

CQL Specification

Draft V2.0

10/11/03

TOSHIBA Corporation

Documentation History

Date	Author	Note of Action	CQLVersion	Document Version
07/07/03	Olivier Lemarchand	Major update	Ver.2.0	1.0
16/09/03	Olivier Lemarchand	added improved support of Collections, removed Extensions mechanism and updated query conditions	Ver.2.0	1.1
17/09/03	Olivier Lemarchand	updated examples and datatypes descriptions	Ver.2.0	1.2
10/11/03	Olivier Lemarchand	Modified Document Layout and work items	Ver.2.0	1.3

1. Scope and Definition

CQL is an abbreviation for Class Query Language and is used to perform queries on information stored in a class-theoretic database (hereafter CDB). The aim of this specification is to define a declarative query language for hierarchical class libraries, i.e., data are assumed to be stored as instances of a class whose collection forms an acyclic graph structure¹ (ACGS).

In this respect, this query language is not only designed for the retrieval of information stored in a PLIB-LMS (ISO13584 based Library Management System), but is rather designed as a general purpose data query language for CDB-like databases. Nevertheless, PLIB-LMS is one of the major applications for which this CQL approach might be especially attractive and practical. In fact, CQL is an abstraction and an extension of the basic concept described in Part42 of ISO13584. Another possible area of application for this language might be ERDL (ISO15926 Part4; Epistle Reference Data Library). CQL is developed as a full scope neutral language for data modeling and manipulation adapted to various class libraries based on different de-jure and de-facto standards in order to ease the migration of data between them.

CQL includes SQL as its subset, so it is a superset of the latter. It does not mean, however, that the implementation of CQL must be based on a Relational Database System. The implementation may be built on an Object-Oriented Database system, regardless of its superficial SQL-like syntax.

One of the major differences between CQL and SQL-99(formerly known as SQL-3) is that CQL assumes every record of data is classified in a node under an ASGS. In other words, every piece of data must be an instance of some class which itself must form a set-theoretic subset of another class, exception made for the root of the class hierarchy. All root classes are bundled together by a virtual entity named “virtual root”, corresponding to Universal Class (Universal Set) in mathematics.

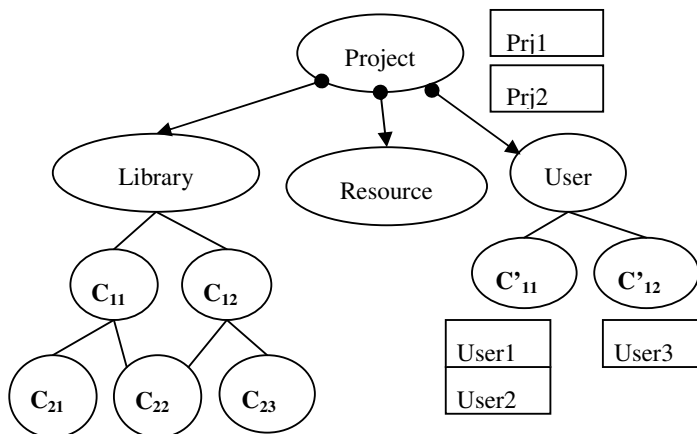
¹An acyclic graph structure is also called a forest in graph theory. Therefore, a tree is a subset of an ACGS

2. CQL architecture and datatypes

2.1. CQL architecture: predefined classes

The Object-oriented concepts are the very core of CQL. Therefore, in order to respect and follow these concepts, CQL provides 4 predefined classes:

- Project
- Library
- Resource
- User



A **Project** instance is the basic unit of data management. If a user wants to know about a product catalogue through CQL, he needs to belong to a user group defined in the concerned instance of **Project**. This concept does not derive from PLIB, but is even though necessary to CQL since the latter assumes the existence of a certain repository of data, or a database system. **Project** is the **whole** class for the three **Part** classes, i.e., **Library**, **Resource**, and **User**. In Object-oriented words, those classes are members of the class **Project**.

The **Library** class is the root class of ordinary class hierarchies modeled in a project. A project can contain any number of libraries (a library means here a class hierarchy with product instance data).

The **Resource** class is the class which stores physical files such as images, sounds, or programs, related to a project, and any type of other resources such as dictionaries, tables, etc.

The **User** class is the class which stores information about users and groups of users defined in the project.

It may have subclasses, and each of them has individual users as instances and specific access rights in order to perform security checks and access controls.

2.2. BSU Mechanism

In CQL every concept is unambiguously identified by a **Basic Semantic Unit** or BSU. The uniqueness of each BSU allows concepts to be referenced externally.

Classes and properties have a special BSU made from the concatenation of several other BSUs:

Class_BSU: a class_BSU in CQL is in fact the concatenation of:

- the identifier of the dictionary where the class is defined
- the BSU of the supplier
- the class' code

```
<class_BSU> ::= <dictionary_id> . <supplier_BSU> . <class_code>
```

Property_BSU: In CQL a property_BSU is the concatenation of:

- the class_BSU of the class where the property is defined visible
- the property's code

```
<property_BSU ::= <class_BSU> . <property_code>
```

2.3. Library: class instances and data containers

Each Project instance has a set of Library instances, each instance being a dictionary. Those dictionaries are in fact hierarchies of classes, following the object-oriented design. To store class instances, CQL uses tables as containers, being held in the class' extension. A class that does not define an extension cannot have instances. Eventually, those extensions are anonymous and do not need to be named since there is one and only one extension per class. Therefore, extensions can be unambiguously referred by a ClassBSU.

2.4. Datatypes in CQL

The datatype system in CQL consists in two different kinds of datatypes:

- simple datatypes
- complex datatypes

The difference between complex and simple datatypes lies in their very structure: complex datatypes are defined with simple datatypes (or other complex datatypes) whereas simple datatypes are stand-alone

```
<data_type> ::= <simple_data_type>
                | <complex_data_type>
```

2.4.1. Simple datatypes

The simple datatypes embedded in CQL are the followings:

INT_TYPE

INT_TYPE stands for relative integers.

REAL_TYPE

REAL_TYPE stands for decimal numbers.

STRING_TYPE

STRING_TYPE stands for both characters and strings of characters.

BOOLEAN_TYPE

BOOLEAN_TYPE stands for classical two-values Booleans: TRUE or FALSE

```
<simple_data_type> ::=  INT_TYPE
                        |  REAL_TYPE
                        |  STRING_TYPE
                        |  BOOLEAN_TYPE
```

2.4.2. Complex datatypes

INT_MEASURE_TYPE

INT_MEASURE_TYPE stands for relative integers associated with a UNIT (kg, m/s, etc.)

INT_CURRENCY_TYPE

INT_CURRENCY_TYPE stands for relative integers associated with a CURRENCY

REAL_MEASURE_TYPE

REAL_MEASURE_TYPE stands for decimal numbers associated with a UNIT (kg, m/s, etc.)

REAL_CURRENCY_TYPE

REAL_CURRENCY_TYPE stands for decimal numbers associated with a CURRENCY

MULTI_LINGUAL_STRING

MULTI_LINGUAL_STRING represents a list of STRING_TYPE values, each value being associated with a language code (2 characters: ja, en, fr, de, etc.).

COLLECTION

a COLLECTION in CQL is a very general type to describe any kind of set: ordered, unordered, homogeneous, heterogeneous, partially ordered, indexed, etc. This type can be used recursively since elements of a COLLECTION can be of any type ; so a COLLECTION OF COLLECTION OF INT_TYPE makes sense in CQL.

CLASS_INSTANCE_TYPE

CLASS_INSTANCE_TYPE corresponds to the notion of composition in Object Modeling. Consequently, on declaration time, a class BSU must be provided. A CLASS_INSTANCE_TYPE value is simply represented by a query.

ENUMERATION

An ENUMERATION declaration is a list of codes, each code being associated to a MULTILINGUAL_STRING. Thus a valuation is only embodied by a code (one listed in the declaration), de facto referencing one and only one MULTILINGUAL_STRING specified in the declaration.

#Work in Progress (syntax for ENUMERATIONs)

LEVEL

This datatype is used to described a range of values, thus defining a MINimum value, a MAXimum value, a NOMinal value and a TYPical value.

DATE_TIME_TYPE

#Work in Progress

```

<complex_data_type> ::= <enum_type_term>
    | <level_type_term>
    | <class_instance_type_term>
    | <collection_type_term>
    | TRANSLATED_STRING_TYPE
    | INT_MEASURE_TYPE
    | INT_CURRENCY_TYPE
    | REAL_MEASURE_TYPE
    | REAL_CURRENCY_TYPE
    | MULTI_LINGUAL_STRING (<language_code> {, <language_code> })

```

```

<enum_type_term> ::= ENUMERATION <enum_term>

<enum_term> ::= ( CODE , <enum_order> ) ( <enum_members_list> )

<enum_order> ::= MEANING.<language_code> { , MEANING.<language_code> }

<enum_members_list> ::= ( <enum_members> ) {, ( <enum_members> ) }

<enum_members> ::= '<identifier>' {, '<identifier>' }

```

```

<level_type> ::= LEVEL ( [<level_term>] ) OF REAL_TYPE
    | LEVEL ( [<level_term>] ) OF INT_MEASURE_TYPE
    | LEVEL ( [<level_term>] ) OF INT_TYPE
    | LEVEL ( [<level_term>] ) OF INT_CURRENCY_TYPE
    | LEVEL ( [<level_term>] ) OF REAL_MEASURE_TYPE
    | LEVEL ( [<level_term>] ) OF REAL_CURRENCY_TYPE

<level_term> ::= <level_possible_value> {, <level_possible_value> }

<level_possible_value> ::= ( MIN | NOM | TYP | MAX )

```

```

<collection_type_term> ::= <collection_property>
                        [ ( <collection_description_list> ) ]

<collection_description_list> ::= <collection_description_item>
                                {, <collection_description_item> }

<collection_description_item> ::= <bound_values>
                                | <of_clause>
                                | <index_clause>
                                | <with_clause>

<collection_property> ::= CYCLIC COLLECTION
                        | ORDERED COLLECTION
                        | COLLECTION

<bound_values> ::= [ <bound_value> ; <bound_value> ]

<bound_value> ::= <number> | ?

<of_clause> ::= OF <data_type>

<index_clause> ::= INDEXED <index_description>

<index_description> ::= WITH <simple_data_type>
                       | BY ( <index_enumeration> )

<index_enumeration> ::= <index_value> <data_type>
                       {, <index_value> <data_type> }

<index_value> ::= '<identifier>'
                | <number>

<with_clause> ::= WITH <with_attributes>

<with_attributes> ::= { <with_attribute> }[2]

```

```
<with_attribute> ::= UNIQUE  
                | SELECT ( <number_select> )
```

```
<number_select> ::= <number>  
                | <bound_values>
```

```
<class_instance_type_term> ::= CLASS_INSTANCE_TYPE ( <class_BSU> )
```

Example

```
MULTI_LINGUAL_STRING('en', 'ja', 'fr', 'de');
```

```
ENUMERATION (CODE, MEANING.en, MEANING.ja, MEANING.fr)  
(  
  ('car', 'car', '車', 'voiture')  
  ('color_red', 'red', '赤い', 'rouge')  
);
```

```
ORDERED COLLECTION (  
  [ 0 ; ? ],  
  OF INT_MEASURE_TYPE,  
  INDEXED WITH INT_TYPE  
);
```

3. Overview of CQL capabilities and commands

Type Name	Command	Definition
Class Modeling Language (CML)	create dictionary create supplier create class create property alter dictionary alter supplier alter class alter property drop dictionary drop supplier drop class drop property rename dictionary rename supplier rename class rename property	Defines and Manipulates dictionaries Part42, including case_of.
Data Representation Language (DRL)	create table create extension alter table drop table drop extension	Defines and Manipulates containers for content.
Library Manipulation Language (LML)	insert update delete	Manipulates contents
Search	select	Searches contents.
transaction control	begin commit rollback	Controls version, status

4. Search scope concept

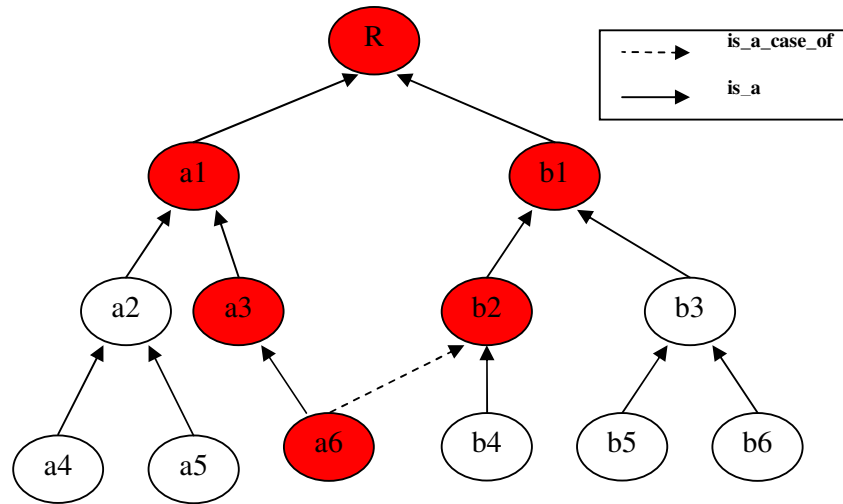
In Object-Oriented formalism, if A is a sub-class of B, any instance of A is also an instance of B thanks to the properties of inheritance. Thus, if we perform a query on class B, we might want to include A in the search. A Search Scope is an extension of this idea to all the relationships of the CQL model: **is_a**, **is_a_super_class_of**, **is_a_case_of**. Besides, Boolean operations can be applied to Search Scope: **union**, **intersection**, **subtraction**. Therefore, a Search Scope is a set of classes having special relationships.

In CQL, we use direction modifiers to extends the search to the super-classes of a class, to its subclasses, etc. The modifiers are the followings:

*	this modifier extends the scope to all the subclasses of the concerned class (it includes the classes that import properties from this class)
\$	this modifiers extends the scope to all the direct subclasses (the case_of relationship is not taken into account)
%	this modifier extends the scope to all the superclasses, including classes from which properties are imported
!	this modifier extends the scope to all the direct superclasses only

Thus, to include subclasses of B in the search, we would describe the search scope as follow:

B* or **B\$**



an example of a class hierarchy

In the above example, R is the root of the hierarchy, a1, a2...a6, b1, ..., b6 are classes. The class a6 is a subclass of a3 and is a case of b2. The search scope defined by **a6%** is represented in red on the illustration: a6 and all its super classes, including classes from which a6 imports properties, are bundled in the SearchScope.

5. CQL: a Class Modeling Language

NB: in the following BNF descriptions, we will use **BOLD CAPITAL** characters to describe terminals and plain characters to describe non-terminals and BNF special characters, such as () [] { } ::=

5.1. Project

5.1.1. Create

```
CREATE PROJECT <project_identifier> ([<project_definition> {, <project_definition> }]);
```

Where <project_definition> is:

```
<project_definition> ::= NAME [. <language_code> ] '<identifier>'
                        | DESCRIPTION [. <language_code> ] '<identifier>'
                        | OWNER '<identifier>'
```

Example

```
CREATE PROJECT demo (  
    DESCRIPTION.en 'Demonstration project',  
    DESCRIPTION.fr 'Projet de démonstration',  
    NAME.en 'DEMO PROJECT',  
    OWNER 'Toshiba Corporation'  
);
```

5.1.2. Alter

```
ALTER PROJECT <project_identifier> (ADD | MODIFY | DROP) ([<project_definition>  
    {, <project_definition> }]);
```

In this expression, the specified <project_definition> items will be modified by the statement.

Example

```
ALTER PROJECT demo ADD(  
    NAME.ja 'デモプロジェクト'  
);  
  
ALTER PROJECT demo DROP(  
    DESCRIPTION.fr  
);
```

5.1.3. Drop

```
DROP PROJECT <project_identifier>;
```

5.1.4. Commit

#Work in Progress

5.2. Dictionary

A project may contain one or more dictionaries. If it contains more than one dictionary, it may be that the “case_of” concept of PLIB (in other words, importation of properties from a class of another dictionary) is used. But this is not a prerequisite. Several dictionaries of different coverage may be used in a project for a certain enterprise, in order to delineate the scope of possible choices of products to be used in a

business activity. In this case, dictionaries contained in a project may serve as “authorized sets of products” for the enterprise.

5.2.1. Create

```
CREATE DICTIONARY <dictionary_identifier>([<dictionary_definition>  
                                         {,< dictionary_definition>}]);
```

Where <dictionary_definition> is:

```
<dictionary_definition> ::= DEFINED_BY <supplier_bsu>  
                           | NAME [,<language_code>] '<identifier>'  
                           | DESCRIPTION [,<language_code>] '<identifier>'
```

Example

```
CREATE DICTIONARY Jemima_dictionary (  
    DESCRIPTION.en 'Jemima dictionary',  
    DESCRIPTION.fr 'Dictionnaire Jemima',  
    NAME.en 'DEMO Dictionary',  
    DEFINED_BY '9901/JEMIMA'  
);
```

5.2.2. Alter

```
ALTER DICTIONARY <dictionary_identifier> (ADD | MODIFY | DROP)  
    ([<dictionary_definition> {,< dictionary_definition>}]);
```

5.2.3. Drop

```
DROP DICTIONARY <dictionary_identifier> ;
```

Example

```
DROP DICTIONARY jemima_dictionary;
```

5.3. Supplier

5.3.1. Create

```
CREATE SUPPLIER <supplier_bsu> ([<supplier_definition>  
                                 {,<supplier_definition>}]);
```

Where <dictionary_definition> is:

```
<supplier_definition> ::= ORGANIZATION_ID '<identifier>'
| ORGANIZATION_NAME '<identifier>'
| ORGANIZATION_DESCRIPTION '<identifier>'
| ADDRESS_INTERNAL_LOCATION '<identifier>'
| ADDRESS_STREET_NUMBER '<identifier>'
| ADDRESS_STREET '<identifier>'
| ADDRESS_POSTAL_CODE '<identifier>'
| ADDRESS_TOWN '<identifier>'
| ADDRESS_REGION '<identifier>'
| ADDRESS_COUNTRY '<identifier>'
| ADDRESS_FACSIMILE_NUMBER '<identifier>'
| ADDRESS_TELEPHONE_NUMBER '<identifier>'
| ADDRESS_ELECTRONIC_MAIL_NUMBER '<identifier>'
| ADDRESS_TELEX_NUMBER '<identifier>'
| CODE '<identifier>'
| ADDRESS_POSTAL_BOX '<identifier>'
```

Example

```
CREATE SUPPLIER 9901/JEMIMA ();
CREATE SUPPLIER 9901/JEMIMA (
    ORGANIZATION_NAME 'JEMIMA',
    ADDRESS_COUNTRY 'Japan'
);
```

5.3.2. Alter

```
ALTER SUPPLIER <supplier_bsu> ( ADD | MODIFY | DROP ) ([<supplier_definition>
{,<supplier_definition>}]);
```

5.3.3. Drop

```
DROP SUPPLIER <supplier_bsu> ;
```

5.4. Class

5.4.1. Create

```
CREATE CLASS <class_bsu> [OF <class_type>] ([<class_definition>  
                                         {,< class_definition>}]);
```

Where <class_type> is:

```
<class_type> ::= ITEM_CLASS  
              | COMPONENT_CLASS  
              | MATERIAL_CLASS  
              | ITEM_CLASS_CASE_OF  
              | COMPONENT_CLASS_CASE_OF  
              | MATERIAL_CLASS_CASE_OF
```

#Class_type may be removed from next revision

And where <class_definition> is:

```
<class_definition> ::= <class_attribute>  
                    | <class_and_property_attribute>
```

```
<class_attribute> ::= SUPER_CLASS <class_bsu>  
                    | IS_CASE_OF ( <class_BSU> {, <class_BSU> } )  
                    | VISIBLE_PROPERTIES ( [ <property_bsu> {,<property_bsu> } ] )  
                    | APPLICABLE_PROPERTIES ( [<property_bsu> {,<property_bsu> } ] )  
                    | CLASS_VALUED_ASSIGNMENT  
                    | IS_CASE_OF (<class_bsu> {,<class_bsu> } )  
                    | IMPORTED_PROPERTIES (<property_bsu> {,<property_bsu> } )  
                    | IMPORTED_TABLES (<property_bsu> {,<property_bsu> } )  
                    | IMPORTED_DOCUMENTS (<property_bsu> {,<property_bsu> } )
```

```

<class_and_property_attribute> ::= SYNONYMOUS_NAME ( <synonyms_declaration> )
    | PREFERRED_NAME [.<language_code>] '<identifier>'
    | SHORT_NAME [.<language_code>] '<identifier>'
    | DEFINITION [.<language_code>] '<identifier>'
    | REMARK [.<language_code>] '<identifier>'
    | NOTE [.<language_code>] '<identifier>'
    | SOURCE_DOC_OF_DEFINITION '<identifier>'
    | GRAPHICS '<identifier>'
    | DATE_OF_ORIGINAL_DEFINITION '<identifier>'
    | DATE_OF_CURRENT_VERSION '<identifier>'
    | DATE_OF_CURRENT_REVISION '<identifier>'

```

```

<synonyms_declaration> ::= <translated_string_value> {, <translated_string_value> }

```

```

<translated_string_value> ::= ( MEANING.<language_code> '<identifier>' )

```

#Work in Progress for <translated_string_value>

Example

```

CREATE CLASS Jemima_dictionary.9901/JEMIMA.Temperature_meter (
    SUPER_CLASS Jemima_dictionary.9901/JEMIMA.Meter,
    PREFERRED_NAME.en 'Temperature Meter'
    APPLICABLE_PROPERTIES (
        Jemima_dictionary.9901/JEMIMA. Temperature_meter.range,
        Jemima_dictionary.9901/JEMIMA. Temperature_meter.diameter
    )
);

```

5.4.2. Alter

```

ALTER CLASS <class_bsu> ( ADD | MODIFY | DROP ) ([<class_definition>
                                                                    {,< class_definition>}]);

```

Example

```

ALTER CLASS Jemima_dictionary.9901/JEMIMA.Temperature_meter ADD(
    PREFERRED_NAME.fr 'Thermomètre'
);

```

5.4.3. Drop

```
DROP CLASS <class_bsu>;
```

Example

```
DROP CLASS Jemima_dictionary.9901/JEMIMA.Temperature_meter;
```

5.4.4. Rename

This command is to change the class_bsu_code.

```
RENAME CLASS <class_bsu> TO <class_bsu>;
```

Example

```
RENAME CLASS Jemima_dictionary.9901/JEMIMA.Temperature_meter  
TO Jemima_dictionary.9901/JEMIMA.TMTR ;
```

5.5. Property

5.5.1. Create

```
CREATE PROPERTY <property_bsu> ([<property_definition>  
                                {,< property_definition>}]);
```

And where <property_definition> is:

```
<property_definition> ::= <property_attribute>  
                        | <class_and_property_attribute>
```

```
<property_attribute> ::= DATA_TYPE <data_type>  
                        | VALUE_FORMAT '<identifier>'  
                        | UNIT '<identifier>'  
                        | CURRENCY '<identifier>'
```

Example

```
CREATE PROPERTY Jemima_dictionary.9901/JEMIMA.THING.PRODUCT_NAME (  
    PREFERRED_NAME.en 'Product Name',  
    PREFERRED_NAME.ja '製品名',  
    NAME_SCOPE 'Jemima_dictionary.9901/JEMIMA.THING',  
    DATA_TYPE 'string_type'  
);
```

```

CREATE PROPERTY Jemima_dictionary.9901/JEMIMA.THING.COLOR (
  PREFERRED_NAME.en 'COLOR',
  PREFERRED_NAME.ja '色',
  NAME_SCOPE 'Jemima_dictionary.9901/JEMIMA.THING',
  DATA_TYPE ENUMERATION(CODE, MEANING.en, MEANING.ja)
    (('BLACK','black','黒'),
    ('WHITE','white','白'),
    ('SILVER','silver','銀'))
);

```

5.5.2. Alter

```

ALTER PROPERTY <property_bsu> ( ADD | MODIFY | DROP) ([<property_definition>
                                                                    {,< property_definition>}]);

```

Example

```

ALTER PROPERTY Jemima_dictionary.9901/JEMIMA.THING.COLOR (
  DATA_TYPE STRING_TYPE
);

```

5.5.3. Drop

```

DROP PROPERTY <property_bsu>;

```

Example

```

DROP PROPERTY Jemima_dictionary.9901/JEMIMA.THING.COLOR;

```

5.5.4. Rename

```

RENAME PROPERTY <property_bsu> TO <property_bsu>;

```

Example

```

RENAME PROPERTY Jemima_dictionary.9901/JEMIMA.THING.COLOR
  TO Jemima_dictionary.9901/JEMIMA.THING.COLOR02;

```

6. CQL: a Data Representation Language

6.1. Extension

6.1.1. Create

```

CREATE EXTENSION ON <class_bsu> ;

```

This command creates an extension on the specified class, thus enabling the creation of tables and therefore the injection of instances

Example

```
CREATE EXTENSION ON Jemima_dictionary.9901/JEMIMA.THING;
```

6.1.2. Drop

```
DROP EXTENSION ON <class_bsu>;
```

Example

```
DROP EXTENSION ON Jemima_dictionary.9901/JEMIMA.THING;
```

6.2. Table

6.2.1. Create

```
CREATE TABLE <table_identifier> ON <class_bsu> ( <table_definition> );
```

Where <table_definition> is:

```
<table_definition> ::= ( <property_bsu> | <constraint> ) {, ( <property_bsu> | <constraint> ) }
```

```
<constraint> ::= CONSTRAINT KEY ( <property_bsu> {, <property_bsu> } )
```

The creation of a table is only possible as long as the specified class has an extension.

Example

```
CREATE TABLE various_things ON Jemima_dictionary.9901/JEMIMA.THING (
    Jemima_dictionary.9901/JEMIMA.THING.COLOR,
    Jemima_dictionary.9901/JEMIMA.THING.PRODUCT_NAME,
    CONSTRAINT KEY(
        Jemima_dictionary.9901/JEMIMA.THING.PRODUCT_NAME
    )
);
```

6.2.2. Alter

```
ALTER TABLE <table_identifier> ON <class_bsu>
    ( ADD | MODIFY | DROP ) ( <table_definition> );
```

Example

```
ALTER TABLE various_things ON Jemima_dictionary.9901/JEMIMA.THING DROP (
    Jemima_dictionary.9901/JEMIMA.THING.COLOR
);
```

6.2.3. Drop

```
DROP TABLE <table_identifier> ON <class_bsu> ;
```

Example

```
DROP TABLE various_things ON Jemima_dictionary.9901/JEMIMA.THING;
```

7. CQL: a Library Manipulation Language

7.1. Insert

```
INSERT INTO <table_name> ON <class_bsu> ( <property_list> )
                                     VALUES ( <value_list> );
```

Where the other terms are:

```
<property_list> ::= <property_bsu>[.<option>] {, <property_bsu>[.<option>]}
```

```
<value_list> ::= <value_declaration> [, <value_list> ]
```

```
<value_declaration> ::= ' <identifier> '
                       | <number>
                       | <collection_instance>
                       | <translated_string_value>
```

```
<collection_instance> ::= [ <value_list> ]
                       | ( <value_list> )
                       | [ <index> <value_declaration> {, <index> <value_declaration> } ]
                       | ( <index> <value_declaration> {, <index> <value_declaration> } )
```

```
<index> ::= <number> | ' <identifier> '
```

#NB: Work in Progress (for heterogeneous and partially ordered collections)

Example

```
INSERT INTO various_things ON Jemima_dictionary.9901/JEMIMA.THING (
    Jemima_dictionary.9901/JEMIMA.THING.COLOR,
    Jemima_dictionary.9901/JEMIMA.THING.PRODUCT_NAME
) VALUES (
    'red',
    'anything'
);
```

7.2. Update

```
UPDATE <table_name> ON <class_bsu> SET <property_update_list>
[ WHERE <logical_term> ]
```

Where the other terms are:

```
<property_update_list> ::= <property_update> { , <property_update> }
```

```
<property_update> ::= <property_bsu>[.<option>] = <value_declaration>
```

```
<logical_term> ::= <logical_factor>
| <logical_term> <operation> <logical_term>
| NOT <logical_term>
| ( <logical_term> )
```

```
<logical_factor> ::= <attribute_reference> ( = | != | LIKE ) '<identifier>'
| <mathematical_expr> <comparison_operation> <mathematical_expr>
| <collection_condition>
```

```
<operation> ::= AND
| OR
| ,
```

```

<comparison_operation> ::= =
    | !=
    | >
    | >=
    | <
    | <=

```

```

<attribute_reference> ::= <property_bsu>[.<option>]
    | <collection_element>[.<option>]
    | <identifier>

```

```

<collection_element> ::= <property_bsu>[ <number> | '<identifier>' ]
    | <property_bsu>( <number> | '<identifier>' )
    | <collection_element> [ <number> | '<identifier>' ]
    | <collection_element> ( <number> | '<identifier>' )

```

```

<mathematical_expr> ::= ( <mathematical_expr> )
    | <mathematical_expr> <mathematical_operator> <mathematical_expr>
    | <number>
    | <attribute_reference>

```

```

<mathematical_operator> ::= + | * | - | ^ | /

```

```

<collection_condition> ::= ALL_OF ( <attribute_reference> , <identifier> , <logical_factor> )
    | SOME_OF ( <attribute_reference> , <identifier> , <logical_factor> )

```

<collection_condition> provides a useful mechanism to specify conditions on collections.

ALL_OF embodies the mathematical \forall whereas **SOME_OF** embodies \exists .

If C is the BSU of a property of type COLLECTION, the following expressions

$$\forall x \in C, x > 4$$

$$\exists x \in C, x = \text{'TOSHIBA'}$$

would be expressed in CQL as follow:

$$\text{ALL_OF} (C , x , x > 4)$$

$$\text{SOME_OF} (C , x , x = \text{'TOSHIBA'})$$

The second parameter is used to give a virtual name to an element of the collection, allowing it to be referenced in the third parameter, i.e. the condition on the element itself.

Example

```
UPDATE various_things ON Jemima_dictionary.9901/JEMIMA.THING
SET Jemima_dictionary.9901/JEMIMA.THING.COLOR = 'blue'
WHERE Jemima_dictionary.9901/JEMIMA.THING.PRODUCT_NAME LIKE '%thing';
```

7.3. Delete

DELETE FROM <table_name> **ON** <class_bsu> [**WHERE** <logical_term>]

Example

```
DELETE FROM various_things ON Jemima_dictionary.9901/JEMIMA.THING
WHERE Jemima_dictionary.9901/JEMIMA.THING.COLOR LIKE 'blue'
OR Jemima_dictionary.9901/JEMIMA.THING.COLOR LIKE 'red';
```

8. Queries in CQL

Queries in CQL can be of two types :

- on content
- on dictionary information

But in fact, since every entity in CQL is a class, both types of queries are very much alike.

8.1. Content queries

SELECT [**DISTINCT**] <select_attribute> {,<select_attribute>} **FROM** <search_scope>
[**WHERE** <logical_term>]
[**ORDER BY** <order_term>]
[<instance_limitation>]

In this select command, the terms are defined as follow:

```
<select_attribute> ::= <property_BSU>
| <class_BSU>.*
| *
| <function_name> ( (<property_BSU> | *) )
```

```

<function_name> ::= COUNT
                |   MAX
                |   MIN
                |   AVG
                |   SUM

```

This functions have the same specifications as their SQL equivalents.

```

<search_scope> ::= <class_BSU> [.<table_name> ] [ <direction_modifier> ]
                [ <search_scope_operator> <search_scope> ]

```

```

<direction_modifier> ::= *
                    |   %
                    |   $
                    |   !

```

```

<search_scope_operator> ::= AND
                        |   &
                        |   +
                        |   -
                        |   ,

```

“AND” and “&” stand for the intersection operation between two search scopes. “+” and “,” stand for the union operation and “-“ stands for the subtraction operation.

```

<order_term> ::= <property_BSU> [ ASC | DESC ] { , <property_BSU> [ ASC | DESC ] }

```

Equivalent to the SQL ordering. Several criteria can be specified.

```

<instance_limitation> ::= INSTANCENUM ( < | <= ) <number>

```

This is a mechanism to limit the number of instances in the result of a query. Note that the effects of this limitation on the result are non predictable as long as the result is not ordered.

Note on <select_attribute>:

there are 4 different kinds of select attributes. One can specify an explicit property BSU, or a class BSU followed by “.*”, this is used to declare as select attributes all the properties declared applicable at the level of the specified class. One can also use “*” which declares all the applicable properties found in the search scope as select attributes. Finally, one can use functions that take a property BSU as parameter. Those functions are the same as the ones found in SQL.

Example

```
SELECT Jemima_dictionary.9901/JEMIMA.THING.*, count(*)
FROM Jemima_dictionary.9901/JEMIMA.ROOTS
     - Jemima_dictionary.9901/JEMIMA.Furnitures*
WHERE Jemima_dictionary.9901/JEMIMA.THING.PRODUCT_NAME LIKE '%thing'
ORDER BY Jemima_dictionary.9901/JEMIMA.THING.Weight ASC;
```

8.2. Queries on dictionary information

#Work in Progress

```
SELECT [ DISTINCT ] <dic_attribute> {,<dic_attribute>} FROM <dic_entities>
                                     [ WHERE <dic_logical_term> ]
                                     [ ORDER BY <dic_order_term> ]
                                     [ <instance_limitation> ]
```

```
<dic_attribute> ::= APPLICABLE_TYPES
                   | APPLICABLE_PROPERTIES
                   | BSU_CODE
                   | CLASS_VALUED_ASSIGNMENT
                   | CURRENCY
                   | DATE_OF_ORIGINAL_DEFINITION
                   | DATE_OF_CURRENT_REVISION
                   | DATE_OF_CURRENT_VERSION
                   | DATA_TYPE
                   | DEFINITION [ . <language_code> ]
                   | GRAPHICS
                   | IS_CASE_OF
                   | IMPORTED_PROPERTIES
                   | IMPORTED_TABLES
                   | IMPORTED_DOCUMENTS
```

```

| NAME_SCOPE
| NOTE [ . <language_code> ]
| PREFERRED_NAME [ . <language_code> ]
| REMARK [ . <language_code> ]
| SHORT_NAME [ . <language_code> ]
| SOURCE_DOC_OF_DEFINITION
| SUPER_CLASS
| SYNONYMOUS_NAME
| UNIT
| VALUE_FORMAT
| VISIBLE_PROPERTIES
| *

```

```

<dic_entities> ::= <dic_entity> { "," <dic_entity> }

```

```

<dic_entity> ::= PROJECT
| CLASS
| DICTIONARY
| PROPERTY
| SUPPLIER
| GROUP

```

```

<dic_logical_term> ::= NOT <dic_logical_term>
| ( <dic_logical_term> )
| <dic_logical_term> <operation> <dic_logical_term>
| <dic_logical_factor>

```

```

<dic_logical_factor> ::= <dic_attribute> <comparison_operation>
( '<identifier>' | <number> )

```

9. Transactions in CQL

#Work in progress

10. Versioning and Data Warehouse capabilities

#Work in progress

11. Security and Access control

In the LMS, each user belongs to a user group. CQL provides mechanisms to grant or revoke privileges to specific user groups. Since everything is a class in CQL, user groups are organized hierarchically, thus providing inheritance mechanisms for the privileges. Therefore, user groups that have the most privileges are found at the bottom of the hierarchy. Note that the system administrator is a special user and does not belong to the hierarchy.

The LMS grants a privilege to a user group (a class under *User*) to do a specific operation (“select”, “alter”, etc) on specific data. The system applies security controls to the class (user group) from the three following viewpoints:

- 1) Class
- 2) Property
- 3) Instance

11.1. Grant

```
GRANT <privilege_type> ON <control_viewpoint> TO <user_group_BSU>  
[ WHERE <search_condition> ] ;
```

Where the other terms are defined as follow:

```
<privilege_type> ::= <command_name> | <role_name>
```

```
<command_name> ::= ALTER  
| COMMIT  
| CREATE  
| DELETE  
| DROP  
| INSERT  
| RENAME  
| SELECT  
| UPDATE  
| ALL
```

```
<controle_viewpoint> ::= <class_BSU> | <property_BSU>
```

<role_name> is a collection of <command_name> items. Implementers are allowed to define <role_name> and their command set in each LMS.

The following is a sample of <role_name>

```
<role_name> ::= " Administrator"  
            | "CLASSEditor"  
            | "OrganizationAdministrator"  
            | "Editor"  
            | "CertifiedUser"
```

#Work in progress

11.2. Revoke

```
REVOKE <privilege_type> ON <controle_viewpoint> FROM <user_group_BSU>  
            [ WHERE <search_condition> ] ;
```

11.3. User Group

11.3.1. Create

```
CREATE USER_GROUP <user_group_BSU> ( [<user_group_definition  
            {,<user_group_definition>} ] ) ;
```

Where the other terms are:

```
<user_group_BSU> ::= <identifier>
```

```
<user_group_definition> ::= SUPER_GROUP <user_group_BSU>  
            | DESCRIPTION[.<language_code>] '<string>'
```

11.3.2. Drop

```
DROP <user_group_BSU> ;
```

11.3.3. Alter

```
ALTER USER_GROUP <user_group_BSU> ( ADD | MODIFY | DROP )  
    ( [<user_group_definition {,<user_group_definition>} ] );
```

11.4. User Management

#Work in progress