CS 210 | File and Data Structures | Fall 2021 | Term Project

Table of Contents

Grading Policies

These grading policies apply to all modules for the course.

A For each module, you must **push a commit** to your *origin* remote by the deadline on the course calendar and **push a tag** for that commit as named in the module instructions. You earn a new or updated grade out of 100% as follows.
  1 A module **graded by unit tests** is graded automatically based on your tagged commit using unit tests from the instructor.
    i Your grade is the percentage of unit tests passed, as reported when the unit tests are compiled and executed by the instructor, either equally weighted or weighted as indicated by the unit tests.
    ii Late submissions are graded with a 30% penalty if you submit before the end of the final session. If you request an extension no more than 48 hours after the deadline and the instructor approves your request, the penalty is waived.
  2 A module **graded by inspection** is graded manually by the instructor based on your tagged commit.
    i You must **demonstrate your work** in a scheduled meeting with the instructor. Alternatively, you can email the instructor a screen recording in which you demonstrate your work.
    ii Your grade is the percentage the instructor assesses on the graded requirements in the module instructions.
    iii Late submissions are graded 0%. If you request an extension no more than 48 hours after the deadline and the instructor approves your request, your submission is graded without penalty.

B If you are in the **honors section** of the course, you must also submit the honors modules.

C At most once before the end of the final session, you can submit an **extra module** and request for the instructor to replace your grade for any previously graded module or honors module with your grade for that extra module.

D The instructor may **audit** your work for any module at any time to ensure that you complied with the requirements and the academic integrity policy. The instructor may retroactively penalize your grade up to 30% for circumventing the requirements, in addition to any academic dishonesty penalties that may be warranted as defined in the syllabus.

E The instructor may release supplemental **lab modules** which don't count in your grade calculations.

Development Environment

You must use a supported development environment for all work on the term project.

A The following development environment is officially supported. All components are free for use on your personal computer. In addition, they can be used on most departmental lab computers and can be virtualized using the LOUD OVA (download).
  1 **Java Development Kit** version 16 by Oracle (exact version required, download and install)
  2 **Eclipse IDE for Java Developers** version 2021-06 or newer (download and install)
    i **EGit** plugin version 5.11 or newer (already included with Eclipse)
    ii **Buildship** plugin version 3.1 or newer or **m2e** plugin version 1.17 or newer (both already included with Eclipse)

B The following components are available for specific needs only.
  1 **IntelliJ IDEA Community** (unsupported alternative to Eclipse)
  2 **Git** terminal client or **GitKraken** or **GitHub Desktop** graphical client (optional for advanced repository maintenance)

Starting the Project

After installing your development environment, to start the project in Eclipse, carefully complete the following steps in order.

A Create an account on GitHub, the popular git service provider.
  1 You may use an account you already own, but your username and profile must be safe for classroom use.
  2 Add your WVU email address to your account for verification purposes.

B Prepare your *origin* repository from the instructor's template in the *upstream* repository.
  1 Accept the GitHub Classroom invitation to access the term project template.
  2 If prompted, authorize GitHub Classroom, a third-party application, to access your GitHub account.
  3 If prompted, join your GitHub account to your name in the GitHub Classroom roster.
  4 Confirm that your *origin* repository is now available under the Reaser Classroom organization. It is private, only visible to you and the instructor. It contains all the code initially released in the instructor's template.

C Create an SSH key using RSA and a passphrase for git authentication with GitHub.
  1 Add the key to your available keys in both your local development environment and your GitHub security settings.
  2 When authenticating git remotely with SSH and your passphrase, use the username *git* and a blank password.
  3 As a fallback, you may authenticate via HTTPS with your GitHub username and password, but this is discouraged.

D Clone your *origin* repository to your local development environment.
  1 Visit your *origin* repository's GitHub page to find the clone URL.
  2 The first time you authenticate, you should accept and store GitHub's key as a known host.
  3 Configure your *origin* remote for fetch/pull and for push, with all branches pushed and all tags force-pushed.
  4 You may need to fetch the remote tracking branch *origin/master* manually.

E Add the *upstream* repository to your local repository as an additional remote.
  1 Visit the *upstream* repository's GitHub page to find the clone URL.
  2 Configure the *upstream* remote for fetch/pull only, not for push.
  3 You may need to fetch the remote tracking branch *upstream/master* manually.

F Confirm that the local *master* branch is checked out and that your remote tracking branches are merged into it.
  1 GitHub has adopted *main* as the default branch name on its service instead of *master*, but this has not been incorporated into the term project because most git clients default to *master*.

G Import the project into your development environment using one of the following dependency management tools.
  1 In the preferred way, import as an existing Gradle project based on the configuration in the *build.gradle* file.
    i If necessary when importing, override the workspace settings to specify Gradle version 7.0 or newer.
    ii Gradle replicates the instructor's grading tools.
  2 In the alternative way, import as an existing Maven project based on the configuration in the *pom.xml* file.
    i Only use Maven if you are unable to import successfully using Gradle.
    ii Maven is comparable to Gradle but may differ in edge cases from the instructor's grading tools.

H Confirm that you can execute the project.
  1 Run `apps.Console` as a Java application. Enter `echo "Hello, world!"` at the prompt. Confirm a successful response.
  2 Run any available test classes in the `grade` package as JUnit tests. Confirm that they run (ignore any failed tests).

I Edit the *README.md* file in the root folder of your local repository as indicated in the file.
  1 Commit this file change and push the commit to your *origin* remote.
  2 You must complete this step to earn any grades in the course.

The following optional Eclipse workspace configuration steps may be helpful and can be completed in any order at any time.

J Enable save actions to organize imports and remove unused imports. This helps avoid grading issues due to illegal imports.
K Add the JUnit stack trace filter pattern `java.base` to hide unhelpful lines in your failure traces and make them easier to read.

Working on a Module

After starting the project above, to work on a module in Eclipse, carefully complete the following steps in order.

A Fetch from the *upstream/master* tracking branch and merge any new commits into your local *master* branch.
  1 The instructor releases unit tests and fixes as commits on the *upstream/master* tracking branch.
  2 Also fetch and merge from your *origin/master* tracking branch if you pushed changes there from another remote.
  3 Fetching allows you to intervene before merging in case the merge might be destructive. Pulling automatically fetches and merges without allowing you to intervene. Pulling is usually safe, but fetching alone is always safe.
  4 If there are merge conflicts, you must resolve them now, then commit the changes to finish the merge.
    i For each conflicted file in the working tree, edit the file to keep the correct version for the conflicted lines only.
    ii Alternatively, for each conflicted file in the index, choose the correct version to keep for the whole file.

B Complete the source code for the module according to its instructions, committing your changes to the local *master* branch one or more times using the workflow that serves you best and pushing periodically to safeguard against lost work.

C Tag the commit you want to be graded for the module using the tag name in its instructions.
  1 Tag names are case-sensitive and must be entered exactly as required (for example, `M1` for Module 1).
  2 The tag message should be the full name of the module (for example, `Module 1` for Module 1).
  3 If you ever need to change the commit for a tag, you can move the tag by forcing reuse or by deleting and recreating it.

D Push your local *master* branch and all tags to your *origin* remote so that the instructor can clone your work for grading.

E Visit your *origin* repository's GitHub page to confirm your tag is on the right commit by the deadline.
  1 If you are unable to move a tag, confirm that your origin remote is configured to force push tags.
  2 If your tag is not pushed, ensure that you pushed the tag itself and not just the commit it points to.

Coding Guidelines

Complete each module using valid and executable **Java** code, observing the following guidelines.

A The `Database` is assessed directly when graded by unit tests. The `Console` is assessed directly when graded by inspection. All other classes are assessed indirectly as dependencies of those two classes.

B Arrange your source code, test code, and packages to match the structure of the instructor's *upstream* repository.

C Some files released for example only or to help with code reuse are marked `@Deprecated` and can be removed if you wish.

D Ensure that all **errors, exceptions, and warnings** are fixed, suppressed, or handled, including the following common issues:
  1 Any *compilation errors* in any classes, even in classes that are not related to the module you're currently testing.
  2 Warnings for using *hidden names* or *raw types* instead of parameterized generic types with distinguishable names.
  3 Warnings for *unused imports*, especially for unit testing libraries imported in classes which aren't unit tests and for any libraries that aren't available in the JDK or in the managed dependencies that *build.gradle* defines.

E Lingering errors, exceptions, or warnings could prevent the instructor's grading tools from being able to compile and test your code even if your local development environment is more permissive. To help detect these issues locally, run the provided JUnit test classes as Gradle tests instead to mimic the compilation environment of the instructor's grading tools.

Document Conventions

The following conventions are used in this document to describe Structured Query Language (SQL) queries.

A All **syntax** is shown in `blue` monospace and is a literal within the query.

B All **metasyntax** is shown in `orange` monospace and represents non-literal abstractions within the query.
  1 An `[`option`]` can be included or excluded. The brackets are not literal.
  2 A `(`list `|` of `|` alternatives`)` must include exactly one alternative. The parentheses and pipes are not literal.
  3 A first case, a second case, `...` a final case implies a pattern or repetition. The ellipsis is not literal.

C A **keyword** is shown in `UPPER` case but must be implemented *case-insensitive* (all case variants are equivalent).

D A **non-keyword** is shown in `lower` case but must be implemented *case-sensitive* (each case variant is distinct).

E A valid **name** of any kind is one letter (`a` to `z` or `A` to `Z`) followed by zero or more letters, digits (`0` to `9`), or underscores (`_`).
  1 When a special case requires, a name can start with an underscore instead of a letter.
  2 The length of a name must be 1 to 15 characters.

F Any **whitespace** is required or optional depending on the context and includes spaces, tabs, and line breaks.
  1 Whitespace is *required* when it disambiguates the boundaries between literals.
  2 Otherwise, whitespace is *optional* unless it ambiguates the boundaries of a literal.

Glossary of Structures

This glossary explains the most important structures in order of easiest comprehension.

A A **table** is a **data or file structure** which is modeled as follows (see the example table below).

example_table		
ps*	**i**	**b**
"abc"	5	false
"xyz"	-1	
"123"	0	true

Schema

Table Name	`"example_table"`
Column Names	`["ps", "i", "b"]`
Column Types	`["string", "integer", "boolean"]`
Primary Index	`0`

State

```
{
  [key "abc", 5, false],
  [key "xyz", -1, null],
  [key "123", 0, true]
}
```

1 The table is organized into columns and rows.
 i A **column** is a vertical abstraction which represents a named **field** with a data type. The columns are ordered within the table from first column to last column.
 ii A **row** is a horizontal abstraction which represents one **field value** per column in the table. The rows are unordered within the table, but the field values are ordered within each row corresponding with the order of the columns.
 iii The **primary column** is the single column in which each field value must be distinct across all rows and not null. The field value in the primary column in each row is called the **key** of that row.

2 The **schema** defines the following properties of the table.
 i The **table name** (string).
 ii The **column names** (list of distinct strings) ordered as the columns within the table.
 iii The **column types** (list of `"string"`, `"integer"`, or `"boolean"`) ordered as the column names.
 iv The **primary index** (integer) corresponding to the index of the primary column within the column names.
 v The schema can be extended to include any other properties that are useful and effective.

3 The **state** defines the rows of the table.
 i The state is a specialization of a **map** (also known as a dictionary or associative array), which is an association of unique keys of a consistent type with corresponding values of a consistent type.
 ii The state maps each distinct field value in the primary column (object) with a row of field values (list of objects).
 iii Unlike a conventional map, each key is stored within its corresponding row at the index of the primary column.

4 The storage and mutability of the table depend on its use case.
 i A conventional **table** is stored in a database and is **mutable** (able to be mutated). It should be a hash file table if available and if the database is persistent, or else a hash array table if available, or else a search table.
 ii A **result table** is not stored in a database and is **immutable** (unable to be mutated). It should be a hash array table if available, or else a search table.

B A **database** is a both a set of tables and an interpreter to operate upon them (the database management system). If the database is **persistent**, then its tables are in persistent storage. Otherwise, its tables are in volatile memory.

C A **query** is a string which contains a single instruction in Structured Query Language (SQL) to access or mutate the database. The SQL used in the project is roughly a subset of the SQL standard, with a few deviations to fit the learning outcomes.
 1 An **accessor query** accesses the database. Its response includes a result table.
 2 A **mutator query** mutates the database. Its response includes no result table.

D A **script** is a string which contains one or more queries with semicolons (`;`) separating or terminating each.

E A **driver** is a component within the database management system which interprets a specific type of query by parsing its syntax using a regular expression, evaluating its semantics using program logic, and then returning its response.

F A **response** is a structure which describes the result of a query interpreted by a driver. When the database management system interprets a list of queries, it returns an ordered list of the responses from its drivers. The response properties follow.
 1 The **query** (string) which was interpreted.
 2 A **status** which indicates the result of the interpretation. The enumeration of statuses follows.
  i A **successful** query was recognized (syntactically valid) and executed without errors (semantically valid).
  ii A **failed** query was recognized (syntactically valid) but executed with errors (semantically invalid).
  iii An **unrecognized** query was unable to be recognized (syntactically invalid) and thus unable to be executed at all.
 3 A **message** (string) which explains the result of the query in natural language, or null if there is no explanation provided.
 4 A **result table** (table reference) created by the query, or null if there is no result table.

G A **console** is a facility for reading queries from a user or other input source, sending queries to a database management system to be interpreted, and displaying their responses to a user or other output source.

MODULE 1 | HASH ARRAY TABLE

Topics: *Data structures, tables, hashing, collision resolution, rehashing, analysis of algorithms*

This module is **graded by unit tests** with the **M1** tag based on the percentage of unit tests passed.

Implement `tables.HashArrayTable` to create a **hash table** as a **data structure**.

A Use an **array** for your hash table.
  1 The array has a **capacity** (its length field) which is always a prime number.
    i Initialize the capacity to a prime number less than 20 (this intentionally forces rehashing soon and often).
    ii *With alternating sign quadratic probing,* use a prime number congruent to 3 modulo 4.
    iii *With cuckoo hashing,* use either two arrays or one array with two address spaces, each with a prime capacity.
    iv *With coalesced chaining,* the address space has a prime capacity and the cellar is at least 15% of the array.
  2 The array has a **size** (integer companion variable) which counts the number of rows in the state.
    i *With removal using tombstones,* the array also has a **contamination** (integer companion variable) which counts the number of tombstones separately from the size.
  3 The array is an `Object[]` populated with elements of the following type.
    i *With any open addressing technique,* each element is a row (list of objects), or null.
    ii *With any separate chaining technique,* each element is a chain (collection of rows), or null.
    iii *With coalesced chaining,* each element is a row (list of objects) paired with a reference to the next element, or null.

B Implement the **constructor** and **methods** according to their documentation.
  1 The 4-ary constructor creates a table by accepting a table name, column names, column types, and primary index in that order as parameters, calling the corresponding methods to initialize the schema properties, and initializing the array.
  2 All algorithms run in **constant time** (or in amortized constant time) when possible or in **linear time** otherwise.
  3 Override inherited methods only when necessary or useful.
  4 Implement helper methods when appropriate to promote code reuse.

C Implement a **hash function** as follows.
  1 For any string key, the hash function returns an integer computed directly from that key object using an original algorithm. The algorithm is deterministic, chaotic, onto, uniform (or approximately so), constant time, and free of modulo bias.
  2 For any non-string key, the hash function returns a call to the `hashCode` method of that key object.
  3 *With double hashing or cuckoo hashing,* also implement a second hash function which returns p (a constant prime number smaller than the hash table capacity) minus the first hash function of the key modulo p.
  4 Do not conflate the hash function with the unrelated `hashCode` method which returns the fingerprint of the table.

D Use a **collision resolution** technique from these options with the given **load factor** upper bound.
  1 Open addressing techniques: alternating sign quadratic probing (75%), double hashing (75%), or cuckoo hashing (50%).
  2 Chaining techniques: separate chaining with lists (5), separate chaining with arrays (5), or coalesced chaining (50%).
  3 Follow any *italicized* special requirements for your chosen technique.

E Perform a **rehash** whenever the load factor is higher at the end of a `put` than the given upper bound.
  1 When rehashing, at least double the capacity while keeping the capacity prime.
  2 *With removal using tombstones*, include the contamination when computing the load factor, discard all tombstones when rehashing, and reuse the first available tombstone when creating or updating a row.

MODULE 2 | HASH FILE TABLE

Topics: *File structures, tables, random access files, memory mapped files, records, bytes*

This module is **graded by unit tests** with the **M2** tag based on the percentage of unit tests passed.

Implement `tables.HashFileTable` to create a **hash table** as a **file structure**.

- A Duplicate your `tables.HashArrayTable` implementation to reuse most of its code.
- B Follow the same requirements as for MODULE 1 with the exceptions below.
  - 1 Do not use an array. The rest of this module explains what to use instead.
  - 2 Implement the constructors as defined in this module instead.
  - 3 If possible, avoid separate chaining techniques in favor of open addressing techniques or coalesced chaining.
  - 4 Rehashing is optional.
- C Use a random access **file channel** and a **memory mapped buffer** for the hash table.
  - 1 Use a `java.nio.channels.FileChannel` instance with one or more `java.nio.MappedByteBuffer` instances as your only file input/output mechanisms. When available, use `java.nio` alternatives to older `java.io` functionality.
  - 2 The **path** is a binary file named after the table name within a data folder which only contains file tables.
  - 3 The **header** stores the capacity, size, and any other companion variables of the table as well as its schema.
  - 4 The **records** are stored contiguously after the header. Each record is a fixed width encoding of a row in the table state.
- D Implement the **constructors** and **methods** according to their documentation.
  - 1 The 4-ary constructor creates a table by accepting a table name, column names, column types, and primary index in that order as parameters, calling the corresponding methods to initialize the schema properties, and initializing the file.
  - 2 The 1-ary constructor reopens a table by accepting a table name as a parameter, opening the file with the given table name, and initializing the schema properties and state rows by reading them from the file.
  - 3 Translate all array operations into equivalent operations on the file structure.
    - i Implement helper methods to write records and mark nulls, replacing array assignments with these.
    - ii Implement helper methods to read records and identify nulls, replacing array lookups with these.
    - iii *With removal using tombstones*, also implement helper methods to write and identify tombstones.
- E Allocate the **records** in the file structure as follows.
  - 1 Use a **bit mask** at the front of each record to distinguish whether the record is null and, if not, which of its fields are null.
    - i *With removal using tombstones,* the bit mask also distinguishes whether the record is a tombstone.
  - 2 Allocate each record with a minimal **fixed width** as a sequence of fields of the data types in the schema.
    - i Each string is 0 to 127 characters encoded in UTF-8 (each character is 1 byte) with either an embedded length or a terminating character. Each string in a schema property is only 0 to 15 characters.
    - ii Each integer is 32 bits (4 bytes).
    - iii Each boolean is 1 byte.

MODULE 3 | APPLICATIONS & TABLE VIEW

Topics: *Tables, iteration, string builders, truncation*

This module is **graded by inspection** with the **M3** tag based on the weighted percentages given.

[**20%**] Implement the `apps.Console.main` method to provide a REPL (read-evaluate-print loop) as an entry point.

A Prompt the user to input a script and split it on semicolons into a list of single queries. It is always safe to split on semicolons because they are used exclusively as delimiters between queries in a script and never within a single query.
B Send the list of queries to the database to be interpreted and receive back a list of responses.
C For each response in the list of responses, display all properties of the response.
D Repeat until the case-insensitive non-query sentinel `EXIT` is entered. Do not implement a driver for the sentinel.

[**20%**] Implement the `apps.Database` constructor and methods to provide a database management system with an interpreter.

E Implement the constructor as follows.
  1 If the database is persistent, then for each file in the data folder, reopen it as a hash file table using the 1-ary constructor and recreate it in the database. Otherwise, if the database is not persistent, then initialize the database without tables.
  2 Initialize the driver list with `ECHO`, `RANGE`, `DUMP TABLE`, and any other implemented drivers.
F Implement the `interpret` method to interpret a list of queries and return a corresponding list of responses.
  1 Create a new list of responses, initialized to empty.
  2 For each query in order, execute the query once per driver until a driver returns a successful or failed response. Then add that response to the list of responses, skip all remaining drivers for that query, and continue to the next query, if any.
    i Do not execute a query more than once per driver or else erroneous mutations may occur.
    ii If no driver returns a successful or failed response for a query, add an unrecognized response as a fallback.
  3 Return the list of responses, ordered to correspond with the list of queries.
G Implement all other methods according to their documentation.

To earn a grade on this portion, you must **demonstrate SQL queries** which prove the functionality of the console and database.

Implement the `tables.Table.toString` method to represent any table in a pretty format similar to the example below.

example_table		
word*	**length**	**is_noun**
"tabular"	7	false
"view"	4	null
"demonstration"	14	true

```
/ example_table \
+---------------------------------------------------+
| word*            |          length | is_noun      |
|===================================================|
| "tabular"        |               7 | false        |
| "view"           |               4 |              |
| "demonstrat...   |              14 | true         |
+---------------------------------------------------+
```

H [**20%**] The string representation of the table must be formatted as follows.
  1 It must be legible, attractive, and free of errors, but it should not be identical to the example above.
  2 It must be generated in efficient time using minimally nested loops.
  3 It must be generated in efficient space using a string builder instead of concatenation when possible.
I [**10%**] The header of the table must include the following features.
  1 The table name must be shown above the header.
  2 The column names must be shown in the header as ordered in the schema.
  3 The primary column must be indicated in the header.
J [**10%**] The columns of the table must include the following features.
  1 Each column must have a constant width.
  2 String and boolean columns must be left-aligned, but integer columns must be right-aligned.
K [**20%**] The rows of the table must include the following features.
  1 String fields must be wrapped in quotation marks to disambiguate them from integer and boolean fields.
  2 Null fields must be displayed as empty cells, not as null literals.
  3 If a string field is too wide for its column then it must be truncated with an ellipsis (`...`) to fit the width.

To earn a grade on this portion, you must **demonstrate a sandbox and/or SQL queries** that create and output the string representations of at least three nontrivial tables, each with at least three columns in its schema and at least five rows in its state.

You must also **prepare for an audit** of the hash tables from MODULE 1 and MODULE 2 at the discretion of the instructor.

MODULE 4 | TABLE DEFINITION

Topics: *Databases, data definition language (DDL) queries, regular expressions, table schemas*

This module is **graded by unit tests** with the **M4** tag based on the percentage of unit tests passed.

Implement drivers for each **mutator query** below.

```
CREATE TABLE table_name (column_def1, column_def2, ... column_defN)
```

A The `table_name` is any valid name not already in the database.

B Each `column_def` has the syntax `column_name (STRING|INTEGER|BOOLEAN) [PRIMARY]` indicating the name of a column, the data type of its field values, and whether it is the primary column for the table. There must be 1 to 15 defined columns, and exactly one of them must be indicated as the primary column for the table. Each `column_name` must be a valid name distinct from the others given. The order the columns are given indicates the order the schema defines them.

C The query should create a new table with the given schema in the database if there is not already a table with that name.

D In a successful response, explain the created table's name and number of columns in the message.

E In a failed response, explain the failure clearly in the message.

```
DROP TABLE table_name
```

F The `table_name` is any valid name belonging to a table that already exists in the database.

G The query should drop the table with the given name from the database if it exists.

H In a successful response, explain the dropped table's name and number of rows in the message.

I In a failed response, explain the failure clearly in the message.

Implement drivers for each **accessor query** below.

```
SHOW TABLES
```

J In a response (which is always successful), explain the database's number of tables in the message, and return this result table.

1 The result table is named `_tables`.
2 The first and primary column is a string named `table_name`.
3 The second column is an integer named `column_count`.
4 The third column is an integer named `row_count`.
5 Each row lists the corresponding table details in the database, if any.

K Refer to the example to the right for a hypothetical database with 3 tables.

_tables		
table_name*	**column_count**	**row_count**
"abc"	1	12
"wxyz"	4	0
"pq"	3	5

MODULE 5 | TABLE MUTATION

***Topics:** Databases, data manipulation language (DML) queries, regular expressions, data types, literals, sanitization, table states*

This module is **graded by unit tests** with the **M5** tag based on the percentage of unit tests passed.

Implement drivers for each **mutator query** below.

```
(INSERT|REPLACE) INTO table_name [(col1, col2, ... colN)]
VALUES (val1, val2, ... valN)
```

A The `table_name` is any valid name belonging to a table that exists in the database. This is the destination table.

B If the optional clause is given, the columns used are defined to be only the columns given in the order listed in the clause. Each `colI` is a valid column name in the destination table. There must be at least one column name, exactly one of the names must be the primary column, and no name may be repeated. The order of the column names need not match the order in the destination table's schema. Instead, they correspond to the order of the given `valI` values by position.

C Otherwise, if the optional clause is not given, the columns used are defined to be all the columns in the schema in the order listed in the schema. The scheme columns are treated as `col1` through `colN` in order.

D Each `valI` is a literal whose data type corresponds to that of the respective column `colI` by position in the destination table (from the optional clause, if given, or otherwise from the schema, as defined above). There must be exactly enough columns and values to associate `col1` with `val1`, `col2` with `val2`, and so on through `colN` with `valN`.

E The rules for the supported literal formats are as follows.

   1 A string literal is a sequence of zero or more Unicode characters in UTF-8 encoding enclosed in double quotation marks (`"`). To simplify parsing, you can assume a string never includes double quotation marks (`"`), semicolons (`;`), or commas (`,`). The length of the string must be 0 to 127 characters. Values in string columns must be stored as `String` objects. Do not store quotation marks as part of a string value's data.

   2 An integer literal is either a `0` or else an optional `+` or `-` sign followed by one or more digits without leading zeroes with a magnitude is from -2^{31} and 2^{31}-1 (in other words, an integer is `-2`, `-1`, `0`, `1` or `+1`, `2` or `+2`, or so on within the 32-bit signed integer bounds). Values in integer columns must be stored as `Integer` objects, not as strings of a sign and digits.

   3 A boolean literal is encoded as either `TRUE` or `FALSE` or any case variant thereof. Values in boolean columns must be stored as `Boolean` objects, not as strings of "true" or "false" words.

   4 A field value in a non-primary column can instead be `NULL` or any case variant thereof, which indicates a null (empty) field value. Empty field values must be stored as null references, not as strings of "null" words.

F The query inserts a new row with the given field value assignments into the destination table. However, if a row already exists with the primary column value of the new row, the behavior varies as follows.

   1 If the query uses the `INSERT` keyword, the query is invalid and the new row is not inserted.

   2 If the query uses the `REPLACE` keyword, the query replaces the existing row with the new one.

G In a successful response, explain the destination table's name and the number of rows inserted or replaced in the message.

H In a failed response, explain the failure clearly in the message.

I Ensure that the `SHOW TABLES` query still produces valid `row_count` results now that tables may contain rows.

Module 6 | Table Access

Topics: *Databases, data manipulation language (DML) queries, regular expressions, relational operators, aliasing, table states*

This module is **graded by unit tests** with the **M6** tag based on the percentage of unit tests passed.

Implement drivers for each **accessor query** below.

```
SELECT (*|col1 [AS alias1], col2 [AS alias2], ... colN [AS aliasN])
FROM table_name [WHERE lhs (=|<>|<|>|<=|>=) rhs]
```

A The table_name is any valid name belonging to a table that exists in the database. This is the source table.

B Each colI is a valid column name within the source table, with no restrictions on order, quantity, or repetition but with the requirement that the primary column is included at least once. If such a list of column names is not given then the asterisk (*) must be given instead, and in this case the column names default to the column names defined and ordered in the schema.

C Each aliasI is a valid column name used to rename colI in the result table without modifying the name of the original colI in the source table. If the given aliases or source column names would result in duplicate column names in the result table such that each column name is not distinguishable from each other, the query is invalid.

D If the WHERE clause is given, the selected rows are defined to be each row in the source table for which the given condition evaluates as true on the values within the row (this may include no rows, some rows, or all rows). Otherwise, the selected rows are defined to be each row in the source table as if the implied condition were always true (this includes all rows).

E The rules for evaluating a WHERE clause condition per row are as follows.

   1 The lhs is a valid column name in the source table. It evaluates as the field value in that column for the current row.

   2 The rhs is a string, integer, boolean, or null literal. It evaluates as defined in INSERT.

   3 The condition evaluates as false if the lhs or rhs is null.

      i This special case takes precedence over the following rules.

      ii This means that null is unequal to null or that all nulls are distinct from each other. This may be surprising or awkward, but it is necessary for databases because null can represent unknown, undefined, or inapplicable.

   4 The condition evaluates the comparison operators =, <>, <, <=, >, and >= as whether the lhs is equal, unequal, lesser, lesser or equal, greater, and greater or equal compared respectively to the rhs as defined below.

      i Compare two strings in ascending lexicographic order as defined by String.compareTo.

      ii Compare two integers in ascending numeric order as defined by Integer.compareTo.

      iii Compare two booleans in ascending canonical order as defined by Boolean.compareTo.

      iv To compare two different types, convert each non-string to a string, then compare as two strings.

F In a successful response, explain the source table's name and the number of rows that were selected from it in the message, and return this result table.

   1 The result table is named _select.

   2 The schema contains only the selected columns of the source table.

   3 The state contains only the selected rows of the source table.

   4 The primary column is the same as in the source table (or the leftmost instance if repeated), with aliasing if applicable.

G In a failed response, explain the failure clearly in the message.

MODULE 7 | SERIALIZATION & DESERIALIZATION

Topics: *Serialization, deserialization, data interchange formats, XML, JSON*

This module is **graded by inspection** with the **M7** tag based on the weighted percentages given.

Implement drivers for each **accessor query** below.

A [**10%**] For **XML** serialization and deserialization, use `XMLStreamWriter` and `XMLStreamReader` in the `javax.xml.stream` package from the StAX API. The JDK includes a library that implements this API.
B [**10%**] For **JSON** serialization and deserialization, use `JsonWriter` and `JsonReader` in the `javax.json` package from the JSON-P API. The JDK doesn't include a library that implements this API, but dependency management provides one.
C [**10%**] File input/output errors must be handled gracefully.

```
EXPORT table_name (TO file_name.(xml|json)|AS (XML|JSON))
```

D [**10% for XML** + **10% for JSON**] Exports the table with the given table name to a file in either the XML or JSON format.
  1 For the `TO` form, the file name is explicitly the given name with an extension indicating the format. For the `AS` form, the file name is implied to be the name of the table with the extension indicated by the keyword.
  2 The exported file must be valid and standard for the format, not a custom format or a binary object.
  3 It must include enough schema properties and state data to precisely replicate the table during a future import.
  4 It must support tables with nontrivial schemas and states including multiple rows, multiple columns, and all data types.
  5 In a successful response, explain the table name, file name, and number of exported rows in the message, and return the result of executing `SELECT * FROM table_name` on the corresponding driver, but with `_export` as the result table name.
  6 In a failed response, explain the failure clearly in the message. The query can fail if a table with the given name doesn't exist, the file name is already in use, the given file format is not supported, there is a file input/output exception, or so on.
E [**10% for XML** + **10% for JSON**] Each file format must be used efficiently.
  1 The exported file must segregate the schema from the state.
  2 It must be as compact as reasonable by avoiding redundancy and cruft.
  3 It must serialize all primitive data types, data structures, and nulls into the equivalent native representation supported by the file format, not into custom encodings requiring additional parsing during deserialization. These requirements do not prohibit whitespace or pretty-printing for legibility, which you should use when available.

```
IMPORT file_name.(xml|json) [TO table_name]
```

F [**10% for XML** + **10% for JSON**] Imports a new table into the database from an XML or JSON file with the given file name.
  1 For the `TO` form, use the given table name as the new table name. Otherwise, use the table name inside the given file.
  2 The new table must be precisely replicated from the schema and state in the file such that it is identical to the original table that was exported to produce the file, except possibly for the table name depending on the form of the query.
  3 In a successful response, explain the table name, file name, and number of imported rows in the message, and return the result of executing `SELECT * FROM table_name` on the corresponding driver, but with `_import` as the result table name.
  4 In a failed response, explain the failure clearly in the message. The query can fail if the file is invalid, the given table name is invalid, there is an input/output exception, or so on.
G [**10%**] When importing an XML or JSON file, if a table name is already in use then generate a new name to use instead by appending a suffix of `_1`, `_2`, `_3`, or so on, using the minimum integer necessary to avoid conflict with existing table names.

To earn a grade, you must **demonstrate SQL queries** that create at least three nontrivial tables as defined in previous modules and exports and imports those tables in both formats in a variety of ways which prove the functionality of the drivers.

You must also **prepare for an audit** of the drivers from MODULE 4, MODULE 5, and MODULE 6 at the discretion of the instructor.

ALTERNATIVE STEPS

You can replace steps D and F for either the XML format or the JSON format with the following alternative steps for the SQL format. If you do so, you are assessed for steps D and F but not for step A/B or step E for the format you replaced.

E [**10% for SQL**] Exports the table to a file in the SQL format as in the original step D with the following modifications.
  1 Serialize the table using a sequence of `CREATE TABLE` and `REPLACE INTO` queries which replicate the schema and state.
  2 Generate the queries directly from the schema and state, not by logging any queries previously executed on the table.
G [**10% for SQL**] Imports a new table into the database from an SQL file as in the original step F with the following modifications.
  1 Deserialize the table by sequentially executing the queries in the file on the database.
  2 If the table name in the queries is already in use then ignore only the `CREATE TABLE` query.

HONORS 1 | ENHANCEMENTS TO MODULE 4

Topics: *Advanced learning outcomes from Module 4*

This module is **graded by inspection** with the **H1** tag based on the weighted percentages given.

Implement drivers for each **accessor query** below.

```
DESCRIBE TABLE table_name
```

A [**20%**] In a response (which is always successful), explain the given table's name and number of columns in the message, and return this result table.
1 The result table is named `_columns`.
2 The first and primary column is an integer named `index`.
3 The second column is a string named `name`.
4 The third column is a string named `type`.
5 The fourth column is a boolean named `is_primary`.
6 Each row lists the corresponding details for a column in the table.

_columns			
index*	**name**	**type**	**is_primary**
0	"ps"	"string"	true
1	"i"	"integer"	false
2	"b"	"boolean"	false

B Refer to the example to the right which corresponds to the example table in the glossary.

Implement drivers for each **mutator query** below.

```
ALTER TABLE table_name ADD COLUMN col_def [FIRST|(BEFORE|AFTER) other_col|LAST]
```

C [**20%**] Creates and adds a new non-primary column with the given definition (as in `CREATE TABLE`), widening all rows accordingly. The column is added either before the first column, before or after the given other column, or after the last column (the default case when the placement clause is omitted).

```
ALTER TABLE table_name DROP COLUMN column_name
```

D [**20%**] Removes the given non-primary column, narrowing all rows accordingly.

```
ALTER TABLE table_name MODIFY PRIMARY column_name
```

E [**20%**] Switches the primary column to the given column, but only if all field values within the column are distinct and not null.

```
ALTER TABLE table_name RENAME COLUMN old_col_name TO new_col_name
```

F **[10%]** Renames the given old column to the given new distinct column name.

```
ALTER TABLE table_name RENAME TO new_table_name
```

G [**10%**] Renames the table to a given new distinct name.

To earn a grade, you must **demonstrate an SQL script** with sufficient examples to cover the requirements.

Honors 2 | Enhancements to Module 5

***Topics:** Advanced learning outcomes from Module 5*

This module is **graded by inspection** with the **H2** tag based on the weighted percentages given.

Implement drivers for each **accessor or mutator query** below.

A Enhance the syntax of column definitions in `CREATE TABLE` so that `INTEGER` is replaced with `INTEGER [AUTO_INCREMENT]` where the optional `AUTO_INCREMENT` keyword modifies the semantics of an integer column as follows.

1 [**20%**] Whenever a table mutation (such as inserting, replacing, or updating a row or adding a column) would cause a field in the column to have a null value, that field instead defaults to the next available integer, which is 1 greater than the maximum integer previously assigned in the column, or defaults to `0` if there are no previous assignments in the column.

2 [**10%**] The next available integer must be found in amortized constant time by defining an additional schema property to track the maximum integer previously assigned in each column.

B Enhance the syntax and semantics of `CREATE TABLE` and related queries to support a `DECIMAL` data type.

1 [**10%**] A decimal literal is the syntax of an integer literal followed by a point (`.`) and then one or more digits. Values in decimal columns must be stored as `BigDecimal` objects of arbitrary scale, not as strings.

2 [**20%**] Support a `DECIMAL(scale)` parameterized data type, which is the same as the `DECIMAL` data type except that it applies a given scale (number of digits after the point). The `scale` is an integer literal with a value of 0 or greater.

```
CREATE TABLE new_table LIKE old_table
```

C [**10%**] Creates a table with the given new name.

1 Copy the schema (excluding the table name) from the given old table in the database.

2 Leave the state empty.

Implement drivers for each **mutator query** below.

```
(INSERT|REPLACE) INTO table_name [(...)] VALUES (...), (...), ... (...)
```

D [**20%**] As defined in the original `INSERT INTO` and `REPLACE INTO` except that more than one row can be inserted or replaced using a single query, with each row represented in its own set of parentheses in a list after the `VALUES` keyword.

```
TRUNCATE table_name
```

E [**10%**] Truncates the given table `table_name` by clearing its state.

To earn a grade, you must **demonstrate an SQL script** with sufficient examples to cover the requirements.

Honors 3 | Enhancements to Module 6

Topics: *Advanced learning outcomes from Module 6*

This module is **graded by inspection** with the **H3** tag based on the weighted percentages given.

Implement drivers for each **accessor or mutator query** below.

A Support iterated selection using the `SELECT` query.
 1 [**10%**] Whenever the `SELECT` query returns a successful response with a result table, also store a copy of that result table renamed `_select` in the database.
 2 [**10%**] All queries must support `_select` as a given table when it exists in the database.

B Modify the `SELECT` query to support enhanced `WHERE` clause conditions as follows.
 1 [**10%**] Support conditions `WHERE lhs IS [NOT] NULL` which evaluates as whether the `lhs` is or isn't null respectively.
 2 This fulfills the intuitive behavior that `lhs (=|<>) NULL` lacks due to the distinctness of nulls.

```
CREATE TABLE new_table AS SELECT ... FROM old_table [WHERE ...]
```

C [**20%**] Creates a table with the given new name.
 1 Execute the nested `SELECT` query on the corresponding driver.
 2 Copy the schema (excluding the table name) from the result of executing the nested query.
 3 Copy the state from the result of executing the nested query.

Implement drivers for each **mutator query** below.

```
UPDATE table_name SET col1 = val1 [WHERE ...]
```

D [**20%**] Updates the given table to set the field value in the given column to the given string, integer, or boolean literal in the rows where the condition in the `WHERE` clause is met as defined in `SELECT` or in all rows if the clause is omitted.

```
UPDATE table_name SET col1 = val1, col2 = val2, ... colN = valN [WHERE ...]
```

E [**10%**] Support a list of multiple assignments separated by commas (`,`) as defined in the original `UPDATE`.

```
DELETE FROM table_name [WHERE ...]
```

F [**20%**] Deletes the rows in the given table where the condition in the `WHERE` clause is met or in all rows if the clause is omitted.

To earn a grade, you must **demonstrate an SQL script** with sufficient examples to cover the requirements.

Extra 1 | Transactions

Topics: *Transactions, reference manipulation, time and space analysis, deep and shallow copying*

This module is **graded by inspection** with the **X1** tag based on the weighted percentages given.

Implement drivers for each **transaction query** below (which behave like mutator queries but do not count as mutator queries).

A [**20%**] Each database copy must be shallow (not deep) by reusing nested object references when possible.

```
BEGIN
```

B [**20%**] Begin a transaction by creating a copy of the database (the backup copy). If another transaction is already in progress, the query fails.

```
SAVEPOINT name
```

C [**10%**] Continue the transaction by creating a copy of the database associated with the given name (a savepoint copy). Unlimited savepoints can be created during a transaction, but each savepoint name must be unique, and a savepoint name cannot be reused. If no transaction is in progress or the given name already exists, the query fails.

```
COMMIT
```

D [**20%**] End the transaction by destroying the backup copy and savepoint copies. If no transaction is in progress, the query fails.

```
ROLLBACK [TO SAVEPOINT name]
```

E [**20%**] If the optional clause is omitted, end the transaction by replacing the database with the backup copy. If no transaction is in progress, the query fails.

F [**10%**] Otherwise, if the optional clause is given, continue the transaction by replacing the database with the savepoint copy associated with the given name. If no transaction is in progress or the given name does not exist, the query fails.

To earn a grade, you must **demonstrate an SQL script** with sufficient examples to cover the requirements.

EXTRA 2 | TABLE JOINS & EXPRESSIONS

Topics: *Set theory, joins, inner joins, equijoins, elliptical and fully-qualified notation*

This module is **graded by inspection** with the **X2** tag based on the weighted percentages given.

Implement drivers for each **accessor or mutator query** below.

A [**10%**] Expressions are the sum or concatenation (+), difference (-), product (*), or quotient (/) of two or more fields or literals.
   1 [**10%**] Support expressions with or without aliases within the list of columns in the `SELECT` clause.
   2 [**10%**] Support expressions on the right hand side of assignments in the `SET` clause.
   3 [**10%**] Support expressions and columns on the left and right hand sides of conditions in the `WHERE` clause.

Implement drivers for each **accessor query** below.

```
SELECT ... FROM left_table
[JOIN right_table ON left_column = right_column] [WHERE ...]
```

B If the optional `JOIN` clause is omitted, this query behaves as the original `SELECT` query.
C [**20%**] The query must return a result table containing only rows built by concatenating each row from a left table with a corresponding row from a right such that the join columns share the same non-null value (this is called an inner equijoin).
   1 The `left_table` and `right_table` are distinct tables in the database referred to respectively as the left and right tables.
   2 The left table must have a column with the given column name `left_column`, and the right table must have a column with the given column name `right_column`. These columns are referred to as the join columns.
   3 The join columns must be the same data type.
   4 One or both of the join columns must be primary in its respective table.
D [**10%**] When a column name is ambiguous in any clause because it appears in both source tables, the usual elliptical notation `column_name` must fail. Support the fully-qualified notation `table_name.column_name` to disambiguate such columns.
E [**10%**] When an elliptical column name is ambiguous in the result table schema, use the fully-qualified column name instead.
F The primary column of the result table is as follows.
   1 [**10%**] If both join columns were primary, use the left join column as the primary.
   2 [**10%**] Otherwise, if one of the join columns was non-primary, avoid duplicates in the one that was primary by using a new column as the primary, filled with strings encoded in the form `(left_join_value, right_join_value)`.

To earn a grade, you must **demonstrate an SQL script** with sufficient examples to cover the requirements.