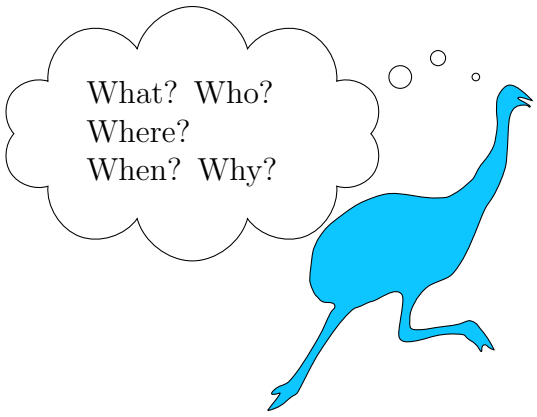
Chapter 5

The query system



This chapter describes the newly implemented query system of the `emuR` package. When developing the new `emuR` package it was essential that it had a query mechanism allowing users to query a database's annotations in a simple manner. The EMU Query Language (EQL) of the EMU-SDMS arose out of years of developing and improving upon the query language of the legacy system (e.g., Cassidy and Harrington, 2001; Harrington, 2010; John, 2012). As a result, today we have an expressive, powerful, yet simple to learn and domain-specific query language. The EQL defines a user interface by allowing the user to formulate a formal language expression in the form of a query string. The evaluation of a query string results in a set of annotation items or, alternatively, a sequence of items of a single annotation level in the `emuDB` from which time information, if applicable (see Section 5.2.6), has been deduced from the time-bearing sub-level. An example of this is a simple query that extracts all strong syllables (i.e., syllable annotation items containing the label

S on the *Syllable* level) from a set of hierarchical annotations (see Figure 5.1 for an example of a hierarchical annotation). The respective EQL query string `"Syllable == S"` results in a set of segments containing the annotation label *S*. Due to the temporal inclusion constraint of the domination relationship, the start and end times of the queried segments are derived from the respective items of the *Phonetic* level (i.e., the *m* and *H* nodes in Figure 5.1), as this is the time-bearing sub-level. The EQL described here allows users to query the complex hierarchical annotation structures in their entirety as they are described in Chapter 3.

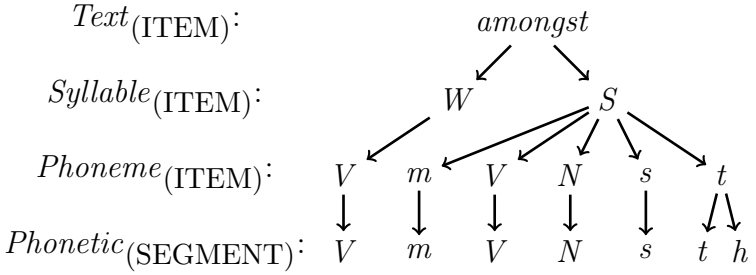


Figure 5.1: Simple partial hierarchy of an annotation of the word *amongst* in the *msajc003* bundle in the *ae* demo `emuDB`.

R Example 5.0.1 shows how to create the demo data that is provided by the `emuR` package followed by loading an example `emuDB` called *ae* into the current R session. This database will be used in all the examples throughout this chapter.

R Example 5.0.1

```
# load package
library(emuR)

# create demo data in directory
# provided by tempdir()
create_emuRdemoData(dir = tempdir())

# create path to demo database
path2ae = file.path(tempdir(), "emuR_demoData", "ae_emuDB")

# load database
ae = load_emuDB(path2ae, verbose = F)
```

5.1 emuRsegs: The resulting object of a query

In emuR the result of a query or requery (see Section 5.2.7) is a pre-specified object which is a superclass of the common data.frame. R Example 5.1.1 shows the result of a slightly expanded version of the above query ("Syllable == S"), which additionally uses the dominates operator (i.e., the ^ operator; for further information see Section 5.2.2) to reduce the queried annotations to the partial hierarchy depicted in Figure 5.1 in the *ae* demo emuDB. In this example, the classes of the resulting object including its printed output are displayed. The class vector of a resulting emuRsegs object also contains the legacy EMU system's emusegs class, which indicates that this object is fully backwards compatible with the legacy class and the methods available for it (see Harrington, 2010, for details). The printed output provides information about which database was queried and what the query was as well as information about the labels, start and end times (in milliseconds), session, bundle, level and type information. The call to colnames() shows that the resulting object has additional columns, which are ignored by the print() function. This somewhat hidden information is used to store information about what the exact items or sequence of items were retrieved from the emuDB. This information is needed to know which items to start from in a requery (see Section 5.2.7) and is also the reason why an emuRsegs object should be viewed as a reference of sequences of annotation items that belong to a single level in all annotation files of an emuDB.

R Example 5.1.1

```
# query database
sl = query(ae, "[Syllable == S ^ Text == amongst]")

# show class vector
class(sl)
```

```
## [1] "emuRsegs"   "emusegs"    "data.frame"
```

```
# show sl object
sl
```

```
## segment  list from database:  ae
## query was:  [Syllable == S ^ Text == amongst]...
##   labels   start     end session   bundle
## 1      S 256.925 674.175    0000 msajc003
##      level type
## 1 Syllable ITEM
```

```
# show all (incl. hidden) column names
colnames(sl)

##  [1] "labels"             "start"
##  [3] "end"                "utts"
##  [5] "db_uuid"            "session"
##  [7] "bundle"             "start_item_id"
##  [9] "end_item_id"        "level"
## [11] "start_item_seq_idx" "end_item_seq_idx"
## [13] "type"               "sample_start"
## [15] "sample_end"         "sample_rate"
```

5.2 EQL: The EMU Query Language version 2

The EQL user interface was retained from the legacy system because it was sufficiently flexible and expressive enough to meet the query needs in most types of speech science research. The EQL parser implemented in `emuR` is based on the Extended Backus-Naur Form (EBNF) (Garshol, 2003) formal language definition of John (2012), which defines the symbols and the relationship of those symbols to each other on which this language is built (see adapted version of entire EBNF in Appendix D). Here we will describe the various terms and components that comprise the slightly adapted version 2 of the EQL. It is worth noting that the new query mechanism uses a relational back-end to handle the various query operations (see Chapter 10 for details). This means that expert users, who are proficient in Structured Query Language (SQL) may also query this relational back-end directly. However, we feel the EQL provides a simple abstraction layer which is sufficient for most speech and spoken language research.

5.2.1 Simple queries

The most basic form of an EQL query is a simple equality, inequality, matching or non-matching query, two of which are displayed in R Example 5.2.1. The syntax of a simple query term is `[L OPERATOR A]`, where `L` specifies a level (or alternatively the name of a parallel attribute definition); `OPERATOR` is one of `==` (equality), `!=` (inequality), `=~` (matching) or `!~` (non-matching); and `A` is an expression specifying

the labels of the annotation items of L [1]. The second query in R Example 5.2.1 queries an event level. The result of querying an event level contains the same information as that of a segment level query except that the derived end times have the value zero.

R Example 5.2.1

```
# query all annotation items containing
# the label "m" on the "Phonetic" level
sl = query(ae, "Phonetic == m")

# query all items NOT containing the
# label "H*" on the "Tone" level
sl = query(ae, "Tone != H*")

# show first entry of sl
head(sl, n = 1)

## event  list from database:  ae
## query was:  Tone != H*...
##   labels start end session   bundle level  type
## 1     L-  1107   0    0000 msajc003  Tone EVENT
```

R Example 5.2.1 queries two levels that contain time information: a segment level and an event level. As described in Chapter 3, annotations in the EMU-SDMS may also contain levels that do not contain time information. R Example 5.2.2 shows a query that queries annotation items on a level that does not contain time information (the *Syllable* level) to show that the result contains deduced time information from the time-bearing sub-level.

R Example 5.2.2

```
# query all annotation items containing
# the label S on the Syllable level
sl = query(ae, "Syllable == S")
```

[1] The examples and syntax descriptions used in this chapter have been adapted from examples by Cassidy and Harrington (2001) and Harrington and Cassidy (2002) and were largely extracted from the EQL vignette of the emuR package. All of the examples were adapted to work with the supplied *ae* emuDB.

```
# show first entry of sl
head(sl, n = 1)

## segment  list from database:  ae
## query was:  Syllable == S...
##   labels   start     end session   bundle
## 1      S 256.925 674.175    0000 msajc003
##      level type
## 1 Syllable ITEM
```

Queries using regular expressions

The slightly expanded version 2 of the EQL, which comes with the `emuR` package, introduces regular expression operators (`=~` and `!~`). These allow users to formulate regular expressions for more expressive and precise pattern matching of annotations. A minimal set of examples displaying the new regular expression operators is shown in Table 5.1.

Query	Function
`"Phonetic =~ '[AIOUEV]'"`	A disjunction of annotations using a RegEx character class
`"Word =~ a.*"`	All words beginning with *a*
`"Word !~ .*st"`	All words not ending in *st*
`"[Phonetic == n ^ #Syllable =~ .*]"`	All syllables that dominate an *n* segment of the Phonetic level

Table 5.1: EQL V2: examples of simple and complex query strings using RegEx operators including their function descriptions.

5.2.2 Combining simple queries

The EQL contains three operators that can be used to combine the simple query terms described above as well as position queries which we will describe below. These three operators are the sequence operator, `->`; the conjunction operator, `&`; and the domination operator, `^`, which is used to perform hierarchical queries. These three types of queries are described below. To start with, we describe the two types of queries that query more complex annotation structures on the same level (sequence

and conjunction queries). This is followed by a description of domination queries that query hierarchically linked annotation structures, sometimes spanning multiple annotation levels.

Sequence queries

The syntax of a query string using the -> sequence operator is `[L == A -> L == B]` where annotation item `A` on level `L` precedes item `B` on level `L`. For a sequence query to work, both arguments must be on the same level. Alternatively parallel attribute definitions of the same level may also be chosen (see Chapter 3 for further details). An example of a query string using the sequence operator is displayed in R Example 5.2.3. All rows in the resulting segment list have the start time of *@*, the end time of *n* and their labels are *@->n*, where the -> substring denotes the sequence.

R Example 5.2.3

```
# query all sequences of items on the "Phonetic" level
# in which an item containing the label "@" is followed by
# an item containing the label "n"
sl = query(ae, "[Phonetic == @ -> Phonetic == n]")

# show first entry of sl
head(sl, n = 1)

## segment  list from database:  ae
## query was:  [Phonetic == @ -> Phonetic == n]...
##   labels    start      end session   bundle
## 1   @->n 1715.425 1791.425    0000 msajc003
##      level    type
## 1 Phonetic SEGMENT
```

Result modifier

Because users are often interested in just one element of a compound query such as sequence queries (e.g., the @s in a *@->n* sequences), the EQL offers a so-called result modifier symbol, `#`. This symbol may be placed in front of any simple query component of a multi component query as depicted in R Example 5.2.4. Placing

the hashtag in front of either the left or the right simple query term will result in segment lists that contains only the annotation items of the simple query term that have the hashtag in front of it. Only one result modifier may be used per query.

R Example 5.2.4

```
# query the "@"s in "@->n" sequences
sl = query(ae, "[#Phonetic == @ -> Phonetic == n]")

# show first entry of sl
head(sl, n = 1)

## segment  list from database:  ae
## query was:  [#Phonetic == @ -> Phonetic == n]...
##   labels    start      end session   bundle
## 1      @ 1715.425 1741.425    0000 msajc003
##      level    type
## 1 Phonetic SEGMENT

# query the "n"s in a "@->n" sequences
sl = query(ae, "[Phonetic == @ -> #Phonetic == n]")

# show first entry of sl
head(sl, n = 1)

## segment  list from database:  ae
## query was:  [Phonetic == @ -> #Phonetic == n]...
##   labels    start      end session   bundle
## 1      n 1741.425 1791.425    0000 msajc003
##      level    type
## 1 Phonetic SEGMENT
```

Conjunction queries

The syntax of a query string using the conjunction operator can schematically be written as: `[L_a1 == A & L_a2 == B & L_a3 == C & L_a4 == D & ... & L_an == N]`, where annotation items on level `L` have the label `A` and also have the parallel labels `B`, `C`, `D`, ..., `N` (see Chapter 3 for more information about parallel labels). By

analogy with the sequence operator, all simple query statements must refer to the same level (i.e., only parallel attributes definitions of the same level indicated by the `a1 - an` may to be chosen). Hence, the conjunction operator is used to combine query conditions on the same level. Using the conjunction operator is useful for two reasons:

- It combines different attributes of the same level: `[Text == always & Accent == S]` where *Text* and *Accent* are additional attributes of level *Word*; and
- It combines a simple query with a function query (see Position Queries Section 5.2.3): `[Phonetic == l & Start(Word, Phonetic) == 1]`.

An example of a query string using the conjunction operator is displayed in R Example 5.2.5.

R Example 5.2.5

```
# query all words with the orthographic transcription "always"
# that also have a strong word accent ("S")
query(ae, "[Text == always & Accent == S]")

## segment  list from database:  ae
## query was:  [Text == always & Accent == S]...
##   labels   start      end session   bundle level
## 1 always 775.475 1280.175    0000 msajc022  Text
##   type
## 1 ITEM
```

R Example 5.2.5 does not make use of the result modifier symbol. However, only the annotation items of the left simple query term (`Text == always`) are returned. This behavior is true for all EQL operators that combine simple query terms except for the sequence operator. As it is more explicit to use the result modifier to express the desired result, we recommend using the result modifier where possible. The more explicit variant of the above query which yields the same result is `"[#Text == always & Word == C]"`.

Domination/hierarchical queries

Compared to sequence and conjunction queries, a domination query using the operator `^` is not bound to a single level. Instead, it allows users to query annotation

items that are directly or indirectly linked over one or more levels. Queries using the domination operator are often referred to as hierarchical queries as they provide the ability to query the hierarchical annotations in a vertical or inter-level manner. Figure 5.2 shows the same partial hierarchy as Figure 5.1 but highlights the annotational items that are dominated by the strong syllable (*S*) of the *Syllable* level. Such linked hierarchical sub-structures can be queried using hierarchical/domination queries.

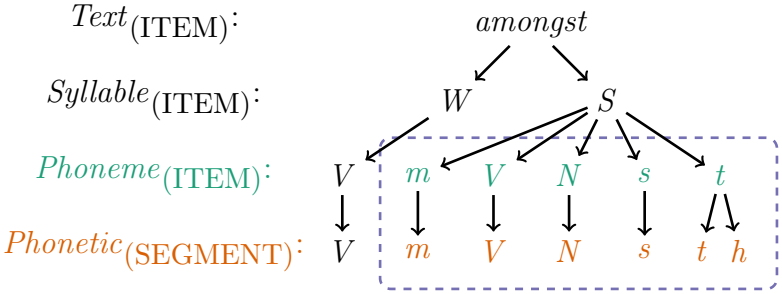


Figure 5.2: Partial hierarchy depicting all annotation items that are dominated by the strong syllable (*S*) of the *Syllable* level (inside dashed box). Items marked green belong to the *Phoneme* level, items marked orange belong to the *Phonetic* level and the purple dashed box indicates the set of items that are dominated by *S*.

A schematic representation of a simple domination query string that retrieves all annotation items `A` of level `L1` that are dominated by items `B` in level `L2` (i.e., items that are directly or indirectly linked) is `[L1 == A ^L2 == B]`. Although the domination relationship is directed the domination operator is not. This means that either items in `L1` dominate items in `L2` or items in `L2` dominate items in `L1`. Note that link definitions that specify the validity of the domination have to be present in the `emuDB` configuration for this to work (see Chapter 4 for details). An example of a query string using the domination operator is displayed in R Example 5.2.6.

R Example 5.2.6

```
# query all "p" phoneme items that belong
# to / are dominated by a strong syllable ("S")
sl = query(ae, "[Phoneme == p ^ Syllable == S]")

# show first entry of sl
head(sl, n = 1)
```

```
## segment  list from database:  ae
## query was:  [Phoneme == p ^ Syllable == S]...
##   labels   start     end session   bundle   level
## 1      p 558.575 639.575    0000 msajc015 Phoneme
##   type
## 1 ITEM
```

As with the conjunction query, if no result modifier is present, a dominates query returns the annotation items of the left simple query term. Hence, the more explicit variant of the above query is `"[#Phoneme == p ^ Syllable == S]"`.

5.2.3 Position queries

The EQL has three function terms that specify where in a domination relationship a child level annotation item is allowed to occur. The three function terms are `Start()`, `End()` and `Medial()`. A schematic representation of a query string representing a simple usage of the `Start()`, `End()` and `Medial()` function would be: `POSFCT(L1, L2) == TRUE`. In this representation `POSFCT` is a placeholder for one of the three functions, at which level `L1` must dominate level `L2`. Where `L1` does indeed dominate `L2`, the corresponding item from level `L2` is returned. If the expression is set to `FALSE` (i.e., `POSFCT(L1, L2) == FALSE`), all the items that do not match the condition of `L2` are returned. An illustration of what is returned by each of the position functions depending on if they are set to `TRUE` or `FALSE` is depicted in Figure 5.3, while R Example 5.2.7 shows an example query using a position query term.

R Example 5.2.7

```
# query all phoneme items that occur
# at the start of a syllable
sl = query(ae, "[Start(Syllable, Phoneme) == TRUE]")

# show first entry of sl
head(sl, n = 1)
```

```
## segment  list from database:  ae
## query was:  [Start(Syllable, Phoneme) == TRUE]...
##   labels   start     end session   bundle   level
## 1      V 187.425 256.925    0000 msajc003 Phoneme
##   type
## 1 ITEM
```

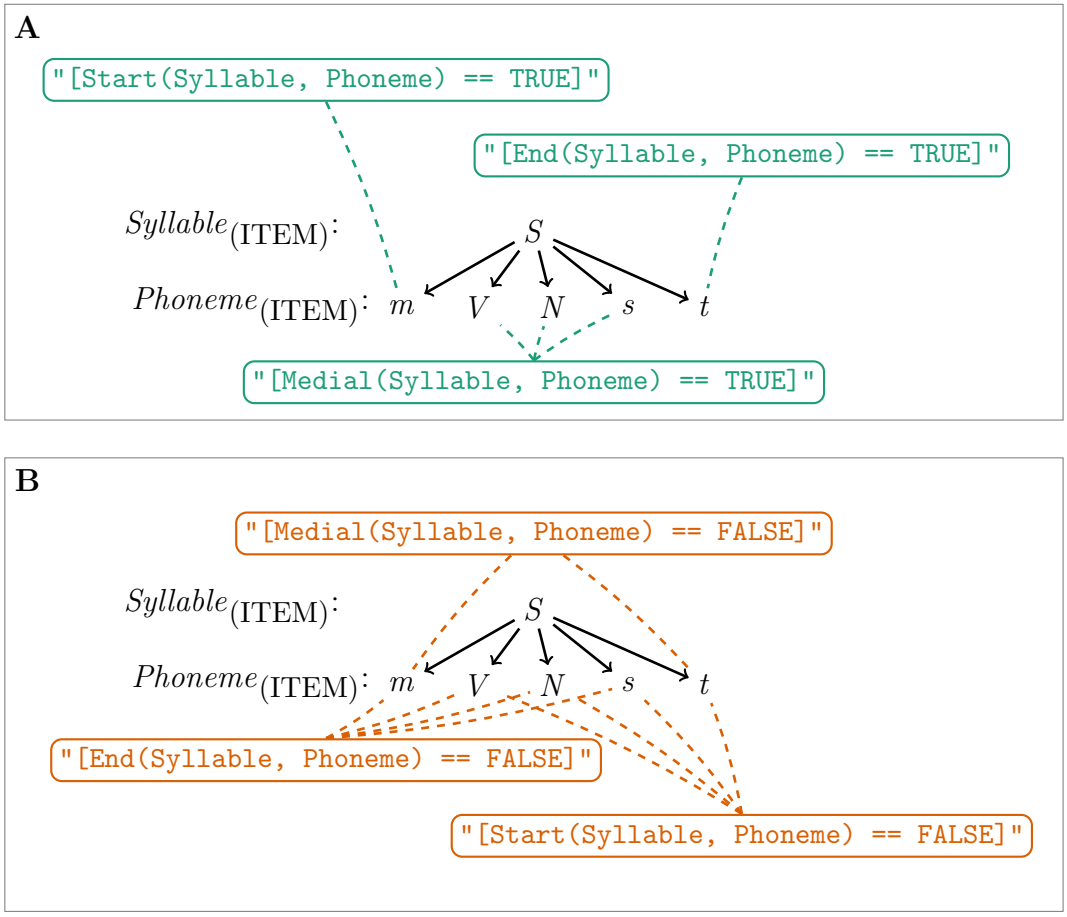


Figure 5.3: Illustration of what is returned by the `Start()`, `Medial()` and `End()` functions depending if they are set to **A:** `TRUE` (green) or **B:** `FALSE` (orange).

5.2.4 Count queries

A further query component of the EQL are so-called count queries. They allow the user to specify how many child nodes a parent annotation item is allowed to have. Figure 5.4 displays two syllables, one containing one phoneme and one phonetic annotation item, the other containing five phoneme and six phonetic items. Using EQL's `Num()` function it is possible to specify which of the two syllables should be retrieved, depending on the number of phonemic or phonetic elements to which it is directly or indirectly linked. R Example 5.2.8 shows a query that queries all syllables that contain five phonemes.

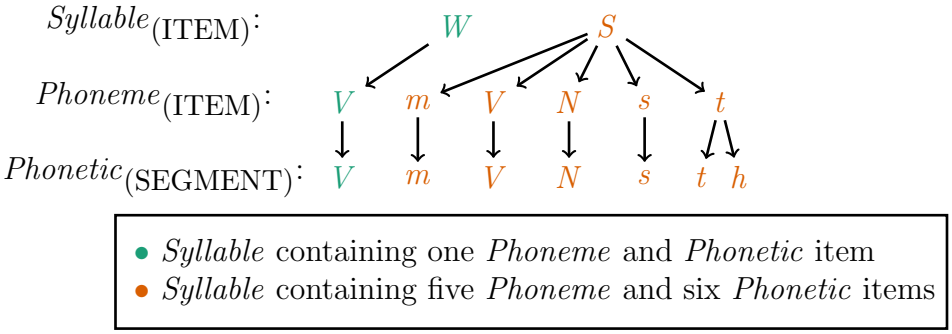


Figure 5.4: Partial hierarchy depicting a *Syllable* containing one *Phoneme* and *Phonetic* item (green) and a *Syllable* containing five *Phoneme* and six *Phonetic* items (orange).

A schematic representation of a query string utilizing the count mechanism would be `[Num(L1, L2) == N]`, where `L1` contains `N` annotation items in `L2`. For this type of query to work `L1` has to dominate `L2` (i.e., be a parent level to `L2`). As the query matches a number (`N`), it is also possible to use the operators `>` (more than), `<` (less than) and `!=` (not equal to). The resulting segment list contains items of `L1`.

R Example 5.2.8

```
# retrieve all syllables that contain five phonemes
query(ae, "[Num(Syllable, Phoneme) == 5]")

## segment  list from database:  ae
## query was:  [Num(Syllable, Phoneme) == 5]...
##   labels    start      end session   bundle
## 1      S  256.925  674.175    0000 msajc003
```

```
## 2      S  739.925 1289.425    0000 msajc003
## 3      W 2228.475 2753.975    0000 msajc010
## 4      S 1890.275 2469.525    0000 msajc022
## 5      S 1964.425 2554.175    0000 msajc023
##      level type
## 1 Syllable ITEM
## 2 Syllable ITEM
## 3 Syllable ITEM
## 4 Syllable ITEM
## 5 Syllable ITEM
```

5.2.5 More complex queries

By using the correct bracketing, all of the above query components can be combined to formulate more complex queries that can be used to answer questions such as: *Which occurrences of the word "his" follow three-syllable words which contain a schwa (@) in the first syllable occur in the database?* Such multi-part questions can usually be broken down into several sub-queries. These sub-queries can then be recombined to formulate the complex query. The steps to answering the above multi-part question are:

1. *Which occurrences of the word "his" ...*: `[Text == his]`
2. *... three-syllable words ...*: `[Num(Text, Syllable) == 3]`
3. *... contain a schwa (@) in the first syllable ...*: `[Phoneme == @ ^Start(Word, Syllable) == 1]`
4. All three can be combined by saying 2 dominates 3 (`[2 ^3]`) and these are followed by 1 (`[2 ^3] -> 1]`)

The combine query is depicted in R Example 5.2.9. This complex query demonstrates the expressive power of the query mechanism that the EMU-SDMS provides.

R Example 5.2.9

```
# perform complex query
# Note that the use of paste0() is optional, as
```

```
# it is only used for formatting purposes
query(ae, paste0("[[[Num(Text, Syllable) == 3] ",
                 "^ [Phoneme == @ ^ Start(Word, Syllable) == 1]] ",
                 "-> #Text = his]"))

## segment  list from database:  ae
## query was:  [[[Num(Text, Syllable) == 3] ^ [Phoneme == @ ^ Sta...
##   labels    start      end session   bundle level
## 1    his 2693.675 2780.725    0000 msajc015  Text
##   type
## 1 ITEM
```

As mastering these complex compound queries can require some practice, several simple as well as more complex examples that combine the various EQL components described above are available in Appendix E. These examples provide practical examples to help users in find queries suited to their needs.

5.2.6 Deducing time

The default behavior of the legacy EMU system was to automatically deduce time information for queries of levels that do not contain time information. This was achieved by searching for the time-bearing sub-level and calculating the start and end times from the left-most and right-most annotation items, which where directly or indirectly linked to the retrieved parent item. This upward purculation of time information is also the default behavior of the new EMU-SDMS. However, a new feature has been added to the query engine which allows the calculation of time to be switched off for a given query using the `calcTimes` parameter of the `query()` function. This is beneficial in two ways: for one, levels that do not have a time-bearing sub-level may be queried and secondly, the execution time of queries can be greatly improved. The performance increase becomes evident when performing queries on large data sets on one of the top levels of the hierarchy (e.g., *Utterance* or *Intonational* in the *ae* `emuDB`). When deducing time information for annotation items that contain large portions of the hierarchy, the query engine has to walk down large partial hierarchies to find the left-most and right-most items on the time-bearing sub-level. This can be a computationally expensive operation and is often unnecessary, especially during data exploration. R Example 5.2.10 shows the usage of this parameter by querying all of the items of the *Intonational* level and displaying the `NA` values for start and end times in the resulting segment list. It is worth noting

that the missing time information excluded during the original query can be retrieved at a later point in time by performing a hierarchical requery (see Section 5.2.7) on the same level.

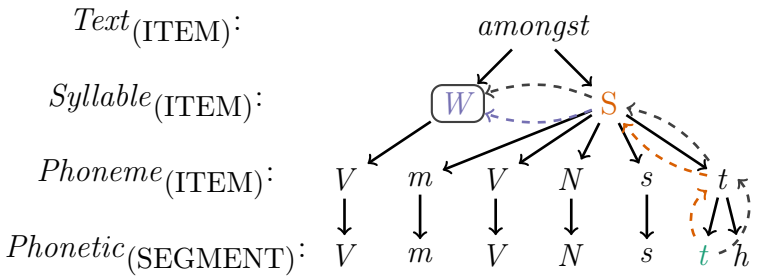
R Example 5.2.10

```
# query all intonational items
sl = query(ae, "Intonational =~ .*", calcTimes = F)

 # show first entry of sl
head(sl, n = 1)

## segment  list from database:  ae
## query was:  Intonational =~ .*...
##   labels start end session   bundle        level
## 1     L%    NA  NA    0000 msajc003 Intonational
##   type
## 1 ITEM
```

5.2.7 Requery

A popular feature of the legacy system was the ability to use the result of a query to perform an additional query, called a requery, starting from the resulting items of a query. The requery functionality was used to move either sequentially (horizontally) or hierarchically (vertically) through the hierarchical annotation structure. Although this feature technically does not extend the querying functionality (it is possible to formulate EQL queries that yield the same results as a query followed by $1 : n$ requeries), requeries benefit the user by breaking down the task of formulating long query terms into multiple, simpler queries. Compared with the legacy system, this feature is implemented in the `emuR` package in a more robust way, as unique item IDs are present in the result of a query, eliminating the need for searching the starting segments based on their time information. Examples of queries and their results within a hierarchical annotation based on a hierarchical and sequential requery as well as their EQL equivalents are illustrated in Figure 5.5.

R Example 5.2.11 illustrates how the same results of the sequential query `[#Phonetic =~.* -> Phonetic == n]` can be achieved using the `requery_seq()` function. Further, it shows how the `requery_hier()` function can be used to move vertically




```
sl1 = query(emuDBhandle, query = "Phonetic == t")
sl2 = requery_hier(emuDBhandle, sl1, level = "Syllable")
sl3 = requery_seq(emuDBhandle, sl2, offset = -1)
```

vs.

```
sl3 = query(emuDBhandle, query = "[#Syllable =~ .* ->
[Syllable =~ .* ^ Phonetic == t]]")
```

Figure 5.5: Three-step (query → requery_hier → requery_seq) requery procedure, its single query counterpart and their color coded movements within the annotation hierarchy.

through the annotation structure by starting at the *Syllable* level and retrieving all the *Phonetic* items for the query result.

R Example 5.2.11

```
########################
# requery_seq()

# query all "n" phonetic items
sl_n = query(ae, "Phonetic == n")

# sequential requery (left shift result by 1 (== offset of -1))
# and hence retrieve all phonetic items directly preceeding
# all "n" phonetic items
sl_precn = requery_seq(ae, seglist = sl_n, offset = -1)

# show first entry of sl_precn
head(sl_precn, n = 1)

## segment  list from database:  ae
```

```
## query was:  FROM REQUERY...
##   labels   start      end session   bundle
## 1      E 949.925 1031.925    0000 msajc003
##      level    type
## 1 Phonetic SEGMENT

########################
# requery_hier()

# query all strong syllables (S)
sl_s = query(ae, "Syllable == S")

# hierarchical requery
sl_phonetic = requery_hier(ae, seglist = sl_s,
                           level = "Phonetic")

# show first entry of sl_phonetic
head(sl_phonetic, n = 1)

## segment  list from database:  ae
## query was:  FROM REQUERY...
##             labels   start     end session
## 1 m->V->N->s->t->H 256.925 674.175    0000
##     bundle    level    type
## 1 msajc003 Phonetic SEGMENT
```

5.3 Discussion

This chapter gave an overview of the abilities of the query system of the EMU-SDMS. We feel the EQL is a expressive, powerful, yet simple to learn and domain-specific query language that allows users to adequately query complex annotation structures. Further we feel that the query system provided by the EMU-SDMS surpasses the querying capabilities of most commonly used systems. As the result of a query is a superclass of the common `data.frame` object, these results can easily be further processed using various R functions (e.g., to remove unwanted segments). Further, the results of queries can be used as input to the `get_trackdata()` function (see Chapter

6) which makes the query system a vital part in the default workflow described in Chapter 1.

Although the query mechanism of the EMU-SDMS covers most linguistic annotation query needs (including co-occurrence and domination relationship child position queries), it has limitations due to its domain-specific nature, its simplicity and its predefined result type. Performing more general queries such as: *What is the average age of the male speakers in the database who are taller than 1.8 meters?* is not directly possible using the EQL. Even if the gender, height and age parameters are available as part of the database's annotations (e.g., using the single bundle root node metadata strategy described in Chapter 3) they would be encoded as strings, which do not permit direct calculations or numerical comparisons. However, it is possible to answer these types of questions using a multi-step approach. One could, for example, extract all height items and convert the strings into numbers to filter the items containing a label that is greater than 1.8. These filtered items could then be used to perform two requeries to extract all male speakers and their age labels. These age labels could once again be converted into numbers to calculate their average. Although not as elegant as other languages, we have found that most questions that arise as part of studies working with spoken language database can be answered using such a multi-step process including some data manipulation in R, provided the necessary information is encoded in the database. Additionally, from the viewpoint of a speech scientist, we feel that the intuitiveness of an EQL expression (e.g., a query to extract the sibilant items for the question asked in the introduction: `"Phonetic == s|z|S|Z"`) exceeds that of a comparable general purpose query language (e.g. a semantically similar SQL statement: `SELECT desired_columns FROM items AS i, labels AS l WHERE i.unique_bundle_item_id = l.uniq_bundle_item_id AND l.label = 's' OR l.label = 'z' OR l.label = 's' OR l.label = 'S' OR l.label = 'Z'`). This difference becomes even more apparent with more complex EQL statements, which can have very long, complicated and sometimes multi-expression SQL counterparts.

A problem which the EMU-SDMS does not explicitly address is the problem of cross-corpus searches. Different `emuDB`s may have varying annotation structures with varying semantics regarding the names or labels given to objects or annotation items in the databases. This means that it is very likely that a complex query formulated for a certain `emuDB` will fail when used to query other databases. If, however, the user either finds a query that works on every `emuDB` or adapts the query to extract the items she/he is interested in, a cross-corpus comparison is simple. As the result of a query and the corresponding data extraction routines are the same, regardless of database they where extracted from, these results are easily comparable. However, it is worth noting that the EMU-SDMS is completely indifferent to the semantics

of labels and level names, which means it is the user's responsibility to check if a comparison between databases is justifiable (e.g., *are all segments containing the label "@" of the level "Phonetic" in all* `emuDB`*s annotating the same type of phoneme?*).

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).

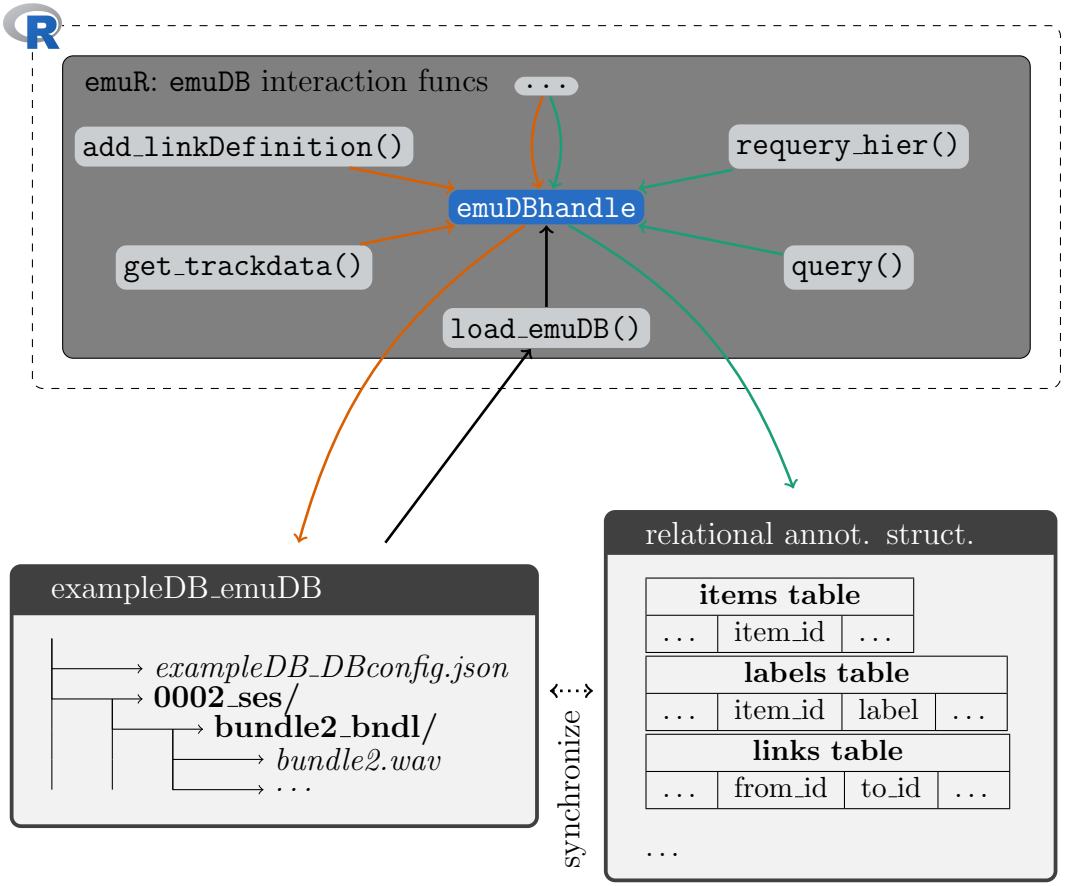


Figure 10.1: Schematic architecture of `emuDB` interaction functions of the `emuR` package. Orange paths show functions interacting with the files of the `emuDB`, while green paths show functions accessing the relational annotation structure. Actions like saving a changed annotation using the `EMU-webApp`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

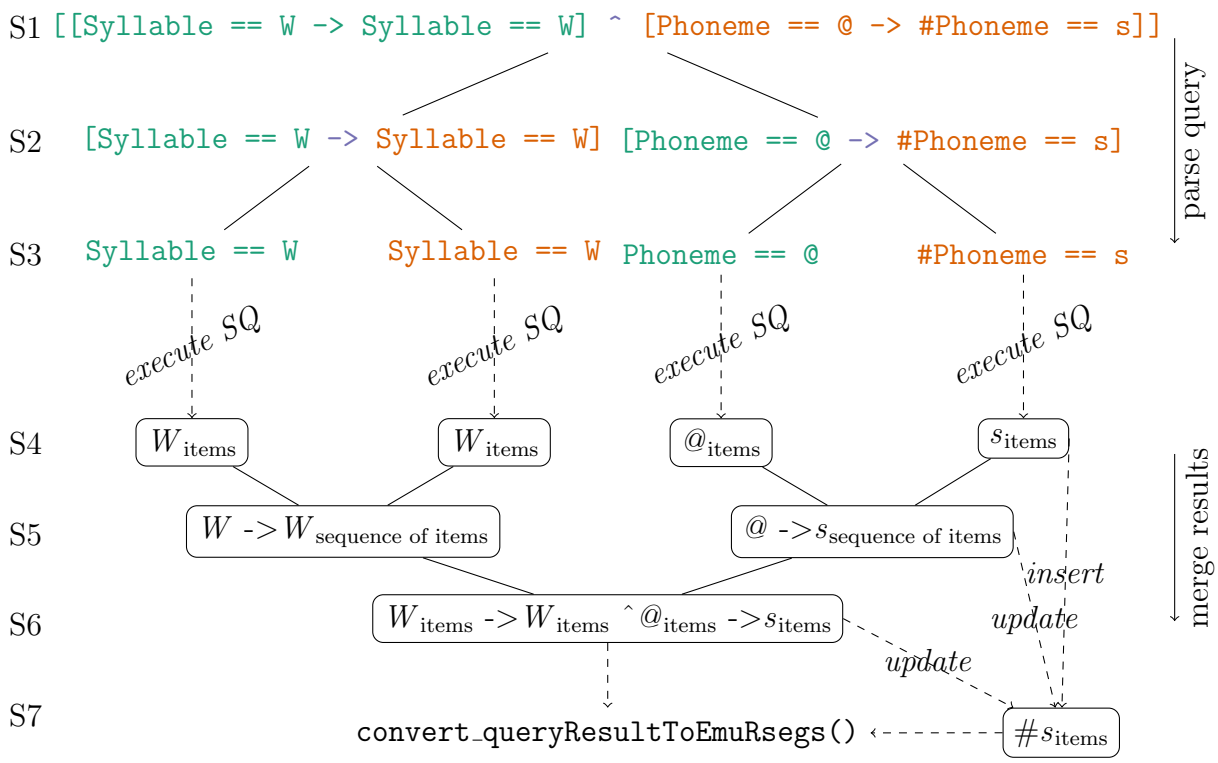


Figure 10.2: Example of how the query expression parser parses and evaluates an EQL expression and merges the result according to the respective EQL operators.

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

```
1: function query_DbEqlFUNCQ(query)
2:     place all parent level items into tmp table
3:     place all child level items into tmp table
4:     query_DbHier(parentItemsTable, childItemsTable)
5:     if Start End or Medial query then
6:         extract parent items and place in tmp result table
7:     else
8:         extract child items and place in tmp result table
9:     end if
10: end function
11: function query_DbEqlLABELQ(query)
12:     splitLabels ← split labels at |
13:     for all splitLabels do
14:         if operator is ==, = or ! = then
15:             extract items that contain labels which are equal or unequal to label
16:         else if operator is =  or ! ∼ then
17:             extract items that contain labels that match or don't match RegEx
18:         end if
19:         merge results in tmp table
20:     end for
21: end function
22: function query_DbEqlSQ(query)
23:     if query contains round brackets then
24:         query_DbEqlFUNCQ(query)
25:     else
26:         query_DbEqlLABELQ(query)
27:     end if
28: end function
29: function query_DbEqlCONJQ(query)
30:     splitItems ← split query at &
31:     for all splitItems do
32:         query_DbEqlSQ(splitItems)
33:         merge results in tmp table
34:     end for
35: end function
36: function query_DbHier(leftTable, rightTable)
37:     hp ← extract hier. paths conn. leftTable and rightTable level names
38:     for all child and parent level pairs in hp do
39:         connect child and parent items using links table
40:         reduce to min seq. idx (left side of trapeze)
41:         and to max seq. idx (right side of trapeze)
42:     end for
43: 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.

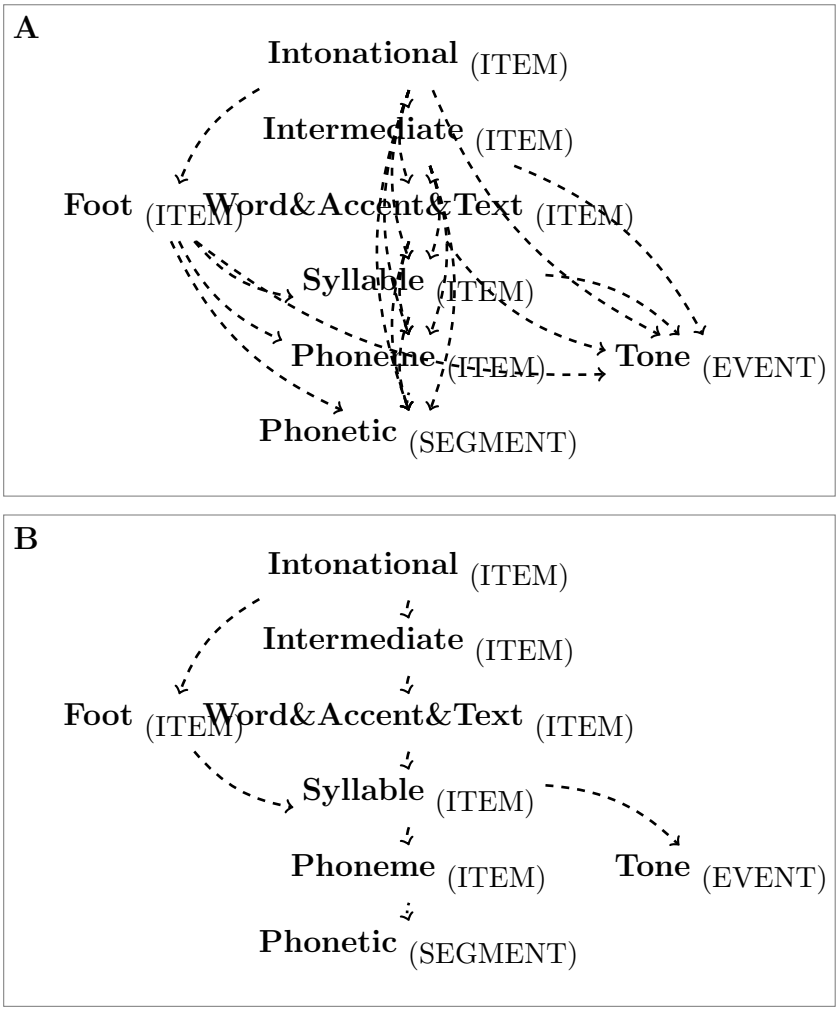


Figure 10.3: Schematic of hierarchy graph *ae*; **A**: legacy cluttered redundant strategy vs. **A**: cleaner non-redundant strategy.

Appendix D

EQL EBNF

This chapter presents the Extended Backus-Naur Form (EBNF) that describes the EQL. As the original EBNF adapted from John (2012) was written in German, some of the abbreviation terms were translated into English abbreviations (e.g., `DOMA` is the abbreviation for the German term **Dom**inanz**a**bfrage and the newly translated `DOMQ` is the abbreviation for the English term **dom**ination **q**uery).

D.0.1 Terminal symbols of EQL2 (operators) and their meaning

The terminal symbols described below are listed in descending order by their binding priority.

Symbol	Meaning
`#`	Result modifier (projection)
`,`	Parameter list separator
`==`	Equality (new in version 2 of the EQL; added for cleaner syntax)
`=`	Equality (optional; for backwards compatibility)
`!=`	Inequality
`=~`	Regular expression matching
`!~`	Regular expression non-matching
>	Greater than
>=	Equal to or greater than
<	Less than
>=	Equal to or less than

Symbol	Meaning
|	Alternatives separator
&	Conjunction of equal rank
^	Dominance conjunction
->	Sequence operator

D.0.2 Terminal symbols of EQL2 (brackets) and their meanings.

Symbol	Meaning
'	Quotes literal string
(	Function parameter list opening bracket
)	Function parameter list closing bracket
[	Sequence or dominance-enclosing opening bracket
]	Sequence or dominance-enclosing closing bracket

D.0.3 Terminal symbols of EQL2 (functions) and their meanings.

Symbol	Meaning
Start	Start
Medial	Medial
End	Final
Num	Count

D.0.4 Formal description of EMU Query Language Version 2

EBNF term	Abbreviation	Conditions
EQL = CONJQ | SEQQ | DOMQ;	**E**MU **Q**uery **L**anguage	

EBNF term	Abbreviation	Conditions
`DOMQ = "[", ( CONJQ | DOMQ | SEQQ ), "^", ( CONJQ | DOMQ | SEQQ ), "]";`	**dom**inance **q**uery	levels must be hierarchically associated
`SEQQ = "[", ( CONJQ | SEQQ | DOMQ ), "->", ( CONJQ | SEQQ | DOMQ ), "]";`	**seq**uential **q**uery	levels must be linearly associated
`CONJQ = { "[" }, SQ, { "&", SQ }, { "]" };`	**conj**unction **q**uery	levels must be linearly associated
`SQ = LABELQ | FUNCQ;`	**s**imple **q**uery	
`LABELQ = [ "#" ], LEVEL, ( "=" | "==" | "!=" | "=~" | "!~" ), LABELALTERNATIVES;`	**label q**uery	
`FUNCQ = POSQ | NUMQ;`	**fun**ction **q**uery	
`POSQ = POSFCT, "(", LEVEL, ",", LEVEL, ")", "=", "0" | "1";`	**position q**uery	levels must be hierarchically associated; second level determines semantics
`NUMQ = "Num", "(", LEVEL, ",", LEVEL, ")", COP, INTPN;`	**number q**uery	levels must be hierarchically associated; first level determines semantics
`LABELALTERNATIVES = LABEL , { "|", LABEL };`	**label alternatives**	
`LABEL = LABELING | ( "'", LABELING, "'" );`	**label**	levels must be part of the database structure; `LABELING` is an arbitrary character string or a label group class configured in the emuDB; result modifier `#` may only occur once

EBNF term	Abbreviation	Conditions
`POSFCT = "Start" | "Medial" | "End";`	**pos**ition **f**un**ct**ion	
`COP = "=" | "==" | "!=" | ">" | "<" | "<=" | ">=";`	**c**omparison **o**perator	
`INTPN = "0" | INTP;`	**int**eger **p**ositive with **n**ull	
`INTP = DIGIT-"0", { DIGIT };`	**int**eger **p**ositive	
`DIGIT = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";`	**digit**	

INFO: The `LABELING` term used in the `LABEL` EBNF term can represent any character string that is present in the annotation. As this can be any combination of Unicode characters, we chose not to explicitly list them as part of the EBNF.

D.0.5 Restrictions

A query may only contain a single result modifier `#` (hashtag).

Appendix E

EQL: further examples

Below are examples of query strings that have been adapted from Cassidy and Harrington (2001) and Harrington and Cassidy (2002) and which are displayed as questions and answers. All examples use the *ae* demo `emuDB`, which is provided by the `emuR` package, and were extracted from the `EQL` vignette of the `emuR` package. Descriptions (some of them duplicates of those in Chapter 5) of the various syntaxes and query components are also included for easier reading. R Example E.0.1 shows how to access the *ae* demo `emuDB`.

R Example E.0.1

```
# load the package
library(emuR)

# create demo data in directory provided by the tempdir() function
create_emuRdemoData(dir = tempdir())

# get the path to emuDB called 'ae' that is part of the demo data
path2directory = file.path(tempdir(), "emuR_demoData", "ae_emuDB")

# load emuDB into current R session
ae = load_emuDB(path2directory)
```

E.0.1 Simple equality, inequality, matching and non-matching queries (single-argument)

The syntax of a simple, equality, inequality, matching and non-matching query is `[L OPERATOR A]` where `L` specifies a level (or alternatively the name of a parallel attribute definition), `OPERATOR` is one of the following operators: `==` (equality); `!=` (inequality); `=~` (matching) or `!~` (non-matching), and `A` is an expression specifying the labels of the annotation items of `L`.

Example questions and answers:

- **Q**: *What is the query to retrieve all items containing the label "m" in the "Phonetic" level?*
- **A**:

```
query(emuDBhandle = ae,
      query = "[Phonetic == m]")
```

- **Q**: *What is the query to retrieve all items containing the label "m" or "n" in the "Phonetic" level?*
- **A**:

```
query(emuDBhandle = ae,
      query = "[Phonetic == m | n]")
```

- **Q**: *What is the query to retrieve all items that do not contain the label "m" or "n"?*
- **A**:

```
query(emuDBhandle = ae,
      query = "[Phonetic != m | n]")
```

- **Q**: *What is the query to retrieve all items in the "Syllable" level?*
- **A**:

```
query(emuDBhandle = ae,
      query = "[Syllable =~ .*]")
```

- **Q**: *What is the query to retrieve all items that begin with "a" in the "Text" level?*
- **A**:

```
query(ae, "[Text =~ a.*]")
```

- **Q**: *What is the query to retrieve all items that do not begin with "a" in the "Text" level?*
- **A**:

```
query(ae, "[Text !~ a.*]")
```

The above examples use three operators that are new to the EQL as of version 2. One is the `==` equal operator, which has the same meaning as the `=` operator of the EQL version 1 (which is also still available) while providing a cleaner, more precise syntax. The other two are `=~` and `!~`, which are the new matching and non-matching regular expression operators. Further, it is worth noting that the use of parentheses, blanks or characters that represent operands used by the EQL (see EBNF in Appendix D) as part of a label matching string (the string on the right hand side of one of the operands mentioned above), must be placed in additional single quotation marks to escape these characters. For example, searching for the items containing the labels `O_'` on the Phonetic level could not be written as `"[Phonetic == O_']"` but would have to be written as `"[Phonetic == 'O_'']"`. Reversing the order of single vs. double quotation marks is currently not supported, that is `'[Phonetic == ''O_''']'` will currently not work. Hence, to avoid this issue only double quotation marks for the outer wrapping of the query string should be used.

E.0.2 Sequence queries using the -> sequence operator

The syntax of a query string using the `->` sequence operator is `[L == A -> L == B]`, where item `A` on level `L` precedes item `B` on level `L`. For a sequential query to

work both arguments must be on the same level (alternatively, parallel attribute definitions of the same level may also be chosen).

Example **Q** & **A**'s:

- **Q**: *What is the query to retrieve all sequences of items containing the label "@" followed by items containing the label "n" on the "Phonetic" level?*
- **A**:

```
# NOTE: all row entries in the resulting
# segment list have the start time of "@", the
# end time of "n", and their labels will be "@->n"
query(ae, "[Phonetic == @ -> Phonetic == n]")
```

- **Q**: *Same as the question above but this time we are only interested in the items containing the label "@" in the sequences.*
- **A**:

```
# NOTE: all row entries in the resulting
# segment list have the start time of "@", the
# end time of "@" and their labels will also be "@"
query(ae, "[#Phonetic == @ -> Phonetic == n]")
```

- **Q**: *Same as the first question but this time we are only interested in the items containing the label "n".*
- **A**:

```
# NOTE: all row entries in the resulting
# segment list have the start time of "n", the
# end time of "n" and their labels will also be "n"
query(ae, "[Phonetic == @ -> #Phonetic == n]")
```

Subsequent sequence queries using nesting of the -> sequence operator

The general strategy for constructing a query string that retrieves subsequent sequences of labels is to nest multiple sequences while paying close attention to the correct placement of the parentheses. An abstract version of such a query string for the subsequent sequence of arguments `A1`, `A2`, `A3` and `A4` would be: `[[[[A1 -> A2] -> A3] -> A4] -> A5]` where each argument (e.g. `A1`) represents an equality, inequality, matching or non-matching expression on the same level (alternatively, parallel attribute definitions of the same level may also be chosen).

Example questions and answers:

- **Q**: *What is the query to retrieve all sequences of items containing the labels "@", "n" and "s" on the "Phonetic" level?*
- **A**:

```
query(ae, "[[Phonetic == @ -> Phonetic == n] -> Phonetic == s]")
```

- **Q**: *What is the query to retrieve all sequences of items containing the labels "to", "offer" and "any" on the "Text" level?*
- **A**:

```
query(ae, "[[Text == to -> Text == offer] -> Text == any]")
```

- **Q**: *What is the query to retrieve all sequences of items containing labels "offer" followed by two arbitrary labels followed by "resistance"?*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formating purposes
query(ae, paste0("[[[Text == offer -> Text => .*] ",
                 "-> Text => .*] -> Text == resistance]"))
```

As the EQL1 did not have a regular expression operator, users often resorted to using queries such as `[Phonetic != XXX]` (where XXX is a label that was not part of the label set of the *Phonetic* level) to match every label on the `Phonetic` level. Although this is still possible in the EQL2, we strongly recommend using regular expressions as they provide a much clearer and more precise syntax and are less error-prone.

E.0.3 Conjunction operator &

The syntax of a query string using the conjunction operator can schematically be written as: `[L == A & L_a2 == B & L_a3 == C & L_a4 == D & ... & L_an == N]`, where items on level `L` have the label `A` (technically belonging to the first attribute of that level, i.e., `L_a1`, which per default has the same name as its level) also have the attributes `B`, `C`, `D`, ..., `N`. As with the sequence operator all expressions must be on the same level (i.e., parallel attribute definitions of the same level indicated by the `a2 - an` may to be chosen).

The conjunction operator is used to combine query conditions on the same level. This makes sense in two cases:

1. when to combining different attributes of the same level: `"[Phonetic == l & sonorant == T]"` when *Sonorant* is an additional attribute of level *Phonetic*;
2. when combining a basic query with a function (see sections Position and Count below): `"[phonetic == l & Start(word, phonetic) == 1]"`.

Example questions and answers:

- **Q**: *What is the query to retrieve all items containing the label "always" in the "Text" attribute definition which also have the label "C" on a parallel attribute definition called "Word"?*
- **A**:

```
query(ae, "[Text == always & Word == C]")
```

- **Q**: *What is the query to retrieve all items of the attribute definition "Text" of the level "Word" that were also labeled as function words (labeled "F" in the "Word" level)?*
- **A**:

```
query(ae, "[Text =~ .* & Word == F]")
```

- **Q**: *What is the query to retrieve all items of the attribute definition "Text" of the level "Word" that were also labeled as content words (labeled "C" in the "Word" level) and as accented (labeled "S" in the attribute definition "Accent" of the same level)?*

- **A**:

```
query(ae, "[Text =~ .* & Word == C & Accent == S]")
```

E.0.4 Domination operator ^ (hierarchical queries)

A schematic representation of a simple domination query string that retrieves all items containing label `A` of level `L1` that are dominated by (i.e., are directly or indirectly linked to) items containing the label `B` in level `L2` is `[L1 == A ^ L2 == B]`. The domination operator is not directional, meaning that either items in `L1` dominate items in `L2` or items in `L2` dominate items in `L1`. Note that link definitions that specify the validity of the domination have to be present in the emuDB for this to work.

Simple domination

Example questions and answers:

- **Q**: *What is the query to retrieve all items containing the label "p" in the "Phoneme" level that occur in strong syllables (i.e., dominated by/linked to items of the level "Syllable" that contain the label "S")?*
- **A**:

```
query(ae, "[Phoneme == p ^ Syllable == S]")
```

- **Q**: *What is the query to retrieve all syllable items which contain a Phoneme item labeled "p"?*
- **A**:

```
query(ae, "[Syllable =~ .* ^ Phoneme == p]")
# or
query(ae, "[Phoneme == p ^ #Syllable =~ .*]")
```

- **Q**: *What is the query to retrieve all syllable items which do not contain a Phoneme item labeled "k" or "p" or "t"?*

- **A**:

```
query(ae, "[Syllable =~ .* ^ Phoneme != p | t| k]")
# or
query(ae, "[Phoneme != p | t | k ^ #Syllable =~ .*]")
```

Even though the domination operator is not directional, what you place to the left and right of the operator does have an impact on the result. If no result modifier (the hash tag `#`) is used, the query engine will automatically assume that the expression to the left of the operator specifies what is to be returned. This means that the schematic query string `[L1 == A ^ L2 == B]` is semantically equal to the query string `[#L1 == A ^ L2 == B]`. As it is more explicit to mark the desired result we recommend you always use the result modifier where possible.

Multiple domination

The general strategy when constructing a query string that specifies multiple domination relations of items is to nest multiple domination expressions while paying close attention to the correct placement of the parentheses. A dominance relationship sequence or the arguments `A1`, `A2`, `A3`, `A4`, can therefore be noted as: `"[[[[A1 ^ A2] ^ A3] ^ A4] ^ A5]"` where `A1` is dominated by `A2` and `A3` and so on.

Example questions and answers:

- **Q**: *What is the query to retrieve all items on the "Phonetic" level that are part of a strong syllable (labeled "S") and belong to the words "amongst" or "beautiful"?*
- **A**:

```
# NOTE: usage of R's paste0() function is optional
# as it is only used for formatting purposes
query(ae, paste0("[[#Phonetic =~ .* ^ Syllable == S] ",
                 "^ Text == amongst | beautiful]"))
```

- **Q**: *The same as the question above but this time we want the "Text" items.*
- **A**:

```
# NOTE: usage of R's paste0() function is optional
# as it is only used for formatting purposes
query(ae, paste0("[[Phonetic =~ .* ^ Syllable == S] ",
                 "^ #Text == amongst | beautiful]"))
```

E.0.5 Position

The EQL has three function terms to specify where in a domination relationship a child level item is allowed to occur. The three function terms are `Start()`, `End()` and `Medial()`.

Simple usage of `Start()`, `End()` and `Medial()`

A schematic representation of a query string representing a simple usage of the `Start()`, `End()` and `Medial()` function would be: `"POSFCT(L1, L2) == 1"` or `"POSFCT(L1, L2) == TRUE"`. In this representation `POSFCT` is a placeholder for one of the three functions where the level `L1` must dominate level `L2`. The `== 1` / `== TRUE` part of the query string indicates that if a match is found (match is `TRUE` or `== 1`), the according item of level `L2` is returned. If this expression is set to `== 0` / `== FALSE` (`FALSE`), all the items that do not match the condition of `L2` will be returned. A visualization of what is returned by the various options of the three functions is displayed in Figure 5.4.

As using `1` and `0` for `TRUE` and `FALSE` is not that intuitive to many R users, the EQL version 2 optionally allows for the values `TRUE`/`T` and `FALSE`/`F` to be used instead of `1` and `0`. This syntax should be more familiar to most R users.

Example questions and answers:

- **Q**: *What is the query to retrieve all word-initial syllables?*
- **A**:

```
query(ae, "[Start(Word, Syllable) == TRUE]")
```

- **Q**: *What is the query to retrieve all word-initial phonemes?*
- **A**:

```
query(ae, "[Start(Word, Phoneme) == TRUE]")
```

- **Q**: *What is the query to retrieve all non-word-initial syllables?*
- **A**:

```
query(ae, "[Start(Word, Syllable) == FALSE]")
```

- **Q**: *What is the query to retrieve all word-final syllables?*
- **A**:

```
query(ae, "[End(Word, Syllable) == TRUE]")
```

- **Q**: *What is the query to retrieve all word-medial syllables?*
- **A**:

```
query(ae, "[Medial(Word, Syllable) == TRUE]")
```

Position and boolean &

The syntax for combining a position function with the boolean operator is `[L == E & Start(L, L2) == TRUE]`, where item `E` on level `L` occurs at the beginning of item `L`. Once again, `L` has to dominate `L2` (optionally, parallel attribute definitions of the same level may also be chosen).

Example questions and answers:

- **Q**: *What is the query to retrieve all "n" Phoneme items at the beginning of a syllable?*
- **A**:

```
query(ae, "[Phoneme == n & Start(Syllable, Phoneme) == 1]")
```

- **Q**: *What is the query to retrieve all word-final "m" Phoneme items?*
- **A**:

```
query(ae, "[Phoneme == m & End(Word, Phoneme) == 1]")
```

- **Q**: *What is the query to retrieve all non-word-final "S" syllables?*
- **A**:

```
query(ae, "[Syllable == S & End(Word, Syllable) == 0]")
```

Position and boolean ^

The syntax for combining a position function with the boolean hierarchical operator is `[L == E ^ Start(L1, L2) == 1]`, where level `L` and level `L2` refer to different levels where either `L` dominates `L2`, or `L2` dominates `L`.

Example questions and answers:

- **Q**: *What is the query to retrieve all "p" Phoneme items which occur in the first syllable of the word?*
- **A**:

```
query(ae, "[Phoneme == p ^ Start(Word, Syllable) == 1]")
```

- **Q**: *What is the query to retrieve all phonemes which do not occur in the last syllable of the word?*
- **A**:

```
query(ae, "[Phoneme =~ .* ^ End(Word, Syllable) == 0]")
```

E.0.6 Count

A schematic representation of a query string using the count mechanism looks like `[Num(L1, L2) == N]`, where `L1` contains `N` items in `L2`. For this type of query to work, `L1` has to dominate `L2`. As the query matches a number (`N`), it is also possible to use the operators > (more than), < (less than) and `!=` (not equal). The resulting segment list contains items of `L1`.

Example questions and answers:

- **Q**: *What is the query to retrieve all words that contain four syllables?*
- **A**:

```
query(ae, "[Num(Word, Syllable) == 4]")
```

- **Q**: *What is the query to retrieve all syllables that contain more than six phonemes?*
- **A**:

```
query(ae, "[Num(Syllable, Phoneme) > 6]")
```

Count and boolean `&`

A schematic representation of a query string combining the count and the boolean operators looks like `[L == E & Num(L1, L2) == N]`, where items `E` on level `L` are dominated by `L1` and `L1` contains `N` `L2` items. Further, `L1` dominates `L2` on the condition that `L` and `L1` (not `L2`) refer to the same level (parallel attribute definitions of the same level may also be chosen).

Example questions and answers:

- **Q**: *What is the query to retrieve the "Text" of all words which consist of more than five phonemes?*
- **A**:

```
query(ae, "[Text =~ .* & Num(Text, Phoneme) > 5]")
# or
query(ae, "[Text =~ .* & Num(Word, Phoneme) > 5]")
```

- **Q**: *What is the query to retrieve all strong syllables that contain five phonemes?*
- **A**:

```
query(ae, "[Syllable == S & Num(Syllable, Phoneme) == 5]")
```

Count and ^

A schematic representation of a query string combining the count and the boolean operators is `[L == E ^ Num(L1, L2) == N]` where items `E` on level `L` are dominated by `L1` and `L1` contains `N` `L2` items. Further, `L1` dominates `L2` on the condition that `L` and `L1` do **not** refer to the same level.

Example questions and answers:

- **Q**: *What is the query to retrieve all "m" phonemes in three-syllable words?*
- **A**:

```
query(ae, "[Phoneme == m ^ Num(Word, Syllable) == 3]")
```

- **Q**: *What is the query to retrieve all "W" syllables in words of three syllables or less?*
- **A**:

```
query(ae, "[Syllable = W ^ Num(Word, Syllable) <= 3]")
```

- **Q**: *What is the query to retrieve all words containing syllables which contain four phonemes?*
- **A**:

```
query(ae, "[Text =~ .* ^ Num(Syllable, Phoneme) == 4]")
```

E.0.7 Combinations

^ and -> (domination and sequence)

A schematic representation of a query string combining the domination and the sequence operators is `[[A1 ^ A2] -> A3]`, where `A1` and `A3` refer to the same level (parallel attribute definitions of the same level may also be chosen).

Example questions and answers:

- **Q**: *What is the query to retrieve all "m" preceding "p" when "m" is part of an "S" syllable?*

- **A**:

```
query(ae, "[[Phoneme == m -> Phoneme =~ p] ^ Syllable == S]")
```

- **Q**: *What is the query to retrieve all "s" preceding "t" when "t" is part of a "W" syllable?*
- **A**:

```
query(ae, "[Phoneme == s -> [Phoneme == t ^ Syllable == W]]")
```

- **Q**: *What is the query to retrieve all "S" syllables, containing an "s" phoneme and preceding an "S" syllable?*
- **A**:

```
query(ae, "[[#Syllable == S ^ Phoneme == s] -> Syllable == S]")
```

- **Q**: *Same question as above but this time we want all "s" items where "s" is part of a "S" syllable and the "S" syllable precedes another "S" syllable.*
- **A**: `"[[Phoneme == s ^ Syllable == S] -> Syllable == S]"` would cause an error as `Phoneme == s` and `Syllable == S` are not on the same level. Therefore, the correct answer is:

```
query(ae, "[[Syllable == S ^ #Phoneme == s] -> Syllable == S]")
```

^ and -> and & (domination and sequence and boolean &)

Example questions and answers:

- **Q**: *What is the query to retrieve the "Text" of all words beginning with a "@" on the "Phoneme" level?*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formatting purposes
query(ae, paste0("[Text =~ .* ^ Phoneme == @ ",
                 "& Start(Text, Phoneme) == 1]"))
```

- **Q**: *What is the query to retrieve all word-initial "m" items in a "S" syllable preceding "o:"?*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formatting purposes
query(ae, paste0("[[Phoneme == m & Start(Word, Phoneme) == 1 ",
                 "-> Phoneme == o:] ^ Syllable == S]"))
```

- **Q**: *Same question as the question above, but this time we want the "Text" items.*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formatting purposes
query(ae, paste0("[[[Phoneme == m & Start(Word, Phoneme) == 1 ",
                 "-> Phoneme == o:] ^ Syllable == S] ",
                 "^ #Text =~ .*]"))
```

E.0.8 A few more questions and answers (because practice makes perfect)

- **Q**: *What is the query to retrieve all "m" or "n" phonemes which occur in the word-medial position?*
- **A**:

```
query(ae, "[Phoneme == m | n & Medial(Word, Phoneme) == 1]")
```

- **Q**: *What is the query to retrieve all "H" phonetic segments followed by an arbitrary segment and then by either "I" or "U"?*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formatting purposes
query(ae, paste0("[[Phonetic == H -> Phonetic =~ .*] ",
                 "-> Phonetic == I | U]"))
```

- **Q**: *What is the query to retrieve all syllables which do not occur in word-medial positions?*
- **A**:

```
query(ae, "[Syllable =~ .* & Medial(Word, Syllable) == 0]")
```

- **Q**: *What is the query to retrieve the "Text" items of all words containing two syllables?*
- **A**:

```
query(ae, "[Text =~ .* & Num(Text, Syllable) == 2]")
```

- **Q**: *What is the query to retrieve the "Text" items of all accented words following "the"?*
- **A**:

```
query(ae, "[Text == the -> #Text =~ .* & Accent == S]")
```

- **Q**: *What is the query to retrieve all "S" (strong) syllables consisting of five phonemes?*
- **A**:

```
query(ae, "[Syllable = S ^ Num(Word, Phoneme) == 5]")
```

- **Q**: *What is the query to retrieve all "W" (weak) syllables containing a "@" phoneme?*
- **A**:

```
query(ae, "[Syllable == W ^ Phoneme == @]")
```

- **Q**: *What is the query to retrieve all Phonetic items belonging to a "W" (weak) syllable?*
- **A**:

```
query(ae,"[Phonetic =~ .* ^ #Syllable == W]")
```

- **Q**: *What is the query to retrieve "W" (weak) syllables in word-final position occurring in three-syllable words?*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formatting purposes
query(ae, paste0("[Syllable == W & End(Word, Syllable) == 1 ",
                 "^ Num(Word, Syllable) == 3]"))
```

- **Q**: *What is the query to retrieve all phonemes dominating "H" Phonetic items at the beginning of a syllable and occurring in accented ("S") words?*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formatting purposes
query(ae, paste0("[[[Phoneme =~ .* ^ Phonetic == H] ",
                 "^ Start(Word, Syllable) == 1] ^ Accent == S]"))
```

E.1 Differences to the legacy EMU query language

In this section summarizes the major changes concerning the query mechanics of `emuR` compared to the legacy R package `emu` Version 4.2. This section is mainly aimed at users transitioning to `emuR` from the legacy system.

E.1.1 Function call syntax

In `emuR` it is necessary to load an `emuDB` into the current R session before being able to use the `query()` function. This is achieved using the `load_emuDB()` function. This was not necessary using the legacy `emu.query()` function.

E.1.2 Empty result

The `query` function of `emuR` returns an empty segment list (row count is zero) if the query does not match any items. If the legacy EMU function `emu.query()` did not find any matches it, returned an error with the message:

```
## Can't find the query results in emu.query: there may have
## been a problem with the query command.
```

E.1.3 The result modifier hash tag

Compared to the legacy EMU system, which allowed multiple occurrences of the hash tag `#` to be present in a query string, the `query()` function only allows a single result modifier. This ensures that only consistent result sets are returned (i.e., all items belong to a single level). However, if multiple result sets in one segment list are desired, this can easily be achieved by concatenating the result sets of separate queries using the `rbind()` function.

E.1.4 Interpretation of the hash tag # in conjunction operator queries

legacy EMU

```
emu.query(template = "andosl",
          pattern = "*",
          query = "[Text=spring & #Accent=S]")}
```

yielded:

```
## moving data from Tcl to R
## Read 1 records
## segment  list from database:  andosl
## query was:  [Text=spring & #Accent=S]
##   labels    start      end     utts
## 1 spring 2288.959 2704.466 msajc094
```

and

```
emu.query(template = "andosl",
          pattern = "*",
          query = "[#Text=spring & #Accent=S]")
```

yielded the identical:

```
## moving data from Tcl to R
## Read 1 records
## segment  list from database:  andosl
## query was:  [#Text=spring & #Accent=S]
##   labels    start      end     utts
## 1 spring 2288.959 2704.466 msajc094
```

Hence, the hash tag **#** had no effect.

emuR

```
query(emuDBhandle = andosl,
      query = "[Text == spring & #Accent == S]",
      resultType = "emusegs")
```

```
## segment  list from database:  andosl
## query was:  [Text=spring & #Accent=S]
##   labels    start      end          utts
## 1      S 2288.975 2704.475 0000:msajc094
```

Returns the same item but with the label of the hashed attribute definition name. The second legacy example is not a valid `emuR` query (two hash tags) and will return an error message.

```
query(dbName = "andosl",
      query = "[#Text == spring & #Accent == S]")
```

```
## Error in query.database.eql.KONJA(dbConfig, qTrim) :
## Only one hashtag allowed in linear query term: #Text=spring & #Accent=S
```

E.1.5 Bugs in legacy EMU function `emu.query()`

Alternative labels in inequality queries

Example:

legacy EMU

It appears that the OR operator | was mistakenly ignored when used in conjunction with the inequality operator `!=`:

```
emu.query(template = "ae",
          pattern = "*",
          query = "[Text != beautiful | futile ^ Phoneme = u:]")
```

yielded:

```
## moving data from Tcl to R
## Read 4 records
## segment  list from database:  ae
## query was:  [Text!=beautiful|futile ^ Phoneme=u:]
```

```
##      labels    start      end     utts
## 1       new  475.802  666.743 msajc057
## 2    futile  571.999 1091.000 msajc010
## 3        to 1091.000 1222.389 msajc010
## 4 beautiful 2033.739 2604.489 msajc003
```

emuR

The query engine of the `emuR` package respects the presence of the OR operator in such queries:

```
query(emuDBhandle = ae,
      query = "[Text != beautiful | futile ^ Phoneme == u:]",
      resultType = "emusegs")
```

```
## segment  list from database:  ae
## query was:  [Text!=beautiful|futile ^ Phoneme=u:]
##   labels    start      end          utts
## 1     to 1091.025 1222.375 0000:msajc010
## 2    new  475.825  666.725 0000:msajc057
```

Errors caused by missing or superfluous blanks or parentheses

Some queries in the legacy EMU system required blanks around certain operators to be present or absent as well as parentheses to be present or absent. If this was not the case the legacy query engine sometimes returned cryptic errors, sometimes crashing the current R session. The query engine of the `emuR` package is much more robust against missing or superfluous blanks or parentheses.

Order of result segment list

To our knowledge, the order of a segment list in the legacy EMU system was never predictable or explicitly defined. In the new system, if the result type of the `query()` function is set to `"emuRsegs"` the resulting list is ordered by UUID, session, bundle and sample start position. If the parameter `calcTimes` is set to `FALSE` it is ordered by UUID, session, bundle, level, seq_idx. If it is set to `"emusegs"` the resulting list is ordered by the fields utts and start.

Additional features

- The query mechanics of `emuR` accepts the double equal character string `==` (recommended) as well as the single `=` equal character string as an equal operator.
- The EQL2 is capable of querying labels by matching regular expressions using the `=~` (matching) and `!~` (non-matching) operators.
    - For example: `query("andosl", "Text =~ .*tz.*")`

Bibliography

Abercombie, D. (1967). *Elements of general phonetics*. Aldine Pub. Company.

Beckman, M. E. and Ayers, G. (1997). Guidelines for ToBI labelling. *The OSU Research Foundation*, 3.

Bird, S. and Liberman, M. (2001). A formal framework for linguistic annotation. *Speech communication*, 33(1):23–60.

Boersma, P. and Weenink, D. (2016). Praat: doing phonetics by computer (Version 6.0.19). `http://www.fon.hum.uva.nl/praat/`.

Bombien, L. (2011). *Segmental and prosodic aspects in the production of consonant clusters: On the goodness of clusters*. PhD thesis, München, Univ., Diss., 2011.

Bombien, L., Cassidy, S., Harrington, J., John, T., and Palethorpe, S. (2006). Recent developments in the Emu speech database system. In *Proc. 11th SST Conference Auckland*, pages 313–316.

Cassidy, S. (2013). The Emu Speech Database System Manual: Chapter 9. Simple Signal File Format. `http://emu.sourceforge.net/manual/chap.ssff.html`.

Cassidy, S. and Harrington, J. (1996). Emu: An enhanced hierarchical speech data management system. In *Proceedings of the Sixth Australian International Conference on Speech Science and Technology*, pages 361–366.

Cassidy, S. and Harrington, J. (2001). Multi-level annotation in the Emu speech database management system. *Speech Communication*, 33(1):61–77.

Coleman, J. and Local, J. (1991). The "no crossing constraint" in autosegmental phonology. *Linguistics and Philosophy*, 14(3):295–338.

Conway, J., Eddelbuettel, D., Nishiyama, T., Prayaga, S. K., and Tiffin, N. (2016). *RPostgreSQL: R interface to the PostgreSQL database system*. R package version 0.4-1 package version 0.4-1.

Draxler, C. and Jänsch, K. (2004). SpeechRecorder - a Universal Platform Independent Multi-Channel Audio Recording Software. In *Proc. of the IV. International Conference on Language Resources and Evaluation*, pages 559–562, Lisbon, Portugal.

Fromont, R. and Hay, J. (2012). LaBB-CAT: An annotation store. In *Australasian Language Technology Association Workshop 2012*, volume 113. Citeseer.

Garshol, L. M. (2003). BNF and EBNF: What are they and how do they work. *acedida pela última vez em*, 16.

Google (2014). AngularJS. `http://angularjs.org/`.

Harrington, J. (2010). *Phonetic analysis of speech corpora*. John Wiley & Sons.

Harrington, J. and Cassidy, S. (2002). The emu-query language (anhang).

Harrington, J., Cassidy, S., Fletcher, J., and Mc Veigh, A. (1993). The mu+ system for corpus based speech research. *Computer Speech & Language*, 7(4):305–331.

Hipp, D. R. and Kennedy, D. (2007). Sqlite. `https://www.sqlite.org/`.

Ide, N. and Romary, L. (2004). International standard for a linguistic annotation framework. *Natural language engineering*, 10(3-4):211–225.

ISO (2012). Language resource management — Linguistic annotation framework (laf). ISO 24612:2012, International Organization for Standardization, Geneva, Switzerland.

John, T. (2012). *Emu speech database system*. PhD thesis, Ludwig Maximilian University of Munich.

Kisler, T., Schiel, F., Reichel, U. D., and Draxler, C. (2015). Phonetic/linguistic web services at bas. ISCA.

Kisler, T., Schiel, F., and Sloetjes, H. (2012). Signal processing via web services: the use case WebMAUS. In *Proceedings Digital Humanities 2012, Hamburg, Germany*, pages 30–34, Hamburg.

Knuth, D. E. (1968). The Art of Computer Programming Vol. 1, Fundamental Algorithms. *Addison-Wesley, Reading, MA*, 9:364–369.

McAuliffe, M. and Sonderegger, M. (2016). Speech Corpus Tools (SCT). `http://speech-corpus-tools.readthedocs.io/`.

Ooms, J. (2014). The jsonlite package: A practical and consistent mapping between json data and r objects. *arXiv:1403.2805 [stat.CO]*.

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

R Special Interest Group on Databases (R-SIG-DB), Wickham, H., and Müller, K. (2016). *DBI: R Database Interface*. R package version 0.4.

Reichel, U. D., Kleber, F., and Winkelmann, R. (2009). Modelling similarity perception of intonation. In *Proc. 10th Interspeech*, pages 1711–1714, Brighton.

Reichel, U. D. and Winkelmann, R. (2010). Removing micromelody from fundamental frequency contours. In *Proc. 5th Speech Prosody Conference*, Chicago. 100923:1-4.

Rivest, R. (1992). The md5 message-digest algorithm. `https://tools.ietf.org/html/rfc1321`.

Rose, Y., MacWhinney, B., Byrne, R., Hedlund, G., Maddocks, K., O'Brien, P., and Wareham, T. (2006). Introducing phon: A software solution for the study of phonological acquisition. In *Proceedings of the... Annual Boston University Conference on Language Development. Boston University Conference on Language Development*, volume 2006, page 489. NIH Public Access.

RStudio and Inc. (2015). *httpuv: HTTP and WebSocket Server Library*. R package version 1.3.3.

Shue, Y.-L., P., K., C., V., and K., Y. (2011). VoiceSauce: A program for voice analysis. In *Proceedings of the ICPhS*, volume XVII, pages 1846–1849.

Wickham, H., James, D. A., and Falcon, S. (2014). *RSQLite: SQLite Interface for R*. R package version 1.0.0.

Winkelmann, R. (2015). Managing speech databases with emur and the emu-webapp. In *Proceedings of the Sixteenth Annual Conference of the International Speech Communication Association*, volume 1, pages 2611–2612.

Winkelmann, R., Harrington, J., and Jänsch, K. (2017). EMU-SDMS: Advanced speech database management and analysis in R. *Computer Speech & Language*, pages –.

Winkelmann, R. and Jänsch, K. (2015). The new R library for the Emu Speech Database System. `https://github.com/IPS-LMU/emuR`.

Winkelmann, R. and Raess, G. (2014). Introducing a Web Application for Labeling, Visualizing Speech and Correcting Derived Speech Signals. In Calzolari (Conference Chair), N., Choukri, K., Declerck, T., Loftsson, H., Maegaard, B., Mariani, J., Moreno, A., Odijk, J., and Piperidis, S., editors, *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, Reykjavik, Iceland. European Language Resources Association (ELRA).

Winkelmann, R. and Raess, G. (2015). EMU-webApp. `http://ips-lmu.github.io/EMU-webApp/`.

Wittenburg, P., Brugman, H., Russel, A., Klassmann, A., and Sloetjes, H. (2006). Elan: a professional framework for multimodality research. In *Proceedings of LREC*, volume 2006.

Zipser, F. and Romary, L. (2010). A model oriented approach to the mapping of annotation formats using standards. In *Workshop on Language Resource and Language Technology Standards, LREC 2010*, La Valette, Malta.