Alexandria University

Faculty of Engineering

Computer and Systems Engineering Dept.

JDBC API

Team Members:

- Salma Yehia 35
- Moamen Raafat 52
- Mohamed Ayman 61
- Nada Ayman 79

1) Introduction:

a. Project Description:

Extending a pre implemented Database Management System (DBMS) with new features which was pretty easy task as the code was well organized, divided into modules, reusable and well designed for extensibility.

New Features:

- Supporting the ALTER TABLE command.
- Supporting the SELECT DISTINCT command.
- New backend writer formats (JSON and Protocol buffer)

Integrating the DBMS with Java Database Connectivity (JDBC) which provides the ability to access databases regardless of the driver and database product by implementing the JDBC standard API.

Complete log of the operations done on the database through the DBMS is shown in a logging file with their timestamp using log4J.

JUnit testing for all the new features and the implemented JDBC interface.

The project in developed in java language, using the several OOP Principles and design patterns to speed up the development process, satisfy certain conditions which allow testing functionalities and improve the code readability in case it's later maintained.

b. Report Overview:

i. Software Design:

Illustrates the design and how the code is organized between different packages and classes, also explains the role of each class and how it interacts with the rest, all of this is supported by the UML design.

ii. Design Patterns:

Shows the design patterns used in the project and why each one of them is the most suitable where it is used.

iii. Design Decisions:

1. Illustrates the reason why we have chosen that design for project.
2. Illustrates the benefits of the current design of the project.
3. Difficult decisions and trade-offs while planning for the design of the project.

iv. Labor division among team members

2) Software Design:

- ### JDBC interface implementation:

1. java.sql.Driver

   The interface that every driver class must implement. Where it takes url to which this driver will be connected and incase of establishing the connection, it takes the connection properties like user name and password.

   - **acceptsURL(String url)**
     Retrieves whether the driver thinks that it can open a connection to the given URL.
   - **connect(String url, Properties info)**
     Attempts to make a database connection to the given URL.
   - **getPropertyInfo(String url, Properties info)**
     Gets information about the possible properties for this driver.

2. java.sql.Connection

   A connection (session) with a specific database. SQL statements are executed and results are returned within the context of a connection. A Connection object's database is able to provide information describing its tables, its supported SQL grammar, its stored procedures, the capabilities of this connection, and so on. This information is obtained with the getMetaData method.

   - **close()**
     Releases this Connection object's database and JDBC resources immediately
   - **createStatement()**
     Creates a `Statement` object for sending SQL statements to the database.

3. java.sql.Statement

The object used for executing a static SQL statement and returning the results it produces. It can be used in several ways whether by adding the SQL statements to the batch and at a certain moment, the developer can execute all the Commands in the batch or commands can be executed individually as in case of execute or executeQuery.

After a query is executed, If it's a select query then the return value will be a ResultSet describing the data which satisfied this query where It was also implemented because it's dependent on the DBMS.
In case it was update, delete or insert Query, then the number of affects column AKA update counter is returned.

- **addBatch(String sql)**
  Adds the given SQL command to the current list of commands for this `Statement` object.
- **clearBatch()**
  Empties this `Statement` object's current list of SQL commands.
- **close()**
  Releases this `Statement` object's database and JDBC resources immediately including the ResultSet and the ResultSetMetaData
- **execute(String sql)**
  Executes the given SQL statement, which may return multiple true if the current query has a ResultSet which can be returned by getResultSet
- **executeBatch()**
  Submits a batch of commands to the database for execution and if all commands execute successfully, returns an array of update counts.
- **executeQuery(String sql)**
  Executes the given SQL statement, which returns a single `ResultSet` object.
- **executeUpdate(String sql)**
  Executes the SQL statement given that it's update, insert or delete statement which has an update counter to be returned after the query is executed.

- **getConnection()**
  Retrieves the `Connection` object that produced this `Statement` object.
- **getResultSet()**
  Retrieves the current result as a `ResultSet` object which is the ResultSet formed by the last select query.
- **getUpdateCount()**
  Retrieves the current result as an update count which is the last update count formed by the last query from the type insert, update or delete.

4. java.sql.Resultset

A table of data representing a database result set, which is usually generated by executing a statement that queries the database such as Select query.

- **absolute(int row)**
  Moves the cursor to the given row number in this ResultSet object.
- **afterLast()**
  Moves the cursor to the end of this `ResultSet` object, just after the last row.
- **beforeFirst()**
  Moves the cursor to the front of this `ResultSet` object, just before the first row.
- **close()**
  Releases this `ResultSet` object's database and JDBC resources immediately .
- **findColumn(String columnLabel)**
  Maps the given `ResultSet` column label to its `ResultSet` column index.
- **first()**
  Moves the cursor to the first row in this `ResultSet` object.
- **getInt(int columnIndex)**
  Retrieves the value of the designated column in the current row of this `ResultSet` object as an `int`.
- **getInt(String columnLabel)**
  Retrieves the value of the designated column in the current row of this `ResultSet` object as an `int`.

- **getDate(int columnIndex)**
  Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date`
- **getDate(String columnLabel)**
  Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date`
- **getString(int columnIndex)**
  Retrieves the value of the designated column in the current row of this `ResultSet` object as a `string`
- **getString(String columnLabel)**
  Retrieves the value of the designated column in the current row of this `ResultSet` object as a `string`.
- **getFloat(int columnIndex)**
  Retrieves the value of the designated column in the current row of this ResultSet object as a float
- **getFloat(String columnLabel)**
  Retrieves the value of the designated column in the current row of this ResultSet object as a float
- **getMetaData()**
  Retrieves the number, types and properties of this `ResultSet` object's columns.
- **getObject(int columnIndex)**
  Gets the value of the designated column in the current row of this `ResultSet` object as an `Object`
- **getStatement()**
  Retrieves the `Statement` object that produced this `ResultSet` object.
- **isAfterLast()**
  Retrieves whether the cursor is after the last row in this `ResultSet` object.
- **isBeforeFirst()**
  Retrieves whether the cursor is before the first row in this `ResultSet` object.
- **isClosed()**
  Retrieves whether this `ResultSet` object has been closed.
- **isFirst()**
  Retrieves whether the cursor is on the first row of this `ResultSet` object.
- **isLast()**

Retrieves whether the cursor is on the last row of this `ResultSet` object.

- **last()**
  Moves the cursor to the last row in this `ResultSet` object.
- **next()**
  Moves the cursor froward one row from its current position.
- **previous()**
  Moves the cursor backwards one row from its current position.

5. java.sql.ResultSetMetaData

An object that can be used to get information about the types and properties of the columns in a “ResultSet” object.

- **getColumnCount()**
  Returns the number of columns in this ResultSet object.
- **getColumnLabel(int column)**
  Gets the designated column's suggested title for use in printouts and displays.
- **getColumnName(int column)**
  Retrieves the designated column's database-specific type name.
- **getColumnType(int column)**
  Retrieves the designated column's SQL type.
- **getTableName(int column)**
  Gets the designated column's table name.

- ## New DBMS features:

1. ALTER TABLE:

   The ALTER TABLE statement is used to add, or delete columns in an existing table.
   Any number of columns can be added or deleted in an alter statement satisfying certain condition where there can be no duplicate columns in the table.

2. SELECT DISTINCT

   In a table, a column may contain many duplicate values; and sometimes you only want to list the different (distinct) values.
   the DISTINCT keyword can be used to return only distinct values.

3. JSON Writer

   JSON is a light weight data format language used to describe data which is easily readable by humans.
   JSON Writer was added to the project where the database information can be saved using the JSON writer.
   Gson was used which is a jar from the Google the supports JSON parsing.

4. Protocol Buffer

   Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data

   Using the Jar developed by Google to give support to protocol buffer where they’re fast and low on memory usage.

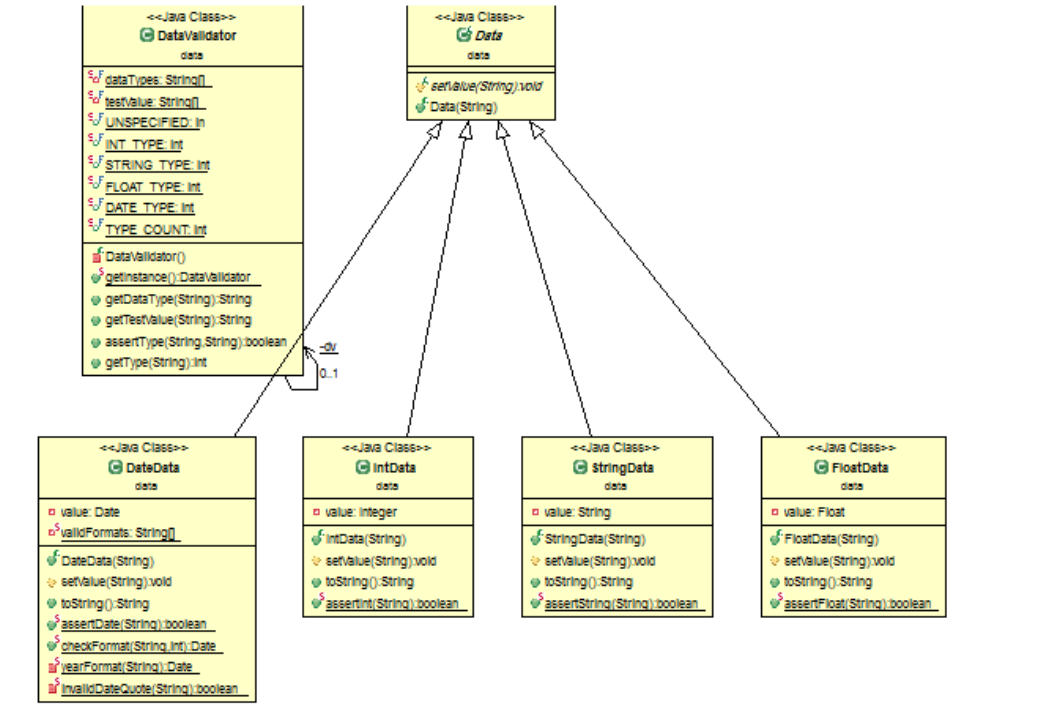
- Added Classes:
  - JDBC:
    Several classes were made implementing the JDBC Interfaces of Driver, Connection, Statement, ResultSet and ResultSetMetaData.

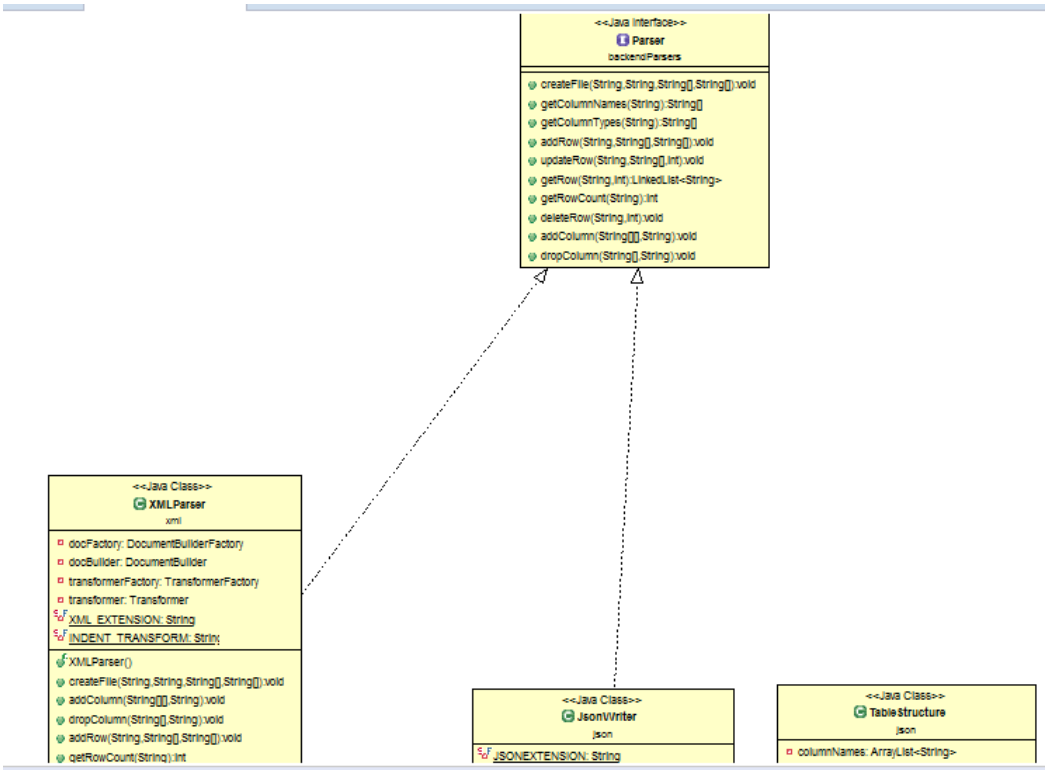
  - Alter and Distinct:
    For both Alter and Distinct a class was made for the parsing process of these two commands extending the main SQLParser class.

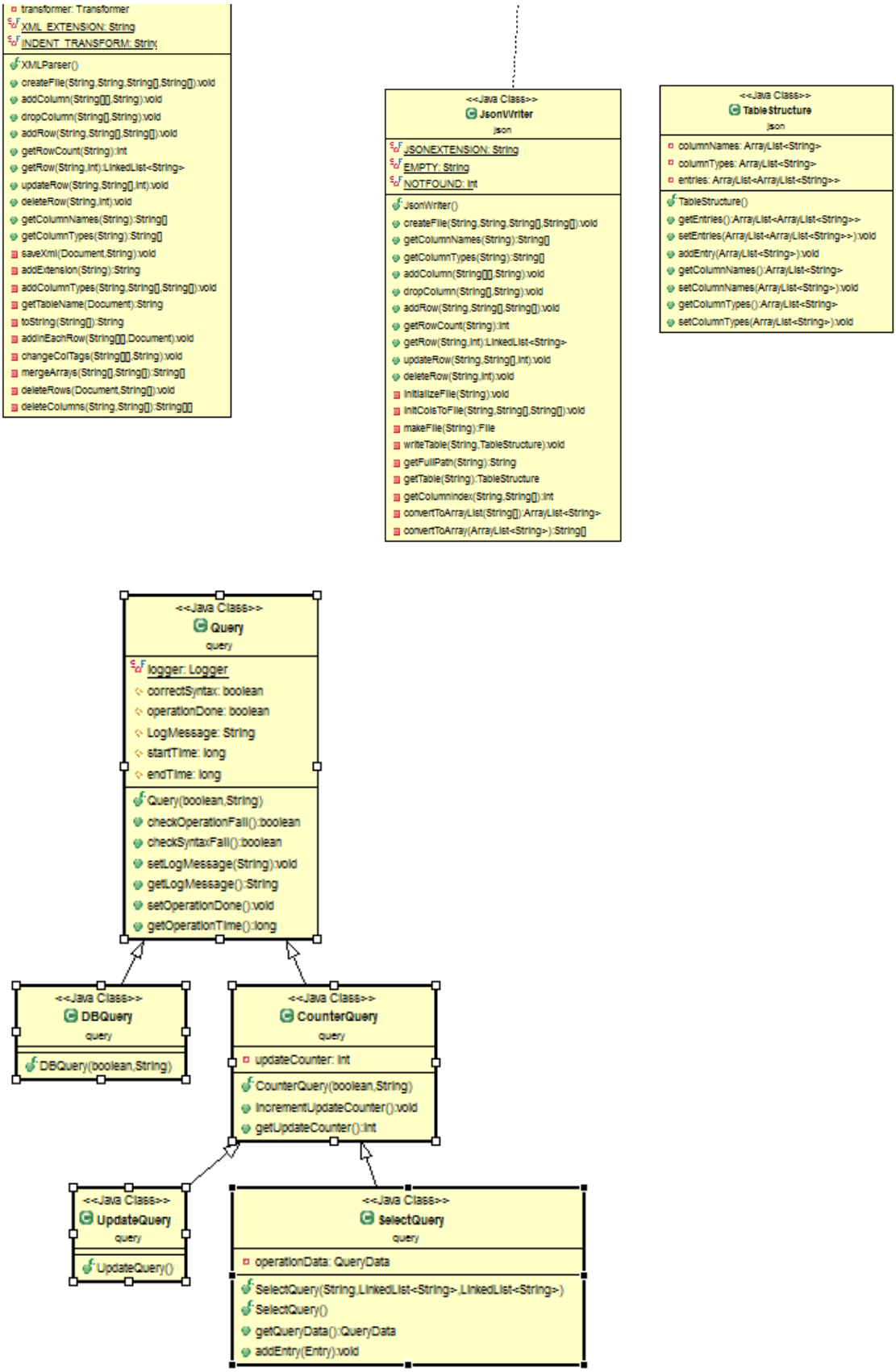
  - Backend writers:
    Several classes were introduced to support the new backend writers, JSON and Protocol Buffer, which are ProtoBufParser which implements the parser interface. JsonParser which implements the parser interface. With each of the newly introduced parsers a tabular structure class was added.

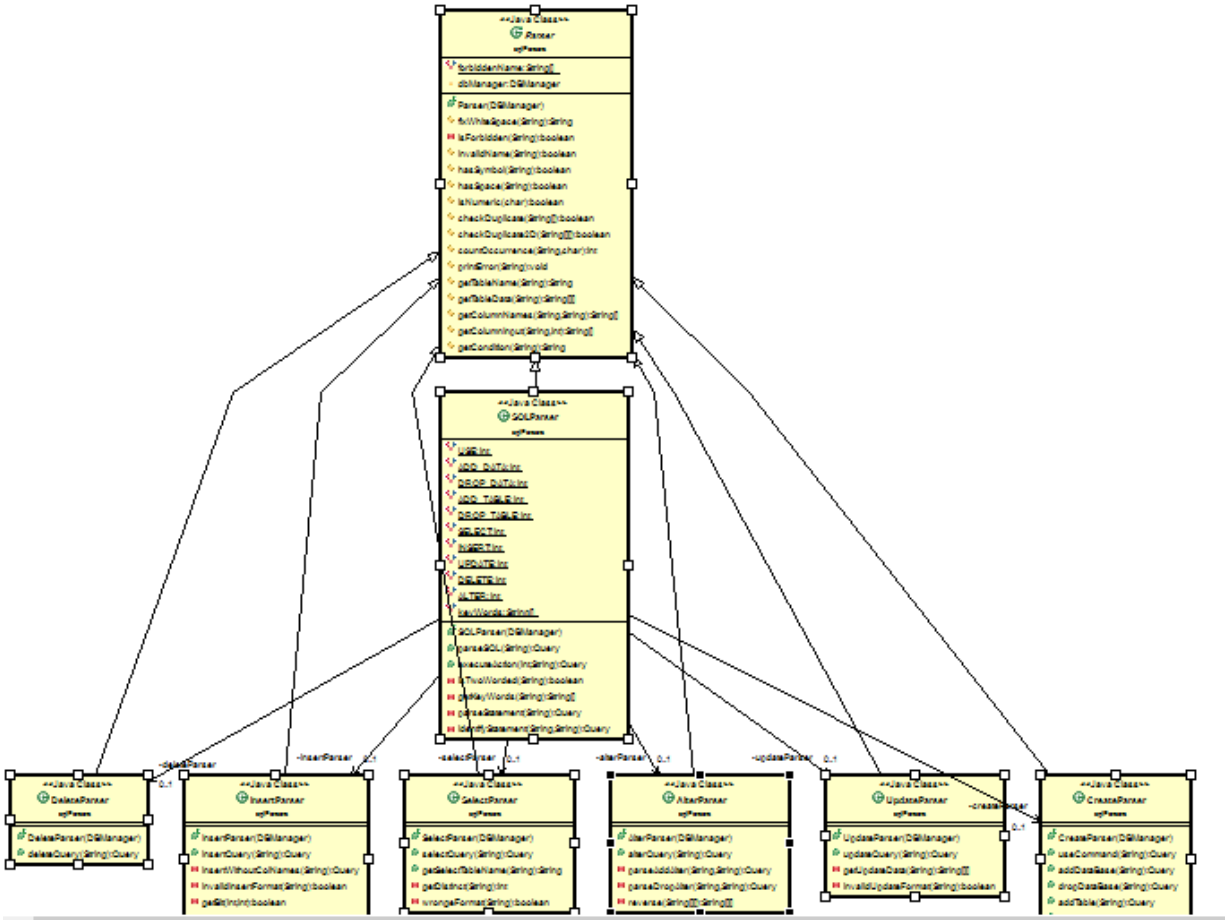
  - Data types:
    Due to the extensibility of the codebase adding new Data types didn't require anything but creating two separate new classes 'Date' and 'Float' where each one handles operations related to this data type. Also there was two Comparator classes introduced one for comparing Float data and one for comparing Date data.

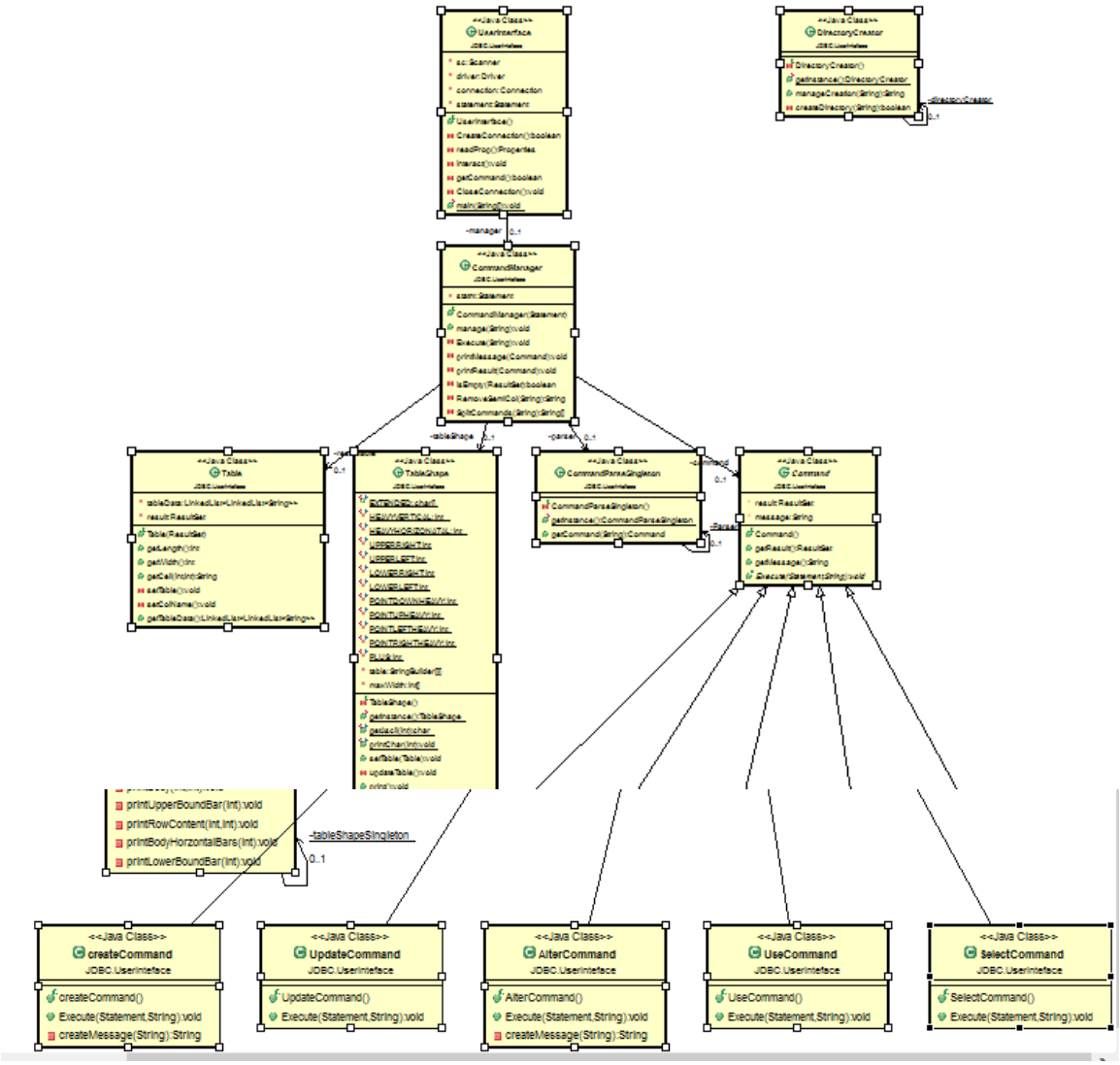
3) UMLS:


<<Java Class>>
DataValidator
data
dataTypes: String[]
testValue: String[]
UNSPECIFIED: in
INT_TYPE: Int
STRING_TYPE: Int
FLOAT_TYPE: Int
DATE_TYPE: Int
TYPE_COUNT: Int
DataValidator()
getInstance():DataValidator
getDataType(String):String
getTestValue(String):String
assertType(String,String):boolean
getType(String):Int
-dv
0..1
<<Java Class>>
Data
data
setValue(String):void
Data(String)
<<Java Class>>
DateData
data
value: Date
validFormats: String[]
DateData(String)
setValue(String):void
toString():String
assertDate(String):boolean
checkFormat(String,Int):Date
yearFormat(String):Date
invalidDateQuote(String):boolean
<<Java Class>>
IntData
data
value: Integer
IntData(String)
setValue(String):void
toString():String
assertInt(String):boolean
<<Java Class>>
StringData
data
value: String
StringData(String)
setValue(String):void
toString():String
assertString(String):boolean
<<Java Class>>
FloatData
data
value: Float
FloatData(String)
setValue(String):void
toString():String
assertFloat(String):boolean


<<Java Interface>>
Parser
backendParsers
createFile(String,String,String[],String[]):void
getColumnNames(String):String[]
getColumnTypes(String):String[]
addRow(String,String[],String[]):void
updateRow(String,String[],Int):void
getRow(String,Int):LinkedList<String>
getRowCount(String):Int
deleteRow(String,Int):void
addColumn(String[][],String):void
dropColumn(String[],String):void
<<Java Class>>
XMLParser
xml
docFactory: DocumentBuilderFactory
docBuilder: DocumentBuilder
transformerFactory: TransformerFactory
transformer: Transformer
XML_EXTENSION: String
INDENT_TRANSFORM: Strin
XMLParser()
createFile(String,String,String[],String[]):void
addColumn(String[][],String):void
dropColumn(String[],String):void
addRow(String,String[],String[]):void
getRowCount(String):Int
<<Java Class>>
JsonVriter
json
JSONEXTENSION: String
<<Java Class>>
TableStructure
json
columnNames: ArrayList<String>


transformer: Transformer
XML_EXTENSION: String
INDENT_TRANSFORM: Strin
XMLParser()
createFile(String,String,String[],String[]):void
addColumn(String[][],String):void
dropColumn(String[],String):void
addRow(String,String[],String[]):void
getRowCount(String):int
getRow(String,int):LinkedList<String>
updateRow(String,String[],int):void
deleteRow(String,int):void
getColumnNames(String):String[]
getColumnTypes(String):String[]
saveXml(Document,String):void
addExtension(String):String
addColumnTypes(String,String[],String[]):void
getTableName(Document):String
toString(String[]):String
addInEachRow(String[][],Document):void
changeColTags(String[][],String):void
mergeArrays(String[],String[]):String[]
deleteRows(Document,String[]):void
deleteColumns(String,String[]):String[][]
<<Java Class>>
JsonWriter
json
JSONEXTENSION: String
EMPTY: String
NOTFOUND: Int
JsonWriter()
createFile(String,String,String[],String[]):void
getColumnNames(String):String[]
getColumnTypes(String):String[]
addColumn(String[][],String):void
dropColumn(String[],String):void
addRow(String,String[],String[]):void
getRowCount(String):int
getRow(String,int):LinkedList<String>
updateRow(String,String[],int):void
deleteRow(String,int):void
initializeFile(String):void
initColsToFile(String,String[],String[]):void
makeFile(String):File
writeTable(String,TableStructure):void
getFullPath(String):String
getTable(String):TableStructure
getColumnIndex(String,String[]):int
convertToArrayList(String[]):ArrayList<String>
convertToArray(ArrayList<String>):String[]
<<Java Class>>
TableStructure
json
columnNames: ArrayList<String>
columnTypes: ArrayList<String>
entries: ArrayList<ArrayList<String>>
TableStructure()
getEntries():ArrayList<ArrayList<String>>
setEntries(ArrayList<ArrayList<String>>):void
addEntry(ArrayList<String>):void
getColumnNames():ArrayList<String>
setColumnNames(ArrayList<String>):void
getColumnTypes():ArrayList<String>
setColumnTypes(ArrayList<String>):void
<<Java Class>>
Query
query
logger: Logger
correctSyntax: boolean
operationDone: boolean
LogMessage: String
startTime: long
endTime: long
Query(boolean,String)
checkOperationFail():boolean
checkSyntaxFail():boolean
setLogMessage(String):void
getLogMessage():String
setOperationDone():void
getOperationTime():long
<<Java Class>>
DBQuery
query
DBQuery(boolean,String)
<<Java Class>>
CounterQuery
query
updateCounter: Int
CounterQuery(boolean,String)
IncrementUpdateCounter():void
getUpdateCounter():Int
<<Java Class>>
UpdateQuery
query
UpdateQuery()
<<Java Class>>
SelectQuery
query
operationData: QueryData
SelectQuery(String,LinkedList<String>,LinkedList<String>)
SelectQuery()
getQueryData():QueryData
addEntry(Entry):void


«Java Class»
Parser
ojPlsql
forbiddenName: String[]
dbManager: DBManager
Parser(DBManager)
fixWhiteSpace(String):String
isForbidden(String):boolean
invalidName(String):boolean
hasSymbol(String):boolean
hasSpace(String):boolean
isNumeric(char):boolean
checkDuplicate(String[]):boolean
checkDuplicate2D(String[][]):boolean
countOccurrences(String,char):int
printError(String):void
getTableName(String):String
getTableData(String):String[][]
getColumnNames(String,String):String[]
getColumnInput(String,int):String[]
getCondition(String):String
«Java Class»
SQLParser
ojPlsql
USE:Int
ADD_OUTA:Int
DROP_OUTA:Int
ADD_TABLE:Int
DROP_TABLE:Int
SELECT:Int
INSERT:Int
UPDATE:Int
DELETE:Int
ALTER:Int
keyWords: String[]
SQLParser(DBManager)
parseSQL(String):Query
executeAction(Int,String):Query
isTwoWorded(String):boolean
getKeyWords(String):String[]
parseStatement(String):Query
identifyStatement(String,String):Query
-deleteParser
0..1
-insertParser
0..1
-selectParser
0..1
-alterParser
0..1
-updateParser
0..1
-createParser
0..1
«Java Class»
DeleteParser
ojPlsql
DeleteParser(DBManager)
deleteQuery(String):Query
«Java Class»
InsertParser
ojPlsql
InsertParser(DBManager)
insertQuery(String):Query
insertWithoutColNames(String):Query
invalidInsertFormat(String):boolean
getSit(Int):boolean
«Java Class»
SelectParser
ojPlsql
SelectParser(DBManager)
selectQuery(String):Query
getSelectTableName(String):String
getDistinct(String):Int
wrongeFormat(String):boolean
«Java Class»
AlterParser
ojPlsql
AlterParser(DBManager)
alterQuery(String):Query
parseAddAlter(String,String):Query
parseDropAlter(String,String):Query
reverse(String[]):String[]
«Java Class»
UpdateParser
ojPlsql
UpdateParser(DBManager)
updateQuery(String):Query
getUpdateData(String):String[]
invalidUpdateFormat(String):boolean
«Java Class»
CreateParser
ojPlsql
CreateParser(DBManager)
useCommand(String):Query
addDataBase(String):Query
dropDataBase(String):Query
addTable(String):Query



4) Design Patterns:

1. Interface pattern: Interface contains method signatures that are guaranteed to be implemented by the implementing class.

   (1) Parser which contains methods implemented by all supported backend writers (XML – JSON – Protocol Buffer)

   (2) DBManager which contains methods that are non-dispensable for any database management system which are the main types of queries.

2. Visitor pattern: a pattern used to execute several operations on objects when they are unrelated and distinct. Where several operations like parsing, executing and getting the return value of statements was executed and the Session object was sent to the 1batch to execute current statement in batch.

3. Singleton pattern: a creational pattern implemented by creating a class with a method that creates a new instance of the class if one does not exist. If an instance already exists, it simply returns a reference to that object. To make sure that the object cannot be instantiated any other way, the constructor is made private.

   It was used in the Program In case of XML, JSON, ProtocolBuffer writers to provide a global access point and ease the testing process. Also it was used in the Boolean evaluation of expression in the condition as the BooleanEvaluator was a Singleton class where it takes a statement returning whether its value is true or false.

5) Design Decisions:

- Using Inheritance (Query)
  Query class was used to store several information regarding the currently being executed query, in case of success of query It can be of many types including CounterQuery which is a Query based on counting such as update, delete, or insert where in the Statement class their return value is updateCount. But in case of Queries such as Select then the Query will be of the type

SelectQuery which is used to save the entries the satisfied the query and data related to it.
Query parent class has the common behavior and data between all queries such as:

a. Saving the time of creating the end of the query.
b. Saving whether there was an error the occurred during the query.

- Using Polymorphism: Query
  One of the main advantages was the polymorphism of Query class where the classes can be used as the parent class not relating to their type.

6) Labor Division:

- Moamen Raafat: JSON, Protocol Buffer, Driver, Connection, JUnit testing.
- Mohamed Ayman: Statement, JUnit testing.
- Nada Ayman: Alter, Distinct, ResultSetMetaData, logging.
- Salma Yehia: ResultSet, XML, CLI.
- All team members contributed in writing the report.