CS44800 - Project 2

Fall 2017

Due: 10/26/2017 at 11:50 PM on Blackboard

**Introduction**

You will be working with an implementation of a simple database built on top of JSON files for storing table data and metadata. The database features limited integrity validation, access control, and querying functionality using relational algebra operators, which you will be responsible for implementing.

You are provided with a framework for parsing JSON files into the relevant components of the database: the schemas, users, row data, and queries to be executed. You do not have to be concerned with the mechanics of parsing data or queries. You only need to focus on the tasks of validation and query execution. This project is a Maven distribution for ease of building and executing the program with the necessary dependencies. The following is a brief description of the provided file structure and framework:

**File Structure**

All JSON files are identified and referenced by the parsing interface with the convention of *identifying file names*. These names should be used when accessing the parsing interface instead of the full qualified filepath. All files should be stored in the src/main/resources folder in order to be visible to the parsing interface.

**Table** data for an individual table is contained in two JSON files – the schema file (“<tableName>.json”) and the data file (“<tableName>Data.json”).

The description of all **users** in the database is included in a single file as well (“<usersFileName>.json”).

Finally, **queries** are defined in json files as well, one query per file (filenames follow the convention of <queryName>.json).

The file **names** should be passed into the program as command line arguments, in the following format:

*java jsonDB.db.simpleDB -t <table1> [-t <table2> …] –u <users> [-q <query1> -q <query2> …]*

where –t specifies that the next argument is the name of a table definition file, -u specifies the argument as the user file, and –q specifies the argument as a query. Arguments can be listed in any order. There may be multiple table and query file arguments, but there should exactly be one user file argument and at least one table argument.

**Parsing Interface**

The Parsing Interface is in the jsonDB.parser package and consists of three static parsing methods for parsing table/user/query definitions, and a JSON\_RowIterator for parsing actual row data.

**Parser.parseTable**

 This will parse the JSON file that describes a Table into a Table object. The Table object defines the fields and constraints for a given relation. For the purposes of this implementation, this has two components: the Schema (which defines the field names and types) and a list of Constraints on that Schema.

**Parser.parseUsers**

 This will parse a JSON file that describes the users into list of User objects. A User object contains a list of *subschemas* that define the access rights a User has. Only schemas and fields listed in the User’s set of subschemas can be accessed – any other access must be denied.

**Parser.parseQuery**

 This will parse a JSON file that describes an individual query into a Query object. A Query object is a definition of which tables need to be accessed, which fields need to be included in the output, and which predicates returned rows need to satisfy in order to be included in the output. This structure means there is no need for complicated parsing of SQL queries into the relevant information.

**JSON\_RowIterator**

 This class is will iterate over a JSON file that defines actual row data. The hasNext() function will return true if there are more rows to return, and the getNext() tuple returns that next row. These rows are returned returned as raw string data, i.e. as a list of Strings, one per each field. When using the JSON\_RowIterator you will be responsible for converting these Strings into typed Java Objects (e.g. Integer, Double, Boolean, etc).

**Other Portions of the Framework**

**Predicate**

 The Predicate class allows for generically-typed evaluation of row data to a pre-defined predicate. These Predicates are used in certain Constraints and for evaluating Queries. The Predicate class may be a good resource example to use for dealing with the generically-typed row data.

**Constraint**

 The Constraint class defines the constraints placed on fields on a table. Each Constraint defines the type of constraint, the field name the constraint is defined on (it is assumed that each Constraint will be associated with a given table elsewhere), and optionally a list of Predicates that need to be satisfied. In this implementation, the evaluation of the Constraint is separated from the definition of it. Calling the getConstraintValidator() method constructs a *ConstraintValidator* class, specific to the type of the Constraint and the Schema the Constraint is defined on. Each ConstraintValidator has a validateConstraint() method that, given a Row, will validate that Row on that Constraint.

**Database**

 The Database can be used to store information about the specific Database instance. It is where the main interface methods you will be writing are located, and a static instance of the class is defined in the SimpleDB class, *SimpleDB.db*. This instance can be used to reference any information about the current running database instance that you have stored in it.

**Tasks**

There are two main portions of this project: the implementation of validation and access control, and the implementation of the relational algebra operators using the Iterator interface. You may implement these components in whichever order you wish, but the suggested order is as follows

For each task, there are pre-defined interface methods that you must implement. You can write as many other methods or even classes as you need in order to implement each step, but the final solution for that task must be executed through the interface method.

**Part 1: Validation and Access Control**

Your first task is to read in and validate the schemas, user access rights, and row data. You may assume that the provided JSON files will not be malformed – that is, you do not have to account for typos or incorrect JSON syntax.

Most of the implementation for this section will take place in the jsonDB.db.ValidationInterface class. This class contains the majority of the interface methods you will have to implement during this project. In addition, you will have to implement the validation of Constraints, which are in the jsonDB.validation package.

Additional information on requirements is given in the documentation comments for the methods.

**ValidationInterface**

For this task, you must implement the following required methods in the ValidationInterface:

**readTables**

 Use the provided Parser interface methods to read in all Tables specified in the command line arguments.

**readAndValidateUsers**

 Use the provided Parser interface methods to read in all Users from the specified user file. Validate the user subschemas against the existing schemas – any user with non-valid subschemas should not be accepted.

**readAndValidateRow**

 Use the provided JSON\_RowIterator to iterate through all data in a given table. Check that the row fits the definition of the schema (i.e. matching fields), and return the Row object representation of that row. Constraints are not checked.

**validateData**

Given a table, iterate through the data in the table and validate each row based on both the schema and the constraints. Print error messages for each invalid row. After finishing validating all data, if any data is not valid, return false.

**validateQuery**

 Given a query, validate that the query accesses valid fields in the database (i.e. the table in the FROM clause exists, and all fields accessed in either the SELECT clause or from predicates in the WHERE clause all exist in that table).

 In addition, check that the User making the query has sufficient access rights to access the table and specific fields accessed by the query (i.e. the table and fields exist in the User’s set of subschemas).

 For simplicity, joins are not implemented in this project. Therefore all queries are single-table queries.

**Constraint Validators**

In order to validate constraints, you must complete the ConstraintValidator classes (in the jsonDB.validation package). There are three different ConstraintValidator classes that must be completed, one for each of the constraint types implemented in this project: NOT\_NULL, UNIQUE, and CHECK constraints.

For each ConstraintValidator you must complete the Constructor, which initializes the state of the ConstraintValidator and the validateConstraint() method, which validates a given row on the constraint.

**CheckConstraintValidator**

 CHECK constraints check that the data satisfies some set of predicates. For simplicity, all predicates in a given CHECK constraint should be connected by the Boolean AND operator – ORs are not allowed in the definition of a CHECK constraint. All Predicates need to be satisfied for the row to be valid.

**NotNullConstraintValidator**

 A NOT\_NULL constraint checks that the data for the field defined by the Constraint is not null. It fails if and only if the data in that field is null.

**UniqueConstraintValidator**

 A UNIQUE constraint checks that the data in the defined field is unique within that table. If there are non-unique values, only the first row checked will be accepted as valid. Any subsequent rows with non-unique data are not valid.

**Part 2: Implementing Relational Operators**

Relational databases execute queries through implementing Relational Algebra operators and executing them in a sufficient matter that takes into account available memory space and efficient disk access. This is accomplished by implementing relational operators through an Iterator interface.

Relational algebra operators are pipelined together into a Query Plan. Rows (or Tuples) “flow” through the plan from the base relations to the final output of the plan. Tuples from each Iterator in the plan are lazily computed only when required through calls to the Iterator interface method getNext(). This allows for results to be computed without needing to materialize and store the results of each operator in-memory.

The Iterator interface is as follows:

**open()**

 Initializes the Iterator state and readies the Iterator to start reporting results. In this database, this is represented by the Iterator constructor.

**close()**

 Closes the Iterator (and recursively close any child Iterators) and releases any resources held by it.

**hasNext()**

 Computes the next result that will be returned by the Iterator and advances the Iterator. Returns true if a result was found, false if no more Tuples remain.

**getNext()**

 Returns the last computed result of the Iterator

**reset()**

 Resets the state of the Iterator to the point at which the Iterator was first opened.

More detailed documentation of this interface is available in the Iterator class in the jsonDB.relops package. You will need to finish the implementation of the following three Iterators in the jsonDB.relops package.

**FileScan**

The FileScan Iterator is essentially a wrapper for the JSON\_RowIterator that converts the next valid row data retrieved by the underlying JSON\_RowIterator into a Row object.

The FileScan Iterator needs to determine whether an incoming row is valid on the Schema of the Iterator and if all constraints of the Iterator are satisfied. Invalid rows should not be included in the result. No error messages should be printed when invalid rows are encountered – simply do not report them as a result.

**Selection**

The Selection Iterator is the implementation of the Selection Relational Algebra operator. This should check each Tuple retrieved from the underlying Iterator to see if the Tuple satisfies the Predicates that are defined for this Selection Iterator. Only Tuples that satisfy the Predicate should be returned.

Predicates for a Query are defined in Conjunctive Normal Form (CNF). A Selection Iterator should OR all Predicates defined on it together. To implement AND, pipeline together multiple Selection Iterators. See the documentation comments for Iterator.createPlan() for a more detailed description and examples of CNF representation of queries.

**Projection**

The Projection Iterator is the implementation of the Projection Relational Algebra operator. This should take an incoming Tuple from the underlying Iterator and construct a new Row with the new Schema defined for this Projection Iterator, containing only the data from the incoming row of the columns defined in this new schema.

For simplicity, “SELECT \*” is not implemented – the fields passed into the Projection Iterator will always be the complete list of fields that should appear in the output.

**Executing a Query**

Now that all the operators are implemented, you need to actually construct the Query Plan out of these Iterators so that the query can be executed. You need to implement the static createPlan() method defined in the Iterator class.

A valid Query object is passed into the createPlan() method. Using the information contained in the Query, you should construct a Query plan consisting of a FileScan to read the base table, Selection operators to evaluate the WHERE predicates, and a Projection operator to provide the appropriate output columns. The final root Iterator the plan should be returned. Then, execute() can be called on this root Iterator in order to recursively and lazily evaluate the results of the query.

The tasks of calling the query validation, plan creation, and query execution methods are done for you in the jsonDB.db.Database.runQuery() method. You are only responsible for implementing the createPlan() method.

**Testing your Code**

Sample test schemas, data, users, and queries are provided to you. This sample data allows you to run some output, illustrates some validation behavior, and provides a reference for all the JSON syntax needed to write your own sample data. However, this sample data is not a complete set of edge case behaviors or invalid data. You should think about what possible cases need to be checked for proper validation and are encouraged to write your own test cases.

After building the project and maven dependencies, you may execute the program with the provided sample data from the command line by running the following command:

A Makefile is provided to facilitate compiling and running the project. Run ‘make’ from the terminal to compile and run your project with the provided sample data as arguments. If you wish to provide your own data you can edit the arguments provided in the xx recipe in the Makefile. Remember to make sure that the bin/ directory exists in the top level of your project directory otherwise make will fail.

What to submit

Include your project src folder and a README file in a zipped archive. The archive name should be **p2\_your\_career\_login.zip**

**Remember to double-check that your zip archive actually contains your project data**

Your code should compile and run on the University linux systems.

The README file should include your name, purdue career login id, and any notes about your project that you’d like the TAs to know for grading.

The project should be submitted on Blackboard.