9. Working with several databases

For purposes of record linkage in particular, i.e. so that you can integrate information held in several independent databases, $\kappa\lambda\epsilon\iota\omega$ also allows you to work with several databases simultaneously.

Three problems must be surmounted in order to do this effectively:

- The system must be able to identify the parts of two databases which you want to link together. This section describes the additional conventions available in the `query` and `write` commands which allow it to do so, and which go far beyond anything you can do with these commands in any one single database.
- You need to connect the items of information in separate databases together. You do so by setting up a link of the `relation` data type, defined in the following section (section 10). The current version of $\kappa\lambda\epsilon\iota\omega$ is *not* yet capable of transferring information physically from one database to another.
- The user must also be able to instruct the system that in addition to information items held in one database, it should use information items in another database which have been explicitly linked to the former. Such links are set up using the network transitions described in section 8.1.1.6.3.

9.1 Paths when you are using several databases simultaneously

Everything we have said so far about the use of paths in tasks which refer to *a single* database also applies to tasks which refer to several databases.

9.1.1 Based paths

You can explicitly relate *any* path to a particular database by prefixing it with a *basis*. Since this particular option is redundant in tasks which only refer to one database, it is normally omitted from them. The formal definition of a *based path* looks like this:
$<\textit{data signal character } 6><\textit{name of database}><\textit{data signal character } 7><\textit{path}>$

If you are using several databases in one task, the following rule applies: *any non-based path refers to the last database to be addressed.* This preserves the rules regarding the relativity of paths. In an analysis command, for example, the first path always refers to the database addressed by the element which terminated the first path after the `query` command's `part=` parameter.

In practice, it is certainly very much easier – in tasks which refer to more than one database – to base *all* paths rather than attempt to remember which rules are currently active.

Every database name used in a based path must first appear in the `query` command's `name=` or `also=` parameters (cf. section 9.2.1.1).

9.1.2 Managing logical objects

If you use logical objects in tasks which refer to more than one database, you can – and should – *base* their names as well; i.e. give their names as follows:

$$< data\ signal\ character\ 6 >< name\ of\ data\ base >< data\ signal\ character\ 7 >$$
$$< name\ of\ definition >$$

9.1.2.1 Restrictions in practice

At present, $\kappa\lambda\epsilon\iota\omega$ transforms logical objects addressed in both databases' permanent environments into *one single* local environment. The user is currently responsible for ensuring that logical objects with the same name have been defined in the same way in both databases. If logical objects with the same name are used in both databases, the system automatically uses the first object to be addressed in the course of the task.

9.2 Commands which refer to several databases at the same time

Currently, you can use the `query` and `write` commands to process several files simultaneously.

9.2.1 The `query` command

The `query` command is used to associate files with one another. For this purpose, one database is processed in the same way as in any other task. In addition, however, for every entry in the first database selected by the appropriate `part=` parameter, the system attempts to find all those entries in the second database which may refer to the information mentioned in the first database. These two entries in both databases are described as *linking entries*.

All the entries in *database 2* which could refer to one and the same entry in *database 1* are described as this entry's *domain* in *database 2*.

A `query` command relating to two databases starts by searching in order for the entry which is being addressed in *database 1*, and then searches in order for all entries in its domain in *database 2*. Every time it finds one of these entries, the system starts processing the evaluating commands (`write` etc.) in the given task.

In order to perform this operation:

- the `query` command must specify which databases are to be linked together *before* the `part=` parameter, and
- you must specify which path in *database 2* should form the domain for the path in *database 1*.

9.2.1.1 The `also=` parameter

`query` commands referring to two databases require an additional parameter, which has the general form:

ALSO=<name of database> (MUST be specified!)

This parameter specifies the database in which the system should search for the domain of the entries in the database specified in the `name=` parameter.

9.2.1.2 Setting up relationships between two databases

9.2.1.2.1 Setting up one database's domain in another

Paths in `query` tasks which are supposed to set up a linkage between two databases *must always* start with the built-in function `catalogue[]` (cf. section 8.1.1.2.2.3).

- The first argument in this built-in function represents a catalogue from the database specified in the `also=` parameter. If no `also=` parameter has been specified, the same construction can be used to create a "linkage of the database to itself".
- The second argument selects the access path in the traditional way.
- The third argument describes an access path in the database specified in the `name=` parameter. In this case, the catalogue entries for the individual entries in the database specified in `name=` form these entries' domain in the database specified in `also=`.

9.2.1.2.2 Limitations of the chosen domain

Once the `catalogue[]` function has been invoked as specified, you can use conditions (which can be as complex as you like) for the same construction as in `query` commands, i.e relating to individual databases.

The `catalogue[]` function corresponds to a number of entries in the database specified in `also=`.

After `negate` and `confirm`, you can set additional conditions which can be as complex as you like.

9.2.1.3 Example

For example, a formally correct `query` command for linking together two databases would look like this:

```
query name=linkage2;also=linkage1;
      part=catalogue[<linkage1>person2,algorithm,<linkage2>:name]
      :firstname=<linkage2>:firstname
```

9.2.2 The `write` command

In tasks relating to two databases, the `write` command has an additional, optional parameter which has the general form:

ONLY=No | Yes (default: No)

9.2.2.1 Alternating data output from several databases

Following a `write` command's `part` parameter in a task relating to more than one database, you can specify alternating paths relating to both databases in any order you choose.

By default, the system processes the *entire* path for every entry in the domain of all entries found for the file specified in `name=`.

9.2.2.1.1 The `only=` parameter

This parameter expects one of the keywords `yes` or `no` as a parameter value.

- `no` causes the system to process the entire path specified in `part` for every entry in the domain of all entries found for the database specified in `name=`, i.e. to behave as it would by default.
- `yes` means that the system only outputs the paths relating to the database specified in `name=` at the beginning of the information covering each domain of this kind, whereas for every entry in the domain it only processes the paths found in the database specified in `also=`.

9.2.2.2 Example

For example, a formally correct `write` command relating to two databases would look like this:

```
write part=<linkage2>:each[],<linkage1>:each[]
```