Milestone 3 >

Detailed specifications

Contents

Table with Columns

The extended version of the storage server will support tables where the records can have multiple columns, each with possibly different data types.

Here's an example of a table to store information about the TTC subway lines. The key is the colour code of the line, and the record consists of columns for the name of the line, the number of stops in each line, and the length of the line in kilometres.

Key	Columns in record		
	Name	Stops	Kilometres
green	Bloor Danforth	31	26
yellow	Yonge University Spadina	32	30
purple	Sheppard	5	5

Configuration file

The only modification to the configuration file is that the table parameter must now specify not just the name of the table but the schema of each table. The schema of a table consists of the name of each column in the table, and the type of the column. The column name is a string of alphanumeric characters, and the type is one of the following:

- `char[SIZE]`: A string with a maximum length of SIZE characters (including the null terminating character). The string may only include alphanumeric characters and spaces.
- `int`: A signed integer corresponding to C's `int` type. The integer value may include an optional sign ("+" or "-").

A column name and its type are separated by a colon, while different column specifications are separated by commas, with optional whitespace before and after the comma. In addition to the columns (which are explicitly specified in the table schema), each table has an implicit *key* which is unique for each record in the table. The table keys are always alphanumeric strings and are used in various client library functions in order to access the data in the table. Here are some examples of the table parameter in the configuration file.

```
table subwayLines name:char[30],stops:int,kilometres:int
table cities lowTemperature:int,highTemperature:int,province:char[20]
table cars brand:char[10],price:int
table students id:int,grade:int
```

Note that the table specification only specifies the columns in the record. The key, which is always an alphanumeric string as in the previous milestones, is implicit. For example, in the `subwayLines` table, the subway line colour code which is used as the key in the above example, is not specified. Similarly, in the cities, the name of the city may be the key, but is not specified.

In case of duplicate table definitions (i.e., the same table is defined with different columns or column types), the server should exit with an appropriate error.

Here are some bad examples of the table parameter in the configuration file.

```
// two definitions of the same table with different columns
table subwayLines name:char[30], stops:int, kilometres:int
table subwayLines stops:int, kilometres:int

// two definition of the same table with different column types (Note
// the numbers 30 and 40 for char[])

table subwayLines name:char[30], stops:int, kilometres:int
table subwayLines name:char[40], stops:int, kilometres:int
```

Client library

The client library is also different in this version of the storage server in order to support the more complex table schema.

There is one new `storage_query` function whose prototype is as follows.

```
int storage_query(const char *table, const char *predicates, char **keys, const int max_keys, void *conn);
```

The table below summarizes the behaviour of the key functions. Note that the parameters passed to our `storage_get` and `storage_set` functions now have a different format as explained below the table.

Function	Parameters	Description	Return value
storage_query	`table`: A table in the database. `predicates`: A comma separated list of predicates. See below for the `predicates` format. `keys`: An array of strings where keys whose records match the specified predicates will be stored. `max_keys`: The number of elements in the `keys` array. `conn`: A connection to the server.	Query the table for records, and retrieve keys whose record matches *all* the specified predicates. It is expected that the caller has allocated enough memory at `keys` for `max_keys` strings, and the function should return no more than `max_keys` keys in the `keys` array. However, if `max_keys` is set to zero, then `keys` could be null.	Return the number of matching keys (which may be more than max_keys) if successful, and -1 otherwise.
storage_get	`table`: A table in the database. `key`: A key in the table. `record`: A pointer to a record structure. See below for the record value format. `conn`: A connection to the server.	The record with the specified key in the specified table is retrieved from the server using the specified connection. If the key is found, the record structure is populated with the details of the corresponding record. Otherwise, the record structure is not modified.	Return 0 if successful, and -1 otherwise.
storage_set	`table`: A table in the database. `key`: A key in the table. `record`: A pointer to a record structure. See below for the record value format. `connection`: A connection to the server.	The key and record are stored in the table of the database using the connection. If the key already exists in the table, the corresponding record is updated with the one specified here. If the key exists in the table and the record is NULL, the key/value pair are deleted from the table.	Return 0 if successful, and -1 otherwise.

The record structure is not changed from the previous assignment, but the `value` field in the structure now holds a set of column name-value pairs each separated by a comma. The column names must be in the same order as specified in the table schema. The column name and column value are separated by whitespace, and there is optional whitespace before and after the comma. "Whitespace can be single or multiple spaces." Here are some examples of the `value` field in the record structure that correspond to the tables specified earlier.

```
name Bloor Danforth , stops 31 , kilometres 26
lowTemperature -7 , highTemperature 28 , province Ontario
brand BMW , price 43210
id 991234567 , grade 87
```

Note again that only the columns in the record are included in the `value` field in the record structure. The key is passed as a separate argument to the `storage_get` and `storage_set` functions. You must check that the column names match those listed in the configuration file, that the columns are listed in the same order as in the config file, and that the column values conform to the specified type. Otherwise, it is an invalid parameter error. Here are some examples of invalid `value` fields in the record structure that correspond to the tables specified earlier.

```
// Columns out of order.  (For the subwayLines table.)
name Bloor Danforth , kilometres 26 , stops 31

// highTemperature should be a int.  (For the cities table.)
lowTemperature -7 , highTemperature twentyeight, province Ontario

// brand column is missing.  (For the cars table.)
price 43210

// Column name is misspelled (note the case-sensitivity).  (For the students table.)
id 991234567 , Grade 87
```

For the `storage_query` function, each predicate consists of a column name, an operator, and a value, each separated by optional whitespace. The operator may be a "=" for string types, or one of "<, >, =" for int types. *There will be at most one predicate per column.* Here are some examples of valid query predicates that correspond to the tables specified earlier.

```
// Find subway lines with more than 10 stops.  (For the subwayLines table.)
stops > 10

// Find cities in Ontario that don't get colder than -10 Celsius.  (For the cities table.)
province = Ontario , lowTemperature > -10

// Find all Toyota cars that cost less $20000.  (For the cars table.)
brand = Toyota , price < 20000

// Find students who are failing.  (For the students table.)
grade < 50
```

For a record to match a set of predicates, the record must match every predicate in the set.

Unlike the `value` field in the record structure, the query predicates do not have to include every column in the corresponding table, and the columns may be in a different order from those in the configuration file. However, it is still invalid if the column name is not in the table, if the column value is not of the correct type, or if an inappropriate operator is used. Here are some examples of invalid query predicates that correspond to the tables specified earlier.

```
// The stops column value must be an integer.  (For the subwayLines table.)
stops > 10.0 , kilometres < 40

// Cannot use ">" operator for a string column type.  (For the cities table.)
provice > Ontario , lowTemperature > -10

// Note the missing comma and "=" operator.  (For the cars table.)
brand Toyota price < 20000

// There is no Grade column name.  (For the students table.)
Grade < 50
```

Server

You need to modify the server to be able to process the configuration file according to the specification described above, as well as communicate with the client library, including the new `storage_query` function, and modified `storage_get` and `storage_set` functions outlined above. The server in this assignment is an extension of the server in the previous assignment.

Simplifying assumptions

You may make the following assumptions:

- The maximum number of columns in a table is `MAX_COLUMNS_PER_TABLE`.
- The maximum length of a column name is `MAX_COLNAME_LEN`.
- The maximum size of string type columns is `MAX_STRTYPE_SIZE`. Note that a column declared of type `char[N]` in a configuration file still means that the column values can have at most `N` characters, but you can assume that `N` will not be more that `MAX_STRTYPE_SIZE`.
- The maximum length of the `value` field in the `storage_record` structure is `MAX_VALUE_LEN`.

The constants above are defined in a `storage.h` file.

Parsing

The parsing in this assignment is more challenging. The configuration file, the data files, and the client/server protocol must all be able to process a more complex table schema. While you may write your own parsing algorithms, it is sometimes easier to use specialized tools to simplify this part of the code.

If you choose to do the parsing yourself, you may find functions such as `sscanf`, `strtok`, and `sprintf` useful.

On the other hand, you may use the Flex and Bison tools to help in this task. If you do use these tools, you can earn bonus marks as outlined here. Consult the relevant lecture slides and resources in the Course Reader for help with these tools.

Unit tests & partial tests

You must use the Check unit test framework to test the functionality of your code. You need to write at least three test suites, each in a separate directory as outlined below.

- In the `test/query` directory, you must write a program to test the `storage_query` client library function.
- In the `test/get` directory, you must write a program to test the `storage_get` client library function.
- In the `test/set` directory, you must write a program to test the `storage_set` client library function. All three behaviours of `storage_set` should be evaluated: adding a new record to a table, modifying an existing record, or deleting a record.

Each test application may use any of the `storage_*` client library functions. For example, the test application in the `test/query` directory may perform a series of `storage_set` calls whose values are then retrieved with calls to `storage_query`. The main purpose of this test, however, should be to test the `storage_query` function.

In all the tests you should consider testing corner cases, such as trying to get a non-existent key from a table, doing a query with column names not defined in a table, or storing a record with columns listed in the wrong order. You are free to define your own table schemas but it would be a good idea to construct tables with columns of all supported types (strings and integers)

In each test directory, it should be possible to run the test by typing `make run`. You can copy and modify `test/a3-partial` or develop your own from scratch.

Each test application should perform all initialization required, such as starting the server. The user should just be able to type `make run` and not have to worry about starting the server in another terminal, or create any required configuration files.

Your deliverable should contain at least 5 added tests.

Some of the marking tests are available at `/cad2/ece297s/public/assignment3/a3-partial.tgz`. You can try running them as follows.

```
> cd ~/ece297/storage/test
> tar zxf /cad2/ece297s/public/assignment3/a3-partial.tgz
> cd a3-partial
> make clean run
```

Suggested Development plan

As in the previous assignment, here are some of the major tasks involved in this assignment.

- Design considerations
- Design and plan the implementation of the specified extensions to the storage server and the client library's application programming interface (API)
- Decide on how to implement the parsing related to the more complex table schema. This includes parsing the data files, the configuration file, the client/server protocol, and the `get`/`set`/`query` function parameters.
- Implement client library support for extensions
- Implement client/server communication protocol extensions
- Implement server side support for extensions
- Implement configuration file extensions
- Implement table management extensions
- Develop meaningful test cases as specified
- Revise design document and update with new design decisions

Again, it is up to you whether you want to divide the tasks as above.

Design considerations

In this section we summarize key design decisions you will have to make when designing your storage server.

These design decisions are:

- What should be the protocol for interacting between client and storage server? Note that the `query` function can return an arbitrary number of keys. How can your protocol handle such unpredictable results?
- What kind of parsing algorithms are you going to develop? Will you write your own or use parsing tools?
- How do the new requirements in this assignment break your code from the previous assignment? Are the versions of the client or server backward or forward compatible?
- What are important unit tests to add? What code coverage do you achieve with the test cases that you developed?

You should argue about why you came to your design decisions and discuss the pros and cons of the decisions, such as the ease of implementation, robustness to failures, performance implications, and so on. Note, for many decisions you will have to balance trade-offs. Try to understand what they are.

Deliverables

Please note, there are two deadlines for this assignment. We first ask you to hand in your design documents, later followed by the code you developed.

By the design document deadline, you must hand in the **M3 Design Document: Extended Storage Server** that includes the following content:

- **M2 Design Document: Basic Storage Server** revised, edited, updated and proofread implementing any corrections and suggestions given to you by your Communication Instructor / Project Manager.
- An **Executive Summary** of no more than 400 words. This summary should show clear understanding of audience and purpose, as discussed in Chapter 1 of *Engineering Communication: From Principles to Practice.* It must be readable as a stand-alone document, clearly differentiated from an introduction, and should explain the context for your project and show the key benefits of its implementation.
- **Use Case Scenario and Diagram:** Your use case scenario will be based on the user profile you developed for the M2 Design Document. Details on what you should include in your use case scenario are discussed **here.** The Use Case Scenario must be accompanied by a UML Use Case Diagram. This section must be integrated into your revised M2 Design Document.
- **New Design Decisions**: Major design decisions are discussed above in Design Considerations. This section must be integrated into your revised M2 Design Document.
- **Bug Report**: A two-page report that identifies bugs your team has encountered during the M3 coding. This report must be submitted to Turnitin.com by the software development deadline.

The maximum length for the main body of your document is eleven (11) pages. This page count does not include Executive Summary, Table of Contents, List of References, and Appendices. For this document, you may include up to five (5) pages of appendices. Please refer to the grading rubric for further information.

By the software development deadline, you must hand in your software artifacts that includes at least the following content.

- Your code including the unit tests.
- Your code documentation (i.e., the doxygen output).

back to top