# Relational Implementation of EMdF and MQL

Ulrik Petersen

April 27, 2013

Copyright (C) 2001-2007 Ulrik Petersen
This document is made available under the Creative Commons Attribution-Sharealike license version 2.5.

See

http://creativecommons.org/licenses/by-sa/2.5/

for what that means.

Please visit the Emdros website for the latest news and downloads:

http://emdros.org

**Abstract**

In this report, I document some of my ideas on implementing the EMdF model in an RDBMS. The emphasis is on showing how the data domains of the EMdF model can be implemented in tables, using SQL2. It documents Emdros version 1.2.0.pre208 and above.

In chapter 1, I give some preliminaries, including conventions used in this document. In chapter 2, I show how to implement the meta-data of the EMdF model in an RDBMS. In chapter 3, I show how to implement the objects in the EMdF model. In chapter 4, I show the way in which all of the commands of the full MQL access language translate into SQL statements.

# Contents


**1 Preliminaries** **6**
1.1 Introduction . . . 6
1.2 Assumptions on the implementation . . . 6
1.2.1 The three sequences of ids . . . 6
1.2.2 All names are stored as lower-case . . . 6
1.3 Conventions used . . . 7

**2 Meta-data** **8**
2.1 Introduction . . . 8
2.2 schema_version . . . 9
2.2.1 SQL template . . . 9
2.2.2 Explanation . . . 9
2.2.3 Example . . . 9
2.3 sequence_0, sequence_1, and sequence_2 . . . 9
2.3.1 SQL template . . . 9
2.3.2 Explanation . . . 10
2.3.3 Example . . . 10
2.4 enumerations . . . 10
2.4.1 SQL template . . . 10
2.4.2 Explanation . . . 10
2.4.3 Example . . . 11
2.5 enumeration_constants . . . 11
2.5.1 SQL template . . . 11
2.5.2 Explanation . . . 11
2.5.3 Example . . . 13
2.6 object_types . . . 13
2.6.1 SQL template . . . 13
2.6.2 Explanation . . . 13
2.6.3 Example . . . 14
2.7 normalized_object_type_names . . . 14
2.7.1 SQL template . . . 14
2.7.2 Explanation . . . 14
2.8 features . . . 15

2.8.1 SQL template . . . 15
2.8.2 Explanation . . . 15
2.8.3 Example . . . 17
2.9 min_m . . . 17
2.9.1 SQL template . . . 17
2.9.2 Explanation . . . 17
2.9.3 Example . . . 17
2.10 max_m . . . 18
2.10.1 SQL template . . . 18
2.10.2 Explanation . . . 18
2.10.3 Example . . . 18
2.11 monad sets . . . 18
2.11.1 SQL template . . . 18
2.11.2 Explanation . . . 18
2.11.3 Example . . . 19

**3 Object_dm data 20**
3.1 Introduction . . . 20
3.2 OT_objects . . . 21
3.2.1 SQL template . . . 21
3.2.2 Explanation . . . 21
3.2.3 Example . . . 22
3.3 Monad set encoding . . . 22
3.4 OT_mdf_FEATURE_NAME_set . . . 23
3.4.1 SQL template . . . 23
3.4.2 Explanation . . . 24
3.4.3 Example . . . 24

**4 Implementing the MQL commands 26**
4.1 Introduction . . . 26
4.2 Database manipulation . . . 26
4.2.1 CREATE DATABASE . . . 26
4.2.2 USE DATABASE . . . 28
4.2.3 DROP DATABASE . . . 29
4.3 Object type manipulation . . . 30
4.3.1 CREATE OBJECT TYPE . . . 30
4.3.2 UPDATE OBJECT TYPE . . . 32
4.3.3 DROP OBJECT TYPE . . . 33
4.4 Enumeration manipulation . . . 34
4.4.1 CREATE ENUMERATION . . . 34
4.4.2 UPDATE ENUMERATION . . . 35
4.4.3 DROP ENUMERATION . . . 37
4.5 Segment manipulation . . . 38

4.5.1 CREATE SEGMENT . . . . . 38
4.6 Querying . . . . . 39
4.6.1 SELECT OBJECTS . . . . . 39
4.6.2 SELECT OBJECTS AT . . . . . 40
4.6.3 SELECT OBJECT TYPES . . . . . 40
4.6.4 SELECT FEATURES . . . . . 41
4.6.5 SELECT ENUMERATIONS . . . . . 42
4.6.6 SELECT ENUMERATION CONSTANTS . . . . . 42
4.6.7 SELECT OBJECT TYPES USING ENUMERATION . . . . . 43
4.7 Object manipulation . . . . . 44
4.7.1 CREATE OBJECT FROM MONADS . . . . . 44
4.7.2 CREATE OBJECT FROM ID_DS . . . . . 45
4.7.3 CREATE OBJECT FROM (focus | all | ) QUERY . . . . . 46
4.7.4 UPDATE OBJECTS BY MONADS . . . . . 47
4.7.5 UPDATE OBJECTS BY ID_DS . . . . . 48
4.7.6 UPDATE OBJECTS BY (focus | all | ) QUERY . . . . . 49
4.7.7 DELETE OBJECTS BY MONADS . . . . . 50
4.7.8 DELETE OBJECTS BY ID_DS . . . . . 50
4.7.9 DELETE OBJECTS BY (focus | all | ) QUERY . . . . . 51
4.8 Feature manipulation . . . . . 52
4.8.1 GET FEATURES . . . . . 52

**A Copying this document** **53**
A.1 Introduction . . . . . 53
A.2 Creative Commons Deed (for all documentation) . . . . . 53

# List of Tables


2.1 enumeration_constants example . . . . . 12
2.2 Bit-set flags for object_types table and object ranges . . . . . 13
2.3 Bit-set flags for object_types table and monad uniqueness . . . . . 13
2.4 Bit-set flags for object_types table and the property of whether the OT name is a string or a plain identifier . . . . . 14
2.5 object_types example . . . . . 14
2.6 Feature type ids for standard atomic types . . . . . 16
2.7 Feature type flags for standard atomic types . . . . . 16
2.8 Examples of features. Note how the feature_type_id has the value of 1 for strings (see table 2.6). . . . . 17
2.9 Example of min_m . . . . . 17
2.10 Example of min_m . . . . . 18

3.1 SQL types corresponding to EMdF types . . . . . 22
3.2 Example of OT_Objects (Phrase_Objects) . . . . . 22

# List of Figures

# Chapter 1

# Preliminaries

## 1.1 Introduction

In this report, it is my aim to succinctly describe most of my ideas on how to implement the EMdF model in a Relational Database Management System, using a subset of SQL2.

The data is split into two neatly segregated kinds of data:

- meta-data and
- object_dm data.

The meta-data maintains information about object types, enumerations, and sequences of ids. The object_dm data is made on a per-object type basis. The structure of this report reflects this segregation: chapter 2 deals with meta-data, whereas chapter 3 deals with object_dm data.

Chapter 4 details how to implement all of the MQL statements using SQL2.

In this chapter, I give some preliminaries.

## 1.2 Assumptions on the implementation

### 1.2.1 The three sequences of ids

Three sequences of ids are assumed to exist in each EMdF database: One sequence for assigning object id_ds, one sequence for assigning type ids (object type ids, enumeration type ids, and feature type ids), and one for everything else (see section 2.3).

When autogenerating an id from a given sequence, we read the number of the relevant sequence from this table, and update the tuple with this value plus one, ready for next time we need an autogenerated id.

### 1.2.2 All names are stored as lower-case

The names of all object types, enumerations, and features are stored as all-lower-case. This makes it easy to search for them later. However, enumeration constants are case-sensitive, so

they are not stored lower-case.

## 1.3 Conventions used

I employ a number of conventions in this document:

1. Throughout, the shorthand "OT" is used to mean "Object Type." This is especially important in SQL templates.

2. When referring to tables in the text, the tables are in the modern typeface, and are enclosed in "double quotes."

3. When referring to table attributes in the text, the attributes are in the typewriter typeface, and are enclosed in "`double quotes`"

4. In SQL code, anything enclosed in { curly braces } is meant to be replaced with a value described within the curly braces. E.g., "SET is_true = { 0 | 1 }" means that, when executing the SQL code, the value used to set "`is_true`" must be either "0" or "1".

5. Throughout, examples of table data are given. Where ids are involved, the ids are meant to be consistent throughout this document, so that you should be able to follow the references to the right tuples.

# Chapter 2

# Meta-data

## 2.1 Introduction

This chapter, I detail all of the tables necessary for maintaining the meta-data in the EMdF database. For each table, I write on three subjects:

1. An SQL template for creating the table,
2. An explanation, including a rationale, and
3. An example, to show the theory in practice.

The following tables are needed for storing meta-data:

- schema_version
- database_metadata
- sequence_0
- sequence_1
- sequence_2
- enumerations
- enumeration_constants
- object_types
- features
- min_m
- max_m

- monad_sets
- OT_objects

These will be described in turn below.

## 2.2 schema_version

### 2.2.1 SQL template

```
CREATE TABLE schema_version (
    dummy_id INTEGER PRIMARY KEY NOT NULL,
    schema_version INT NOT NULL
);
```

### 2.2.2 Explanation

This table contains, in numerical form, the version of the schema in use. The values are defined in emdf.h in the sources. This was added in version 1.2.0.pre59. The dummy_id is always 0, and there is always exactly one row in the table.

### 2.2.3 Example

The table looks like this:

| dummy_id | schema_version |
|---|---|
| 0 | 5 |

## 2.3 sequence_0, sequence_1, and sequence_2

### 2.3.1 SQL template

```
CREATE TABLE sequence_0 (
    sequence_id INTEGER PRIMARY KEY NOT NULL,
    sequence_value INT NOT NULL
);
CREATE TABLE sequence_1 (
    sequence_id INTEGER PRIMARY KEY  NOT NULL,
    sequence_value INT NOT NULL
);
CREATE TABLE sequence_2 (
    sequence_id INTEGER PRIMARY KEY  NOT NULL,
    sequence_value INT NOT NULL
);
```

### 2.3.2 Explanation

These tables are for maintaining information on the three sequences of ids that must exist in an EMdF database. See section 1.2.1 for background information.

The "`sequence_id`" attribute is meant to take on one of the following two values:

| Value | C/C++ preprocessor #DEFINE | Meaning |
|---|---|---|
| 0 | SEQUENCE_OBJECT_ID_DS | The sequence is for object id_ds |
| 1 | SEQUENCE_TYPE_IDS | The sequence for object type ids, enumeration type ids, and featu |
| 2 | SEQUENCE_OTHER_IDS | The sequence is for all other ids |

The "`sequence_value`" attribute then lists the value of the next id to be taken for that sequence.

All three must be initialized to 1. However, when drawing from SEQUENCE_TYPE_IDs, the actual value will be shift-lefted SEQUENCE_TYPE_IDS_FREE_LOWER_BITS. This is currently 16, meaning that the seuqence can in reality only go as far as 2^15 (32768) before wrapping around into negative numbers.

NOTE: This may be implemented differently for each backend.

### 2.3.3 Example

The tables should look like this right after initialization of the database:

| sequence_id | sequence_value |
|---|---|
| 0 | 1 |

| sequence_id | sequence_value |
|---|---|
| 1 | 1 |

| sequence_id | sequence_value |
|---|---|
| 2 | 1 |

## 2.4 enumerations

### 2.4.1 SQL template

```
CREATE TABLE enumerations (
    enum_id INTEGER PRIMARY KEY NOT NULL,
    enum_name VARCHAR(255) NOT NULL
);
```

### 2.4.2 Explanation

This table is the master table for the data domain of enumerations. It lists, for each enumeration in the database, its enum_id and its human-readable name. Another table, "enumeration_constants," then lists all of the constants for each enumeration type.

The "`enum_id`" attribute is taken from the "sequence_1" table, i.e., from the SEQUENCE_TYPE_IDS sequence.

The "`enum_name`" attribute is what the user entered when creating the enumeration.

### 2.4.3 Example

As an example, the following enumerations might be defined:

| enum_id | enum_name |
|---|---|
| 65536 | phrase_type_t |
| 131072 | part_of_speech_t |
| 983040 | clause_type_t |
| ⋮ | ⋮ |

## 2.5 enumeration_constants

### 2.5.1 SQL template

```
CREATE TABLE enumeration_constants (
    enum_id INT NOT NULL,
    enum_value_name VARCHAR(255) NOT NULL,
    value INT NOT NULL,
    is_default CHAR(1) NOT NULL,
    PRIMARY KEY (enum_id, enum_value_name)
);
```

### 2.5.2 Explanation

This table lists, for each enumeration specified in the table "enumerations," data pertaining to all of the constants in the enumeration:

- The enum_id (see below).
- The human-readable name of the constant ("`enum_value_name`"),
- The value itself ("`value`"), and
- A boolean specifying whether this is the default or not ("`is_default`"). The only valid values for this attribute are 'Y' and 'N'.

The enum_ids are drawn from SEQUENCE_TYPE_IDS, but of course shift-lefted as explained in Section 2.3 which starts on page 9.

| enum_id | enum_value_name | value | default_value |
|---|---|---|---|
| 65536 | ptNotAppliccable | -1 | 'Y' |
| 65536 | VP | 1 | 'N' |
| 65536 | NP | 2 | 'N' |
| 65536 | NPpers | 3 | 'N' |
| ⋮ | ⋮ | ⋮ | ⋮ |
| 131072 | pspNotAppliccable | -1 | 'Y' |
| 131072 | psp_article | 0 | 'N' |
| 131072 | psp_verb | 1 | 'N' |
| 131072 | psp_noun | 2 | 'N' |
| 131072 | psp_proper_noun | 3 | 'N' |
| 131072 | psp_adverb | 4 | 'N' |
| ⋮ | ⋮ | ⋮ | ⋮ |
| 196608 | prsNotAppliccable | -1 | 'Y' |
| 196608 | prs_singular | 1 | 'N' |
| 196608 | prs_dual | 2 | 'N' |
| 196608 | prs_plural | 3 | 'N' |
| ⋮ | ⋮ | ⋮ | ⋮ |
| 262144 | gndNotAppliccable | -1 | 'Y' |
| 262144 | gnd_masculine | 1 | 'N' |
| 262144 | gnd_feminine | 2 | 'N' |
| ⋮ | ⋮ | ⋮ | ⋮ |
| 983040 | ct_Way0 | 1 | 'N' |
| 983040 | ct_Xqtl | 2 | 'N' |
| ⋮ | ⋮ | ⋮ | ⋮ |

Table 2.1: enumeration_constants example

| #define | value | meaning |
|---|---|---|
| OT_RANGE_MASK | 0x00000007 | Bit-mask for these values |
| OT_WITH_MULTIPLE_RANGE_OBJECTS | 0x00000000 | Object type has multiple-range objects |
| OT_WITH_SINGLE_RANGE_OBJECTS | 0x00000001 | Object type has single-range objects |
| OT_WITH_SINGLE_MONAD_OBJECTS | 0x00000002 | Object type has single-monad objects |

Table 2.2: Bit-set flags for object_types table and object ranges

| #define | value | meaning |
|---|---|---|
| OT_MONAD_UNIQUENESS_MASK | 0x00000078 | Bit-mask for these values |
| OT_WITHOUT_UNIQUE_MONADS | 0x00000000 | Monads may not be unique |
| OT_HAVING_UNIQUE_FIRST_MONADS | 0x00000008 | All first monads are unique |
| OT_HAVING_UNIQUE_FIRST_AND_LAST_MONADS | 0x00000010 | All first and last monads are unique |

Table 2.3: Bit-set flags for object_types table and monad uniqueness

### 2.5.3 Example

For the two enumerations defined in the previous section, the values in table 2.1 might be defined.

## 2.6 object_types

### 2.6.1 SQL template

```
CREATE TABLE object_types (
    object_type_id INTEGER PRIMARY KEY NOT NULL,
    object_type_name VARCHAR(255) NOT NULL,
    object_type_flags INT NOT NULL
);
```

### 2.6.2 Explanation

This table is the master table for object types. It stores, for each object type, its id, its human-readable name, and an "INT"-encoded set of integers of flags. The id is autogenerated, upon creation of the object type, from the "sequence_1" table, i.e., using the "SEQUENCES_TYPE_IDS" sequence. The flags are taken from Tables 2.2 and 2.3. Note that the flags in Table 2.2 are not bitfield flags, but form three-bit integer. The flags in Table 2.3 form a four-bit integer, and the flags in Table 2.4 form a two-bit integer.

Special mention should be made of the bit-set in Table 2.4. As will be shown in Section XXX, the object type names sometimes map to their lower-case equivalents, and sometimes to a name derived from a CRC32-hash of the original name.

| #define | value | meaning |
|---|---|---|
| OT_NAME_IS_STRING_MASK | 0x00000300 | Bit-mask for this property |
| OT_NAME_IS_STRING | 0x00000100 | This OT name is a string, and is therefore mapped to a CRC |

Table 2.4: Bit-set flags for object_types table and the property of whether the OT name is a string or a plain identifier

### 2.6.3 Example

Table 2.5 shows some sample object types.

| type_id | type_name | object_type_flags |
|---|---|---|
| 327680 | Word | 0x00000001 |
| 851968 | Phrase | 0x0 |
| 1048576 | Clause | 0x0 |
| ⋮ | ⋮ | ⋮ |

Table 2.5: object_types example

## 2.7 normalized_object_type_names

### 2.7.1 SQL template

```
CREATE TABLE normalized_object_type_names (
    object_type_id INT NOT NULL,
    object_type_name VARCHAR(255) NOT NULL,
    normalized_object_type_name VARCHAR(255) NOT NULL,
    PRIMARY KEY (object_type_id)
);
```

### 2.7.2 Explanation

Whenever an object type name is a C identifier “A”, the name used for the table to hold the objects of that feature is formed as “A’_objects”, where A’ is the lowercased version of A. However, starting with Emdros version 3.2.1.pre16, it is possible to use arbitrary strings as object type names. Since many database backends cannot use arbitrary strings in the table names, we need a way of mapping the “real”, MQL-defined object type name to something that will be accepted by the database backend.

Thus the EMdF backend “normalizes” object type names before forming database table names. The algorithm used to determine the normalized object type name is as follows:

1. Let OT = the object type name as given in the MQL.

2. Let OTL = OT, with all ASCII capital letters lower-cased.

3. If OTL is a C identifier (i.e., starts with a letter or an underscore, and consists only of (in this case, lower-case) letters, underscores, and digits 0-9), then return OTL. Otherwise, go on

4. Let CRC = the 8-character lower-case hexadecimal representation of the CRC32 hash of the string ("EMdF" + OTL), as defined in EMdF/crc32.cpp.

5. return "ot" + CRC.

The result of this algorithm is then used to form the table name for the object tables, as well as any other table names, such as sets for sets of strings. For object tables, the method is to prepend "_objects" to the normalized table name.

Before Emdros version 3.2.1.pre16, the algorithm would stop at step #3, since all object type names were required to be C identifiers.

## 2.8 features

### 2.8.1 SQL template

```
CREATE TABLE features (
    object_type_id INT NOT NULL,
    feature_name VARCHAR(255) NOT NULL,
    feature_type_id INT NOT NULL,
    default_value VARCHAR(1000) NOT NULL,
    computed CHAR(1) NOT NULL DEFAULT 'N',
    PRIMARY KEY (object_type_id, feature_name)
);
```

### 2.8.2 Explanation

This table is analogous to the "enumeration_constants" table. It lists, for each feature:

1. The object type id denoting the object type with which this feature is associated ("`object_type_id`"),

2. The feature name in human-readable form ("`feature_name`"),

3. A feature type id ("`feature_type_id`"). More on this in a moment,

4. A string representing the default value ("`default_value`"), and

5. A one-CHAR boolean indicating whether the feature is computed ('Y') or stored ('N') ("`computed`").

The attribute "`object_type_id`" references the "`object_type_id`" attribute of the "object_types" table.

The attribute "`feature_type_id`" can take on values from the following two sources:

1. For standard atomic types, the value will be composite: A bitwise OR of one of the values described in table 2.6 and possibly one of the values described in table 2.7. Note that of the standard atomic types, only INTEGER and ID_D can have the FEATURE_TYPE_LIST_OF bit set.

2. For enumerations, the value will be any value from the "`enum_id`" attribute of the "enumerations" table, and with the FEATURE_TYPE_ENUM #define from table 2.6 bitwise-OR'ed in. Thus

Only FEATURE_TYPE_STRING and FEATURE_TYPE_ASCII can have the FEATURE_TYPE_AS_SET bit set. If set, there is an additional table, OT_mdf_FEATURE_NAME_set (described in Section 3.4 on page 23), which holds the strings as well as an id_d. Then this id_d is used in OT_objects in lieu of the string. This is more compact, and may give a speed increase.

Only the standard atomic types (not enumerations) may have the FEATURE_TYPE_WITH_INDEX bit set. If set, the EMdF layer will put an index on the feature. The index may be dropped again with the DROP INDEXES MQL statement, or with the external manage_indices(1) program. However, this bit will not be cleared by such operations. It stays there and tells the EMdF layer to add the index to the feature if a CREATE INDEXES MQL statement is issued for the object type, or if manage_indices(1) is invoked to create indexes on the object type.

| value | C/C++ preprocessor #DEFINE | SQL-type in object tables |
|---|---|---|
| 0 | FEATURE_TYPE_INTEGER | INT |
| 1 | FEATURE_TYPE_STRING | TEXT |
| 2 | FEATURE_TYPE_ASCII | TEXT |
| 3 | FEATURE_TYPE_ID_D | INT |
| 4 | FEATURE_TYPE_ENUM | INT |
| 8 | FEATURE_TYPE_LIST_OF_INTEGER | TEXT |
| 11 | FEATURE_TYPE_LIST_OF_ID_D | TEXT |
| 12 | FEATURE_TYPE_LIST_OF_ENUM | TEXT |

Table 2.6: Feature type ids for standard atomic types

| value | C/C++ preprocessor #define | Meaning |
|---|---|---|
| (0x00000100L) | FEATURE_TYPE_WITH_INDEX | If set, the feature is indexed |
| (0x00000200L) | FEATURE_TYPE_FROM_SET | If set, the feature's value is drawn from a set |

Table 2.7: Feature type flags for standard atomic types

### 2.8.3 Example

Examples of features are given in table 2.8.

| object_type_id | feature_name | feature_type_id | default_value | computed |
|---|---|---|---|---|
| 327680 | psp | 131072 | pspNotAppliccable | 'N' |
| 327680 | person | 196608 | prsNotAppliccable | 'N' |
| 327680 | gender | 262144 | gndNotAppliccable | 'N' |
| 327680 | surface | 1 | “” | 'N' |
| 327680 | lexeme | 1 | “” | 'N' |
| ⋮ | ⋮ | ⋮ | | 'N' |
| 851968 | phrase_type | 65536 | pt_NotAppliccable | 'N' |
| ⋮ | ⋮ | ⋮ | | 'N' |
| 1048576 | clause_type | 983040 | ctWay0 | 'N' |

Table 2.8: Examples of features. Note how the feature_type_id has the value of 1 for strings (see table 2.6).

## 2.9 min_m

### 2.9.1 SQL template

```
CREATE TABLE min_m (
    dummy_id INTEGER PRIMARY KEY NOT NULL,
    min_m INT NOT NULL
);
```

### 2.9.2 Explanation

This table stores the smallest monad in the database. dummy_id is always 0.

### 2.9.3 Example

An example is given in table 2.9.

| dummy_id | min_m |
|---|---|
| 0 | 1 |

Table 2.9: Example of min_m

## 2.10 max_m

### 2.10.1 SQL template

```
CREATE TABLE max_m (
    dummy_id INTEGER PRIMARY KEY NOT NULL,
    max_m INT NOT NULL
);
```

### 2.10.2 Explanation

This table stores the largest monad in the database. dummy_id is always 0.

### 2.10.3 Example

An example is given in table 2.10.

| dummy_id | max_m |
|---|---|
| 0 | 138019 |

Table 2.10: Example of min_m

## 2.11 monad sets

### 2.11.1 SQL template

```
CREATE TABLE monad_sets (
    monad_set_id INTEGER PRIMARY KEY NOT NULL,
    monad_set_name VARCHAR(255) NOT NULL
);
CREATE TABLE monad_sets_monads (
    monad_set_id INT NOT NULL,
    mse_first INT NOT NULL,
    mse_last INT NOT NULL,
    PRIMARY KEY (monad_set_id, mse_first)
);
```

### 2.11.2 Explanation

The "`monad_sets`" table is for storing monad set IDs (built from the "sequence_2" table, i.e., from the SEQUENCE_OTHER_IDS sequence) and monad set names. The "`monad_sets_monad`" table is for storing the actual monad sets, mse by mse.

### 2.11.3 Example

As an example, consider the following tables:

| monad_set_id | monad_set_name |
|---|---|
| 131072 | Pentateuch |
| 196608 | My_book_collection |

| monad_set_id | mse_first | mse_last |
|---|---|---|
| 131072 | 1 | 113226 |
| 196608 | 1 | 52547 |
| 196608 | 176800 | 212900 |
| 196608 | 394700 | 430154 |

The “Pentateuch” monad-set consists of the monads { 1-113226 }, whereas the “My_book_collection” monad-set consists of the monads { 1-52547, 176800-212900, 394700-430154 }.

# Chapter 3

# Object_dm data

## 3.1 Introduction

In this chapter, I describe the tables needed for each object type.

There are three basic schemas for object types. The first is valid when the object type has been declared WITH MULTIPLE RANGE OBJECTS, or hasn't been given any RANGE declaration. The second is valid when the object type has been declared WITH SINGLE RANGE OBJECTS. The third is valid when the object type has been delcared WITH SINGLE MONAD OBJECTS.

In all three cases, the only table involved is:

- OT_objects

An object type that has been declared WITH SINGLE RANGE OBJECTS can only hold objects that consist of a single monad span, i.e., a single monad set element, from A to B. An object type that has been declaed WITH SINGLE MONAD OBJECTS can only hold objects that are singleton sets (i.e., have only 1 monad in their monad set). An object that has been declared WITH MULTIPLE RANGE OBJECTS can hold arbitrary monad sets.

The "range types" just described have a bearing on *which* fields are present. There is an additional distinction, namely "WITHOUT UNIQUE MONADS", "HAVING UNIQUE FIRST MONADS", and "HAVING UNIQUE FIRST AND LAST MONADS" . This distinction has a bearing on what the primary key is:

1. If "WITHOUT UNIQUE MONADS" is specified (or none of these three is specified), then the primary key will be the object_id_d. This means that there is no restriction on the uniqueness of the first (and last) monads.

2. If "HAVING UNIQUE FIRST MONADS" is specified, then the primary key is first_monad. This means that the user promises never to create any two objects with this object type which have the same first monad. Objects need not be unique in their first monads across object types: It is only within an object type that this needs to hold.

3. If "HAVING UNIQUE FIRST AND LAST MONADS" is specified, then the primary key is (first_monad, last_monad). This means that the user promises never to create any two

objects with this object type which have the same first and the same last monads, regardless of whether the two objects have the same monad set or not.

If a STRING or ASCII feature is declared "FROM SET", then a special table is created for that feature:

- OT_mdf_FEATURE_NAME_set

This is described in Section 3.4 on page 23.

## 3.2 OT_objects

### 3.2.1 SQL template

```
CREATE TABLE OT_objects(
    object_id_d INTEGER PRIMARY KEY NOT NULL,
    -- first_monad is always there
    first_monad INT NOT NULL,
    -- last_monad is not there for WITH SINGLE MONAD OBJECTS
    last_monad INT NOT NULL,
    -- monads is not there except for WITH MULTIPLE RANGE
    monads TEXT NOT NULL,  OBJECTS
    [ ... list of stored features ... ]
);
```

### 3.2.2 Explanation

This table is the master table for storing objects of type OT. For each object, the following are given:

1. The object id_d ("object_id_d"),
2. The first monad, for easy reference ("first_monad"),
3. The last monad, for easy reference ("last_monad"), and
4. The monad-set, encoded in a special way (see below).
5. Values for all of the stored features of the object.

The "object_id_d" attribute is either auto-generated from the "sequence_0" table, i.e., using the SEQUENCE_OBJECT_ID_DS sequence, or it is explicitly given. The "object_id_d" attribute is also the source for the special, read-only feature "self" that is on each object_dm type.

The reason why the first and last monads are here will become apparent when we discuss how to implement MQL queries.

| EMdF type | SQL type | Comment |
|---|---|---|
| INTEGER | INTEGER | 32-bit integer |
| ID_D | INTEGER | 32-bit integer |
| ASCII | SQL_TEXT_TYPE | |
| STRING | SQL_TEXT_TYPE | |
| Enumeration constants | INTEGER | 32-bit integer |
| List of INTEGER | SQL_TEXT_TYPE | |
| List of ID_D | SQL_TEXT_TYPE | |
| List of Enumeration constants | SQL_TEXT_TYPE | |

Table 3.1: SQL types corresponding to EMdF types

The last_monad column is not present if the object type has been declared WITH SINGLE MONAD OBJECTS. The monads column is only present when the object type has been declared WITH SINGLE RANGE OBJECTS or WITH MULTIPLE RANGE OBJECTS.

The types of the stored features are given in Table 3.1.

Note that ASCII, STRING, and lists are stored as the SQL_TEXT_TYPE, which varies between the backends. It is basically a long string. For lists, the value is a space-surrounded, space-delimited list of integers. For example, the list (1,2,3) would be represented as:

```
' 1 2 3 '
```

This makes for searching with LIKE '% 1 %' and the like.

### 3.2.3 Example

In table 3.2, I have listed four objects of type Phrase.

| object_id_d | first_monad | last_monad | phrase_type |
|---|---|---|---|
| 201 | 4 | 7 | 5 |
| 202 | 8 | 8 | 1 |
| 203 | 9 | 10 | 2 |
| 203 | 12 | 15 | 2 |
| ⋮ | ⋮ | ⋮ | ⋮ |

Table 3.2: Example of OT_Objects (Phrase_Objects)

## 3.3 Monad set encoding

The monad set encoding (in column OT_objects.monads) stores an arbitrary monad set efficiently, as a text-string. The format is as follows:

1. Each number is stored in a base-64 encoding that is described below.
2. The monad set is seen as a series of numbers. The current number is stored as the difference between the actual number and the preview number (where the previous number is defined as 0 for the first number).
3. The separator between monad set elements is the character 'y'. Thus the monad set element chain is a 'y'-separated list of monad set elements.
4. Singleton monad set elements are just stored as that single number.
5. Non-singleton monad set elements are stored as two numbers with the character 'z' in between.

The base-64 encoding will be explained below. For now, let me give an example to illustrate the principles above.

The monad-set { 1-3, 5, 7-10 } would, if we were using base-10 to store the numbers, be stored as "1z2y2y2z3". Let us break this down. There are three monad-set elements: "1z2", "2", and "2z3". The first translates to "1-3" because 1+2 = 3 (taking the previous "1" and adding "2" makes "3"). The second translates to "5" because the previous was 3, and when we add 2, we get 5. The third monad set element translates to "7-10" because "5+2=7" and "7+3=10" (again taking the previous number and adding the current number).

The base-64 encoding is very straightforward: The 32-bit number is broken down into 5 6-bit chunks and one 2-bit chunk (the 2 most significant bits). Starting from the chunk that has the most significant non-null bit, each chunk is written as the 6-bit value plus 48 (i.e., ASCII '0'). Thus the above set would be written as exactly "1z2y2y2z3".

## 3.4 OT_mdf_FEATURE_NAME_set

### 3.4.1 SQL template

```
-- This is optimized for finding string
-- string values from id_ds (for
-- querying.)
CREATE TABLE OT_mdf_FEATURE_NAME_set (
   id_d INTEGER PRIMARY KEY NOT NULL,
   string_value TEXT NOT NULL
);
-- This is so we can also quickly
-- find id_ds from string values
-- (for inserting/updating)
CREATE INDEX OT_mdf_FEATURE_NAME_set_i
ON OT_mdf_FEATURE_NAME_set
(string_value)
;
```

### 3.4.2 Explanation

If a STRING or ASCII feature of an object type is declared "FROM SET", this table will be created. Any strings which are assigned to this feature of an object when it is created or updated will be drawn from this table. Instead of storing the string in the feature, the id_d is stored instead. This gives a space savings, and often also a time savings, especially on MySQL and PostgreSQL. SQLite and SQLite3 may see no difference, or even worse performance. However, FROM SET should only be used with data sets which have low cardinality. The declaration should not be used with, e.g., "surface" or "lemma", since they are likely to have a large number of individual values. Better candidates would be "case", "number", "gender", "part of speech", etc., since they usually have low cardinality. Thus FROM SET should be seen as a way of getting the same effect as an enumeration, but with arbitrary strings instead of C identifiers as enumeration constants.

When an object is created or updated, and the object type to which it belongs has a STRING or ASCII feature which is declared "FROM SET", then this table is consulted to see if the string exists in it already. If it does not, then it is added, and an id_d is assigned from the SEQUENCE_OTHER_ID_DS sequence. Then that id_d is used in the in lieu of the string in the object's feature. If the string does exist in this table, the id_d from that row is used.

Note that features of type ID_D, INTEGER, and ENUM cannot be declared "FROM SET". This is because it makes no sense: There is no space savings, and certainly no time savings, since in all these cases, the integer can be stored directly.

### 3.4.3 Example

NOTE: the following example uses FROM SET with a "surface" feature *against* the recommendation used above.

```
CREATE OBJECT TYPE [Word surface : STRING FROM SET;]
```

word_mdf_surface_set:

| id_d | string_value |
|---|---|
| 21 | A |
| 22 | horse |
| 23 | is |
| 24 | a |
| 25 | horse. |

word_objects ("A horse is a horse is a horse."):

| object_id_d | first_monad | mdf_surface |
|---|---|---|
| 1 | 1 | 21 |
| 2 | 2 | 22 |
| 3 | 3 | 23 |
| 4 | 4 | 24 |
| 5 | 5 | 22 |
| 6 | 6 | 23 |
| 7 | 7 | 24 |
| 8 | 8 | 25 |

# Chapter 4

# Implementing the MQL commands

## 4.1 Introduction

In this chapter, I treat all of the commands of the new MQL and show in some detail how they can be implemented using fragments of SQL. I follow the structure of chapter 2 of "Towards a new MQL."

## 4.2 Database manipulation

### 4.2.1 CREATE DATABASE

#### 4.2.1.1 Weeder

Nothing to do.

#### 4.2.1.2 Symbol-checker

Nothing to do.

#### 4.2.1.3 Type-checker

Nothing to do.

#### 4.2.1.4 Monads-checker

Nothing to do.

#### 4.2.1.5 Interpreter

The following needs to be done when creating a database:

1. Create the physical database in the server.

2. Create and initialize the “schema_version” table.
3. Create and initialize the “sequence_0”, “sequence_1”, and “sequence_2” tables.
4. Create the “enumerations” table.
5. Create the “enumeration_constants” table.
6. Create the “object_types” table.
7. Create the “features” table.
8. Create the “monad_sets” table.
9. Create the “monad_sets_monads” table.

#### 4.2.1.6 SQL fragments

The SQL to do the above is as follows (in one transaction):

```
CREATE DATABASE { database_name }
CREATE TABLE schema_version (
    dummy_id INTEGER PRIMARY KEY NOT NULL,
    schema_version INT NOT NULL
);
INSERT INTO schema_version (dummy_id, schema_version)
VALUES (0, { schema-version } );
CREATE TABLE sequence_0 (
    sequence_id INTEGER PRIMARY KEY NOT NULL,
    sequence_value INT NOT NULL
);
CREATE TABLE sequence_1 (
    sequence_id INTEGER PRIMARY KEY NOT NULL,
    sequence_value INT NOT NULL
);
CREATE TABLE sequence_2 (
    sequence_id INTEGER PRIMARY KEY NOT NULL,
    sequence_value INT NOT NULL
);
INSERT INTO sequence_0 (sequence_id, sequence_value)
VALUES ( { SEQUENCES_OBJECT_ID_DS } , 1)
INSERT INTO sequence_1 (sequence_id, sequence_value)
VALUES ( { SEQUENCES_TYPE_IDS } , 1)
INSERT INTO sequence_2 (sequence_id, sequence_value)
VALUES ( { SEQUENCES_OTHER_IDS } , 1)
```

```
CREATE TABLE enumerations (
    enum_id INTEGER PRIMARY KEY NOT NULL,
    enum_name VARCHAR(255) NOT NULL
)
CREATE TABLE enumeration_constants (
    enum_id INT NOT NULL,
    enum_value_name VARCHAR(255) NOT NULL,
    value INT NOT NULL,
    is_default CHAR(1) NOT NULL,
    PRIMARY KEY (enum_id, enum_value_name)
)
CREATE TABLE object_types (
    object_type_id INTEGER PRIMARY KEY  NOT NULL,
    object_type_name VARCHAR(255) NOT NULL
)
CREATE TABLE features (
    object_type_id INT NOT NULL,
    feature_name VARCHAR(255) NOT NULL,
    feature_type_id INT NOT NULL,
    computed CHAR(1) NOT NULL DEFAULT 'N',
    PRIMARY KEY (object_type_id, feature_name)
)
CREATE TABLE monad_sets (
    monad_set_id INTEGER PRIMARY KEY  NOT NULL,
    monad_set_name VARCHAR(255) NOT NULL
);
CREATE TABLE monad_sets_monads (
    monad_set_id INT NOT NULL,
    mse_first INT NOT NULL,
    mse_last INT NOT NULL,
    PRIMARY KEY (monad_set_id, mse_first)
);
```

### 4.2.2 USE DATABASE

#### 4.2.2.1 Weeder

Nothing to do.

#### 4.2.2.2 Symbol-checker

The symbol checker should check that the database exists.

#### 4.2.2.3 Type-checker

Nothing to do.

#### 4.2.2.4 Monads-checker

Nothing to do.

#### 4.2.2.5 Interpreter

How to do this will vary from database server to database server. I don't think it can always be done in SQL. On the contrary, PostgreSQL seems to couple connections tightly with databases, so it should rather be on a connection-level

### 4.2.3 DROP DATABASE

#### 4.2.3.1 Weeder

Nothing to do.

#### 4.2.3.2 Symbol-checker

The symbol checker should check that the database exists.

#### 4.2.3.3 Type-checker

Nothing to do.

#### 4.2.3.4 Monads-checker

Nothing to do.

#### 4.2.3.5 Interpreter

- Drop the database. This is usually an easy DROP DATABASE statement.

#### 4.2.3.6 SQL fragments

```
DROP DATABASE { database_name }
```

## 4.3 Object type manipulation

### 4.3.1 CREATE OBJECT TYPE

#### 4.3.1.1 Weeder

- Check that the feature "self" is not declared.

#### 4.3.1.2 Symbol-checker

- Check that the object type does not already exist.
- Check that the enumerations exist for the features whose types are enumerations.
- Check that, within these enumerations, any default specification which is an enumeration constant, does exist in that enumeration.

#### 4.3.1.3 Type-checker

- Assign type-ID to each feature, based on the type-name. If it is one of the standard types, then assign its corresponding ID (see table 2.6). If it is an enumeration type, then assign the enum_id of the enumeration.
- Check that the type of each feature matches the type of any default specification. In doing so, provide, in the AST, a string representing the default value for any feature that does not have a default specification. It is an error to specify an integer if the type is an enumeration. It must be an enumeration constant. The reason is that we must have data integrity, and this is an easy way of ensuring that for enumerations.

#### 4.3.1.4 Monads-checker

Nothing to do.

#### 4.3.1.5 Interpreter

- Create the object type in table "object_types"
- Create all the tables associated with the object type (OT_objects, etc.)
- Create all the features

#### 4.3.1.6 SQL fragments

##### 4.3.1.6.1 Checking for (non-)existence of object type

```
SELECT object_type_id
FROM object_types
WHERE object_type_name = '{ object_type_name }'
```

#### 4.3.1.6.2 Checking for (non-)existence of enumeration

```
SELECT enum_id
FROM enumerations
WHERE enum_name = '{ enumeration-name }'
```

#### 4.3.1.6.3 Checking for (non-)existence of enumeration constant

```
SELECT enum_value_name
FROM enumeration_constants EC, enumerations E
WHERE EC.enum_value_name = '{ enumeration-constant-name }'
      AND EC.enum_id = E.enum_id
      AND E.enum_name = '{ enumeration-name }'
```

#### 4.3.1.6.4 Creating the object type

```
INSERT INTO object_types (object_type_id, object_type_name)
VALUES ( { auto-generated id }, { object_type_name } )
```

#### 4.3.1.6.5 Creating the tables associated with the object type

```
CREATE TABLE OT_objects(
    object_id_d INTEGER PRIMARY KEY NOT NULL,
    first_monad INT NOT NULL,
    last_monad INT NOT NULL,
    monads TEXT NOT NULL,
    [ ... list of stored features ... ]
)
```

#### 4.3.1.6.6 Creating all the features

For each feature:

```
INSERT INTO features (
    object_type_id,
    feature_name,
    feature_type_id,
    default_value,
    computed
)
VALUES (
    { object_type_id : from the creation of the object type },
    { feature_name : feature_name },
    { feature_type_id : taken from AST },
```

```
        { default_value : string from AST },
        { computed : 'Y'/'N' based on the presence or
                     absence of the T_KEY_COMPUTED keyword}
    )
```

### 4.3.2 UPDATE OBJECT TYPE

#### 4.3.2.1 Weeder

- Check that the feature "self" is neither added nor removed.

#### 4.3.2.2 Symbol-checker

- Check that the object type already exists. In doing so, store the object type_id somewhere in the AST.
- Check that all the features that are to be removed do exist.
- Check that all the features that are to be added do not exist.
- Check that the enumerations exist for the new features whose types are enumerations.
- Check that, within these enumerations, any default specification which is an enumeration constant, does exist in that enumeration.

#### 4.3.2.3 Type-checker

- Assign type-ID to each feature that is to be added, based on the type-name. If it is one of the standard types, then assign its corresponding ID (see table 2.6). If it is an enumeration type, then assign the enum_id of the enumeration.
- Check that the type of each feature matches the type of any default specification. In doing so, provide, in the AST, a string representing the default value, both for those feature additions that do and those that don't have a default specification. It is an error to specify an integer if the type is an enumeration. It must be an enumeration constant. The reason is that we must have data integrity, and this is an easy way of ensuring that for enumerations.

#### 4.3.2.4 Monads-checker

Nothing to do.

#### 4.3.2.5 Interpreter

- Add the features that are to be added.
- Remove the features that are to be removed.

#### 4.3.2.6 SQL fragments

##### 4.3.2.6.1 Checking for (non-)existence of a feature

```
SELECT feature_type_id, default_value, computed
FROM features
WHERE object_type_id = { object type_id }
      AND feature_name = '{ feature-name }'
```

##### 4.3.2.6.2 Adding a feature to the OT_objects table

```
ALTER TABLE OT_objects ADD { encoded feature-name }
      { SQL-type } NOT NULL
```

##### 4.3.2.6.3 Removing a feature from the OT_objects table

```
ALTER TABLE OT_objects DROP { encoded feature-name }
```

##### 4.3.2.6.4 Removing a feature from the features table

```
DELETE FROM features
WHERE object_type_id = { object type_id }
      AND feature_name = '{ feature-name }'
```

### 4.3.3 DROP OBJECT TYPE

#### 4.3.3.1 Weeder

Nothing to do.

#### 4.3.3.2 Symbol-checker

- Check that the object type exists. In doing so, it should store the object type id in the AST.

#### 4.3.3.3 Type-checker

Nothing to do.

#### 4.3.3.4 Monads-checker

Nothing to do.

#### 4.3.3.5 Interpreter

- Drop all the tables associated with the object type.
- Delete all features associated with the object type.
- Delete the object type from the "object_types" table.

#### 4.3.3.6 SQL fragments

```
DROP TABLE OT_objects
DELETE FROM features
WHERE object_type_id = { object type_id from AST }
DELETE FROM object_types
WHERE object_type_id = { object type_id from AST }
```

## 4.4 Enumeration manipulation

### 4.4.1 CREATE ENUMERATION

#### 4.4.1.1 Weeder

- Check that at most one "`ec_declaration`" has the "DEFAULT" keyword, and set a boolean for each member of the list of declarations saying whether it is the default or not. If none has the "DEFAULT" keyword, then set the boolean of the first item in the list to "true."

#### 4.4.1.2 Symbol-checker

- Check that no other enumeration by the same name exists already.
- Check that no enumeration constant already exists by the name given in any of the ec-declarations.
- Assign a value in the AST to each ec-declaration, either based on its position in the sequence, or based on its initialization.

#### 4.4.1.3 Type-checker

Nothing to do.

#### 4.4.1.4 Monads-checker

Nothing to do.

#### 4.4.1.5 Interpreter

- Create the enumeration using an autogenerated ID.
- Add all the enumeration constants to the table.

#### 4.4.1.6 SQL fragments

##### 4.4.1.6.1 Creating the enumeration

```
INSERT INTO enumerations (enum_id, enum_name)
VALUES ( { auto-generated id }, { name } }
```

##### 4.4.1.6.2 Add an enumeration constant

```
INSERT INTO enumeration_constants (
    enum_id,
    enum_value_name,
    value,
    is_default
)
VALUES (
    { enum_id : The auto-generated id used to create the enum },
    { enum_value_name : ec-name },
    { value : ec-value},
    { is_default : 'Y'/'N' }
)
```

### 4.4.2 UPDATE ENUMERATION

#### 4.4.2.1 Weeder

- Check that at most one enumeration-constant update has the "DEFAULT" keyword, and set a boolean for each member of the list of updates saying whether it is the default or not. If none has the "DEFAULT" keyword, then none of these booleans should be true. Either set a boolean in the top-level AST node of the MQL statement, or provide a function which lets one know, whether one of the additions or updates has the "DEFAULT" keyword.

#### 4.4.2.2 Symbol-checker

- Check that the enumeration exists already.
- Check that for all additions, the enumeration constants added do not exist already.
- Check that, for all updates, the enumeration constants updated already exist.

- Check that all constants being removed do exist.
- Check whether the current default is being removed. If it is, then another default should be specified, either as an update or as an addition (use the boolean or function mentioned under "weeder" above).

#### 4.4.2.3 Type-checker

Nothing to do.

#### 4.4.2.4 Monads-checker

Nothing to do.

#### 4.4.2.5 Interpreter

- Remove all the constants being removed.
- Add all the constants being added.
- Update all the constants being updated.
- If there was a new specification of the "DEFAULT" constant:
  - Remove the "`is_default`" status from the current default.
  - Update the new default constant so that it "`is_default`".

#### 4.4.2.6 SQL fragments

##### 4.4.2.6.1 Checking which is the default enumeration constant

```
SELECT enum_value_name
FROM enumeration_constants
WHERE enum_id = { enumeration-id }
      AND is_default = 'Y'
```

##### 4.4.2.6.2 Checking for the (non)-existence of an enumeration

See section 4.3.1.

##### 4.4.2.6.3 Checking for the (non)-existence of an enumeration constant

See section 4.3.1.

##### 4.4.2.6.4 Removing a constant

```
DELETE
FROM enumeration_constants
WHERE enum_id = { enumeration-id }
      AND enum_value_name = { name of constant to delete }
```

**4.4.2.6.5 Adding a constant** See section 4.4.1.

**4.4.2.6.6 Updating a constant**

```
UPDATE enumeration_constants
SET value = { new value }
WHERE enum_id = { enumeration-id }
      AND enum_value_name = { name of constant to update }
```

**4.4.2.6.7 Removing the "is_default" status from the current default**

```
UPDATE enumeration_constants
SET is_default = 'N'
WHERE enum_id = { enumeration-id }
```

**4.4.2.6.8 Set the new default**

```
UPDATE enumeration_constants
SET is_default = 'Y'
WHERE enum_id = { enumeration-id }
      AND enum_value_name = { name of new default }
```

### 4.4.3 DROP ENUMERATION

#### 4.4.3.1 Weeder

Nothing to do.

#### 4.4.3.2 Symbol-checker

- Check that the enumeration does exist. In doing so, store the "enum_id" of the enumeration in the AST.

#### 4.4.3.3 Type-checker

Nothing to do.

#### 4.4.3.4 Monads-checker

Nothing to do.

#### 4.4.3.5 Interpreter

- Remove all the enumeration constants associated with the enumeration from table "enumeration_constants".
- Remove the enumeration itself from table "enumerations"

#### 4.4.3.6 SQL fragments

##### 4.4.3.6.1 Checking that the enumeration exists

See section 4.3.1.

##### 4.4.3.6.2 Removing all enumeration constants associated with the enumeration

```
DELETE
FROM enumeration_constants
WHERE enum_id = { enumeration-id }
```

##### 4.4.3.6.3 Removing the enumeration itself

```
DELETE
FROM enumerations
WHERE enum_id = { enumeration-id }
```

## 4.5 Segment manipulation

### 4.5.1 CREATE SEGMENT

#### 4.5.1.1 Weeder

- Check that the range is monotonic, i.e., that the second integer is greater than or equal to the first integer.
- Check that the range consists of positive numbers.

#### 4.5.1.2 Symbol-checker

Nothing to do.

#### 4.5.1.3 Type-checker

Nothing to do.

#### 4.5.1.4 Monads-checker

Nothing to do.

#### 4.5.1.5 Interpreter

- Currently, nothing. In the future: Add as a single-range monad set.

#### 4.5.1.6 SQL fragments

None.

## 4.6 Querying

### 4.6.1 SELECT OBJECTS

#### 4.6.1.1 Weeder

- Check everything as described in the "MQL query-subset" document.
- Check that the monad set in the AST consists of only positive, monotonic ranges.

#### 4.6.1.2 Symbol-checker

- Check everything as described in the "MQL query-subset" document.

#### 4.6.1.3 Type-checker

- Check everything as described in the "MQL query-subset" document.

#### 4.6.1.4 Monads-checker

- Build the monad set of the "IN" clause, if it is there. Store the monad set in the AST. If it isn't there, store "1..MAX_MONAD."

#### 4.6.1.5 Interpreter

This should follow the retrieval functions given in the "MQL Query subset" document. Below I list some of the SQL fragments which are needed for implementing these functions.

#### 4.6.1.6 SQL fragments

##### 4.6.1.6.1 Getting inst(T,U)

```
SELECT object_id_d
FROM OT_objects
WHERE { U.first() } <= first_monad
      AND last_monad <= { U.last() }
```

##### 4.6.1.6.2 Retrieve features from an object

```
SELECT { feature-names }
FROM OT_objects
WHERE object_id_d = { object id_d }
```

### 4.6.2 SELECT OBJECTS AT

#### 4.6.2.1 Weeder

- Check that the integer is positive.

#### 4.6.2.2 Symbol-checker

- Check that the object type exists.

#### 4.6.2.3 Type-checker

Nothing to do.

#### 4.6.2.4 Monads-checker

Nothing to do.

#### 4.6.2.5 Interpreter

Just asks the SQL database.

### 4.6.3 SELECT OBJECT TYPES

#### 4.6.3.1 Weeder

Nothing to do.

#### 4.6.3.2 Symbol-checker

Nothing to do.

#### 4.6.3.3 Type-checker

Nothing to do.

#### 4.6.3.4 Monads-checker

Nothing to do.

#### 4.6.3.5 Interpreter

Just asks the SQL database.

#### 4.6.3.6 SQL fragments

##### 4.6.3.6.1 Asking for the object types available

```
SELECT object_type_name
FROM object_types
```

### 4.6.4 SELECT FEATURES

#### 4.6.4.1 Weeder

Nothing to do.

#### 4.6.4.2 Symbol-checker

- Check that the object type actually exists. In doing so, store its object type_id in the AST.

#### 4.6.4.3 Type-checker

Nothing to do.

#### 4.6.4.4 Monads-checker

Nothing to do.

#### 4.6.4.5 Interpreter

- Ask the database server for the answer.
- Translate feature type_ids to strings. Only for enumeration constants does this involve querying the database.
- Translate the "computed" 'Y'/'N' boolean to a real boolean.

#### 4.6.4.6 SQL fragments

##### 4.6.4.6.1 Asking for the features of an object type

```
SELECT feature_name, feature_type_id, default_value, computed
FROM features
WHERE object_type_id = { object type_id }
```

##### 4.6.4.6.2 Translating feature type_ids to strings

```
SELECT enum_name
FROM enumerations
WHERE enum_id = { feature type_id }
```

### 4.6.5 SELECT ENUMERATIONS

#### 4.6.5.1 Weeder

Nothing to do.

#### 4.6.5.2 Symbol-checker

Nothing to do.

#### 4.6.5.3 Type-checker

Nothing to do.

#### 4.6.5.4 Monads-checker

Nothing to do.

#### 4.6.5.5 Interpreter

- Just ask the database server.

#### 4.6.5.6 SQL fragments

##### 4.6.5.6.1 Asking for the enumerations available

```
SELECT enum_name
FROM enumerations
```

### 4.6.6 SELECT ENUMERATION CONSTANTS

#### 4.6.6.1 Weeder

Nothing to do.

#### 4.6.6.2 Symbol-checker

- Check that the enumeration actually exists. In doing so, store the enum_id in the AST.

#### 4.6.6.3 Type-checker

Nothing to do.

#### 4.6.6.4 Monads-checker

Nothing to do.

#### 4.6.6.5 Interpreter

- Ask the database server for the answer.
- Convert the value to an integer and the "`is_default`" 'Y'/'N' boolean to a real boolean.

#### 4.6.6.6 SQL fragments

##### 4.6.6.6.1 Asking for the enumeration constants of an enumeration

```
SELECT enum_value_name, value, is_default
FROM enumeration_constants
WHERE enum_id = { enumeration id from AST }
```

### 4.6.7 SELECT OBJECT TYPES USING ENUMERATION

#### 4.6.7.1 Weeder

Nothing to do.

#### 4.6.7.2 Symbol-checker

- Check that the enumeration exists. In doing so, store its enum_id in the AST.

#### 4.6.7.3 Type-checker

Nothing to do.

#### 4.6.7.4 Monads-checker

Nothing to do.

#### 4.6.7.5 Interpreter

- Ask the database server for the answer

#### 4.6.7.6 SQL fragments

```
SELECT object_type_name
FROM object_types
WHERE object_type_id IN
   (SELECT object_type_id
    FROM features
    WHERE feature_type_id = { enumeration id }
   )
```

## 4.7 Object manipulation

### 4.7.1 CREATE OBJECT FROM MONADS

#### 4.7.1.1 Weeder

- Check that "`object_type_name`" is neither all_m, nor any_m, nor pow_m.
- Check that the feature "self" is not assigned a value.
- Check that all the ranges of monads are positive and monotonic.

#### 4.7.1.2 Symbol-checker

- If the user specified an id_d, check that this id_d is not in use already.
- Check that the object type exists. In doing so, store its object type_id in the AST.
- Check that no feture is assigned which the object type does not have.
- Make sure that all features are given a value. If a feature is not given a value, then use the default value.

#### 4.7.1.3 Type-checker

- Assign a type to each feature-assignment.
- Check for type-compatibility.

#### 4.7.1.4 Monads-checker

- Build the set of monads from the monads in the AST.

#### 4.7.1.5 Interpreter

- If the user did not specify an id_d, autogenerate one.
- Insert the object and monads in "OT_objects"

#### 4.7.1.6 SQL fragments

##### 4.7.1.6.1 Getting the default value of all features for an object id_d.

```
SELECT feature_name, default_value
FROM features
WHERE object_type_id = { object type_id }
```

##### 4.7.1.6.2 Inserting the object in "OT_objects."

```
INSERT INTO OT_objects (
    object_type_id,
    first_monad,
    last_monad,
    ... /* features */
)
VALUES (
   { object_type_id },
   { first monad },
   { last monad },
   ... /* features */
)
```

### 4.7.2 CREATE OBJECT FROM ID_DS

#### 4.7.2.1 Weeder

- Check that "`object_type_name`" is neither all_m, nor any_m, nor pow_m.
- Check that the feature "self" is not assigned a value.
- Check that none of the id_ds in the list are NIL.

#### 4.7.2.2 Symbol-checker

- If the user specified an id_d, check that this id_d is not in use already.
- Check that the object type exists. In doing so, store its object type_id in the AST.
- Check that no feture is assigned which the object type does not have.

- Make sure that all features are given a value. If a feature is not given a value, then use the default value.

#### 4.7.2.3 Type-checker

- Assign a type to each feature-assignment.
- Check for type-compatibility.

#### 4.7.2.4 Monads-checker

- Get the set of monads from the id_ds.

#### 4.7.2.5 Interpreter

- If the user did not specify an id_d, autogenerate one.
- Insert the object and monads in "OT_objects"

#### 4.7.2.6 SQL fragments

### 4.7.3 CREATE OBJECT FROM (focus | all | ) QUERY

#### 4.7.3.1 Weeder

- Check everything that must be checked for a SELECT OBJECTS query.
- Check that the feature "self" is not assigned a value.
- Check that "`object_type_name`" is neither all_m, pow_m, or any_m.

#### 4.7.3.2 Symbol-checker

- Check everything that must be checked for a SELECT OBJECTS query.
- If the user specified an id_d, check that this id_d is not in use already.
- Check that the object type exists. In doing so, store its object type_id in the AST.
- Check that no feture is assigned which the object type does not have.
- Make sure that all features are given a value. If a feature is not given a value, then use the default value.

#### 4.7.3.3 Type-checker

- Check everything that must be checked for a SELECT OBJECTS query.
- Assign a type to each feature-assignment.
- Check for type-compatibility.

#### 4.7.3.4 Monads-checker

- Check everything that must be checked for the SELECT OBJECTS query.
- Run the query.
- Get the set of monads from the query.

#### 4.7.3.5 Interpreter

- If the user did not specify an id_d, autogenerate one.
- Insert the object and monads in "OT_objects"

#### 4.7.3.6 SQL fragments

### 4.7.4 UPDATE OBJECTS BY MONADS

#### 4.7.4.1 Weeder

- Check that the object type is neither all_m, nor any_m, nor pow_m.
- Check that self is not assigned to.
- Check that all the ranges of monads are positive and monotonic.

#### 4.7.4.2 Symbol-checker

- Check that the object type actually exists. In doing so, store its object type_id in the AST.
- Check that the object type has all the features that are assigned a new value.
- Check that the features which are assigned are not computed features.

#### 4.7.4.3 Type-checker

- Check that there is type-compatibility between the features and their values.

#### 4.7.4.4 Monads-checker

- Build the set of monads from the monads in the AST.

#### 4.7.4.5 Interpreter

- Get the objects which are part_of the set of monads. This is a two-step process:
  - Find all the objects which are wholly contained within the borders of the set of monads.
  - Load each object one by one and check whether it should be included because it is part_of the set of monads.
- Update the objects

#### 4.7.4.6 SQL fragments

##### 4.7.4.6.1 Updating an object

```
UPDATE OT_objects
SET { (feature = value), (feature = value), ... }
WHERE object_id_d = { object id_d }
```

### 4.7.5 UPDATE OBJECTS BY ID_DS

#### 4.7.5.1 Weeder

- Check that the object type is neither all_m, nor any_m, nor pow_m.
- Check that self is not assigned to.
- Check that none of the id_ds in the list are NIL.

#### 4.7.5.2 Symbol-checker

- Check that the object type actually exists. In doing so, store its object type_id in the AST.
- Check that the object type has all the features that are assigned a new value.
- Check that the features which are assigned are not computed features.
- Check that the objects with the id_ds exist.

#### 4.7.5.3 Type-checker

- Check that there is type-compatibility between the features and their values.
- Check that the objects with the id_ds are of the specified type.

#### 4.7.5.4 Monads-checker

Nothing to do.

#### 4.7.5.5 Interpreter

- Update the objects

#### 4.7.5.6 SQL fragments

### 4.7.6 UPDATE OBJECTS BY (focus | all | ) QUERY

#### 4.7.6.1 Weeder

- Check that the object type is neither all_m, nor any_m, nor pow_m.
- Check that self is not assigned to.
- Check everything that must be checked for a SELECT OBJECTS query.

#### 4.7.6.2 Symbol-checker

- Check everything that must be checked for a SELECT OBJECTS query.
- Check that the object type actually exists. In doing so, store its object type_id in the AST.
- Check that the object type has all the features that are assigned a new value.
- Check that the features which are assigned are not computed features.

#### 4.7.6.3 Type-checker

- Check everything that must be checked for a SELECT OBJECTS query.
- Check that there is type-compatibility between the features and their values.

#### 4.7.6.4 Monads-checker

- Check everything that must be checked for the query.

#### 4.7.6.5 Interpreter

- Run the query.
- Get the set of objects:
  - If it is an ALL query, filter the returned set of objects by the given object type.

  - If it is a FOCUS query, first filter by focus, then filter by object type.

- Update the objects

### 4.7.7 DELETE OBJECTS BY MONADS

#### 4.7.7.1 Weeder

- Check that all the ranges of monads are positive and monotonic.

#### 4.7.7.2 Symbol-checker

- Check that the object type exists.

#### 4.7.7.3 Type-checker

Nothing to do.

#### 4.7.7.4 Monads-checker

- Build the set of monads from the monads in the AST.

#### 4.7.7.5 Interpreter

- Get the object id_ds of the objects which are part_of the set of monads. See the section on UPDATE OBJECTS BY MONADS for how to do this.

- Delete the objects and monads from OT_objects

#### 4.7.7.6 SQL fragments

##### 4.7.7.6.1 Deleting an object from OT_objects

```
DELETE
FROM OT_objects
WHERE object_id_d = { object id_d }
```

### 4.7.8 DELETE OBJECTS BY ID_DS

#### 4.7.8.1 Weeder

- Check that none of the id_ds in the list are NIL.

#### 4.7.8.2 Symbol-checker

- Check that the object type exists.
- Check that all the id_ds refer to objects that exist and are of the given type.

#### 4.7.8.3 Type-checker

Nothing to do

#### 4.7.8.4 Monads-checker

Nothing to do.

#### 4.7.8.5 Interpreter

- Delete the objects from OT_objects

### 4.7.9 DELETE OBJECTS BY (focus | all | ) QUERY

#### 4.7.9.1 Weeder

- Check everything that must be checked for a SELECT OBJECTS query.

#### 4.7.9.2 Symbol-checker

- Check everything that must be checked for a SELECT OBJECTS query.
- Check that the object type exists.

#### 4.7.9.3 Type-checker

- Check everything that must be checked for a SELECT OBJECTS query.

#### 4.7.9.4 Monads-checker

- Check everything that must be checked for a SELECT OBJECTS query.

#### 4.7.9.5 Interpreter

- Run the query.
- Get the object id_ds from the query:
  - If it is an ALL query, filter the returned set of objects by the given object type.
  - If it is a FOCUS query, first filter by focus, then filter by object type.

- Delete the objects from OT_objects

## 4.8 Feature manipulation

### 4.8.1 GET FEATURES

#### 4.8.1.1 Weeder

Nothing to do.

#### 4.8.1.2 Symbol-checker

- Check that the object type exists. In doing so, store its object type_id in the AST.
- Check that the objects with the given id_ds exists.
- Check that the objects all belong to the same type, namely the one given.
- Check that the features exist for the given object type.

#### 4.8.1.3 Type-checker

Nothing to do

#### 4.8.1.4 Monads-checker

Nothing to do.

#### 4.8.1.5 Interpreter

- Ask the database for the answer.

# Appendix A

# Copying this document

## A.1 Introduction

All Emdros documentation is provided under the Creative Commons Attribution-Sharealike license version 2.5. This license is included below, and can also be seen at:

`http://creativecommons.org/licenses/by-sa/2.5/`

## A.2 Creative Commons Deed (for all documentation)

Attribution-ShareAlike 2.5

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

### *License*

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. **Definitions**

(a) **"Collective Work"** means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

(b) **"Derivative Work"** means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.

(c) **"Licensor"** means the individual or entity that offers the Work under the terms of this License.

(d) **"Original Author"** means the individual or entity who created the Work.

(e) **"Work"** means the copyrightable work of authorship offered under the terms of this License.

(f) **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

(g) **"License Elements"** means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.

2. **Fair Use Rights.** Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. **License Grant.** Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

(a) to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;

(b) to create and reproduce Derivative Works;

(c) to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

(d) to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.

(e) For the avoidance of doubt, where the work is a musical composition:

i. **Performance Royalties Under Blanket Licenses.** Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.

ii. **Mechanical Rights and Statutory Royalties.** Licensor waives the exclusive right to collect, whether individually or via a music rights society or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

(f) **Webcasting Rights and Statutory Royalties.** For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. **Restrictions.** The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

(a) You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the

Collective Work any credit as required by clause 4(c), as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any credit as required by clause 4(c), as requested.

(b) You may distribute, publicly display, publicly perform, or publicly digitally perform a Derivative Work only under the terms of this License, a later version of this License with the same License Elements as this License, or a Creative Commons iCommons license that contains the same License Elements as this License (e.g. Attribution-ShareAlike 2.5 Japan). You must include a copy of, or the Uniform Resource Identifier for, this License or other license specified in the previous sentence with every copy or phonorecord of each Derivative Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Derivative Works that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder, and You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Derivative Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Derivative Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Derivative Work itself to be made subject to the terms of this License.

(c) If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or (ii) if the Original Author and/or Licensor designate another party or parties (e.g. a sponsor institute, publishing entity, journal) for attribution in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. **Representations, Warranties and Disclaimer**

UNLESS OTHERWISE AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE MATERIALS, EXPRESS, IMPLIED, STATU-

TORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. **Limitation on Liability.**

   EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. **Termination**

   (a) This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   (b) Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. **Miscellaneous**

   (a) Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   (b) Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   (c) If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

(d) No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

(e) This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, neither party will use the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time.

Creative Commons may be contacted at http://creativecommons.org/.