Project 4: Database

Avi Saven

April 11 2018

Database

Database Relations

Database Relations in my database are implemented as a structure `database_t` which contains a linked list of `database_table_t` structures, which each represent a relation in the database. Then within each `database_table_t` theres a `database_header_t` which represents the database schema, and a `database_hash_table_t` which represents the primary index containing the data. The `database_hash_table_t` allows the user to key the hash table with `database_tuple_t` objects, which is a fixed sized array of `database_val_t` objects, which themselves contain a union called `database_val_val_t`, which represents either a string, a unsigned or signed number, a floating point number. It can also be used to represent the in queries (the `database_tuple_t` objects are used in a variety of situations). Inserting data, requesting data, or removing data is all handled by the primary index right now. The primary index will use optimized versions of the primary algorithms if the query is usable by the primary index, or else it will fall back to a $O(n)$ approach. However I have already laid the ground work and made it trivial to add secondary indices (see comments throughout the `table.c` file).

Single Relation Functions

As state before, the `database_hash_table_t` is where the single relation functions are primarily handled, in the form of `database_hash_table_add`, `database_hash_table_get`, and, `database_hash_table_rem`. These functions are wrapped by the `database_table_t` class for ease of access (and for making the potential for adding secondary indices simpler). These hash tables use the `database_tuple_hash` function, which itself is simply a bitwise exclusive or of the outut of `database_val_hash` from its children. Then to get to the buckets of the hashtable that hash is taken modulo the size of the hash table in order to get to the linked list of buckets,

implemented as `database_hash_table_bucket_t`. Each of these objects stores a reference to the next `database_hash_table_bucket_t`, the previous `database_hash_table_bucket_t`, and, the `database_tuple_t` that it uses. A simple method for getting the bucket that a certain tuple belongs to (works for full tuples or queries), `database_hash_table_get_buckets`, is provided. It however may return `NULL` in the event that a query is provided who has a on one of the indexed keys. In this case the add, remove, and getter functions will resort to looping through each bucket and in order to figure out which bucket is needed. In the case of adding, this will never happen since they are not queries, but for remove and get this is a possiblility. The adder will make sure that the current added tuple isn't replicated in the bucket already, and if it isn't will prepend it to the bucket. The getter will create a `database_tuple_vector_t` (representing a variable sized array of `database_tuple_T` objects) and use the indexed bucket if possible (if not it will loop through each bucket in the table), and select which tuples in the bucket(s) are needed and add them to the vector. The deleter will use a similar algorithm to the getter but instead of adding to a vector, it'll remove them from their linked list and release the memory.

Saving and Loading

Saving (and consequentially loading) is handled in the `database_t` class. The function will iterate through each table and write the given format. First the table's name is written, then a the headers are written. The headers are written name, then the type is written. The type will be marked with a `*` if the column is a primary index. Once those structures are written then the database will be dumped into a `database_tuple_vector_t` instance, which is iterated through and each row is written from that vector. Each write operation is terminated with a unique non-ASCII byte (unique to the type of object written that is, e.g. the head is terminated a byte thats different than what the headers are terminated with, for ease of detection). When loading, the file is read from disk and is parsed through a parser that is similar to a recursive descent parser with some iterative touches in the process. For example, the parser will call a function to parse rows, which will grab each row from the string, and pass that to an individual row parsing function. However the individual row parsing function will actually iterate through each column manually and create the tuple. The values within the rows will be seperated by a byte, however when the `database_val_t` objects are created, they are not reallocated and copied, but rather the byte will be replaced with a null byte and the pointer into the file data will be used instead, to conserve memory.

Relational Algebra

All relational algebra operations are implemented through a `database_query_t` object, which itself has an enum which represents whether it is a selection, projection or a join operation. It also contains a union type which itself contains 3 pointers, one to the arguments of each respective argumnet. Then a generic `database_query_execute` is provided to execute the given query.

Selection

Selection is implemented in the `database_query_execute_select` function. The arguments are passed as a vector of binary tuples, where the tuples represent pairs of a mapping (key/value pairs), as would the normal select operator work. This is then converted into a tuple query that can be passed directly to the `database_table_get` function. This is accomplished by going through the keys, finding the correct location in a tuple based on the header, and inserting the values into the tuple. The values that aren't mapped in the selection arguments are filled with a $*$. Then the standard `database_table_get` is called.

Projection

The projection takes its arguments as a tuple of strings, each representing which column is desired. First the offsets into the header are determined from the arguments and are loaded as integers into an array, as well as the primary keys being determined for the resulting table. Then all the rows are grabbed from the table. Then they are loaded into the new table, and as they are loaded into the offset table is used to load the old vlaues into the tuples.

Join

The merging algorithm can be overall described as: the first input is added on, then as the second table is merged into the first one (in the resulting table) the input equivelence relations are referred to, to make sure that no duplicate columns are generated. Specifically, the arguments are passed as a list of binary tuples, each representing an equivelance relation for the joining. First the header is generated using the above merging algorithm, in order to create the table object. During this time the primary keys (a merge of the two input tables) will also be generated. Using this information a table object can be constructed. Then each row from the first input is iterated through, for each row it will iterate through the rows of the second input. Each of these pairs are considered as a possibility and it creates a tuple to put into the output table and loads that with the first row's input, and then as the second row's input is loaded in it checks against the input equivalence relations, and if they are valid based on the input

relations, the merged row will be added to the outputting table, and if not it will be deallocated and discarded. The final table is then output to the caller.