DuckDB version: v0.9.1

git hash: 401c8061c6ece35949cac58c7770cc755710ca86

Disclaimer

All writing is my own. I've collaborated & shared ideas with Johan Laursen on the written parts, and discussed high level coding ideas, but all coding is my own with the aid of Copilot & stackoverflow where cited.

1. Out of Tree Extension (20%)

Create an out of tree extension, named oml, using the DuckDB extension template: https://github.com/duckdb/extension-template

Question 1. Describe in one paragraph the build process when you compile the extension.

DuckDB uses "make" as a build entrypoint. Upon calling "make", the "release" action is called by default in "Makefile:58". This calls "CMake" twice: Once for the sub-repository "duckdb" which compiles the core of duckdb, and secondarily on the current directory (our extension). Both look for the "CMakeLists.txt" file, which describes high-level build goals, including other "CMakeLists.txt"s in other subdirectories [duckdb/CMakeLists.txt:1067~1079]. These high-level goals are then used to build lower level build goals in e.g. make or ninja. These tools also avoid recompiling code which has not changed, which means that any change to our extension only needs to recompile the modified files, and not the entire duckdb extension. The resulting binaries are put into ./build/release

2. Database Load (60%)

Question 2 (20%). Describe how the single threaded version of the “read_csv” table function is defined in DuckDB (see src/include/duckdb/function/table/read_csv.hpp and src/function/table/read_csv.cpp).

There's three core components to the read_csv table function. The bind-, init- and the table function. The bind function establishes metadata about the file to read and the inputs provided by the user. This reader supports either receiving the dtypes and column names as input parameters, or by inferring them using a "Sniffer" [duckdb/src/function/table/read_csv.cpp:110~111], which assigns a "most probably type" to each of the columns found in the csv. Finally, the csv may also detect a Hive partitioning folder structure [duckdb/src/function/table/read_csv.cpp:98] which can skip reading entire files if a selection predicate excludes them, or it may detect multiple files to be read [duckdb/src/function/table/read_csv.cpp:93].

The init function constructs a file reader from the user input, which the next function reads from.

Finally, the core table function parses the csv at a character-level in [duckdb/src/function/table/read_csv.cpp:798 & duckdb/src/execution/operator/csv_scanner/buffered_csv_reader.cpp:194~432], which keeps reading until a full vector is constructed, after which it is flushed [duckdb/src/execution/operator/csv_scanner/base_csv_reader.cpp:270~273]. This table function is called multiple times, producing multiple DataChunks, which are passed on to the next step in the pipeline (e.g. print to terminal).

Question 3 (40%) & 3. Database Generation (20%)

The following two questions will be answered together, as one is an extension of the other.

Design, implement and test a table function Power_Consumption_load(filename) that reads from the oml file ‘filename’ generated by an IoTLab measurement point and loads the tuples it contains into a table that corresponds to the Power_Consumption table above. (hint: iot_load is a simplified version of the single threaded csv reader).

Question 5. Design, implement and test a table function OmlGen(filename) that reads an oml file, create a schema based on the metadata the oml file contains and loads the tuples that the oml file contains.

The design aims to borrow as much as possible from the existing csv implementation. This is primarily to avoid implementing the character level csv parsing, but it also has the benfits of converting the parsed strings to the correct datatypes. This is achieved by realizing that the parse_csv(...) takes all the necessary parameters in order to parse an OML file: sep, parallel, skip and columns. Esentially, we will be wrapping the csv implementation, modifying its input parameters and parsing its output.

The first two parameters are fixed, since 1) I'm assuming the seperator for values within a line is always a tab for OML files and 2) we should not run a parallel implementation. Omitting 2) can also cause segmentation faults, which I attribute to simultaneous insertion into the same table when using the InternalAppender [duckdb/src/include/duckdb/main/appender.hpp:129] without locking mechanisms.

The final two parameters are dynamic, which must be parsed from the file header. The CSV implementation uses the Bind function to parse metadata from a file (e.g. the CSVSniffer), so I open the file via the filename, which is provided as a bind parameter, and parse the fixed headers along with the dynamic length schema. The "skip" and "columns" parameters is then assigned from the number of fixed and dynamic lines, and the schema definition.

Upon defining these parameters, we can pass them along to the ReadCSVBind directly, which will indirectly pass it to ReadCSVFunction through the DuckDB architecture. As mentioned in the previous section, the ReadCSVBind does a lot of unnecessary work for our limited usecase, but it does helpfully convert our "fake user input" parameters into the format which the

ReadCSVFunction expects. We define an `global_init` function on the table function, which passes the bind output to CreateTableFromCSVMetadata, which creates a table (assuming the "memory" catalog and "main" table). One thing to note, is that this should be in the init function and not in the bind function, since (for some reason, maybe updated catalog?) when placed in the bind function, the bind function is called multiple times, which can cause name clashes since the table is created twice. For the same reason, the table creating shouldn't be in the ReadCSVFunction, as this is also called multiple times.

This has some pros and cons. One is the fact that we're allowing the user to provide the same parameters as read_csv expects, which is entirely bad for our concrete implementation. First, it may give the user the impression that they can override the header by providing "columns", but this will just be overridden in the bind function. Second, since we're required to implicitly insert the values into a hard-coded table (in contrast to an explicit SQL statement of INSERT INTO foobar VALUES FROM ReadOML(...)), we do not gain the benefits of the modular SQL syntax. Third, the potential upsides and capabilities of the existing read_csv implementation is unused, such as potentially supporting multi-file parsing (with identical schemas) using glob filename matching, parallel parsing. With some extra work in the bind function and proper locking, these capabilities could be introduced with relatively little work, and it may even be able to handle unions of table schemas as the current CSV implementation does with the parameter "union_by_name". If we loosened our function requirements to just creating a temporary table instead of a static one, these values could then be inserted into a (new) table or be aggregated. The reason the hardcoded insertion of values is an issue is described in the following section

The DuckDB architecture calls ReadCSVFunction multiple times (thereby possibly introducing concurrency issues as mentioned), which each produce DataChunks. My implementation parses these DataChunks, which means that if read_csv does NOT produce DataChunks in the form of a temporary table, then my function parsing the output will not be parsing anything.

To be more concrete, consider the following call which uses the ReadCSVFunction: `ReadCSVTableFunction::GetFunction().function(context, data_p, output);`. If I simply `call read_csv(...)`, this `output` will contain a temporary table of values, whereas if i instead perform an aggregation such as `select count(*) from read_csv(...)`, then the datachunk will contain a single value, that being the count. The values will therefore be unavailable for insertion into the table. It is unclear how the function ends up aggregating the input, as there's no transparent or obvious place where this would occur in the ReadCSVFunction. I would expect the aggregation logic to be outside of the csv function, since all input formats should support aggregations, and it seems unmaintainable to implement all operators in all input readers.

In conclusion, the user may believe they can use the temporary output of `ReadOml(...)`, but if they do, they will encounter an error as my implementation tries to iterate over the output. An idiomatic and explicit SQL table insertion would avoid these issues.

The functions are implemented in src/oml_parser_extension.cpp and tested in test/sql/oml_parser.test. To run the tests, type `make test`. For example/interactive usage, run `source handin/example.sh`.

```
call Power_Consumption_load('handin/st_lrwan1_11.oml');
call OmlGen('handin/st_lrwan1_11.oml');
call OmlGen('handin/st_lrwan1_15.oml');
v0.9.1 401c8061c6
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
D
```

Execute one of the three commands mentioned above, which will create a table POWER_CONSUMPTION

```
D call OmlGen('handin/st_lrwan1_15.oml');
┌──────────────┬─────────┬─────────┬─────────────┬──────────────┬─────────┬
────────┐
│   subject    │   key   │  value  │ timestamp_s │ timestamp_us │ channel │
rssi  │
│   varchar    │ varchar │ varchar │   uint32    │    uint32    │ uint32  │
int32 │
├──────────────┼─────────┼─────────┼─────────────┼──────────────┼─────────┼
────────┤
│ 6.921095     │ 2       │ 1       │  1689001669 │       909685 │      11 │
-91 │
│ 6.921713     │ 2       │ 2       │  1689001670 │       909600 │      11 │
-91 │
...
```

```
D SELECT DISTINCT channel FROM POWER_CONSUMPTION;
┌─────────┐
│ channel │
│ uint32  │
├─────────┤
│      11 │
└─────────┘
```