Chapter 10

Implementation of the query system[*]

Compatibly with other query languages, the EQL defines the user a front-end interface and infers the query's results from its semantics. However, a query language does not define any data structures or specify how the query engine is to be implemented. As mentioned in Chapter 1, a major user requirement was database portability, simple package installation, and a system that did not rely on external software at runtime. The only available back-end implementation that met those needs and was also available as an R package at the time was (R)SQLite (Hipp and Kennedy, 2007; Wickham et al., 2014). Because of this, `emuR`'s query system could not be implemented so as to use directly the primary data sources of an `emuDB`, that is, the JSON files described in Chapter 4. A syncing mechanism that maps the primary data sources to a relational form for querying purposes had to be implemented. This relational form is referred to as the `emuDBcache` in the context of an `emuDB`. The data sources are synchronized while an `emuDB` is being loaded and when changes are made to the annotation files. To address load time issues, we implemented a file check-sum mechanism which only reloads and synchronizes annotation files that have a changed MD5-sum (Rivest, 1992). Figure 10.1 is a schematic representation of how the various `emuDB` interaction functions interact with either the file representation or the relational cache.

Despite the disadvantages of cache invalidation problems, there are several advantages to having an object relational mapping between the JSON-based annotation structure of an `emuDB` and a relation table representation. One is that the user still has full access to the files within the directory structure of the `emuDB`. This means

[*]Sections of this chapter have been published in Winkelmann et al. (2017).

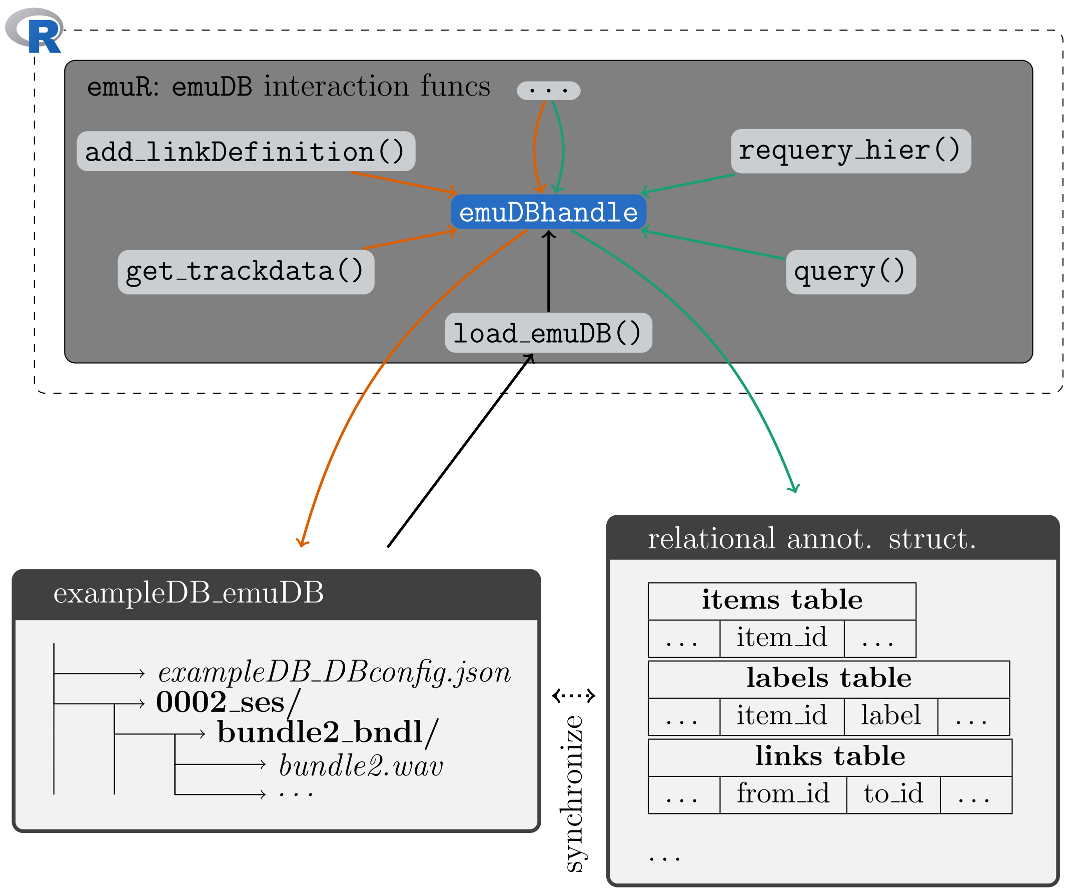

Figure 10.1: Schematic architecture of `emuDB` interaction functions of the `emuR` package. Orange paths show functions interacting with the files of the `emuDB`, while green paths show functions accessing the relational annotation structure. Actions like saving a changed annotation using the `EMU-webApp` first save the `_annot.json` to disk then update the relational annotation structure.

that external tools can be used to script, manipulate or simply interact with these files. This would not be the case if the files were stored in databases in a way that requires (semi-)advanced programming knowledge that might be beyond the capabilities of many users. Moreover, we can provide expert users with the option of using other relational database engines such as PostgreSQL, including all their performance-tweaking abilities, as their relational cache. This is especially valuable for handling very large speech databases.

The relational form of the annotation structure is split into six tables in the relational database to avoid data redundancy. The six tables are:

1. `emu_db`: containing `emuDB` information (columns: `uuid`, `name`),
2. `session`: containing `session` information (columns: `db_uuid`, `name`),
3. `bundle`: containing `bundle` information (columns: `db_uuid`, `session`, `name`, `annotates`, `sample_rate`, `md5_annot_json`),
4. `items`: containing all annotation items of `emuDB` (columns: `db_uuid`, `session`, `bundle`, `item_id`, `level`, `type`, `seq_idx`, `sample_rate`, `sample_point`, `sample_start`, `sample_dur`),
5. `labels`: containing all labels belonging to all items (columns: `db_uuid`, `session`, `bundle`, `item_id`, `label_idx`, `name`, `labels`), and
6. `links`: containing all links between annotation items of `emuDB` (columns: `db_uuid`, `session`, `bundle`, `from_id`, `to_id`, `labels`).

While performing a query the engine uses an aggregate key to address every annotation item and its labels (`db_uuid`, `session`, `bundle`, `item_id`) and a similar aggregate key to dereference the links (`db_uuid`, `session`, `bundle`, `from_id` / `to_id`) which connect items. As the records in relational tables are not intrinsically ordered a further aggregate key is used to address the annotation item via its index and level (`uuid`, `session`, `bundle`, `level` / `seq_idx`). This is used, for example, during sequential queries to provide an ordering of the individual annotation items. It is worth noting that a plethora of other tables are created at query time to store various temporary results of a query. However, these tables are created as temporary tables during the query and are deleted on completion which means they are not permanently stored in the `emuDBcache`.

10.0.1 Query expression parser

The query engine parses an EQL query expression while simultaneously executing partial query expressions. This ad-hoc string evaluation parsing strategy is different from multiple other query systems which incorporate a query planner stage to pre-parse and optimize the query execution stage (e.g., Hipp and Kennedy, 2007; Conway et al., 2016). Although no pre-optimization can be performed, this strategy simplifies the execution of a query as it follows a constant heuristic evaluation strategy. This section describes this heuristic evaluation and parsing strategy based on the EQL expression `[[Syllable == W -> Syllable == W] ^[Phoneme == @ -> #Phoneme == s]]`.

The main strategy of the query expression parser is to recursively parse and split an EQL expression into left and right sub-expressions until a so-called Simple Query (SQ) term is found and can be executed (see EBNF in Appendix D for more information on the elements comprising the EQL). This is done by determining the operator which is the first to be evaluated on the current expression. This operator is determined by the sub-expression grouping provided by the bracketing. Each sub-expression is then considered to be a fully valid EQL expression and once again parsed. Figure 10.2, which is split into seven stages (marked S1-S7), shows the example EQL expression being parsed (S1-S3) and the resulting items being merged to meet the requirements of the individual operator (S4-S6) of the original query. S1 to S3 show the splitting operator character (e.g., `->` in purple) which splits the expression into a left (green) and right (orange) sub-expression.

The result modifier symbol (`#`) is noteworthy for its extra treatment by the query engine as it places an exact copy of the items marked by it into its own intermediary result storage (see $\#s_{\text{items}}$ node on S7 in Figure 10.2). After performing the database operations necessary to do the various merging operation which are performed on the intermediary results, this storage is updated by removing items from it that are no longer present due to the merging operation. As a final step, the query engine evaluates if there are items present in the intermediary result storage created by the presence of the result modifier symbol. If so, these items are used to create an `emuRsegs` object by deriving the time information and extracting the necessary information from the intermediate result storage. If no items are present in the result modifier storage, the query engine uses the items provided by the final merging procedure in S3 instead (which is not the case in the example used in Figure 10.2).

A detailed description of how this query expression parser functions is presented in a pseudo code representation in Algorithms 1 and 2[1]. For simplicity, this repre-

[1]The R code that implements this pseudo code can be found here: `https://github.com/`

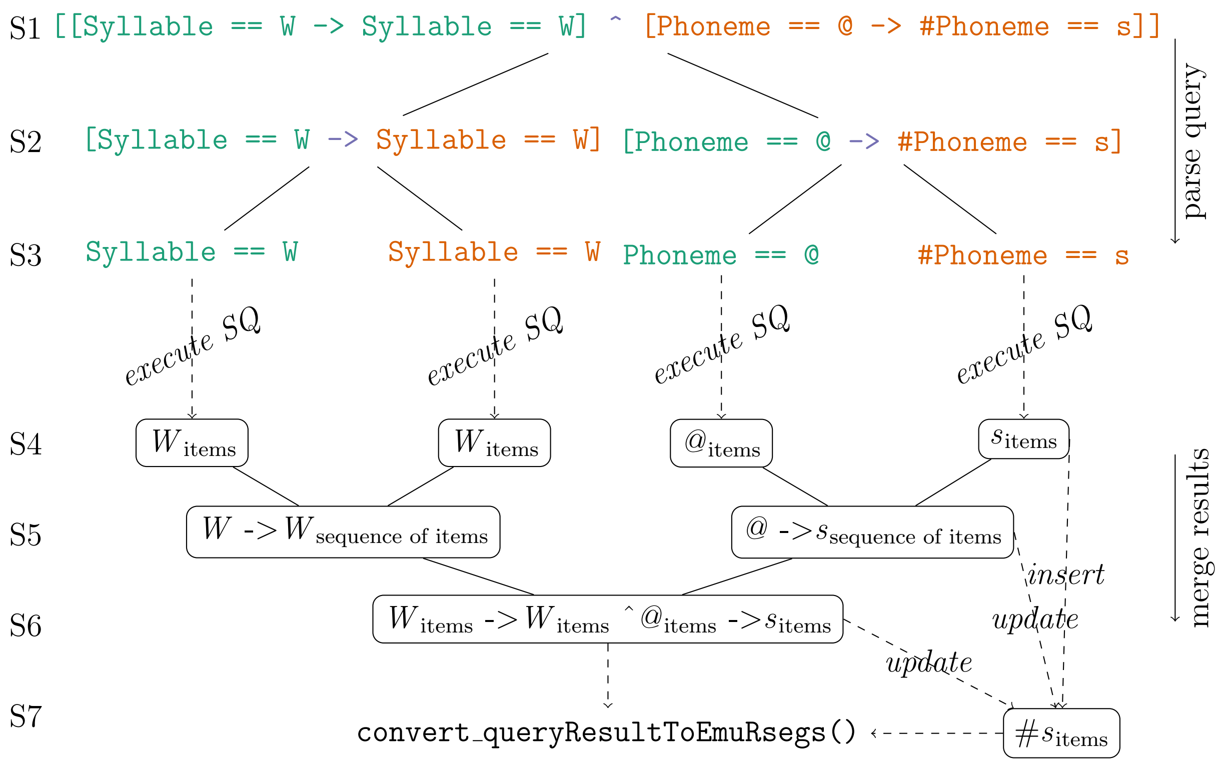
S1 [[Syllable == W -> Syllable == W] ^ [Phoneme == @ -> #Phoneme == s]]
S2 [Syllable == W -> Syllable == W] [Phoneme == @ -> #Phoneme == s]
S3 Syllable == W Syllable == W Phoneme == @ #Phoneme == s
execute SQ
S4 W_{items} W_{items} $@_{\text{items}}$ s_{items}
S5 W -> $W_{\text{sequence of items}}$ @ -> $s_{\text{sequence of items}}$
S6 W_{items} -> W_{items} ^ $@_{\text{items}}$ -> s_{items}
S7 convert_queryResultToEmuRsegs() $\#s_{\text{items}}$
insert
update
update
parse query
merge results

Figure 10.2: Example of how the query expression parser parses and evaluates an EQL expression and merges the result according to the respective EQL operators.

sentation ignores the treatment of the result modifier symbol (#) and focuses on the parsing and evaluation strategy of the query expression parser. As stated previously, the presence of the result modifier before an SQ triggers the query engine to place a copy of the result of that SQ into an additional result table, which is then updated throughout the rest of the query. The starting point for every query is the `query()` function (see line 64 in Algorithm 2). This function places the filtered items, links and labels entries that are relevant for the current query into temporary tables. Depending on which query terms and operators are found, the EQL query engine uses the various sub-routines displayed in Algorithms 1 and 2 to parse and evaluate the EQL expression.

10.0.2 Redundant links

A noteworthy difference between the legacy and the new EMU system is how hierarchies are stored. The legacy system stored the linking information of a hierarchy in so-called hierarchical label files, which were plain text files that used the `.hlb` extensions. Within the label files this information was stored in space/blank separated lines:

```
111 139 140 141 173 174 175 185
112 142 143 176 177
113 144 145 146 178 179 180
114 147
115 148
116 149,
```

where the first number (green) of each line was the parent's ID and the following numbers (orange) indicated the annotation items the parent was linked to. However, it was not just links to the items on the child level that were stored in each line. Rather, a link to all children of all levels below the parent level was stored for each parent item. This was likely due to performance benefits in parsing and mapping onto the internal structures used by the legacy query engine. A schematic representation of this cluttered form of linking is displayed in Figure 10.3A. As these redundant links are prone to errors while updating the data model and lead to a convoluted annotation structure models (see excessive use of dashed lines in Figure 10.3A), we chose to eliminate them and opted for the cleaner, non-redundant representation displayed in Figure 10.3B. Although this led to a more complex query parser engine

`IPS-LMU/emuR/blob/master/R/emuR-query.database.R`.

Algorithm 1 Pseudo Code for Query Engine Algorithm - Part 1

```
function QUERY_DBEQLFUNCQ(query)
    place all parent level items into tmp table
    place all child level items into tmp table
    QUERY_DBHIER(parentItemsTable, childItemsTable)
    if Start End or Medial query then
        extract parent items and place in tmp result table
    else
        extract child items and place in tmp result table
    end if
end function
function QUERY_DBEQLLABELQ(query)
    splitLabels ← split labels at |
    for all splitLabels do
        if operator is ==, = or ! = then
            extract items that contain labels which are equal or unequal to label
        else if operator is =  or ! ∼ then
            extract items that contain labels that match or don't match RegEx
        end if
        merge results in tmp table
    end for
end function
function QUERY_DBEQLSQ(query)
    if query contains round brackets then
        QUERY_DBEQLFUNCQ(query)
    else
        QUERY_DBEQLLABELQ(query)
    end if
end function
function QUERY_DBEQLCONJQ(query)
    splitItems ← split query at &
    for all splitItems do
        QUERY_DBEQLSQ(splitItems)
        merge results in tmp table
    end for
end function
function QUERY_DBHIER(leftTable, rightTable)
    hp ← extract hier. paths conn. leftTable and rightTable level names
    for all child and parent level pairs in hp do
        connect child and parent items using links table
        reduce to min seq. idx (left side of trapeze)
        and to max seq. idx (right side of trapeze)
    end for
end function
```

Algorithm 2 Pseudo Code for Query Engine Algorithm - Part 2

```
44: function QUERY_DBEQLINBRACKET(query)
45:     qTrim ← remove outer square brackets
46:     leftQuery, rightQuery ← split qTrim at cur. operator
47:     QUERY_DATABASEWITHEQL(leftQuery)              ▷ recursive part of query
48:     QUERY_DATABASEWITHEQL(rightQuery)             ▷ recursive part of query
49:     if cur. operator is domintation operator then
50:         QUERY_DBHIER(leftQueryResultTable, rightQueryResultTable)
51:     else if cur. operator is seq. operator then
52:         find seq. of leftQueryResultTable and rightQueryResultTable items
53:     else
54:         QUERY_DATABASEWITHEQL(qTrim)
55:     end if
56: end function
57: function QUERY_DBWITHEQL(query)
58:     if query isn't wrapped in brackets then
59:         QUERY_DBEQLCONJQ(query)
60:     else
61:         QUERY_DBEQLINBRACKET(query)
62:     end if
63: end function
64: function QUERY(query, sesPattern, bndlPattern)
65:     filter items in relational tables by sesPattern
66:     filter items in relational tables by bndlPattern
67:     QUERY_DBWITHEQL(query)
68:     seglist ← CONVERT_QUERYRESULTTOEMURSEGS(tmpResultTableName)
69:     return seglist
70: end function
```

for hierarchical queries and functions, we feel it is a cleaner, more accurate and more robust data representation.

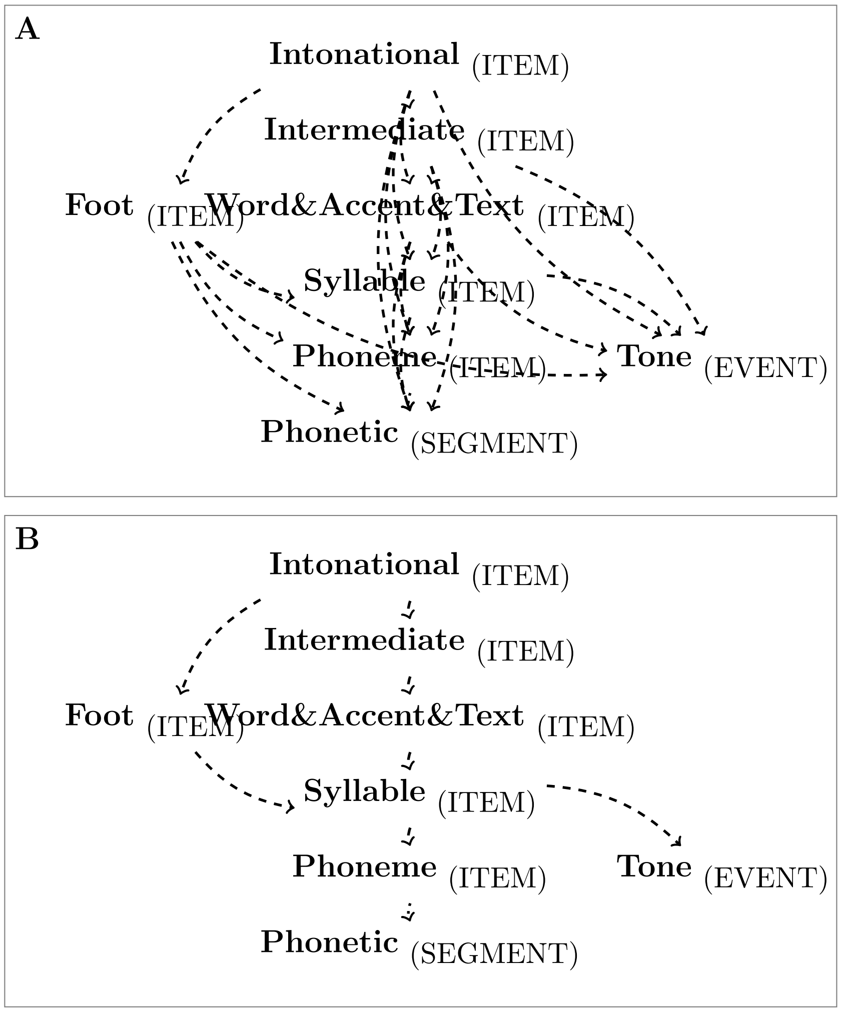

Figure 10.3: Schematic of hierarchy graph *ae*; **A**: legacy cluttered redundant strategy vs. **A**: cleaner non-redundant strategy.