ReMoDeL

Database and Query Model

White Paper

Version: 0.5

Date: 24 May 2010

Anthony J H Simons, Ahmad F Subahi and Meghna Ram
Department of Computer Science
University of Sheffield

Introduction:

This working paper describes the developing specification for *ReMoDeL DBQ*, a high-level database design language. It serves to outline some suggested syntax and its possible uses in data design, query design, data transformation and query optimisation. Like other modelling languages in the *ReMoDeL* stable, *DBQ* directly encodes the abstract syntax tree using a domain-specific dialect of *XML*. However, this document focuses as much on the processes that will manipulate *DBQ*, as on the language itself, as a way of motivating the different suggested constructions. These are therefore introduced in a bottom-up fashion, for ease of comprehension.

Basic Data Types:

The atomic data type in DBQ is called a *Basic* type. We assume that a number of basic types could exist, even types which are considered structured types in other contexts, like *Date, Time* or *Money*. Examples include:

```
<Basic name="Integer" />
<Basic name="Natural" />
<Basic name="String" />
```

Record Types:

The conceptual entities of a database are called *Records*. These may only contain fields of *Basic* data types (otherwise some kind of relationship is required between *Records*).

```
<Record name="Person">
  <Field name="forename" type="String" size="30" />
  <Field name="surname" type="String" size="30" />
  <Field name="gender" type="Character" range="{m, f}" />
  <Field name="age" type="Natural" range="{0-120}" />
</Record>
```

We assume that entities are defined as *Record* types, consisting of *Fields*. Certain fields may express constraints on their value, such as the possible range of admissible values. A range is expressed as a set of permissible values and may contain enumerations, a single subrange, or disjoint subranges of contiguous elements. The *String* type may specify a maximum size for efficient storage in a database.

```
<Record name="Student">
  <Field name="number" type="Natural" key="total" />
  <Field name="level" type="Natural" range="{1-4}"/>
</Record>

<Record name="Degree">
  <Field name="code" type="String" size="8" key="total" />
  <Field name="name" type="String" size="30" />
</Record>
```

```
<Record name="Module">
  <Field name="code" type="String" size="8" key="total" />
  <Field name="name" type="String" size="30" />
  <Field name="credits" type="Natural"
           range="{5, 10, 15, 20, 30, 40, 60}" />
</Record>
```

Certain fields may be marked as *key* fields. A *key* with the value *total* indicates a unique primary key. Several *key*s with the value *partial* indicate a compound key. A key with the value *auto* indicates an automatically-generated primary key (see below).

Association Types:

We assume both *binary* and higher-*arity* relationships, involving multiple entities as their end-roles (although the problem of data normalisation is significantly simpler if the conceptual model contains only binary associations – see below). For example, a *Student* who registers for a *Degree* is a kind of *binary Association* between the *Student* and *Degree* records:

```
<Association name="Register">
  <Role name="student" type="Student" multiple="zeromany" />
  <Role name="degree" type="Degree" multiple="mandatory" />
</Association>
```

One could also choose to define this relationship instead as a *ternary Association* involving the *Department* owning the degree:

```
<Association name="Register">
  <Role name="student" type="Student" multiple="zeromany" />
  <Role name="degree" type="Degree" multiple="mandatory" />
  <Role name="owner" type="Department" multiple="mandatory" />
</Association>
```

or indeed as a *quaternary* (or higher-arity) *Association* involving other partner *Departments* which help to deliver that degree:

```
<Association name="Register">
  <Role name="student" type="Student" multiple="zeromany" />
  <Role name="degree" type="Degree" multiple="mandatory" />
  <Role name="owner" type="Department" multiple="mandatory" />
  <Role name="partner" type="Department" multiple="zeromany" />
</Association>
```

Each *Association* is uniquely named within its data *Schema* (recommended, in case it should later be promoted to a named *Table*) and each end-*Role* records the *type* and *multiple* of related entities. The *names* of the end-*Roles* are often distinct from the names of the related *types* – the above example shows how *owner* and *partner* refer to two different roles played by a *Department*. Otherwise, the default convention is to use the *Record* type name converted to "camel-case".

Associations relate *Records* to each other in the multiples: {*mandatory, optional, zeromany, onemany*}. Transformation rules will operate on this information, to convert some

Associations into foreign key fields (those with *mandatory* at one end and *optional, zeromany* or *onemany* at the other) and promoting other *Associations* to full *Tables* in their own right (those with *optional, zeromany* or *onemany* at both ends). As an example, the following will be promoted, by virtue of the *zeromany-onemany* multiplicity, to a *Table* called *Study*, whose instances each relate exactly one *Student* and one *Module*:

```
<Association name="Study">
  <Role name="student" type="Student" multiple="zeromany" />
  <Role name="module" type="Module" multiple="onemany" />
</Association>
```

This is not the only reason for promoting an *Association* to a *Table*, for example, such a measure is also required if the *Association* has attributes of its own, recording properties of the relationship, rather than of the participating entities:

```
<Association name="Study">
  <Role name="student" type="Student" multiple="zeromany" />
  <Role name="module" type="Module" multiple="onemany" />
  <Field name="marks" type="Natural" range="{0-100}" />
  <Field name="level" type="Natural" range="{1-4}" />
</Association>
```

Here, the *marks* relate to the *Study* relationship (linking a particular *Student* and *Module*) rather than to the *Student*, or *Module* individually. Similarly, the *level* relates to the level at which the *Student* studied the *Module* (assuming that *Modules* may be studied by *Students* at different *levels*, according to which *Degree* they follow).

Associations may be more precisely constrained. An end-*Role* may specify an exact *quantity*, an integer denoting the exact number of participants at that end of the *Association*. Similarly, an end-*Role* may specify a permissible *range* of participants in the style: `range="{low-high}"`. The value of *range* is always a contiguous range in this case. Any rules that operate on multiplicity may infer this from *quantity* or *range* attributes.

Generalisation and Aggregation:

Generalisation and *Aggregation* are special kinds of directed structural relationship, in which it is important to express the direction. A *Generalisation* is a relationship between a subtype and a supertype. For example, to express that a *Student* is a kind of *Person*, we specify that the *Person* record type is the head of the *Generalisation*. This corresponds to the triangular generalisation arrowhead in a UML class diagram:

```
<Generalisation head="Person">
  <Role name="person" type="Person" multiple="mandatory" />
  <Role name="student" type="Student" multiple="optional" />
</Generalisation>
```

Note that the multiples on the end-*Roles* are always *optional-mandatory* in this case, since each *Student* must be a kind of *Person*, but each *Person* need not also relate to a *Student*. This is provided for translations that convert specialised classes into multiple *Tables*, that is, which treat generalisation just like any other kind of association.

It is also possible to specify a group of generalisation relationships together (sometimes known as a *generalisation set* in UML). The following declares that a *Student* is a kind of *Person*; and also that a *Lecturer* is a kind of *Person*. This kind of declaration is equivalent to making several pair-wise declarations between each related subtype and supertype.

```
<Generalisation head="Person" disjoint="true">
  <Role name="person" type="Person" multiple="mandatory" />
  <Role name="student" type="Student" multiple="optional" />
  <Role name="teacher" type="Lecturer" multiple="optional" />
</Generalisation>
```

Furthermore, it is possible to specify whether such a family of specialisations is *disjoint* or *overlapping*. By specifying `disjoint="true"` then every *Person* record will be related exclusively either to a *Student* or to a *Lecturer* record, and not both. A translator may make the decision to merge the *Person* table's fields into the tables for *Student* and *Lecturer* and generate just these two *Tables*, instead of three. However, if `disjoint="false"` (or simply not specified) it is understood that the specialisations are *overlapping* by default. This means that the same *Person* could be both a *Student* and a *Teacher* (e.g. a Teaching Assistant), in which case three *Tables* would be needed to relate a unique *Person* record to each of the *Student* and *Teacher* extension records.

An *Aggregation* is a relationship between a whole and its parts. To express that a *Bicycle* is a composite whole consisting of many parts, we specify that the *Bicycle* record type is the *head* of an *Aggregation* (a type cannot aggregate itself recursively). This corresponds to the diamond-shaped aggregation arrowhead in a UML class diagram:

```
<Aggregation head="Bicycle">
  <Role name="bicycle" type="Bicycle" multiple="mandatory" />
  <Role name="frame" type="Frame" multiple="mandatory" />
  <Role name="fork" type="Fork" multiple="mandatory" />
  <Role name="wheel" type="Wheel" multiple="onemany" />
</Aggregation>
```

This specifies that a *Bicycle* consists of a *Frame*, a *Fork* and some *Wheels* (at least one). The notion is that a *Bicycle* cannot eventually exist, unless it has these parts; however, in this default form of *Aggregation*, the parts are assumed to exist independently of the whole. It is possible to specify the numbers of participants in an *Aggregation* more precisely, using the *quantity* or *range* attribute, instead of the *multiple* attribute. The following states that a *Bicycle* contains one *Frame*, one *Fork* and exactly two *Wheels*:

```
<Aggregation head="Bicycle" composite="true">
  <Role name="bicycle" type="Bicycle" quantity="1" />
  <Role name="frame" type="Frame" quantity="1" />
  <Role name="fork" type="Fork" quantity="1" />
  <Role name="wheel" type="Wheel" quantity="2" />
</Aggregation>
```

Furthermore, it is possible to specify that an aggregate structure is either *composite* or *separate*. By specifying `composite="true"`, this indicates that all of the aggregate's parts are created and deleted together, therefore it would be possible to merge the fields of all the parts: *Frame, Fork* and two instances of *Wheel*, into a *Table* standing for the whole *Bicycle*. Otherwise, if `composite="false"` (or simply not specified), it is understood that the parts

are *separate* by default. They must then be modelled in separate *Tables*, since they may be added to, and removed from, the aggregate structure.

Note that saying `quantity="1"` is equivalent to declaring a *mandatory* multiplicity; also saying `range="{0-1}"` is equivalent to declaring an *optional* multiplicity; also saying `range="{0-n}"` is equivalent to declaring an *zeromany* multiplicity; and finally saying `quantity="n"` or `range="{1-n}"` is equivalent to declaring a *onemany* multiplicity. Rules which expect to operate on multiplicity information may infer this.

Data Tables:

The intention is for DBQ to express both high-level conceptual schemas, and low-level database schemas that can be directly implemented as relational tables. The DBQ language will support translation from the high-level to the low-level view. The entities declared at the lower level are called *Table* types, since they define the shape of rows in a database table. A *Table* may be derived from a *Record* from the conceptual schema:

```
<Table name="Degree">
  <Field name="code" type="String" size="8" key="total" />
  <Field name="name" type="String" size="30" />
</Table>
```

A *Table* may also be derived from an *Association* from the conceptual schema:

```
<Table name="Register">
  <Field name="student.number" type="Natural" key="partial"
      refer="Student" />
  <Field name="degree.code" type="String" key="partial"
      refer="Degree" />
</Table>
```

The normalisation process is described below. We assume that *Records* do not contain repeating groups of data (1NF) and that all non-key attributes in each *Record* depend on the primary key (2NF). Essentially, all *Records*, relationships and *Queries* from the conceptual schema are processed until the normal schema contains only *Table* definitions and normalised *Queries*. The resulting *Schema* is typically in third normal form (3NF) and may achieve 4NF or 5NF (depending on the treatment of higher-arity *Associations*).

Primary and Foreign Keys:

Any *Table* that does not obtain a primary *key* from its original Record (possibly unique, possibly a compound key) must be provided with an automatic primary key field, always named *identity*. This field stores a unique serial number that is incremented for each new instance, when it is first added to the database. The attribute `key="auto"` is set to indicate that the field is an automatic primary key.

The *Person* record above did not originally specify any explicit primary key field, so the corresponding *Person* table must be given one:

```
<Table name="Person">
  <Field name="identity" type="Natural" key="auto" />
```

```
  <Field name="forename" type="String" size="30" />
  <Field name="surname" type="String" size="30" />
  <Field name="gender" type="Character" range="{m, f}" />
  <Field name="age" type="Natural" range="{0-120}" />
</Table>
```

Any binary *Association* that was one-to-one (*mandatory* on both sides) must be eliminated by merging the two related *Records* in a single *Table* to satisfy 3NF (unless the association is *involuted* – see below). The name of the merged *Table* is equal to the name of the more significant *Record* (called *major*), chosen automatically or with help from the designer. The *Field* names of the other (*minor*) *Record* are made unique, by prefixing them with an identifier created from the minor type, to avoid the possibility of name-clashes. For example, if a *Student* is uniquely associated with a *UCard* in a one-to-one association, after merging *UCard* (minor) into *Student* (major), we have:

```
<Table name="Student">
  <Field name="number" type="Natural" key="total" />
  <Field name="level" type="Natural" range="{1-4}"/>
  <Field name="uCard.number" type="Natural" />
  <Field name="uCard.expiry" type="Date" />
</Table>
```

Note how the names of the merged fields are prefixed with "*uCard*" to prevent duplicate occurrences of the *number* field. Although the *UCard's number* was originally marked as a primary key, in the merger this becomes dependent on the *Student's* primary key (suppressing the *key* attribute). If an auto-generated *identity* field existed in the minor *Record*, it would disappear altogether in the merger, since it is no longer needed.

Any binary *Association* that was one-to-many (*mandatory* on one side, *optional, zeromany* or *onemany* on the other) is eliminated in 3NF by inserting the primary key of the mandatory *Record* as a foreign key in the *Table* for the other *Record*. The name of the foreign key *Field* is created by concatenating the end-*Role* name from the *Association* with the *Field*-name in the related *Record*. For example, where many *Students* are related to one *Degree*, the *Student Table* acquires a foreign key, based on the *Degree's* primary key *code*, but renamed *degree.code* in *Student*:

```
<Table name="Student">
  <Field name="number" type="Natural" key="total" />
  <Field name="level" type="Natural" range="{1-4}" />
  <Field name="degree.code" type="String" refer="Degree" />
</Table>
```

Note how the synthesized name is derived from the end-*Role* name, not the type-name (in this example they just happen to be the same), because of the possibility of having several, distinct associations with the same type, with different end-*Role* names. The type of the foreign key *Field* is the same as the type of the related primary key. The field also sets an attribute `refer="Degree"` to identify the related *Table*.

Coordinator (Linker) Types:

Any other kind of binary *Association* must be promoted to a full *Table* in its own right, co-ordinating the participants in the association. The end-*Roles* are converted into named *Fields* storing the primary keys of the participants as foreign keys in the new *Table*. Promoted *Tables* act as co-ordinators for the relationship expressed by the original *Association*. They are known as *linker tables* in a database.

There are three different cases. Firstly, an *Association* which has its own attributes must be promoted to a *Table* containing these attributes as its *Fields*. Secondly, a many-to-many, or many-to-optional *Association* must always be promoted, since there is no way of storing the foreign key fields in either related *Table* type alone. An example is the *Study* association between *Student* and *Module*, which is both many-to-many and has attributes. After normalisation, it is promoted to a *Table* with the same name:

```
<Table name="Study">
  <Field name="student.number" type="Natural" key="partial"
     refer="Student" />
  <Field name="module.code" type="String" key="partial"
     refer="Module" />
  <Field name="marks" type="Natural" range="{0-100}" />
  <Field name="level" type="Natural" range="{1-4}" />
</Table>
```

Whereas the *Roles* originally referred directly to the participating entities, the new *Fields* are *foreign key* fields, storing the corresponding *primary keys* of the related entities. The foreign key fields are named using the concatenation rule described above. Furthermore, the primary key of any such co-ordinating *Table* is always a *compound key*, consisting of the foreign keys describing the related participants (every instance of *Study* uniquely relates a tuple of *Student* and *Module* instances). This is indicated by `key="partial"` in each foreign key field.

Thirdly, any *involuted* binary *Associations* must be promoted. An *involuted* association is one which relates a given *Record* type back to itself. This cannot be handled using foreign keys in the usual way, which would leave null references for some *Records* (or else seem to require a merger, doubling the fields redundantly). For example, if a *Person* record was related to itself by the involuted association *Marry,* relating the (optional) *husband* and *wife* roles, this could not be handled using foreign keys in 3NF, since null references would exist for the unmarried. Instead, promoting the *Marry* association yields the following:

```
<Table name="Marry">
  <Field name="husband.identity" type="Natural" key="partial"
     refer="Person" />
  <Field name="wife.identity" type="Natural" key="partial"
     refer="Person" />
</Table>
```

This completes the normalisation process for binary *Associations*.

Higher-Arity Relationships:

While it might be possible to promote ternary, quaternary or higher-arity *Association*s in a similar way, in general this is not safe and leads to violations of 4NF. This is because the dependencies that exist between all participants in higher-arity associations may or may not

be independent. The safest thing to do is to split these into binary associations first; and this process usually requires human intervention, since it requires some understanding of the meaning of the data.

In general, the same related sets of instances should be reachable in the transformed result as in the original model. For example, the ternary *Association*:

```
<Association name="Register">
  <Role name="student" type="Student" multiple="zeromany" />
  <Role name="degree" type="Degree" multiple="mandatory" />
  <Role name="owner" type="Department" multiple="mandatory" />
</Association>
```

relates multiple *Students* to one *Degree* and one *Department*. It could be split into any two binary *Associations* between pairs of participants, for example, one solution is:

```
<Association name="RegisterFor">
  <Role name="student" type="Student" multiple="zeromany" />
  <Role name="degree" type="Degree" multiple="mandatory" />
</Association>
<Association name="RegisterIn">
  <Role name="student" type="Student" multiple="zeromany" />
  <Role name="owner" type="Department" multiple="mandatory" />
</Association>
```

This solution yields two many-to-one *Associations*. For each *Student*, we can determine both the related *Degree* (via the *RegisterFor* association) and the related *Department* (via the *RegisterIn* association). However, this might not be the best solution in a domain where every *Degree* always belongs to a unique *Department*. In this case, we would do better to arrange the split this way:

```
<Association name="Register">
  <Role name="student" type="Student" multiple="zeromany" />
  <Role name="degree" type="Degree" multiple="mandatory" />
</Association>
<Association name="Administer">
  <Role name="degree" type="Degree" multiple="mandatory" />
  <Role name="owner" type="Department" multiple="mandatory" />
</Association>
```

Now, a *Student* merely has to register for a *Degree*, which determines the related *Department* by a one-to-one mandatory association. This solution is better, in the sense that the act of registering for a degree (which happens more frequently than providing a new degree course) is made less complicated. Pursuing this example further, the *Degree* and *Department* records would eventually have to be merged in 3NF (see above).

The splitting process might have to be handled differently for the example originally given as a quaternary *Association*. Here, the *Department* type is related multiple times, via two different roles named *owner* and *partner*, to the *Student* and *Degree* types:

```
<Association name="Register">
  <Role name="student" type="Student" multiple="zeromany" />
  <Role name="degree" type="Degree" multiple="mandatory" />
  <Role name="owner" type="Department" multiple="mandatory" />
  <Role name="partner" type="Department" multiple="zeromany" />
</Association>
```

If we assume, as above, that *Degrees* are uniquely owned and administered by one *Department*, we could convert the above into three binary *Associations*:

```
<Association name="Register">
  <Role name="student" type="Student" multiple="zeromany" />
  <Role name="degree" type="Degree" multiple="mandatory" />
</Association>
<Association name="Administer">
  <Role name="degree" type="Degree" multiple="mandatory" />
  <Role name="owner" type="Department" multiple="mandatory" />
</Association>
<Association name="Partner">
  <Role name="degree" type="Degree" multiple="mandatory" />
  <Role name="partner" type="Department" multiple="zeromany" />
</Association>
```

This would seem to be the logical extension of the earlier transformation. However, further processing of this example would seem to require merging *Degree* and *Department* in one association, but not in the other! A much better translation, which is only really found by human insight into the domain, is the following:

```
<Association name="Register">
  <Role name="student" type="Student" multiple="zeromany" />
  <Role name="degree" type="Degree" multiple="mandatory" />
</Association>
<Association name="Administer">
  <Field name="owner" type="Boolean" />
  <Role name="degree" type="Degree" multiple="mandatory" />
  <Role name="department" type="Department" multiple="onemany" />
</Association>
```

This captures the relationship with multiple *Departments* more succinctly, and identifies one of the participating *Departments* as the owner of the *Degree* through an attribute, recorded as a *Field* of the *Association*. These examples show how higher-arity associations can be somewhat problematical to eliminate automatically!

Structural Relationships:

Generalisation and *Aggregation* relationships must also be eliminated, when converting into the tabular format of a database. Both kinds of structural relationship can be translated in several ways, depending on the trade-off between expressiveness (achieved by full normalisation) and efficiency (achieved with pre-normal forms).

The usual translation of a (by default, overlapping) *Generalisation* introduces a foreign key field in the *optional* subtype *Record* relating it to the *mandatory* supertype *Record*. This may

use the standard algorithm for creating a foreign key field, based on the end-*Role* name and primary key name in the related type:

```
<Table name="Student">
  <Field name="person.identity" type="Natural" refer="Person" />
  <Field name="number" type="Natural" key="total" />
  <Field name="level" type="Natural" range="{1-4}"/>
</Table>
```

The special translation of a *disjoint Generalisation* eliminates the supertype record altogether and introduces its fields into each of its subtypes separately. This is a pre-normal, or *de-normalised* form, since there is replication of fields in several tables in the data schema, but it saves computing joins between tables. The alternative translation of *Student* is given as:

```
<Table name="Student">
  <Field name="identity" type="Natural" key="auto" />
  <Field name="forename" type="String" size="30" />
  <Field name="surname" type="String" size="30" />
  <Field name="gender" type="Character" range="{m, f}" />
  <Field name="age" type="Natural" range="{0-120}" />
  <Field name="number" type="Natural" />
  <Field name="level" type="Natural" range="{1-4}" />
</Table>
```

This special translation is probably the only one which does not need to rename the fields merged from the supertype in the subtype, since these are distinct in any case. Note how the original primary key for *Person* is retained, despite the fact that this was an auto-generated *identity*, and the *Student number* key has been suppressed. This is for consistency across all the disjoint types, allowing for a common code-generation strategy further down the line.

The usual translation of a (separate) *Aggregation* introduces a foreign key field in every component part *Record* relating it to the whole *Record*. This may use the standard algorithm for creating a foreign key field, based on the end-*Role* name and primary key name in the related type. Taking the *Wheel* component as an example, we have:

```
<Table name="Wheel">
  <Field name="number" type="Natural" key="total" />
  <Field name="manufacturer" type="String" />
  <Field name="diameter" type="Natural" />
  <Field name="bicycle.identity" type="Natural" refer="Bicycle" />
</Table>
```

The same would be done for the components *Frame* and *Fork*. This translation assumes that all the different parts exist independently of the *Bicycle* itself, and so are *optionally* related to the particular whole. The remaining fields of *Wheel* are assumed to have been present in the original *Wheel* record (not elaborated above).

The special translation of a *composite Aggregation* embeds the fields of all of the component-parts *inside* the *Bicycle Table* directly. This may require clever synthesis of field name prefixes, to distinguish when components are replicated several times inside the aggregate structure:

```
<Table name="Bicycle">
  <Field name="identity" type="Natural" key="auto" />
  <Field name="frame.number" type="Natural" ... />
  <Field name="frame.size" type="Natural" ... />
  ...
  <Field name="fork.number" type="Natural" ... />
  ...
  <Field name="wheel[1].number" type="Natural" ... />
  <Field name=" wheel[1].manufacturer" type="String" />
  <Field name=" wheel[1].diameter" type="Natural" />
  <Field name="wheel[2].number" type="Natural" ... />
  <Field name="wheel[2].manufacturer" type="String" />
  <Field name="wheel[2].diameter" type="Natural" />
</Table>
```

This de-normalised translation might be very efficient for reading and writing whole *Bicycles* from the database, although it replicates fields. It still allows unusual combinations, such as different-diameter front and back wheels (chopper-style bicycle), or matching up a pair of wheels from different manufacturers (unusual). Above, we suggest the numbering notation *[n]* to distinguish replicated fields. Alternatively, the user could be asked to rename the fields to more meaningful names such as *frontWheel*, *backWheel*.

Package and Schema:

As with all *ReMoDeL* dialects, the outermost linguistic specification unit is the *Package*. A *Package* is the root element of any XML tree. For DBQ, the kind of *Package* is given by the value of the *model* attribute, and the package may optionally define a *name*-space, or may specify a file system *location* where it is to be stored. A *Package* is a translation unit, that is, *ReMoDeL* transformers and generators should be able to translate a *Package* without needing to load other packages.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Package name="Uni" model="DBQ" location="uni.data.model">
  <Schema name="Registration">
    <!-- other conceptual schema data inserted here -->
  </Schema>
</Package>
```

A *Package* contains one or more data *Schemas* (typically just one). A *Schema* is the metadata for a database, consisting of data and query definitions. Initially, a *Schema* is constructed by considering what entities the database will contain (named *Records*, consisting of named *Fields*) and what relationships will exist between them (*Association*, *Generalisation* or *Aggregation* relationships) and finally what kinds of *Query* will be executed over the data. This kind of *Schema* is called a *conceptual* schema, which is closest to the data model end-users have in their minds.

An alternative kind is a *normal* Schema, which instead consists of normalised data *Tables* and *Queries* that have been optimised to execute over these tables, ready for implementation in a relational database. A normal Schema is distinguished by the *normal* attribute. If `normal="true"`, the *Schema* is normal, otherwise it is conceptual.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Package name="Uni" model="DBQ" location="uni.data.normal">
  <Schema name="Registration" normal="true">
    <!-- other normal schema data inserted here -->
  </Schema>
</Package>
```

To satisfy the requirements of a translation unit, all elements that are imported from outside the current *Schema* must be declared. The following shows the declarations imported by a *Schema*:

```
<Schema name="Registration">
  <Employ type="Integer" basic="true" />
  <Employ type="Natural" basic="true" />
  <Employ type="String" from="DBType" location="data.type" />
  <Record ... />
  <Association ... />
  <!—other records, associations, etc. inserted here -->
</Schema>
```

In this, it is imagined that the elements of the *Schema* either have to be defined locally (as the *Record* and *Association* types here) or imported from other places. *Basic* types may be assumed to exist in any implementation, but some types, such as the *String* type here, are assumed to be imported from other packages, corresponding to libraries.

Query Language:

The format for query expressions must be sufficiently general that these could be translated into a variety of different query models, such as SQL, or DAPLEX or OCL or perhaps even the XML-based tree searching languages XQuery and XPath. This means that a neutral query expression language must support navigation-style queries (like DAPLEX, OCL, XPath) and constraint-matching queries (like SQL, OCL).

Queries will be expressed initially in a *high-level* query language (a conceptual query language), which operates upon the *Records, Associations, Generalisations* and *Aggregations* stored in a conceptual data *Schema*. These will be converted into a *low-level* query language (an optimised query language) that operates upon normalised data tables. The high-level query language may look something like OCL, Daplex or XPath, based on navigating from concept to concept. The low-level transformations of these queries may look more like SQL queries, acting upon primary and foreign key fields of tables.

The query language will need to distinguish between different kinds of *Query*, such as side-effect free *Select* expressions and table-modifying *Insert* and *Update* expressions. A *Select* expression should always denote (be considered to return) a set of records matching some criterion. Similarly, an *Update* expression will use a *Select* statement to specify a set of records to be modified, but specify one or more assignments to their fields. Some kind of expression will be required to relate two sets of records (similar to a *join*). An *Insert* expression will specify a set of new records to be added to the current set maintained by the database – this requires a syntax for expressing record instances, with literal representations of their field values. It is unlikely that the query language will need runtime expressions to

manipulate metadata, corresponding to the *Create* and *Drop Table* instructions in SQL, since any changes to the data *Schema* will result in re-generation of the system.

Path Expressions:

The high-level query language manipulates entities in a data model consisting of collections of *Records*, related to each other mainly by *Associations*. We assume that both concepts will eventually be stored as tables, and that queries may navigate from one table to another and return subsets of *Records*, or project out *Tuples* of values. A key concept is a *Path*, denoting a route navigating from some starting point to an element, or to a set of elements.

All *Path* expressions typically start with a locally-bound variable to denote the starting point. The elements of a *Path* expression are identifiers separated by dots, describing how to navigate to a *Field* or *Role*. For example, if *person* is a variable denoting a *Person* instance, the following paths navigate to its *age* and *surname* fields:

```
<Path name="person.age" type="Natural" />
<Path name="person.surname" type="String" />
```

Likewise, if *study* is a variable denoting an instance of the *Study* association, the following navigate to the *module* and *student* end-*Roles* of that association; or access the *marks* attribute of the association. Note that you can only navigate from an *Association* to its associated *Records*, not the other way around:

```
<Path name="study.module" type="Module" />
<Path name="study.student" type="Student" />
<Path name="study.marks" type="Natural" />
```

Paths may be longer expressions, navigating to attributes of the related *Records*. This is expressive in the high-level query language, but may correspond to table joins in the low-level query language translation. In the following, the first two *Paths* require a join across tables, but the last may not, since the result is a foreign key field:

```
<Path name="study.module.name" type="String" />
<Path name="study.student.surname" type="String" />
<Path name="study.module.code" type="String" />
```

The expressible limit for a *Path* expression is any *Path* whose result is a set of values. No further navigation can be applied to a set (only searching may be performed).

A further kind of *Path* expression refers to complete datasets. By convention, the database is accessed via a global variable called *data*; and each of its datasets is accessed using the name of the stored type (converted to "camel case"). For example, the following *Path* expressions refers to the set of all *Student* records, and *Study* associations, in the database:

```
<Path name="data.student" type="Set[Student]" />
<Path name="data.study" type="Set[Study]" />
```

Select Query:

The query language is based upon very few fundamental functions. These include the *Select* query, which maps a predicate over a set, returning a subset of records which pass the predicate test. This is similar to a filter-function in a functional algebra, which accepts a predicate and a list as arguments, returning a list as the result. The following are some examples.

Select 1: uses a trivial predicate to return everything. Note the structure of the *Function* element, which consists of a formal argument and a body-expression (this is the return value of the function).

```
<Select label="all persons" type="Set[Person]">
  <Function>
    <Variable name="person" type="Person" />
    <Literal value="true" type="Boolean" />
  </Function>
  <Path name="data.person" type="Set[Person]" />
</Select>
```

Select 2: uses a more substantial predicate to return a subset of records.

```
<Select label="all persons under 50" type="Set[Person]">
  <Function>
    <Variable name="person" type="Person" />
    <Operator symbol="lessThan">
      <Path name="person.age" type="Natural" />
      <Literal name="50" type="Natural" />
    </Operator>
  </Function>
  <Path name="data.person" type="Set[Person]" />
</Select>
```

Select 3: returns a singleton set for a uniquely-identified record. All *Select* statements return a set of records, which must be anticipated in the rest of the algebra.

```
<Select label="a given student" type="Set[Student]">
  <Function>
    <Variable name="student" type="Student" />
    <Operator symbol="equals">
      <Path name="student.number" type="Natural" />
      <Literal value="27639421" type="Natural" />
    </Operator>
  </Function>
  <Path name="data.student" type="Set[Student]" />
</Select>
```

Project Query:

The second fundamental function in the query language is the *Project* function, because it projects out a set of (reachable, or computable) values from a set of records. This corresponds to a map-function in functional algebra, which accepts a function and a list as arguments, and returns a list, consisting of the results of mapping the function over every element in the input

list. *Project* is often used in combination with *Select* to return data of interest after a search. The following are some examples:

Project 1: projects out one field from each record in a set of records. Note how the single body expression is understood to return the value defined by the path expression. Also, the *Project* function is here a true projection, that returns a set, rather than a list, of values:

```
<Project label="age of all persons" type="Set[Natural]">
  <Function>
    <Variable name="person" type="Person" />
    <Path name="person.age" type="Natural" />
  </Function>
  <Path name="data.person" type="Set[Person]" />
</Project>
```

Project 2: projects out two fields from a set of records. Note that tuple-types are defined using comma-separated types; and the *tuple* operator constructs the result.

```
<Project label="age of all persons" type="Set[Natural, String]">
  <Function>
    <Variable name="person" type="Person" />
    <Operator name="tuple">
      <Path name="person.age" type="Natural" />
      <Path name="person.surname" type="String" />
    </Operator>
  </Function>
  <Path name="data.person" type="Set[Person]" />
</Project>
```

Project 3: projects out one field from a selection of a subset of records. This is where the power of *Project* and *Select* together are shown. Instead of supplying the whole dataset as input, a filtered set is provided. This also shows the importance of indicating the result-type of each query operation, since it may be nested inside other queries.

```
<Project label="age of persons under 50" type="Set[Natural]">
  <Function>
    <Variable name="person" type="Person" />
    <Path name="person.age" type="Natural" />
  </Function>
  <Select label="all persons under 50" type="Set[Person]">
    <Function>
      <Variable name="person" type="Person" />
      <Operator symbol="lessThan">
        <Path name="person.age" type="Natural" />
        <Literal name="50" type="Natural" />
      </Operator>
    </Function>
    <Path name="data.person" type="Set[Person]" />
  </Select>
</Project>
```

Relate Query:

The third fundamental function is *Relate*. This maps from one dataset to its image in the other dataset. In functional algebra, this is like a double-filter function, which accepts two lists, and returns all instances in the first list, for which the predicate relates it to any instances in the second list. Note how the predicate is a function of two arguments, one from each list. Relate serves a similar purpose to the traditional relational join.

Relate 1: returns a matching set of linker records for a given set of records.

```
<Relate label="study of a given student" type="Set[Study]">
  <Function>
    <Variable name="study" type="Study" />
    <Variable name="student" type="Student" />
    <Operator symbol="and">
      <Operator symbol="equals">
        <Path name="student.number" type="Natural" />
        <Literal value="27639421" type="Natural" />
      </Operator>
      <Operator symbol="equals">
        <Path name="study.student" type="Student" />
        <Path name="student" type="Student" />
      </Operator>
    </Operator>
  </Function>
  <Path label="data.study" type="Set[Study]" />
  <Path name="data.student" type="Set[Student]" />
</Relate>
```

Relate 2: optimises the above search, by filtering the second set, before passing this to the Relate function. The matching predicate is split over the two searches.

```
<Relate label="study of a given student" type="Set[Study]">
  <Function>
    <Variable name="study" type="Study" />
    <Variable name="student" type="Student" />
    <Operator symbol="equals">
      <Path name="study.student" type="Student" />
      <Path name="student" type="Student" />
    </Operator>
  </Function>
  <Path label="data.study" type="Set[Study]" />
  <Select label="a given student" type="Set[Student]">
    <Function>
      <Variable name="student" type="Student" />
      <Operator symbol="equals">
        <Path name="student.number" type="Natural" />
        <Literal value="27639421" type="Natural" />
      </Operator>
    </Function>
    <Path name="data.student" type="Set[Student]" />
  </Select>
</Relate>
```

The above *Relate*-expressions search through both the *Student* and *Study* datasets. The first example has *n-squared* complexity; the second varies between this and linear, depending on the size of the pre-filtered dataset. Sometimes, it may be possible to eliminate the double search altogether. The above example can be converted into a *Select*, which has linear complexity. The join operation is avoided, because the related attribute in the *Student* dataset is stored as a foreign key in the *Study* dataset.

Select 4: sometimes an alternative to *Relate*, returns a matching set of linkers for a related record. This does not need to search the *Student* dataset, since *student.number* is directly accessible as a foreign key in the *Study* dataset.

```
<Select label="study of a given student" type="Set[Study]">
  <Function>
    <Variable name="study" type="Study" />
    <Operator symbol="equals">
      <Path name="study.student.number" type="Natural" />
      <Literal value="27639421" type="Natural" />
    </Operator>
  </Function>
  <Path name="data.study" type="Set[Study]" />
</Select>
```

Relate 3: returns a matching set of linker records for a given set of records. This version must use the *Relate* function, because the compared values are not foreign keys.

```
<Relate label="study of a named student" type="Set[Study]">
  <Function>
    <Variable name="study" type="Study" />
    <Variable name="student" type="Student" />
    <Operator symbol="and">
      <Operator symbol="equals">
        <Path name="student.surname" type="String" />
        <Literal value="Smith" type="String" />
      </Operator>
      <Operator symbol="equals">
        <Path name="study.student" type="Student" />
        <Path name="student" type="Student" />
      </Operator>
    </Operator>
  </Function>
  <Path label="data.study" type="Set[Study]" />
  <Path name="data.student" type="Set[Student]" />
</Relate>
```

Once more, this is the least efficient form of the query, since it searches the entire *Student* dataset for each instance of the *Study* dataset (the complexity is of the order *n-squared*), using a combined predicate inside *Relate*. It could be made more efficient by pre-filtering the *Student* dataset with a query on the *student.surname*, then passing this much smaller set as the second set-argument to *Relate*.

The above examples assume that it is possible to compare record-references for equality (the *Role study.student* is tested for equality with a locally-bound *student* variable). In practice,

this equality might be tested using the foreign key field in the *Study* record and primary key field in the *Student* record.

Relate 4: equivalent *Relate* search, using primary and foreign keys, instead of direct comparison of *Record* references. This might be how searches are performed in the low-level query language.

```
<Relate label="study of a given student" type="Set[Study]">
  <Function>
    <Variable name="study" type="Study" />
    <Variable name="student" type="Student" />
    <Operator symbol="and">
      <Operator symbol="equals">
        <Path name="student.surname" type="String" />
        <Literal value="Smith" type="String" />
      </Operator>
      <Operator symbol="equals">
        <Path name="study.student.number" type="Natural" />
        <Path name="student.number" type=" Natural" />
      </Operator>
    </Operator>
  </Function>
  <Path label="data.study" type="Set[Study]" />
  <Path name="data.student" type="Set[Student]" />
</Relate>
```

Relate 5: optimises the above search slightly by searching for a filtered subset of *Student* records, before passing this as the second argument to *Relate*, returning a matching set of *Study* records. Note how the predicate is split over the two searches.

```
<Relate label="study of a given student" type="Set[Study]">
  <Function>
    <Variable name="study" type="Study" />
    <Variable name="student" type="Student" />
    <Operator symbol="equals">
      <Path name="study.student.number" type="Natural" />
      <Path name="student.number" type="Natural" />
    </Operator>
  </Function>
  <Path label="data.study" type="Set[Study]" />
  <Select label="a given student" type="Set[Student]">
    <Function>
      <Variable name="student" type="Student" />
      <Operator symbol="equals">
        <Path name="student.surname" type="String" />
        <Literal value="Smith" type="String" />
      </Operator>
    </Function>
    <Path name="data.student" type="Set[Student]" />
  </Select>
</Relate>
```

Combined Queries:

Project 4: in combination with *Relate* and *Select*, returns a set of records from one dataset that are related to records in a different dataset.

```
<Project label="modules of a given student" type="Set[Module]">
  <Function>
    <Variable name="study" type="Study" />
    <Path name="study.module" type="Module" />
  </Function>
  <Relate label="study of a given student" type="Set[Study]">
    <Function>
      <Variable name="study" type="Study" />
      <Variable name="student" type="Student" />
      <Operator symbol="equals">
        <Path name="study.student" type="Student" />
        <Path name="student" type="Student" />
      </Operator>
    </Function>
    <Path label="data.study" type="Set[Study]" />
    <Select label="a given student" type="Set[Student]">
      <Function>
        <Variable name="student" type="Student" />
        <Operator symbol="equals">
          <Path name="student.number" type="Natural" />
          <Literal value="27639421" type="Natural" />
        </Operator>
      </Function>
      <Path name="data.student" type="Set[Student]" />
    </Select>
  </Relate>
</Project>
```

Project 5: is the most efficient transformation of the above query, using the technique from the example *Select* 4 to eliminate the *Relate* expression altogether.

```
<Project label="modules of a given student" type="Set[Module]">
  <Function>
    <Variable name="study" type="Study" />
    <Path name="study.module" type="Module" />
  </Function>
  <Select label="study of a given student" type="Set[Study]">
    <Function>
      <Variable name="study" type="Study" />
      <Operator symbol="equals">
        <Path name="study.student.number" type="Natural" />
        <Literal value="27639421" type="Natural" />
      </Operator>
    </Function>
    <Path name="data.study" type="Set[Study]" />
  </Select>
</Project>
```

Project 6: accesses the attributes of an association. This is just as simple as other searches, and reinforces the case for allowing searches to start from *Association* types.

```
<Project label="module code and marks" type="Set[String, Natural]">
  <Function>
    <Variable name="study" type="Study" />
    <Operator name="tuple">
      <Path name="study.module.code" type="String" />
      <Path name="study.mark" type="Natural" />
    </Operator>
  </Function>
  <Select label="study of a given student" type="Set[Study]">
    <Function>
      <Variable name="study" type="Study" />
      <Operator symbol="equals">
        <Path name="study.student.number" type="Natural" />
        <Literal value="27639421" type="Natural" />
      </Operator>
    </Function>
    <Path name="data.study" type="Set[Study]" />
  </Select>
</Project>
```

Update Query:

The *Update* function performs an update operation on every instance of a dataset. Here, we find it most simple to style this as a kind of assignment operation, although in functional algebra, this can also be thought of as a kind of map from one set of records to another set, in which the desired changes have been effected by the map-function equivalent to the assignment. *Update* is typically applied together with *Select*. The following are some examples of usage:

Update 1: initialises a set of records. It is designed like *Project*, in that it applies a function to every instance of a dataset, and returns the modified dataset.

```
<Update label="set level of all students to 1" type="Natural">
  <Function>
    <Variable name="student" type="Student" />
    <Operator symbol="assign">
      <Path name="student.level" type="Natural" />
      <Literal value="1" type="Natural" />
    </Operator>
  </Function>
  <Path name="data.student" type="Set[Student]" />
</Project>
```

Note: the *Natural* return type indicates that the *Update* function is assumed to return a count of the number of records that were updated. (The assign operator can be assumed to return the value that was assigned, but this value is not used by *Update* in any way).

Update 2: applies an update to a subset of records. This shows the most common circumstance, in which *Update* is applied to the result of *Select*:

```
<Update label="set level of a given student to 2" type="Natural">
  <Function>
    <Variable name="student" type="Student" />
    <Operator symbol="assign">
      <Path name="student.level" type="Natural" />
      <Literal value="2" type="Natural" />
    </Operator>
  </Function>
  <Select label="a given student" type="Set[Student]">
    <Function>
      <Variable name="student" type="Student" />
      <Operator symbol="equals">
        <Path name="student.number" type="Natural" />
        <Literal value="27639421" type="Natural" />
      </Operator>
    </Function>
    <Path name="data.student" type="Set[Student]" />
  </Select>
</Project>
```

These handle all the necessary combinations of selections, joins, projections and updates from a database.