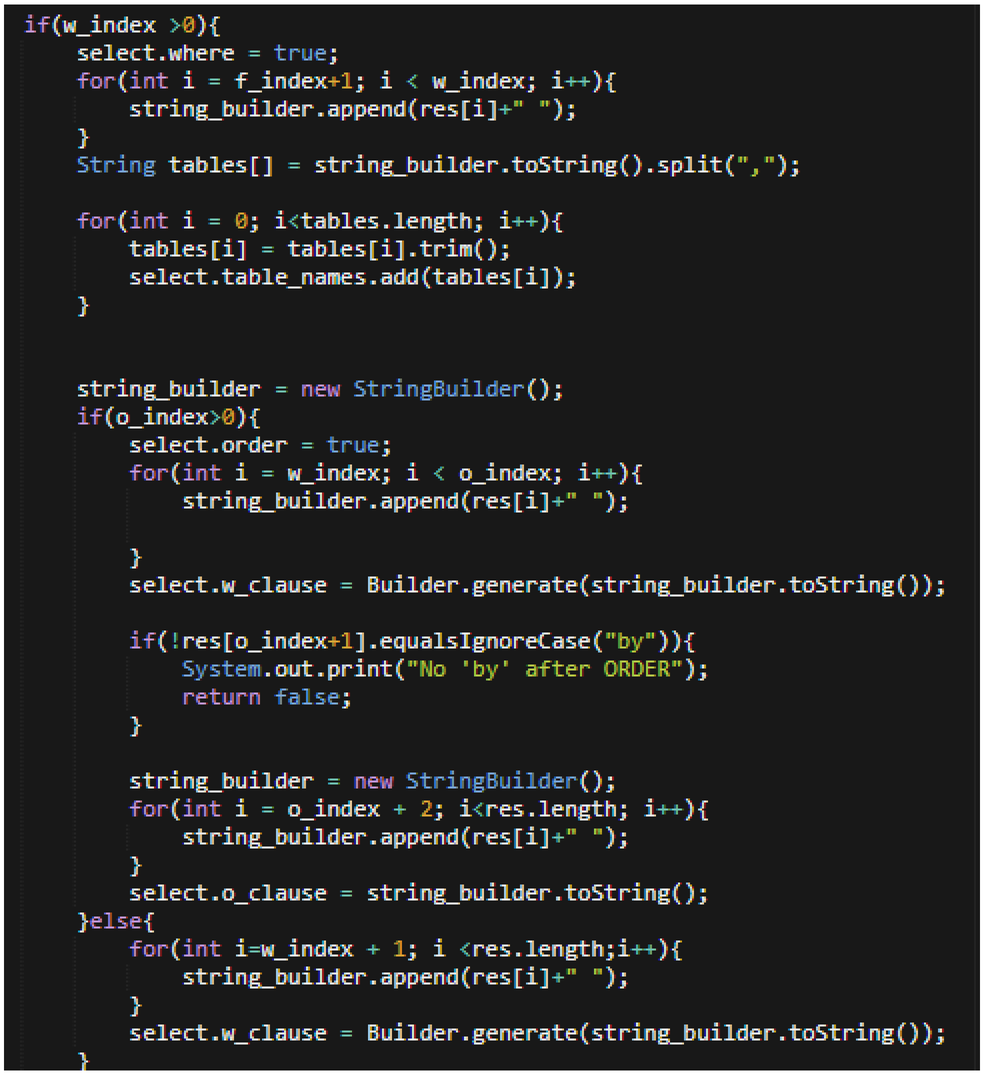
```
private boolean selectedParse(String[] res){
    select = new TreeNode();
    key_word.add("select");

    int f_index = -1;
    int d_index = -1;
    int w_index = -1;
    int o_index = -1;



    for(int i = 1; i < res.length; i++){
        if(res[i].equalsIgnoreCase("distinct")){
            d_index = i;
        }

        if(res[i].equalsIgnoreCase("from")){
            f_index = i;
        }

        if(res[i].equalsIgnoreCase("where")){
            w_index = i;
        }

        if(res[i].equalsIgnoreCase("order")){
            o_index = i;
        }

    }
    if( d_index == 1){
        select.distinct = true;
    }else if (d_index >=0){
        return false;
    }

    if(w_index>0) {
        if(o_index>0) {
            if (w_index > o_index) {
                System.out.print("Order can not be found in front of WHERE!");
                return false;
            }
        }
    }

    if(f_index < 0){
        System.out.print("FROM is not after SELECT");
        return false;
    }

    StringBuilder string_builder = new StringBuilder();
```

Fig 12, Select parse (part 1)



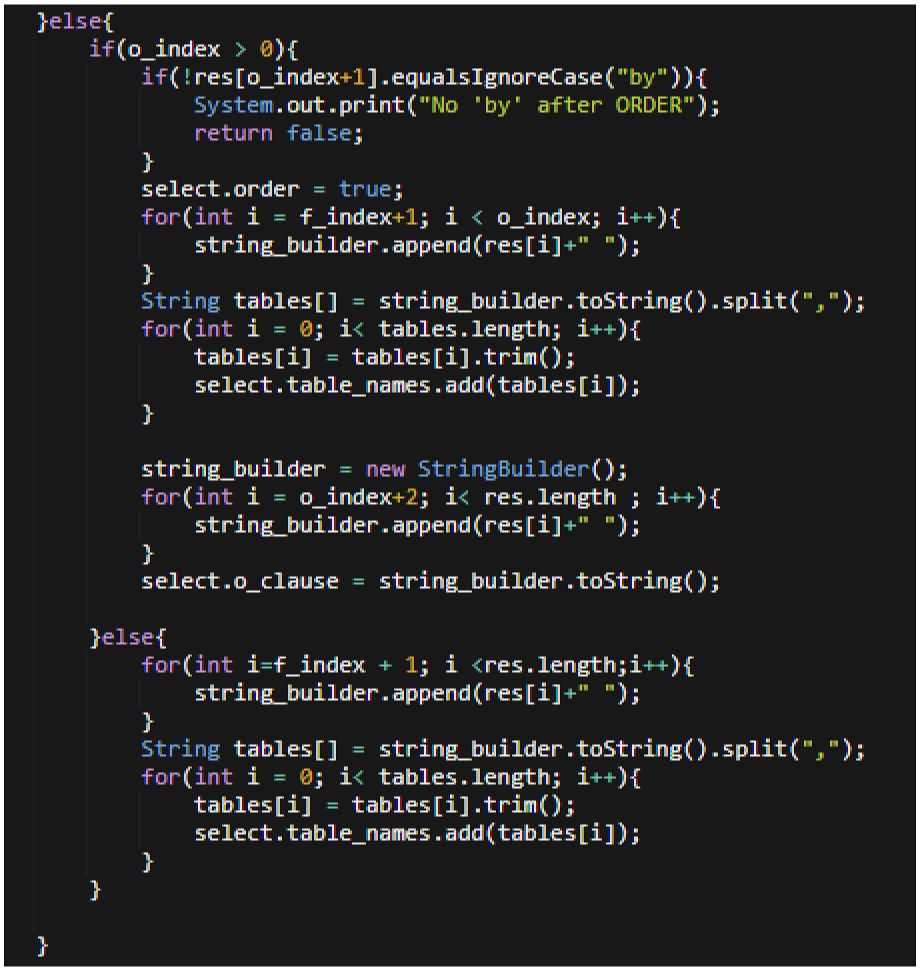
```
// adding * char when not distinct
if(d_index >0){
    for(int i=2;i<f_index;i++){
        string_builder.append(res[i]+" ");
    }
    String[] arg_statement = string_builder.toString().split(",");
    if(arg_statement[0].trim().equalsIgnoreCase("*")){
        if(arg_statement.length==1) {
            select.argument.add("*");
        }
        else{
            System.out.print("check if two characters are connected to *");
            return false;
        }

    }else{
        for(int i=0;i<arg_statement.length;i++){
            arg_statement[i]=arg_statement[i].trim();
            select.argument.add(arg_statement[i]);
        }
    }
}else{

    for(int i=1;i<f_index;i++){
        string_builder.append(res[i]+" ");
    }
    String[] arg_statement = string_builder.toString().split(",");

    if(arg_statement[0].trim().equalsIgnoreCase("*")){
        if(arg_statement.length==1) {
            select.argument.add("*");
        }
        else{
            System.out.print("check if two characters are connected to *");
            return false;
        }

    }else{
        for(int i=0;i<arg_statement.length;i++){
            arg_statement[i] = arg_statement[i].trim();
            select.argument.add(arg_statement[i]);
        }
    }
}

string_builder = new StringBuilder();
```

Fig 13, Select parse (part 2)



```
if(w_index >0){
    select.where = true;
    for(int i = f_index+1; i < w_index; i++){
        string_builder.append(res[i]+" ");
    }
    String tables[] = string_builder.toString().split(",");

    for(int i = 0; i<tables.length; i++){
        tables[i] = tables[i].trim();
        select.table_names.add(tables[i]);
    }


    string_builder = new StringBuilder();
    if(o_index>0){
        select.order = true;
        for(int i = w_index; i < o_index; i++){
            string_builder.append(res[i]+" ");

        }
        select.w_clause = Builder.generate(string_builder.toString());

        if(!res[o_index+1].equalsIgnoreCase("by")){
            System.out.print("No 'by' after ORDER");
            return false;
        }

        string_builder = new StringBuilder();
        for(int i = o_index + 2; i<res.length; i++){
            string_builder.append(res[i]+" ");
        }
        select.o_clause = string_builder.toString();
    }else{
        for(int i=w_index + 1; i <res.length;i++){
            string_builder.append(res[i]+" ");
        }
        select.w_clause = Builder.generate(string_builder.toString());
    }
```

Fig 14, Select parse (part 3)



```
}else{
    if(o_index > 0){
        if(!res[o_index+1].equalsIgnoreCase("by")){
            System.out.print("No 'by' after ORDER");
            return false;
        }
        select.order = true;
        for(int i = f_index+1; i < o_index; i++){
            string_builder.append(res[i]+" ");
        }
        String tables[] = string_builder.toString().split(",");
        for(int i = 0; i< tables.length; i++){
            tables[i] = tables[i].trim();
            select.table_names.add(tables[i]);
        }

        string_builder = new StringBuilder();
        for(int i = o_index+2; i< res.length ; i++){
            string_builder.append(res[i]+" ");
        }
        select.o_clause = string_builder.toString();

    }else{
        for(int i=f_index + 1; i <res.length;i++){
            string_builder.append(res[i]+" ");
        }
        String tables[] = string_builder.toString().split(",");
        for(int i = 0; i< tables.length; i++){
            tables[i] = tables[i].trim();
            select.table_names.add(tables[i]);
        }
    }

}
```

Fig 15, Select parse (part 4)