StormSQL internals

SQL server in C++

Table representation

Internally, tables are represented as a dynamic byte buffer. Therefore each row must have a fixed length. This allows direct random access to any row in the table without the need to decode all previous ones.

The dynamic buffer itself (DynamicBuffer template class) works sort of like a vector with 2 differences.

First, when asked for more available space, it does **not** create an entirely new buffer and copy all the data to the new one. DynamicBuffer allocates a new buffer, which only holds a chunk of the table data – the old buffer is still used and **no data is copied when adding rows.** This is maintained with a dynamic array of pointers to the different chunks of data (and their size). One chunk contains more than one row.

The second difference is that, when removing rows DynamicBuffer makes it quickly by just marking the row (in its chunk) to be removed. It doesn't remove anything but acts like it has, because otherwise copying data is necessary. When data from DynamicBuffer is written/read to/from disk it removes these "deleted" rows, and merges the chunks to one big chunk.

Table schema

The table schema is an array of Field structs. Each field type could be one of: byte, int32, uint32 and fixedchar. The last one is used for strings and represents a C string (NULL terminated array of chars). Even though the size of the field must be fixed across all rows, the length of the string in it can be smaller.

When serializing the tables, the table schema is written just before the actual table data. When reading from tables, the table schema is consulted to calculate the row size (in bytes), the offset for a field (also in bytes) and how to interpret the pointer to the field data.

Iterating table rows

Almost every query that operates on a table needs to iterate the rows. Also, most of the time, there is an enforced WHERE condition which should skip some of them.

The TableDataIterator class is responsible for iterating the rows (in straight, reversed order and also supports seek to row index). It can be given a TableDataPredicate object which is used to check the rows and skip the ones not needed. By default it uses a TruePredicate object which is a dummy predicate that just returns true for all rows.

Lexer

The lexer class is given an istream and is responsible for breaking the SQL query into tokens. It does it token by token when it's needed. A token has a type – Keyword (select, where, insert, etc.), Identifier (table names, column names, etc.), Separator (only ","), Parenthesis ("(" and ")"), Operator (==, !=, !, +, -, /, *, <, >, <=, >=), StringValue and IntValue. Lexer also provides methods to put back tokens and validating the next token is of certain type.

Parsing SQL queries

Each query class (Select, Update, Insert, Delete, ShowTables, ShowCreate, DropTable and CreateTable) inherits the abstract class Query. It has a method that should accept a parameter of type Lexer and uses it to parse itself.

Expressions and functions

StormSQL has full-featured expression parser which is used to parse boolean, integer and string functions, variables and constants. It is also able to detect where the expression ends without throwing errors or using delimiters.

In order to parse expressions the hierarchy is made of 4 classes – one base abstract one (Expression) and 3 that inherit it: ConstExpression (constants), VarExpression (variables/table columns) and CompExpression (takes multiple expressions and a function).

The functions can be infix or prefix ones, as well as C-like syntax (" strlen(string) ", " strcat(string, string) ") etc. All functions inherit the abstract class Operation and have their own unique signature.

In order to use expressions in the SELECT clause (SELECT …, strlen(firstName) AS nameLen …) every expression and function has a way of detecting their result type (so a suitable column is created for the resulting table). This is done based on the types of the nested expressions and the type of the columns used. Take for example SELECT …, strcat(firstName, strcat(" ", lastName)) AS fullName …. Now suppose the columns firstName and lastName have a length of maximum 20 chars. The resulting fullName field will have a type of fixedchar(41) because the function strcat detects its suitable containing field as fixedchar(sum of arguments' lengths). It does that before actually knowing the data behind firstName and lastName, because the resulting table should have fixed schema before actually adding any rows to it.

The currently implemented functions and operators are: or, and, !, <, <=, >, >=, =, !=, +, -, *, /, toint, tostr, tobool, strcat, strlen, substr

WHERE clause

The where clause is essentially parsed as an expression. This expression is given to a special predicate class (ExpressionPredicate), which, in turn, is used in the TableDataIterator, which the query uses. This way there is no special logic for the WHERE in every query class that might use it.