CS 210 » File and Data Structures

Modules
FALL 2016

INSTRUCTIONS

Complete each module detailed in this document using Java code (or Groovy if available) and submit your solution according to the deadline. This is a living specification, and any necessary revisions (including actual due dates and the finalization of draft material) will be edited into this document during the semester, so you are encouraged to check for revisions on eCampus regularly.

Submissions assessed via automated grading will be graded during a window of up to 14 days including the day the module is due, after which late submissions will be accepted for at most half credit for another 7 days not to exceed the last day of classes. An exception is the Final Module, whose grading window is from approximately the semester midpoint through the day of the scheduled final session.

Submissions assessed via inspection will be graded during lectures, labs, appointments, or other designated sessions during a window of 7 days including the day the module is due, after which no late submissions will be accepted for credit.

Your code will be assessed based on its correctness and conformity to the specification. It will be compared to that of your peers and online examples to detect plagiarism. You must understand your own code thoroughly and be able to explain all aspects of it.

PROJECT STRUCTURE

Arrange your files according to the following project structure when submitting your code for automated grading. You will be provided a framework of files by your instructor which complies with this structure. Any additional files you implement should be organized appropriately, including unit tests or any optional files or samples provided by the instructor.

`src/`	**implementation organized into appropriate packages**
`adt/`	**package for abstract data types**
`HashMap.java`	required, do not modify protocols, must implement methods
`Database.java`	required, do not modify protocols, may extend features
`Table.java`	required, do not modify protocols, may extend features
`Row.java`	required, do not modify protocols, may extend features
`Response.java`	required, do not modify
`core/`	**package for core functional components**
`Console.java`	required, do not modify protocols, must implement methods
`Server.java`	required, do not modify protocols, must implement methods
`driver/`	**package for query drivers corresponding to required queries**
`Driver.java`	required, do not modify
`test/`	**optional, unit testing code arranged to mirror the `src` hierarchy**

Modules graded using automated testing will directly assess the `Server` class, while modules graded using inspection will directly assess the `Console` class. All other aspects of implementation will be assessed indirectly as dependencies of those two classes.

TABLE SCHEMA

All tables for the project, whether stored or computed, must contain their own schemas according to the following standardized format. Each table should map its `null` key to a map containing the following entries. This row constitutes the schema for the table.

- A string key `"table_name"` whose value is a string containing the name of the table or `null` for a computed table.
- A string key `"primary_column_name"` whose value is a string containing the name of the primary key column, which is not required to be the first column in the list of ordered column names.
- A string key `"column_names"` whose value is a `java.util.List` of strings containing the ordered names of the columns.
- A string key `"column_types"` whose value is a `java.util.List` of strings containing the ordered type names respectively according to the list of ordered column names, using the string values `"string"`, `"integer"`, or `"boolean"` as appropriate.

SYNTACTIC CONVENTIONS

By convention in this document, keywords are shown in `UPPER` case but must be case-insensitive, and names and data are shown in `lower` case but must be case-sensitive. Regardless of context, a valid name is understood to be a letter followed by zero or more letters, digits, or underscores. Whitespace is only required where it is necessary to separate tokens. Metasymbols are used to indicate `[optional]` portions, to indicate one of `(several | possible | alternatives)`, and to imply `...` sequences. They are not part of actual query syntax.

MODULE 1 » TABLES

Automated **10:00pm Sept. 30**

TOPICS: RELATIONAL DATABASES, DATA DEFINITION QUERIES, REGULAR EXPRESSIONS, TABLE SCHEMAS.

Enhance the provided `core.Server` class according to the following specification.

Encapsulate a non-static, volatile `adt.Database` instance within each server instance. Note that the database class is an alias of the provided noncompliant hash map implementation. In the future as part of the Final Module you will replace the noncompliant hash map with a compliant one, but until then you may continue to use the noncompliant one as the basis of all of your other modules.

Implement the `database` method to return a reference to the database instance associated with the server such that the database can be both accessed and mutated externally to the server.

Implement the `interpret` method to accept a script string which is composed of a semicolon-delimited list of one or more queries (semicolons will be used exclusively as delimiters between queries and will never appear within a single query) and return a `java.util.List` of one `adt.Response` instance per each of the queries in the script in execution order.

Implement new drivers to support the following queries according to the given syntax and semantics. In subsequent modules, you will further enhance the class by adding support for additional queries to the `interpret` method by implementing more drivers.

`CREATE TABLE table_name (column_def, column_def, ..., column_def)`

- The `table_name` is any valid name not already in the database.
- Each `column_def` has the syntax `[PRIMARY] (STRING|INTEGER|BOOLEAN) column_name` indicating whether the column is the primary column for the table, the data type of the column's field values, and the name of the column. There must be at least one `column_def` among which exactly one must be the indicated primary column for the table. Each `column_name` must be a valid name distinct from the rest. The order the columns are given indicates the order they must be defined in the schema.
- The query should add a new stored table with the given name and column definitions to the database backed by the server if there is not already a table with the given name. The created table must include its own schema according to the standard below.
- In a successful response, state the name of the created table and how many columns it has, and return the empty created table.
- In a failed response, state an appropriate explanation of the failure, and do not return any table.

`DROP TABLE table_name`

- The `table_name` is any valid name belonging to a table that already exists in the database.
- The query should remove the stored table with the given name from the database backed by the server if there is such a table.
- In a successful response, state the name of the dropped table and how many rows it had, and return the dropped table.
- In a failed response, state an appropriate explanation of the failure, and do not return any table.

`SHOW TABLES`

- The query should always succeed, because it is a pure accessor of data that is always defined and cannot possibly fail.
- In a successful response, state how many tables there are, and return a computed table whose rows list all table names in the database as the primary column values with their respective row counts (refer to the example map view and tabular view to the right).

```
{
  abc → {
    table_name → abc,
    row_count → 20
  },
  pq → {...},
  table_1 → {...},
  table_2 → {...},
  wxyz → {...}
}
```

table_name*	row_count
abc	20
pq	8
table_1	12
table_2	11
wxyz	17

Note that any invalid or unrecognized queries should return a failure response with a message indicating a reason for the failure.

Enhance the provided `core.Console` class according to the following specification.

Implement the `prompt` method to provide a prompt at which the user can enter one or more semicolon-delimited queries per line to be processed by a locally constructed server instance. Users should be able to enter one or more queries and see their immediate responses from the server's interpretation until terminating the console using the non-query sentinel value `EXIT` which does not require a driver. Responses should be displayed in a legible format including the success flag, a message if any, and the map-view of a table if any.

MODULE 2 » INSERTION

Automated **10:00pm October 14**

TOPICS: MUTATION QUERIES, DATA TYPES, NESTED MAPS.

Implement new drivers to support the following queries according to the given syntax and semantics.

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

- The `table_name` is any valid name belonging to a table that exists in the database. This is the destination table.
- Each `colI` is a valid column name in the destination table. There must be at least one column name, exactly one of the column names must be the primary column, and no column name may be repeated. The order of the column names need not correspond to the order in the destination table's schema but rather corresponds with the order of the given `valI` values by position. If the column names clause is omitted, the column names default to the column names defined and ordered in the schema.
- Each `valI` is a field value whose data type corresponds to the data type of the respective column `colI` in the destination table. There must be enough columns and values to assign `col1` to `val1`, `col2` to `val2`, and so on through `colN` to `valN` exactly.
  - The format for a String value is any finite Unicode character sequence (excluding internal `"` or `;` or `,` characters which will never be included in any string data to simplify processing) enclosed in double quotation marks. Valid examples are `""` and `"a string"` and `"'hi'"` but not `";"` and `""hi""` which contain invalid characters.
  - The format for an Integer field is `0` or any contiguous sequence of digits starting with a non-zero digit optionally preceded by a `-` or `+` sign. In other words, an integer is `-2`, `-1`, `0`, `1` or `+1`, `2` or `+2`, or so on without any defined upper or lower bounds.
  - The format for a Boolean field is either `true` or `false` or any case variant thereof.
  - The format for *any* type of field can also include `null` or any case variant thereof, which indicates an unmapped field value.
- The query should insert a new row with the given field assignments into the destination table as long as doing so would not create a duplicate primary column value. Note that a given field value of `null` indicates that the field's value in the row is to be left unmapped. In other words, for space efficiency, do not actually map a field to null. Instead, skip mapping the field at all.
- In a successful response, state the destination table name and the number of rows that were inserted into the table, and return a computed table with a clone of the schema from the destination table (excluding the table name, since this is a computed table and it should not purport to be the destination table) and exclusively the newly inserted rows of the destination table.
- In a failed response, state an appropriate explanation of the failure, and do not return any table.

OPTIONAL

`DUMP TABLE table_name`

- The `table_name` is any valid name belonging to a table that already exists in the database. If the given source table exists, the query succeeds and returns it from the database. Otherwise, the query fails with an appropriate explanation.

Ensure that the `SHOW TABLES` query still produces valid `row_count` results.

MODULE 3 » SELECTION

Automated **10:00pm October 28**

TOPICS: ACCESSOR QUERIES, VALIDATION ALGORITHMS, SHALLOW CLONING.

Implement new drivers to support the following queries according to the given syntax and semantics.

`SELECT (*|col1 [AS alias1], col2 [AS alias2], ..., colN [AS aliasN]) FROM table_name`

- The `table_name` is any valid name belonging to a table that exists in the database. This is the source table.
- 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 `*` symbol must be given instead, and in this case the column names default to the column names defined and ordered in the schema.
- Each `aliasI` is a valid column name used to rename `colI` in the computed 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 computed table such that each column name is not distinguishable from each other, the query is invalid.
- The query should return a computed table containing all rows of the source table but only the indicated columns, which may be more or fewer than existed in the source table and which may have different column names than the source table due to aliases. The primary column of the computed table should be the primary column of the source table (or the first of its repeated instances).
- In a successful response, state the source table name and the number of rows in the computed table (not the source table, even though in this case those numbers are the same), and return the computed table, including an appropriate schema.
- In a failed response, state an appropriately descriptive error, and do not return any table.

Ensure that the `INSERT` query produces valid source tables for selection.

MODULE 4 » TABULAR VIEWS

Inspection **10:00pm November 4**

TOPICS: TABULAR REPRESENTATION OF MAPS, TRUNCATION, MAP VIEWS, MAP TRAVERSALS.

Modify the `prompt` method to output a pretty-printed tabular view for any stored or computed table returned by an executed query rather than outputting the raw map view of such a table. That is, display tables in a text-based format similar to the example below.

word*	length	is_noun
cat	3	true
tabular	null	false
defenestration	14	false

```
 /table_name\
+-----------------------------------------+
| word*          | length     | is_noun    |
|=========================================|
| "cat"          |          3| true       |
| "tabular"      |           | false      |
| "defenestr..." |         14| false      |
+-----------------------------------------+
```

Although the appearance of the pretty-printed tabular view is not required to exactly match the example shown above, it should be legible and comply with the following requirements. The tabular view must show the columns with the ordering and names (or aliases) defined in the table's schema and should indicate which column is primary. If a field value or column name is wider than the visual width of its column then it must be truncated. Unmapped or `null` fields should be displayed blank instead of displaying the word `null`. Each field value should have a discernible type, meaning that strings should display quotation marks to disambiguate them from integers and booleans. Integer types should be right-aligned, while string and boolean types should be left-aligned within their column.

You must prepare a series of queries that demonstrate that your solution fulfills each of these requirements for grading purposes. You may ask for feedback from the instructor as necessary, but first ensure that you meet all the listed requirements.

Additionally, you must ensure that your `Console` behaves as required by previous module specifications.

MODULE 5 » DATABASE PERSISTENCE

Inspection **10:00pm November 18**

TOPICS: SERIALIZATION AND DESERIALIZATION, XML, JSON, FILE STRUCTURES AND LOGS.

Implement new drivers to support the following queries according to the given syntax and semantics.

`EXPORT table_name file_name`

- Exports the table with the given table name to a file with the given file name in either XML or JSON format (your choice).

`IMPORT file_name`

- Imports a new table from the file with the given file name into the database if the table name does not already exist.

For the above query drivers, you are encouraged to use any reputable XML or JSON libraries for Java.

Separately, add support for database persistence such that the database is maintained in storage while the `Server` is powered down (not running) and can be reloaded when the `Server` is next powered up (running). Expected power downs can be handled by automatically serializing from memory to storage when powering down and automatically deserializing from storage to memory when powering up. Unplanned power downs can be handled by maintaining a log script of all mutation queries that have been processed since the last serialization and using this log after an unexpected `Server` power down to recreate the missing mutations when deserializing.

Each table in the database should be serialized to its own compact file. The `Table` class should be made `Serializable` and custom `readObject` and `writeObject` methods should be implemented in such a way that they are as time and space optimal as feasible.

Note that, for both a table's schema and its rows, only the values need to be serialized, not the keys. This is because the schema keys are already known by definition and because the row keys can be determined during deserialization based on the column order in the schema and the order in which the fields of the row were serialized. Also, only primitive data types and strings should be serialized, not data structures such as arrays, lists, or maps. Serializing map keys or data structures would not be space optimal.

MODULE 6 » MUTATION AND CONDITIONS

Automated TBA

TOPICS: MAP TRAVERSAL, SEARCH STRATEGIES.

Implement new drivers or modify previous drivers to support the following queries according to the given syntax and semantics.

`UPDATE ... WHERE` › Details are pending.
`DELETE FROM ... WHERE` › Details are pending.
`SELECT FROM ... WHERE` › Details are pending.

Continue progress on any other components that were not yet completed correctly for previous modules, as this module may expose previously unknown flaws in previous modules.

For this semester, there will be no graded Module 6. Your instructor will discuss the grading adjustments due to this removal.

MODULE 7 » CHOSEN FEATURE

Last Day of Class

TOPICS: MISCELLANEOUS.

Select one supplement from the supplemental modules section of this specification and complete it as your submission for this module. You may not pick a supplement that you are already completing for another purpose, such as for a substitution or honors credit.

This module will be graded via automated testing or via inspection as per the instructions given for your chosen supplement.

TOPICS: MAPS, HASHING, COLLISION RESOLUTION TECHNIQUES, LOAD FACTORS, ITERATORS, UNIT TESTING.

Write a generic class `adt.HashMap<K,V>` which implements `java.util.Map` according to the official API documentation for Java 6 (or later if available). Your class must implement hash-based mapping from scratch without utilizing any prewritten hash functions, libraries, abstract data types, or other code which circumvents the learning goals of the project, with the exceptions that you may reuse any code provided by the instructor or API classes explicitly authorized by the instructor, such as `java.util.AbstractSet` and `java.util.AbstractCollection` for implementing the three view methods of the map. The instructor reserves the right to reject your solution in whole or part, even after grading, if it is not sufficiently original, if it uses unauthorized content or techniques, or if you are incapable of accurately explaining the functionality of your submitted solution.

Null keys and values are to be fully supported with the sole exception that a map's `null` key should never be exposed in any traversals provided by its view methods or corresponding iterators. This is because this project uses the `null` key for reserved purposes. Effectively, this means that when implementing the iterators for the view methods the `null` key should be skipped over, and all mutators and accessors should be implemented with null-safety for both keys and values.

You may implement your hash function(s) and data structure(s) as you see fit as long as they facilitate constant time for the canonical operations and as long as you use a collision resolution technique covered in lecture or approved by the instructor. For any string key, your hash function must be an original algorithm not utilizing the string object's `hashCode()` method. For any non-string key, your hash function should simply return a call to the key object's `hashCode()` method. Note that your hash function, which should be a private method with an unambiguous but meaningful name such as `myHash()`, `keyHash()`, or `localHash()` and which provides hashes for keys within your table, has no relationship to the required public methods `Map.hashCode()` and `Map.Entry.hashCode()`, which must be implemented according to the Java documentation and which provide hashes for the map and entry structures themselves.

You must implement your own class conforming to the `Map.Entry` interface from scratch. It is recommended you do not call your class `Entry` to avoid accidental type hiding when `Map.Entry` is not fully qualified. Instead, pick an unambiguous but meaningful name such as `MyEntry`, `LocalEntry`, or `Tuple`. In addition to the aforementioned `hashCode()` methods, you must implement both `Map.equals()` and `Map.Entry.equals()` according to the Java documentation or else several other methods will fail to behave correctly. Note that your IDE may not automatically include stubs for the `equals()` and `hashCode()` methods. Do not use raw types when fully qualified generics are necessary for type safety, and ensure that all type warnings are fixed or suppressed appropriately.

To assess whether your hash function and collision resolution techniques perform well, consider the following suggestions, where the load factor α for a table is defined to be $\alpha = \frac{m}{n}$ for table size n with m entries, and where a chain at index i has m_i entries.

If you are using a chaining technique for collision resolution, compute the ideal load factor for the table (the length of each chain if all entries were uniformly distributed over the chains) and the average chain length $(\sum_i m_i)/n$ for the table (the average number of entries over all chains) and compare these figures. If the average chain length is close to the ideal load factor, your implementation performs well. Otherwise, consider whether your hash function is chaotic and explosive enough to facilitate a uniform distribution.

If you are using a probing (open addressing) technique, compute the ideal probing sequence length $1/(1-\alpha)$ for the table (the expected number of steps in a probing sequence for an absent key given uniform hashing) and the average probing sequence length for the table (measured as a running average over the table's lifetime) and compare these figures. If the average probing sequence length is close to the ideal length while $\alpha < 0.8$, your implementation performs well. Otherwise, consider refining your probing sequence to limit clustering.

Supplemental Modules

The following supplements are provided as choices for your Module 7 submission, as substitutions for required modules in special circumstances by instructor permission only, or as options for honors credit. You are not required to attempt or submit any supplements for your project, with the exception that you must choose one of them for your Module 7 submission requirement. Each supplement lists a suggested module with which it should be completed, but the actual due dates for non-required supplements are negotiable.

If you are seeking contract credit for the **honors program** or other extracurricular credit, you may attempt **two or more** honors-eligible supplements. To receive honors credit, you must adequately explain your designs and efficiency analysis to the instructor and receive a grade of B or better on each such submitted supplement. With instructor approval, you may combine multiple non-honors-eligible supplements together to constitute enough work to stand in for one honors-eligible supplement. You should confirm your supplement selections with the instructor to ensure that your plans meet the requirements for honors credit before submission.

Supplement 1A » Enhanced Schemas

🎓 **Honors** 🔍 **Inspection** 📅 **With Module 1**

Topics: schema modifications, static incrementation.

Implement new drivers or modify previous drivers to support the following queries according to the given syntax and semantics.

`ALTER TABLE table_name INSERT COLUMN column_def (FIRST|AFTER pred_col_name|LAST)`

- Alters the table with the given name to add a new non-primary column defined by `column_def` (as in the `CREATE TABLE` query) to the table's schema. The column is either inserted as the first column, as the successor to `pred_col_name`, or as the last column. The query succeeds with a response including the stored table.

`ALTER TABLE table_name RENAME COLUMN old_name new_name`

- Alters the table with the given name to rename the existing column named `old_name` to a non-duplicate name `new_name` in the table's schema. The query succeeds with a response including the stored table.

`ALTER TABLE table_name DROP COLUMN column_name`

- Alters the table with the given name to drop the non-primary column named `column_name` from the table's schema and purge all field mappings for that column in all rows. The query succeeds with a response including the stored table.

`ALTER TABLE old_table_name RENAME TO new_table_name`

- Alters the table with the given existing name to rename it to the given new name.

Enhance all queries to support the new data type `AUTOINTEGER` which behaves exactly as an `INTEGER` except as follows. Whenever a table mutation would cause an `AUTOINTEGER` field to be unmapped or mapped to `null` then that field's value defaults instead to an integer not already in use in the column, such that the integer is the minimum integer greater than all other integers in the column (or `0` if the column is empty) and such that the integer can be computed in constant time.

Supplement 2A » Multiple Insertion

🔍 **Inspection** 📅 **With Module 2**

Topics: code reuse.

Implement new drivers or modify previous drivers to support the following queries according to the given syntax and semantics.

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

- As in the original `INSERT INTO` query except that more than one row can be inserted using a single query.

Ensure that traditional `INSERT INTO` queries still function correctly by returning a computed table containing all of the inserted rows for the query even when there are multiple rows given.

SUPPLEMENT 3A » ITERATED SELECTION

Inspection With Module 3

TOPICS: SIMULATED RECURSION.

Modify all `SELECT` queries that return a computed table such that the returned table is also stored into the database as a special table whose name is `_answer` right before returning the table. Allow `SELECT` queries to use `_answer` as a source table. Successive `SELECT` queries will replace `_answer` with the most recent query response table. In practice, the effect is to partially simulate recursive `SELECT` queries capable of selecting from the results of other `SELECT` queries, in this case using an iterated implementation.

Exclude the `_answer` table from all serialization and deserialization processes, but allow it to be manually exported and imported.

SUPPLEMENT 3B » JOINED SELECTION

Honors Inspection With Module 3

TOPICS: MAP INTERSECTION.

Implement new drivers or modify previous drivers to support the following queries according to the given syntax and semantics.

`SELECT ... FROM left_table_name [JOIN right_table_name]`

- The `left_table_name` and `right_table_name` are any two distinct valid names belonging to tables that exist in the database, referred to respectively as the left and right tables. These are the left and right source tables.
- In this context, the format `table_name.field_name` will be referred to as `field_name`'s *fully qualified* name, whereas `field_name` alone without the `table_name` to which it belongs is called its *elliptical* name. Fully qualified names are always unambiguous, but elliptical names are only unambiguous when the other table does not contain a field with the same name.
- The `...` above represents all of the column selection syntax previously supported between the `SELECT` and `FROM` keywords, including aliases, except that explicitly listed column names can include a mix of fully qualified and elliptical names from both the left and right tables as long as the resulting list is unambiguous and contains a primary key from either table. If instead the `*` is given then the list is implicitly taken to be all names in order from the left table followed by those from the right table, using elliptical names when possible and using fully qualified names where necessary to avoid an ambiguity between the two tables.
- The query should return a computed table containing only rows which can be built by joining a row from the left table with a row with a row from the right table such that the left row and right row share the same primary key value. In other words, the computed table is the map intersection of the left and right tables on the primary keys (which is properly called an *inner* join). The primary column of the computed table should be the first eligible primary key in the list of selected columns.
- In a successful response, state the left and right table names and the number of rows in the computed table (which will be the number of rows from the left and right tables that share a common primary key), and return the computed table.
- In a failed response, state an appropriately descriptive error, and do not return any table.

Ensure that traditional `SELECT` queries still work when `JOIN` is not used.

SUPPLEMENT 3C » ORDERED SELECTION

Honors Inspection With Module 3

TOPICS: MAP TRAVERSAL, SORTING ALGORITHMS.

Implement new drivers or modify previous drivers to support the following queries according to the given syntax and semantics.

`SELECT ... ORDER BY ...`

- Details are pending.

Enhance the `SHOW TABLES` query to order its rows by table name in alphabetical order.

Ensure that traditional `SELECT` queries still function and that the pretty-printed tabular view outputs table rows in their correct order.

SUPPLEMENT 4A » GRAPHICAL CONSOLE

Inspection With Module 4

TOPICS: GRAPHICAL USER INTERFACES.

Supersede the `Console` class with a new graphical user interface which contains a prompt and an area for displaying tabular results. When any query is executed that returns a table, display the table using the tabular region instead of text-based console output.

SUPPLEMENT 5A » TRANSACTIONS

Honors Inspection With Module 5

TOPICS: REFERENCE MANIPULATION, SPACE ANALYSIS AND EFFICIENCY.

Add support for the `BEGIN`, `COMMIT`, and `ROLLBACK` queries whose syntax respectively is just the single keyword. When a `BEGIN` is interpreted, a transaction is started (or the query is ignored if a transaction is already in progress). During a transaction, mutation queries do not permanently mutate the database. If a `COMMIT` is interpreted during a transaction, all mutation queries since `BEGIN` are permanently applied to the database and the transaction is ended. If a `ROLLBACK` is interpreted during a transaction, all mutation queries since `BEGIN` are discarded without mutating the database and the transaction is ended. During a transaction, access queries must still behave as if the intervening mutation queries have actually succeeded, even thought they might later be subject to `ROLLBACK`.

Support for transactions must be implemented in an efficient fashion using shallow copies and reference manipulation, since deep copies are inefficient in terms of both time and space analysis.

MISCELLANEOUS SUPPLEMENT

Inspection By Approval

TOPICS: MISCELLANEOUS.

Discuss with the instructor an appropriately significant database, server, or SQL feature that was not implemented in any previous module. With instructor approval, add the supplemental features to your existing project. The features should be nontrivial and should be considered functionality a realistic database server would be expected to provide. The design of the features should be tied into the learning goals for the course and should emphasize areas of weakness elsewhere in the curriculum.