Document Number: DSP0202

Date: 2007-08-13

Version: 1.0.0

CIM Query Language Specification

Document Type: Specification

Document Status: Final

Document Language: E

Copyright notice

CONTENTS

Tables

Foreword

The *CIM Query Language Specification* (DSP0202) was prepared by the DMTF WBEM Infrastructure & Protocols Working Group.

DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability.

Introduction

Common Information Model (CIM) and Web-Based Enterprise Management (WBEM) support a query mechanism that is used to select sets of properties from CIM object instances. Query support is available in some operations defined by the *CIM Operations over HTTP Specification* (DSP0200) and some CIM classes within the *CIM Event Model v2.14* and *CIM Policy Model v2.14*. Query definitions allow a WBEM client to specify the nature and the number of instances that are selected and what information is returned from those instances. This enables a WBEM-managed environment to place less burden on the network infrastructure. The precise mechanics for delivering query requests and receiving query results are specified as a part DSP0200.

A CIM service implements a Query Engine to parse the query and evaluate its results. Parsing enables the server to understand the query sufficiently to determine where it should be processed (even if the query is executed by some other process acting as a data provider for the server). The Query Language is divided into a base level of functionality and a number of optional features, which determine the complexity of the syntax and semantics. These features enable CIM service implementations, especially in simple or resource-sensitive installations, to support a query interpreter that best suits the needs of clients while also taking the capabilities of the server into account.

CIM implementations that support query may also support a query template mechanism. A query template can be used to model a generic query and can be processed into a valid query. An optional pre-processing facility may be implemented to convert a valid query template into a valid query string. This feature allows for the writer of a query template to provide a model for a query but defer the decision on specific query elements to a processing point further along. It is important to note that the query template language can be used to support the query engine, but the query template language is not part of the formal query language itself.

The CIM query design is based on concepts from both the ISO/IEC Structured Query Language (SQL-92) and the W3C XML Query. Basic understanding of the use of relational databases is required. However, specific knowledge of these other works is not required in order to understand the CIM Query Language.

CIM Query Language Specification

1 Scope

The DMTF Common Information Model (CIM) uses a basic object-oriented structure and conceptualization techniques in its approach to managing hardware, software, systems, and networks. This approach provides a formal consistent model that enables cooperative development of an object-oriented schema across multiple organizations and problem domains.

This document describes a query language used to extract data from a CIM-based management infrastructure.

2 Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2.1 Approved References

DMTF, DSP0200, *CIM Operations over HTTP Specification* v1.2, 2003.

DMTF, DSP0201, *Specification for the Representation of CIM in XML*, v2.2 2007.

DMTF, *CIM Event Model v2.14* , 2006.

DMTF, *CIM Policy Model v2.14* , 2006.

DMTF DSP0004, *CIM Infrastructure Specification v2.3.0, 2005.*

DMTF DSP0207, *WBEM URI Specification v1.0, 2006.*

2.2 Other References

ISO/IEC 9075:1992, *Database Language SQL*, July 30, 1992 (see http://www.iso.org for the latest version).

W3C, *XML Query (XQuery)*, 2001.

IETF, RFC 2234, *Augmented BNF for Syntax Specifications: ABNF*, 1997.

IETF, RFC 3629, *UTF-8, a transformation format of ISO 10646*, 2003.

IETF, RFC 1034, *Domain Names – Concepts and Facilities*, 1987.

IETF, RFC 1123, *Requirements for Internet Hosts – Application and Support*, 1989.

Unicode, Inc., *Unicode Technical Standard #10: Unicode Collation Algorithm*, 2006.

ISO/IEC 14651, *Information Technology – International string ordering and comparison – Method for comparing character strings and description of the common template tailorable ordering*, 2000.

ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*, Fifth edition, 2004.

3 Terms and Definitions

The terms, definitions, and constraints on terms of ISO/IEC Directives, Part 2 apply to this document. In particular, the constraints on the use of the words shall, shall not, must, should, should not, may, need not, possible, impossible, can, and cannot apply to this document.

For the purposes of this document, the following additional terms and definitions apply.

3.1 conditional

Indicates requirements to be followed strictly in order to conform to the document when the specified conditions are met

3.2 mandatory

Indicates requirements to be followed strictly in order to conform to the document and from which no deviation is permitted

3.3 optional

Indicates a course of action permissible within the limits of the document

3.4 unspecified

Indicates that this profile does not define any constraints for the referenced CIM element or operation

3.5 Common Information Model (CIM)

An object-oriented definition of a managed enterprise or Internet environment

3.6 CIM indications

A CIM class hierarchy, starting at `CIM_Indication`, which defines the data in various types of management notifications

3.7 CIM service

A service that provides access to CIM object instances

3.8 `From-criteria`

A definition of the range of data over which a query is conducted

3.9 Query

The act of asking for specific data; for the purposes of this document, a query specifies the range of data of interest (the `from-criteria`), the conditions under which data should be returned in the query result (the `search-condition`), and the specific data to be returned (the `select-list`), as well as other processing options

3.10 `Search-condition`

A specification of the criteria or conditions that select data to be returned in a query result

3.11 Select list

A definition of the specific data to be returned in a query result

3.12 WBEM protocol

A protocol specified by the DMTF for accessing a CIM service over the Internet (One of these is defined by DSP0200)

3.13 WBEM service

A CIM service that supports WBEM protocol interfaces

3.14 XML-Query
An XML-based Query Language from W3C

4 Symbols and Abbreviated Terms

The following symbols and abbreviations are used in this document.

4.1 ABNF
Augmented Backus-Naur Form

4.2 BNF
Backus-Naur Form

4.3 CIM
Common Information Model

4.4 CQL
CIM Query Language

4.5 CQLT
CIM Query Template Language

4.6 MOF
Managed Object Format

4.7 SQL
Structured Query Language

4.8 WBEM
Web-Based Enterprise Management

4.9 WQL
WBEM Query Language

5 Requirements and Concepts

The CIM Query Language (CQL) has been exploited in the *CIM Operations over HTTP Specification* (DSP0200), by the *CIM Event Model v2.14*, and by the *CIM Policy Model v2.14*. The language defines the desired instance-level data ranging over a certain set of objects to be returned as the result of an ExecQuery, OpenQueryInstances (see DSP0200, *CIM Operations over HTTP Specification* v1.3), CIM operation. CQL also defines the conditions and data for Indications returned as a result of one of the following events:

- subscription to `CIM_IndicationFilter` within the event model
- use of `CIM_QueryCondition` or `CIM_MethodAction` instances used within a `CIM_PolicySet`

Query semantics shall include instance property projection (for example, a SQL SELECT clause) and a range (for example, a SQL FROM clause) and may include predicate logic (for example, a SQL Where clause). This support (defined specifically using the keywords Select-From-Where) was included in a preliminary version of the *CIM Query Language Specification*, called the WBEM Query Language (WQL), and implemented in various code bases. Although the preliminary specification was never released, it is

important to maintain these keywords and concepts (unless a critical performance or operational error is found), in order to prevent unnecessary code churn.

The instance property projection, which shall be supported as mentioned previously, is a mechanism to select particular properties from a class to be included in a query response or Indication object. The projection may include "static" entries that can be used for tagging the response or Indication object. (These requirements are provided by the specific or `array class-property-identifier` and `select-string-literal` constructs, respectively.) In addition, the CQL shall:

- Support the ability to project metadata, such as instance name and instance class into a response. See 7.2.4 for descriptions of the OBJECTPATH() and CLASSPATH() methods, respectively.
- Support the ability to query class versioning information (see the `CLASSQUALIFIER` production in 6.4.6).
- Define and support a mechanism for querying class inheritance or hierarchy in a query predicate (provided using the ISA operator).
- Support the ability to query all data types as well as the entries of an array, because CIM defines arrays of simple data types as valid class properties.

Various other requirements for the query language have arisen over the last few years, as work on the Event Model continued. Additional Event Model requirements are specific to Indication processing, but they must be defined in the basic query language in order to have a consistent BNF and query engine. These requirements are as follows:

- the ability to set a returned property value (such as an Indication Priority, which could be overridden by a customer)
- the ability to specify a constant value set of properties to be returned
- Support for accessing property `VALUES` of an `EMBEDDEDOBJECT`

CQL is designed to operate on instances of one or more classes. Query operations on the schema are not in the scope of CQL. However, referencing a certain set of class-level information such as class names or qualifier `VALUES` is supported within the "Extended Select List" feature.

CQL shall support polymorphism. This means that if a query is issued against a base class, all derived class instances will be considered as well. For instance, consider the following SELECT statement:

```
SELECT *
FROM CIM_Indication
```

This SELECT statement would match all instances of derived classes of `CIM_Indication`.

6 CIM Query Language (CQL)

In its simplest form, the CQL is a subset of SQL-92 with some extensions specific to CIM. It supports queries specified as follows:

```
SELECT <select-list>
FROM <class-list>
WHERE <selection expression>
```

Where:

- A <select-list> is a comma-separated list of any of the following:
  - CIM property names (optionally qualified by their class name) related to the individual classes specified in the FROM clause. The asterisk (*) can be used to specify all the properties of a class. The resultant column is named by the property name, but this may be modified using the AS keyword followed by a new name.
  - Literals, named through the AS keyword followed by a name.
  - Function results, named through the AS keyword followed by a name.
- The <class-list> is a comma-separated list of class names.
- A <selection expression> specifies the criteria by which results are selected. It is limited to relatively simple property comparisons.

Moving beyond the simple SELECT-FROM-WHERE format, the ORDER BY functionality of SQL is added. Other capabilities of the language, unique to CIM, are as follows:

- the ability to process arrays through indices
- the ability to query the properties of embedded objects
- the ability to traverse associations (based on the VALUES of their REF properties)

Queries are used to define the operation of some CIM classes (for example, CIM_IndicationFilter, CIM_MethodAction and CIM_QueryCondition). If using CIM operations, a client may issue a query through the ExecQuery or OpenQueryInstances operations if these operations are supported.

CQL operates on instances of one or more class. Operations against the set of classes are not supported. Some class-level information such as class names and qualifier VALUES are folded into the instances.

6.1 Identifying the CIM Query Language

In order to ensure uniqueness, valid VALUES for query-language should conform to the following syntax:

```
<organizationId>":"<languageId>
```

<organizationId> shall not include a colon (:) and shall include a copyrighted, trademarked, or otherwise unique name that is owned by the entity that has defined the query language. For DMTF-defined query languages, the <organizationId> is "DMTF".

The <languageId> shall include a unique (in the context of the identified organization) name for the query language.

Following this convention, the string "DMTF:CQL" identifies the CIM Query Language.

6.2 Query Language Type Lattice

The CQL type system incorporates the type system of the *CIM Infrastructure Specification v2.3.0* (DSP0004), but it also extends that type system as follows:

- For every class *C,* there is an "object of *C*" type, whose VALUES may be either
  - instances of *C* (including instances of any subclasses of *C*)
  - the class *C* itself, or one of *C*'s subclasses

  NOTE: Classes arise as CQL VALUES only when they appear as embedded objects, and that support for embedded objects is an optional feature of CQL. CQL implementations that do not support embedded objects may consider the VALUES for "object of C" to be limited to instances of C (including instances of any subclasses of C).
- The "object of C" types recapitulate the CIM class hierarchy, in that if C1 is a superclass of C2 then "object of C1" is a supertype of "object of C2".
- For every class *C,* there is an "object" type that is a supertype of "object of *C*" type.
- For every class *C,* there is a "reference" type that is a supertype of "C REF" type.
- There is a "`boolean`" type.
- There is an "`unsigned integer`" type that is a supertype of `uint8`, `uint16`, `uint32`, and `uint64`.
- There is a "`signed` integer" type that is a supertype of `sint8`, `sint16`, `sint32`, and `sint64`.
- There is an "integer" type that is a supertype of `unsigned integer` and `signed integer`.
- There is a "`real`" type that is a supertype of `real32` and `real64`.
- There is a "`numeric`" type that is a supertype of `integer` and `real`.

NOTE 1: CIM defines a "`datetime`" type, which contains either timestamp or interval VALUES . CIM does not explicitly define timestamp and interval, but they are defined in Annex D. A timestamp with the year field set to "0000" is interpreted as the year "1 BCE". A year field set to "0001" is interpreted as the year "1 CE".

NOTE 2: There is a "string" type that is the CIM datatype string. It contains a sequence of Unicode characters. The range of allowed code points is the same as the CIM datatype string. The encoding form is defined by the specification that is using CQL.

NOTE 3: There is a "char16" type that is the CIM datatype char16. It contains one Unicode character. The range of allowed code points is the same as the CIM datatype char16. The encoding form is defined by the specification that is using CQL.

DSP0004 also defines a system of array types, which is similarly extended. In that system every non-array type, *T,* in the CQL type lattice has a corresponding array type, array of *T.* The structure of the array type lattice exactly matches that of the non-array types (that is, if T_1 and T_2 are non-array types, then array of T_1 is a supertype of array of T_2 if and only if T_1 is a supertype of T_2).

CQL expressions are assigned types according to the rules described in the subsequent clauses. Any CQL construct that has been assigned a particular type is said also to "have" all the supertypes of that type. (For example, an expression that has been assigned type "object of `CIM_ManagedElement`" also "has" type "object".)

6.3 Query Functional Description

CIM environments vary greatly in terms of processing capabilities and required functionality. CQL is segmented based on functionality, with the assumption that a reduction in functionality is equivalent to reduced processing requirements.

Section 6.4 defines the features required for CQL support. The subsections of Annex E each describe the BNF for optional experimental CQL features.

Discovery of CQL features is enabled through the CQLFeatures enumeration property of the QueryCapabilities class. Each optional feature shall be fully supported before it is advertised as being supported.

If a query includes valid clauses or constructs that are not supported by the infrastructure, the error `CIM_ERR_INVALID_QUERY` shall be returned on a request made through `ExecQuery, the error CIM_ERR_QUERY_FEATURE_NOT_SUPPORTED` shall be returned on a request made through `OpenQueryInstances,` or the error `CIM_ERR_FAILED` shall be returned for all other CIM operations.

If a query includes invalid clauses or constructs, the error `CIM_ERR_INVALID_QUERY` MUST be returned on a request made through `ExecQuery or OpenQueryInstances,` or the error `CIM_ERR_FAILED` must be returned for all other CIM operations.

6.4 Query Language Grammar

The CQL grammar in the following subclauses shall be supported by all implementations conforming to this specification. The grammar is described using the BNF defined in 0. As much as possible, this grammar is constructed to be LALR(1)-parsable.

6.4.1 Reserved Words

The following words are reserved for CQL. A property name that is a reserved word shall be scoped by className, (see the E.15).

```
AND = "AND"
ANY  =  "ANY"
AS  =  "AS"
ASC  =  "ASC"
BY  =  "BY"
CLASSQUALIFIER = "CLASSQUALIFIER"
DESC  =  "DESC"
DISTINCT  =  "DISTINCT"
EVERY  =  "EVERY"
FALSE  =  "FALSE"
FIRST  =  "FIRST"
FROM  =  "FROM"
IN  =  "IN"
IS  =  "IS"
ISA  =  "ISA"
LIKE  =  "LIKE"
NOT  =  "NOT"
NULL  =  "NULL"
OR  =  “OR”
ORDER  =  "ORDER"
PROPERTYQUALIFIER  =  "PROPERTYQUALIFIER"
SATISFIES  =  "SATISFIES"
SELECT  =  "SELECT"
TRUE  =  "TRUE"
```

```
WHERE  =  "WHERE"
```

6.4.2 Identifiers

The following productions define how identifiers are represented in CQL.

```
identifier-start  =  UNICODE-S1
```

```
identifier-subsequent  =  identifier-start  |  DECIMAL-DIGIT
```

```
identifier  =  identifier-start,  *( identifier-subsequent )
```

6.4.3 Class Paths

The following productions define how class paths are represented in CQL.

```
class-name  = identifier
```

The `identifier` shall be in accordance with the definition of className in DSP0004.

```
class-path  = class-name
```

6.4.4 Numeric Literals

The following productions define how numeric literals are represented in CQL. The numeric literals are intended to agree with the numeric literals of MOF, as defined in DSP0004.

```
sign = "+"  |  "-"
```

```
binary-digit = "0" | "1"
```

```
binary-value = [sign]  1*( binary-digit )  "B"
```

Because ABNF is not case sensitive, this production defines both upper and lower case.

```
decimal-digit = binary-digit  | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
"9"
```

```
hex-digit = decimal-digit  | "A" | "B" | "C" | "D" | "E" | "F"
```

Because ABNF is not case sensitive, this defines both upper and lower case.

```
hex-digit-value = [sign]  "0X"  1*( hex-digit )
```

Because ABNF is not case sensitive, this defines both upper and lower case.

```
unsigned-integer = 1*( decimal-digit )
```

```
decimal-value = [sign] unsigned-integer
```

```
exact-numeric = unsigned-integer  "."  [unsigned-integer] | "."  unsigned-integer
```

```
real-value = [sign] exact-numeric ["E" decimal-value]
```

Because ABNF is not case sensitive, this defines both upper and lower case.

6.4.5 String Literals

The following productions define how string literals are represented in CQL.

```
single-quote  =  "'"
```

```
literal-string  =  single-quote,  *( UNICODE-CHAR | char-escape ,  single-quote )
```

The use of **char-escape** is mandatory for the non-printable Unicode characters that these escape sequences represent.

```
char-escape ="\", ( "\" | single-quote | "b" | "t" | "n" | "f" | "r" | ( "u",        4*4( hex-digit ) ) |  ( "U", 8*8( hex-digit ) ))
```

The escape characters directly following the initial backslash are case sensitive, even though ABNF is not case sensitive. The meaning of these escape characters is as follows:

\\ – Backslash (U+005C)

\' – Single Quote (U+0027)

\b – Backspace (U+0008)

\t – Horizontal Tab (U+0009)

\n – Line Feed (U+000A)

\f – Form Feed (U+000C)

\r – Carriage Return (U+000D)

\u<hex> – One Unicode character, with <hex> being exactly 4 hexadecimal digits in any lexical case, to be interpreted as a Unicode code point.

NOTE: The hexadecimal value is not in an encoded form, but it is given as a code point.

\U<hex> – One Unicode character, with <hex> being exactly 8 hexadecimal digits in any lexical case, to be interpreted as a Unicode code point.

NOTE: The hexadecimal value is not in an encoded form, but it is given as a code point. The range of allowed code points is \u0 to \u10FFFF, unless restricted by the range of the char16 CIM datatype.

NOTE: The escaping of double quotes is not necessary within a literal string, because only single quotes can be used to delimit string literals. If the entire CQL string is put into an environment that uses double quotes to delimit a string (for example, as a default value for properties in the MOF), then that environment must define the escape rules for double quotes.

6.4.6 Expressions

Expressions describe the calculation of VALUES used in the SELECT and WHERE clauses.

```
literal = literal-string
```

A literal-string has a string type.

```
|  decimal-value
```

A decimal-value has an integer type.

```
|  binary-value
```

A binary-value has an integer type.

```
|  hex-digit-value
```

A hex-digit-value has an integer type.

```
|  real-value
```

A real-value has real type.

```
|  TRUE | FALSE
```

These literals have a Boolean type. Because ABNF is not case sensitive, this defines both upper and lower case.

```
|  "{" [[literal] *( "," [literal] )] "}"
```

All literals in the list shall be of the same type: (string, integer, real, or boolean).

```
arg-list = "*" |  expr
```

This production defines the basic arguments that are allowed in a query language function.

```
chain = literal
```

The type of the literal is taken as the type for this production.

```
|  "(" expr ")"
```

The type of the expr is taken as the type for this production.

```
|  identifier
```

The identifier is interpreted as one of the following:

- If the identifier matches the name bound by an enclosing SATISFIES production for array-compcomp, then the identifier is treated as a variable whose type is determined by the SATISFIES expression. Variables bound by a SATISFIES expression are described at that production.

- If the `identifier` matches a class alias that appears in a FROM criterion on a class *C*, then the `identifier` refers to an instance of *C* and has a type of object of *C*.
- If the `identifier` matches the name of a class *C* that appears in a FROM criterion without a class alias, then the `identifier` refers to an instance of *C,* and has a type of object of *C*.
- If exactly one property defined by the CIM classes in the FROM clause, or their superclasses, matches `identifier`, then the `identifier` refers to that property (see 7.1.1), and the type of the `identifier` is determined by that property.
- For Basic Query, properties qualified with `EMBEDDEDINSTANCE` or `EMBEDDEDOBJECT` shall be treated as type character string.
- If the "Embedded Properties" query feature is supported, then the ability to directly access properties of the embedded instance shall be supported. Otherwise, the query is invalid.
- If type is Array, then this form without a following "`[`" is equivalent to "`Identifier [ * ]`", and only "`=`" and "`<>`" comparisons are allowed.

```
|  identifier "#" literal-string
```

`identifier` shall unambiguously identify a property (see 7.1.1). The type of the property is taken as the type of this production. This production forms a symbolic constant based on the `VALUES` and `VALUEMAP` qualifiers (see 7.1.4).

```
|  chain "." identifier
```

`chain` shall have a type of object of *C* for some class *C*.

`identifier` shall be the name of a property. For details on the selection of the identified property, see 7.1.1. The type of this production is the type of the property.

For Basic Query, `chain` is restricted to a single class name or class alias bound in the `FROM` clause because Basic Query does not support extraction of properties from embedded objects.

```
|  chain "."  identifier "#" literal-string
```

`chain`, `property-scope` (if present), and `identifier` together identify a property, as described in 7.1.1. This production forms a symbolic constant based on the `VALUES` and `VALUEMAP` qualifiers (see 7.1.4). The type of this expression is the type of the identified property.

For Basic Query, `chain` is restricted to a single class name or class alias bound in the FROM clause because Basic Query does not support extraction of properties from embedded objects.

```
|  chain "[" array-index-list "]"
```

`chain` shall have type array of *T*. If `array-index`-list comprises just a single expression, then this production has type *T*; otherwise, the production has type array of *T*.

```
concat = chain
```

The type of the `chain` is taken as the type of this production.

```
|  concat "||" chain
```

`concat` and `chain` shall have a type of string or char16, and the result has string type.

```
factor = concat
```

The type of the `concat` is taken as the type of this production.

```
term = factor
```

The type of the `factor` is taken as the type of this production.

```
arith = term
```

The type of the `term` is taken as the type of this production.

```
value-symbol = "#" literal-string
```

This is a degenerate syntax for symbolic constants, which is used only for direct comparison; type is determined by context. See productions for `comp`.`comp-op = "=" | "<>" | "<" | "<=" | ">" | ">="``arith-or-value-symbol`

```
comp = arith
```

The type of the arith is taken as the type of this production.

```
|  arith comp-op arith
```

This production has a `Boolean` type for all cases in which it applies. See 7.1.6 for a more detailed description of comparisons.

If either `arith` is `NULL`, then the production evaluates to `NULL`.

```
|  chain comp-op value-symbol
```

The left side shall be a property reference, and that property shall be used as the context for the value-symbol (see 7.1.4).

This production has a `Boolean` type for all cases in which it applies. See 7.1.6 for a more detailed description of comparisons.

If `chain` or the value-symbol is `NULL`, then the production evaluates to `NULL`.

```
|  value-symbol comp-op chain
```

The right side shall be a property reference, and that property shall be used as the context for the value-symbol (see 7.1.4).

This production has a `Boolean` type for all cases in which it applies. See 7.1.6 for a more detailed description of comparisons.

If `chain` or the value-symbol is `NULL`, then the production evaluates to `NULL`.

```
|  arith IS [ NOT ] NULL
```

This production has a `Boolean` type.

```
|  arith ISA identifier
```

The left side shall be either an instance or a property containing an EMBEDDEDOBJECT or EMBEDDEDINSTANCE. The right side shall be the name of a class or a class alias.

The ISA tests whether the left side is of the class or a subclass of the class named by the identifier on the right side.

If arith is NULL, then the production evaluates to NULL.

The production has a Boolean type.

```
|  arith LIKE literal-string
```

arith shall have a string or char16 type; the result has a Boolean type.

If arith is NULL, then the production evaluates to NULL.

The Basic Query feature includes only the Like features described in C.1.

```
expr-factor = comp
```

The type of the comp is taken as the type for this production.

```
|  NOT comp
```

comp shall have a Boolean type; this production has a Boolean type.

Table 1 defines the result of the NOT expression:

Table 1 – NOT Expression

Comp	NOT comp
TRUE	FALSE
FALSE	TRUE
NULL	NULL

```
expr-term = expr-factor
```

The type of the expr-factor is taken as the type for this production.

```
|  expr-term AND expr-factor
```

expr-term and expr-factor must both have a Boolean type; the production has a Boolean type.

Table 2 defines the result of the AND expression:

Table 2 – AND Expression

expr-term	expr-factor	Expr-term AND expr-factor
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
TRUE	NULL	NULL
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE
FALSE	NULL	FALSE

```
expr = expr-term
```

The type of the `expr-term` is taken as the type for this production.

```
|  expr OR expr-term
```

`expr` and `expr-term` must both have a `Boolean` type; the production has a `Boolean` type.

Table 3 defines the result of the OR expression:

Table 3 – OR Expression

expr-term	Expr-factor	Expr-term OR expr-factor
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	NULL	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE
FALSE	NULL	NULL

```
array-index = expr
```

`expr` shall have an unsigned integer type.

Array indices are zero-relative. Note that arrays defined with the qualifier 'ArrayType ("Bag")' should not be referenced using specific indices because these may vary across retrievals and time.

```
array-index-list = array-index
```

The `array-index-list` specifies one array element.

6.4.7 Select List

The productions in this section define the select list of the query. Each entry of the select list defines one column of the query result.

```
star-expr = "*"
```

This production refers to all the properties exposed by all classes defined in the `from-criteria`. This includes uncovered properties of superclasses of the `from-criteria` classes. Properties of subclasses of the `from-criteria` classes are not included.

Covered properties (that is, properties of the same name that are not overridden) may be explicitly referenced by using the scoping operator "`::`" in the `expr` of the `selected-entry` production.

As a consequence of these rules, the property list produced does not vary over the query. For example, if referencing a CIM 2.8 schema and the `from-criteria` includes `CIM_ManagedSystemElement`, then the properties "`Caption`", "`Description`", "`ElementName`", "`InstallDate`", "`Name`", "`OperationalStatus`", "`StatusDescriptions`", and "`Status`" would be included.

```
|  chain "."  "*"
```

`chain` shall have type object of *C* for some class *C*. This production refers to all the properties exposed by *C*, including those of *C*'s superclasses. Properties of subclasses of *C* are not included in the set.

```
selected-entry  =  expr
```

`expr` may have any primitive, reference, or array type, and defines the type of the column defined by this production. If the type of the `expr`ession is an object type, then the corresponding result column shall have a string type and be populated with string representations of the `VALUES` .

The set of column names in the query result shall not contain duplicates.

Basic Query allows only properties in the `select-list`. If there is more than one entry in the FROM list, then each selected entry that is a property reference shall be a `chain` `expr`ession starting with either a class name or an alias that is included in the `FROM` list.

```
|  star-expr
```

This generates a set of selected entries in the query result where the "`*`" is enumerated to be a list of properties. The set of selected entries is taken as the default names for a subset of the columns returned.

If there is more than one entry in the `FROM` list, then each `star-expr` shall be a `chain` `expr`ession starting with either a class name or an alias that is included in the `FROM` list.

```
select-list  =  selected-entry  *( ","  selected-entry )
```

If the `select-list` contains any aggregating expressions, then all items in the `select-list` shall be aggregating expressions.

6.4.8 From Criteria

The productions in this section define the `from-criteria` of the query. The `from-criteria` is generically a list of data sources over which the query is processed. In basic query, the list is limited to one data source.

```
from-specifier  =  class-path  [[AS] identifier]
```

Each `from-specifier` using this production identifies a CIM class that will participate in the query, along with a name by which instances of that class will be referenced in the query. If the explicit `identifier` is present, it is the name that will be used; otherwise, the name of the class will be used as the name.

Even if the explicit `identifier` is present, the name of the class may also be used as an alternative name for instances of the class, provided such use would not conflict with a name established by any other `from-specifier`.

Additionally, each property of the class identified by class-path can be accessed by its name alone, provided that name doesn't conflict with any other property or class name in the `from-criteria`.

```
from-criteria  =  from-specifier
```

6.4.9 Select Statement

The productions in this section define the Select statement, which is a directive to the query processor create an output table with columns defined by a `select-list` and rows containing data specified by the `from-criteria` and `search-condition`.

```
search-condition = expr
```

`expr` shall have a `Boolean` type.

```
select-statement = SELECT select-list
   FROM  from-criteria
   [WHERE  search-condition]
```

This clause produces information that represents the rows returned by the query. Each row has an entry for each selected entry.

The FROM clause produces a candidate set of rows from instances identified by the `from-criteria`.

When present, the WHERE clause rejects all rows of the candidate set produced by the FROM clause except those for which the `search-condition` evaluates to TRUE. (Evaluation to `NULL` is not the same as TRUE.)

The `select-list` selects particular columns of the rows of the candidate set and also may introduce additional derived columns.

```
start  =  select-statement
```

7 CIM Query Language Considerations

This section provides additional details on the use of both basic query features and the extended query features described in Annex E.

7.1 Considerations of the Constructs in the BNF

The CIM Query Language does not currently define "data change" operations (`INSERT`, `DELETE`, or `UPDATE`). These may be added at a later time, but they are not currently required. Today, these operations are supported by invoking individual operations defined in DSP0200.

CQL queries operate only against instances and their properties. CQL does not support the ability to query the supported schema or invoke methods of instances. Through the `ISA` operator, CQL supports the ability to determine if an instance is a member of a class.

Several of the constructs in the BNF require usage information or some additional explanation, which is provided in the following subclauses.

7.1.1 Property Identification

A CIM class may expose more than one property with a given name, but it is not permitted to define more than one property with a particular name. However, more than one property with a particular name can happen if a base class defines a property with the same name as a property defined in a derived class without overriding the base class property. The scoping operator ("`::`") is used to provide an explicit context for resolving `identifiers` to properties.

The general syntax by which a property is identified is as follows:

```
[ chain "."]  [property-scope] identifier
```

In this expression, `chain` shall have type object of *C*.

Property names identify properties relative to a class context. Given a class context *C,* the search for the property begins at *C* and selects a property defined on *C* whose name matches the identifier, if there is one; if *C* does not define a property with this name, then the search continues with *C*'s direct superclass, and so on. If no property is located with this search, then the property reference is invalid.

The class context is determined according to the following rules:

- If `property-scope` is present, then it declares the class context *C* and one of the following rules applies:
  - If the scoped `identifier` does not name a property exposed by *C*, then the query is invalid.
  - If `chain` is not present, *C* shall be the same as, a superclass of, or a subclass of exactly one entry in the FROM list. In this case, `chain` is inferred to refer to instances produced by that FROM list entry.
  - If `chain` is present and it has type object of *D*, for some *D*, then *C* shall be the same as, a superclass of, or a subclass of *D*.
  - If `chain` is present and it does not have type object of *D* for some *D*, then `chain` must have type object.
  - If the value of the `chain` expression is not of class *C* (or subclasses of *C*), then the application of the property produces `NULL`.
- If `chain` "`.`" is present, then `chain` shall be of type object of *C*, and *C* is the class context.

- If neither `property-scope` nor `chain` are present, then the `identifier` must be declared in at most one of the classes named in the FROM list.
- If none of the preceding rules applies, the context cannot be determined and the query is invalid.

7.1.2 Arrays

For properties of type Array, [*] is implicitly used if no specific `array-index`-list is given; therefore, for example, "`OperationalStatus`" has the same semantic meaning as "`OperationalStatus[*]`". For more details on Arrays, please refer to the *CIM Infrastructure Specification v2.3.0* (DSP0004).

7.1.3 Embedded Objects

An embedded object is conveyed as a property of a string type annotated only with the `EMBEDDEDOBJECT` qualifier. This qualifier indicates that the property's value is to be interpreted as an embedded object, but it identifies neither whether the embedded object will be a class or an instance, nor the class to which the embedded instances belong. For this reason, expressions in CQL that refer to string properties with the `EMBEDDEDOBJECT` qualifier are assigned type object. References to the embedded properties of that property have their native type unless they too are qualified with `EMBEDDEDOBJECT`.

The actual type of an `EMBEDDEDOBJECT` is not known until an instance is selected. This uncertainty can lead to situations in which the type of a projected result cannot be determined in advance of the query's execution, and, indeed, may vary even within the execution of a single query. This circumstance affects the resolution of properties of the embedded object. To remove ambiguity, queries that concern themselves with properties of embedded objects shall use the scoping operator ("`::`") to scope those properties, (see E.15). A CQL implementation shall reject any query that involves expressions whose type cannot be determined.

For example, the following query would be permitted because both properties of `SourceInstance` were provided a scope. The `DeviceID` property would be returned as `NULL` when `SourceInstance` is a `CIM_PhysicalElement`.

```
SELECT  SourceInstance.CIM_LogicalDevice::DeviceID,
        SourceInstance.CIM_ManagedSystemElement::OperationalStatus
FROM CIM_InstIndication
WHERE   SourceInstance ISA CIM_LogicalDevice
OR      SourceInstance ISA CIM_PhysicalElement
```

7.1.4 Symbolic Constants

The "`#`" syntax uses the `VALUES` and `VALUEMAP` qualifiers of a property to look up an enumerated value that a particular property may take. The property shall expose a `VALUES` qualifier, and the accompanying literal-string shall match one of the strings in the `VALUES` qualifier's value.

If the property does not also expose a `VALUEMAP` qualifier, then the property shall have an integer type, and the index of the literal-string among the `VALUES` qualifier's value is taken as the value of this production. If, conversely, the property also exposes a `VALUEMAP` qualifier, then the value for this production shall be based on the value in the `VALUEMAP` array corresponding to the selected value of the `VALUES` array, as follows:

1) If the property has a string type, then the `VALUEMAP` entry itself is the value of the production.
2) Otherwise, the property shall have an integer type, the `VALUEMAP` entry shall not include the sequence "`..`", and the `VALUEMAP` entry is converted into an integer of the appropriate type

   NOTE: `CIM_FCPort.OperationalStatus#'OK'` is equivalent to the constant 2, and `CIM_FCPort.OperationalStatus#'Predictive Failure'` is equivalent to 5.

If the expression on one side of a comparison identifies exactly one property, then the # syntax may be used in a standalone form on the opposite side of the comparison. The identified property becomes the defining context of the symbolic constant. For example:

```
CIM_FCPort.OperationalStatus[3] > #'OK'
```

is equivalent to

```
CIM_FCPort.OperationalStatus[3] > CIM_FCPort.OperationalStatus #'OK'
```

If a class name is used to qualify a symbolic constant, that class does not need to be related to any class in the query. For example the following query is valid even though `CIM_LogicalDevice` has nothing to do with the query:

```
SELECT * FROM CIM_AlertIndication WHERE AlertType >
   CIM_LogicalDevice.OperationalStatus#'OK'
```

7.1.5 Computation and Types

The use of arithmetic operators causes numeric types to be "widened" as necessary to minimize the loss of precision. Unless both operands are unsigned, addition, subtraction, and multiplication among integer types results in sint64. If both operands are unsigned, then the result is uint64. Otherwise, all arithmetic operations (that is, all cases of division, as well as addition, subtraction, or multiplication involving at least one non-integer type) produce real64 type. If an overflow or underflow occurs, an error is returned.

Arithmetic and comparisons on `datetime` types are defined in Annex D.

7.1.6 Comparisons

Comparison is supported between all numeric types. When comparisons are made between different numeric types, comparison is performed using the type with the greater precision.

Comparison between strings and between char16 `VALUES` is supported and is done case-sensitively on a Unicode-character basis. A comparison between a string and a char16 value is accomplished by treating the char16 value as a single-character string. For string and char16 comparison and sort operations, the Default Collation Algorithm as defined in ISO/IEC 14651 and Unicode Technical Report #10 shall be applied. Unicode character-based comparison is done as follows:

The `"="` and `"<>"` operators shall use the string identity matching rules defined in W3C *Character Model for the World Wide Web 1.0: Normalization*, section 4 "String Identity Matching".

The following rules apply to comparison between strings and char16 `VALUES` using the `"<"` and `">"` operators:

1) For Basic Query, these operators shall behave as if the normalization defined in *Character Model for the World Wide Web 1.0: Normalization*, section 4 "String Identity Matching", was applied and then the comparison was performed on the resulting strings. The strings are compared from the beginning, on a Unicode character basis. Each character is compared based on its Unicode code point order. The first character found to be different determines the result of the comparison. If the strings are of different lengths, but are otherwise equal, then the longer string is greater than the shorter string.

   NOTE: For implementations that use UTF-8 or UTF-32 as the encoding, the binary order of the encoded characters matches the Unicode code point order. For UTF-16, the binary order of the supplementary characters does not match their Unicode code point order. For more information, refer to section 2.5 of *The Unicode Standard*.

2) For the Full Unicode feature, these operators shall behave as if the normalization defined in *Character Model for the World Wide Web 1.0: Normalization*, section 4 "String Identity Matching", was applied and then the default collation order defined in the Unicode Collation

Algorithm was used on the resulting strings. Note that this collation order accommodates most languages, without having to take any locales into account.

Comparison between `datetime` types is supported and is defined in Annex D.

Comparison between `Boolean` VALUES, complete Arrays, and References is supported, but this support is limited to the "=" and "<>" operators.

Reference comparison is performed through a process of comparing certain components of the references. The components to be compared are the namespace type, namespace handle, and model path, as defined by DSP0004. Two references are considered to be equal if all of the following conditions are true:

- For the model path, all of the following conditions must be true:
  - There must be the same number of key property name/value pairs.
  - For each key property name/value pair in one reference, exactly one matching key property name/value pair must be found in the other reference.
  - The order of the key property name/value pairs does not affect the comparison. Comparison is not done case sensitively for key property names.
  - Key property VALUES are compared according to their type, as defined in this section.
- For all components except the model path, the comparison is not done case sensitively.

NOTE: The implementation may perform reference comparisons using alternative, but equivalent, paths or representations.

Comparison of classes requires that the `ClassName` is the same and that the properties and property types defined by this class and by each superclass in the classes hierarchy compare equal. The comparison of class names, property names and property types is not done case sensitively. The set of qualifiers defined on each class shall be the same and evaluate to the same VALUES .

Comparison of instances requires that the instances be of the same class and that all property VALUES either compare as equal or are both `NULL`. The comparison of the property VALUES is done case-sensitively.

For comparison between an array property and a non-array property, see 7.1.7.

NOTE: This type of comparison shall be supported if the Array Range query feature is supported.

Comparison of complete arrays shall be supported in Basic Query. Comparison of parts of arrays shall be supported in the Array Range query feature. The ArrayType governs how matches are made. There are three types of arrays: Bag, Ordered, and Indexed. If one of the arrays is a Bag, then comparison rules for Bags are used. As defined in DSP0004, a bag is an unordered multiset. Two arrays of ArrayType "Bag" are equal if and only if the number of elements is equal and if it is possible to find a permutation for one of the arrays so that for an element-by-element comparison, all elements of the compared arrays are equal. Equality for Bag-type arrays may be tested by sorting both arrays and then doing an element-by-element comparison. For comparison of Ordered and Indexed arrays, an element-by-element comparison is performed. Arrays that have different numbers of elements do not compare as equal.

Other than the cases described in this section, comparisons among disparate types are not supported in CQL.

7.1.7 Comparisons of Array and Scalar

This section only applies to comparison operations between array properties and non-array properties, as part of the Array Range and Satisfies Array query features. A comparison between an array property and a non-array property is illegal if neither the EVERY keyword nor the ANY keyword is used. If multiple

elements of an array property are compared, the operation evaluates to TRUE if and only if the specified comparison is TRUE for all the indicated Array Range.

EXAMPLE: The following are examples of comparison operations in array processing:

- EVERY `CIM_LogicalDevice.OperationalStatus[*]` <> 2 is TRUE if and only if every value of the `OperationalStatus` array is not 2.
- EVERY `CIM_LogicalDevice.OperationalStatus[*]` = 2 is TRUE if and only if all of the `VALUES` of `OperationalStatus` are 2.
- EVERY `CIM_LogicalDevice.OperationalStatus[*]` < 2 is TRUE if and only if all of the `VALUES` of `OperationalStatus` are less than 2.
- ANY `CIM_LogicalDevice.OperationalStatus[*]` > 2 is TRUE if and only if any the `VALUES` of the `OperationalStatus` array are greater than 2.
- ANY `CIM_LogicalDevice.OperationalStatus[*]` <> 2 is TRUE if and only if any of the `VALUES` of the `OperationalStatus` array are NOT 2.
- NOT EVERY `CIM_LogicalDevice.OperationalStatus[*]` = 2 is TRUE if and only if any of the `VALUES` of the `OperationalStatus` array are <> 2.
- `CIM_LogicalDevice.OperationalStatus[0]` = 2 is TRUE if the first value of the array is set to 2.
- EVERY `CIM_LogicalDevice.OperationalStatus[0..3]` > 2 is TRUE if the first four `VALUES` of the `OperationalStatus` array are each greater than 2.
- ANY stat IN `CIM_LogicalDevice.OperationalStatus[*]` SATISFIES (stat = 3 OR stat > 5) is TRUE if any value of the `OperationalStatus` array is equal to 3 or greater than 5.

7.2 Query Language Functions

This section describes the functions available for CQL.

If the arguments of these functions do not conform to the defined constraints, then the query will be in error.

7.2.1 Numeric Functions

The following define the numeric functions that may be used within a query statement.

- **DATETIMETOMICROSECOND(`expr`)**

  The argument shall have a `datetime` type, and the result has type uint64. If the argument is a timestamp, it is converted to the number of microseconds since 00:00:00.000000UTC on 1/1/0000; otherwise (that is, if the argument is an interval), it is converted to microseconds.

  If `expr` computes to a time before 00:00:00.000000UTC on 1/1/0000 the result is an arithmetic underflow error. If `expr` computes to a time after 23:59:59.999999 UTC on 12/31/9999, the result is an arithmetic overflow error. In either case, the query will result in an error.

- **STRINGTOUINT(`expr`)**

  The argument shall have a string or char16 type and must be a binary-value, hex-digit-value, decimal-value, or real-value in the range of 0 to 264-1. The result has type uint64. The fractional portion of any real-value is discarded.

- **STRINGTOSINT(`expr`)**

  The argument shall have a string or char16 type and must be a binary-value, hex-digit-value, decimal-value, or real-value in the range of -263 to 263-1. The result has type sint64. The fractional portion of any real-value is discarded.

- **STRINGTOREAL(`expr`)**

  The argument shall have a string type and must be a binary-value, hex-digit-value, decimal-value, or real-value. The result has type real64.

7.2.2 String Functions

The following define the string functions that may be used within a query statement.

- **UPPERCASE(`expr`)**

  The argument shall have a string or char16 type, and the result has a string type. This function canonicalizes its argument by converting all lowercase characters to uppercase. For Basic Query, this function converts lowercase characters in the US-ASCII range (U+0000...U+007F) to uppercase. Characters outside of the US-ASCII range are not changed. For the Full Unicode feature, this function performs Case Mapping, as defined in the Unicode standard, on all characters.

- **NUMERICTOSTRING(`expr`)**

  The argument shall have a numeric type, and the result shall have a string type. This function constructs a string representation of its argument, using the following rules:

  – If the argument is of one of the integer types, it is represented using decimal radix. Positive numbers do not have a plus sign, and negative numbers have a preceding minus sign.

  – If the argument is of one of the real types, it is represented using decimal mantissa. If an exponent is needed, it uses decimal radix, follows after an upper case "E", and does not have any leading zeros. If the mantissa has more than one digit, the decimal point is always after the first digit. Positive mantissas and exponents do not have a plus sign, and negative mantissas and exponents have a preceding minus sign.

  – If the argument has a value of zero, it is represented as the single character “0”.

- **REFERENCETOSTRING(`expr`)**

  The argument shall have a reference type, and the result shall have a string type. This function returns an object path string based exclusively on the information in the input reference. Canonicalization may be accomplished by using the Path Functions.

7.2.3 Instance Functions

The following functions operate on objects, references, or strings whose content is a WBEM-URI, as defined in the *WBEM URI Mapping Specification* (DSP0207).

- **INSTANCEOF([`expr`])**

  The argument shall be an instance, an embedded instance, an embedded object, a reference to an instance, or a string containing a WBEM-URI to an instance. If the argument is of type embedded object, it shall represent an instance and shall be scoped using the `property-scope` syntax. In all cases using valid input, if the instance is of type *C*, the result of this function is an embedded instance of type *C*. In all other cases, the query is invalid.

7.2.4 Path Functions

These functions operate on objects, references, or strings whose content is a WBEM-URI, as defined in DSP0207.

- **CLASSPATH([`expr`])**

  The argument shall be an object, a reference, or a string containing a WBEM-URI. The result of this function is of a reference type. If the argument is of a reference or string type and it refers to

a class, the result of this function refers to that class. If the argument is of reference or string type and it refers to an instance, the result of this function refers to the creation class of that instance. If the argument is of type object, it shall be an instance value that is not an Indication or an embedded instance, and the result of this function refers to the creation class of that instance. In all other cases, the query is invalid. Whether the class or instance referenced by the argument exists does not matter for the successful execution of the function. The function does not add any missing components to the namespace path of the resulting reference.

- **OBJECTPATH([expr])**

  The argument shall be an object, a reference, or a string containing a WBEM-URI. The result of this function is of a reference type. If the argument is of type reference or string and it refers to a class, the result of this function refers to that class. If the argument is of type reference or string and it refers to an instance, the result of this function refers to that instance. If the argument is of type object, it shall be an instance value that is not an Indication or an embedded instance, and the result of this function refers to that instance. In all other cases, the query is invalid. Whether the class or instance referenced by the argument exists does not matter for the successful execution of the function. The function does not add any missing components to the namespace path of the resulting reference.

7.2.5 Datetime Functions

- The following define the time and date conversion functions that may be used within a query statement.**CURRENTDATETIME( )**

  This function returns the "current" `datetime` as determined by the implementation.

- **DATETIME(expr)**

  The argument shall be of a string type and at runtime shall take on a 25-character value conformant with a `datetime` specification (either timestamp or interval). The result shall have a `datetime` type.

- **MICROSECONDTOTIMESTAMP(expr)**

  The argument shall be of an integer type, and the result shall have a `datetime` type. The argument will be interpreted as a number of microseconds since 00:00:00.000000UTC on 1/1/0000, and the result shall be a timestamp.

- **MICROSECONDTOINTERVAL(expr)**

  The argument shall be of an integer type, and the result shall have an interval (`datetime`) type. The argument will be interpreted as a number of microseconds, and the result shall be an interval.

7.3 Query Considerations

The result of a query is a table that contains a set of zero or more rows that contain the columns defined in the `select-list`. This table is not stored beyond the execution of a particular invocation of the query. These instances have the following additional characteristics:

- Each column has a type and a distinct name.
- Each classname in the FROM list is considered by query as a table that has one row for each class instance and where the properties of the class are mapped to columns of the table.
- Subqueries are considered by query to produce tables.
- On the relation to classes, instances, and properties the following assertions apply:

  - Each table may be considered as a class. However, it is not required to conform to the definition of a CIM class.
  - Each row may be considered as an instance. However, it is not required to conform to the definition of a CIM instance.
  - Each column may be considered a property that conforms to the definition of a CIM Property.

- A query may be specified as part of a class definition (such as `CIM_IndicationFilter`, `CIM_QueryCondition`, and `CIM_MethodAction`). The implementation of the class is responsible for processing queries specified in instances of that class. For example, `CIM_IndicationFilter` subclasses constrain the `select-list` to produce entries that conform to the `CIM_Indication` subclass that is used in the FROM clause. The results are then typically delivered by the `CIM_ListenerDestination` subclass as instances of the named `CIM_Indication` subclass.

7.4 Query Errors

It is not in the scope of this specification to specify errors returned as a result of processing CQL queries. Specifications that specify the use of CQL should specify the type of errors that might return. For instance, (see DSP0200 for errors returned by ExecQuery and DSP0200, *CIM Operations over HTTP Specification* v1.3 for errors returned by OpenQueryInstances.) Use of CQL queries in the context of class definitions should be documented in the class definition.

In the future, the `CIM_Error` class will be used to expand on the errors defined in this clause..

Annex A Examples (Informative)

This section provides a number of sample queries to illustrate the use of CQL.

A.1 Information Gathering Examples

The following eight examples show example Select statements that each might be used to provide information about the current state of an implementation.

1) Get the object path, `ElementName`, and `Caption` for all `CIM_StorageExtents`.

   Required Features: Basic Query, Extended Select List, Conversion Utilities

   ```
   SELECT OBJECTPATH(CIM_StorageExtent) AS Path,
      ElementName, Caption
   FROM CIM_StorageExtent
   ```

   A set of instances would be returned with three properties: the object path of the instance, as well as the `ElementName` and `Caption` properties.

2) Select all LogicalDevices on a particular `CIM_ComputerSystem` that have an `OperationalStatus` not equal to "OK" (value = 2), and return their object paths and `OperationalStatus`.

   Required Features: Basic Query, Extended Select List, Complex Join, Array Range, Conversion Utilities

   ```
   SELECT OBJECTPATH(CIM_LogicalDevice) AS Path,
      CIM_LogicalDevice.OperationalStatus[*]
   FROM CIM_LogicalDevice,
      CIM_ComputerSystem,
      CIM_SystemDevice
   WHERE CIM_ComputerSystem.ElementName = 'MySystemName'
      AND CIM_SystemDevice.GroupComponent =
         OBJECTPATH(CIM_ComputerSystem)
      AND CIM_ SystemDevice.PartComponent =
                OBJECTPATH(CIM_LogicalDevice)
           AND ANY CIM_LogicalDevice.OperationalStatus[*] <> 2)
   ```

   A set of instances would be returned, each with the following properties: a string containing the object path of the instance of `CIM_LogicalDevice` and the `OperationalStatus` array property.

3) Get all `CIM_StorageExtent` and `CIM_MediaAccessDevice` instances. Note that the projection is limited to instances that are either `CIM_StorageExtent` or `CIM_MediaAccessDevice`; however, only properties of `CIM_LogicalDevice` and its superclasses are returned.

   Required Features: Basic Query

   ```
   SELECT *
   FROM CIM_LogicalDevice
   WHERE CIM_LogicalDevice ISA CIM_StorageExtent OR
      CIM_LogicalDevice ISA CIM_MediaAccessDevice
   ```

A set of instances would be returned with a complete `select-list` as defined by `CIM_LogicalDevice`.

4) List all `CIM_ComputerSystem` instances and the object paths of any instances dependent on the system as described by the `CIM_Dependency` association.

Required Features: Basic Query, Extended Select List, Complex Join, Conversion Utilities

```
SELECT CIM_ComputerSystem.*,
      OBJECTPATH(CIM_ManagedElement) AS MEObjectName
FROM CIM_ComputerSystem,
   CIM_ManagedElement,
   CIM_Dependency
WHERE CIM_Dependency.Antecedent =
      OBJECTPATH(CIM_ComputerSystem)
   AND CIM_Dependency.Dependent =
      OBJECTPATH(CIM_ManagedElement)
```

This query returns a set of instances defined by the references of the `CIM_Dependency` association's instances. The instances that are created contain all the properties of `CIM_ComputerSystem` and a string representing the related or associated ManagedElement's object path.

5) Traverse from a resource (`CIM_ComputerSystem`) to the `CIM_BaseMetricValue` instances associated through the `CIM_MetricForME` association. The resource instance is known by its keys, many (more than 10000) `CIM_BaseMetricValue` objects are associated with it, and the selection criteria is such that only a handful of them match.

Required Features: Basic Query, Extended Select List, Complex Join, Conversion Utilities

```
SELECT OBJECTPATH(CIM_ComputerSystem) AS CSOBJECTPATH,
   CIM_BaseMetricValue.*
  FROM CIM_ComputerSystem,
       CIM_BaseMetricValue,
       CIM_MetricForME
  WHERE CIM_ComputerSystem.Name = 'MySystem1'
       AND CIM_BaseMetricValue.TimeStamp >
  DATETIME('200407101000********+300')
   AND CIM_BaseMetricValue.TimeStamp <
DATETIME('200407101030********+300')
   AND CIM_BaseMetricValue.Duration =
DATETIME('000000000005********:000')
   AND CIM_MetricForME.Antecedent =
      OBJECTPATH(CIM_ComputerSystem)
   AND CIM_MetricForME.Dependent =
              OBJECTPATH(CIM_BaseMetricValue)
```

As in example 4), this query returns a set of instances defined by the query's join. The instances that are returned contain all properties of `CIM_BaseMetricValue` and the associated `CIM_ComputerSystem` instance's object path.

The query in this example is very selective: Only six instances are returned, while the combined number of instances in the classes selected from can be in the tens of thousands. This shows that it is essential that these instances never be enumerated or "walked" in the implementation of the query engine, because this would likely result in huge computational penalties. It is critical to appropriately break down the query to the different providers involved.

6) Display all the Settings for a particular `CIM_ManagedSystemElement` in a Composite Setting that is associated with the MSE.

Required Features: Basic Query, Complex Join, Conversion Utilities

```
SELECT SD.*
FROM CIM_SettingData CSD,
   CIM_SettingData SD,
   CIM_ManagedSystemElement MSE,
   CIM_ElementSettingData ESD,
   CIM_ConcreteComponent CC
WHERE OBJECTPATH(MSE) = 'some desired key'
   AND ESD.ManagedElement = OBJECTPATH(MSE)
   AND ESD.SettingData = OBJECTPATH(CSD)
   AND CC.GroupComponent = OBJECTPATH(CSD)
   AND CC.PartComponent = OBJECTPATH(SD)
```

A set of instances would be returned (that meet the association criteria) with properties as specified by `CIM_SettingData`.

7) Get a storage array's LUN masking and mapping for a failed `CIM_FCPort`. This query uses aliasing in the `FROM` clause and a series of subqueries. The use of nested subqueries guides the query engine through a step-wise process that is similar to one that would be used by a client executing a series of CIM intrinsic operations. Use of subqueries is recommended to limit the complexity of otherwise very large joins. The principle advantage over the series of intrinsic operations is that the query is a single operation that returns only the final results.

Required Features: Basic Query, Extended Select List, Complex Join, Subquery, Array Element, Property Scope, Conversion Utilities

```
SELECT OBJECTPATH(pms) AS PrivilegeMgmtServiceInst,
 Oh AS StorageHardwareIDInst, Op AS AuthorizedPrivilegeInst,
 Ov AS StorageVolumeInst
FROM CIM_HostedService hs,
 CIM_PrivilegeManagementService pms,
 ( SELECT OBJECTPATH(cs) AS Oc, O.Op, O.Oh, O.Ov
  FROM CIM_ComputerSystem cs, CIM_SystemDevice sd,
   ( SELECT OBJECTPATH(v) AS Ov, P.Op, P.Oh
    FROM CIM_AuthorizedTarget t,
     CIM_StorageVolume v,
      ( SELECT OBJECTPATH(p) AS Op,
       OBJECTPATH(hi) AS Oh
      FROM CIM_StorageHardwareID hi, CIM_AuthorizedPrivilege p,
           CIM_AuthorizedSubject s,
       ( SELECT  SourceInstance.
        CIM_FCPort ::PermanentAddress
       FROM CIM_InstModification
       WHERE SourceInstance ISA CIM_FCPort
        AND ANY SourceInstance.CIM_FCPort::OperationalStatus[*]
         <> #'OK'
       ) fc
       WHERE fc.PermanentAddress = hi.StorageID
        AND s.PrivilegedElement = OBJECTPATH(hi)
        AND s.Privilege = OBJECTPATH(p)
     ) P
       WHERE t.Privilege = P.Op AND t.TargetElement = OBJECTPATH(v)
  ) O
  WHERE sd.PartComponent = Ov
   AND sd.GroupComponent = OBJECTPATH(cs)
 ) C
WHERE hs.Antecedent = Oc AND hs.Dependent = OBJECTPATH(pms)
```

Without the use of subqueries, but keeping the same color codes to relate to the subqueries of the above query, an equivalent query can be expressed as:

Required Features: Basic Query, Extended Select List, Complex Join, Array Element, Property Scope, Conversion Utilities

```
SELECT OBJECTPATH(pms) AS PrivilegeMgmtServiceInst,
  OBJECTPATH(hi) AS StorageHardwareIDInst,
  OBJECTPATH(p) AS AuthorizedPrivilegeInst,
  OBJECTPATH(v) AS StorageVolumeInst
FROM CIM_InstModification im,
    CIM_StorageHardwareID hi,
    CIM_AuthorizedSubject s,
    CIM_AuthorizedPrivilege p,
    CIM_AuthorizedTarget t,
    CIM_StorageVolume v,
    CIM_SystemDevice sd,
    CIM_ComputerSystem cs,
    CIM_HostedService hs,
    CIM_PrivilegeManagementService pms
WHERE im.SourceInstance ISA CIM_FCPort
  AND ANY im.SourceInstance.CIM_FCPort::OperationalStatus[*] <> #'OK'
  AND im.SourceInstance.CIM_FCPort::PermanentAddress = hi.StorageID
  AND s.PrivilegedElement = OBJECTPATH(hi)
  AND s.Privilege = OBJECTPATH(p)
  AND t.Privilege = OBJECTPATH(p)
  AND t.TargetElement = OBJECTPATH(v)
  AND sd.PartComponent = OBJECTPATH(v)
  AND sd.GroupComponent = OBJECTPATH(cs)
  AND hs.Antecedent = OBJECTPATH(cs)
  AND hs.Dependent = OBJECTPATH(pms)
```

The primary difference is that without the use of subqueries, the query implementation would have to determine how to optimize this query to avoid an uncorrelated join across all of the instances belonging to the 10 classes named in the FROM clause. This level of analysis is beyond the capability of most expected implementations.

8) Example of mathematical aggregation function

Required Features: Basic Query, Extended Select List, Aggregation, Result Set Operations, Subquery

```
SELECT DISTINCT            OBJECTPATH(sv) AS VolumePath,
        (sv.BlockSize * sv.NumberOfBlocks) AS Size
FROM   CIM_StorageVolume sv,
       ( SELECT MAX(v.BlockSize*v.NumberOfBlocks) AS Maxbytes
         FROM   CIM_StorageVolume v) mv
WHERE (sv.BlockSize * sv.NumberOfBlocks) = mv.Maxbytes
```

A.2 Event Detection Examples

The following examples define queries that might be contained in the Query property of a `CIM_IndicationFilter` instance within a storage management implementation. The corresponding QueryLanguage property would contain the value "`DMTF:CQL`".

1) Using the lifecycle indication classes, the following query would be stored in the Query string property of an instance of `CIM_IndicationFilter` and its delivery defined by a `CIM_IndicationSubscription` association to a ListenerDestination (see the CIM Event Model). A `CIM_InstCreation` notification would be delivered any time that a `CIM_FCPort` was created. The notification would consist of a single instance with a `select-list` as defined by the `CIM_InstCreation` class.

   Required Features: Basic Query

   ```
   SELECT *
     FROM CIM_InstCreation
     WHERE SourceInstance ISA CIM_FCPort
   ```

2) As in the previous example, this query would be stored in the Query string property of an instance of IndicationFilter and its delivery would be defined by a `CIM_IndicationSubscription` association. A `CIM_InstModification` notification would be delivered any time that a `CIM_FCPort` was modified and its first array property had changed. The notification would consist of a single instance with a `select-list` as defined by the `CIM_InstModification` class.

   Required Features: Basic Query, Embedded Properties, Property Scope

   ```
   SELECT *
   FROM CIM_InstModification
   WHERE SourceInstance ISA CIM_FCPort
      AND PreviousInstance ISA CIM_FCPort
         AND SourceInstance.CIM_FCPort::OperationalStatus[0] <>
            PreviousInstance.CIM_FCPort::OperationalStatus[0]
   ```

3) Send an Indication consisting of EventTime, AlertingManagedElement, PerceivedSeverity and ProbableCause whenever AlertingManagedElmeent is '/dev/tty0p1' and ProbableCause=20.

   Required Features: Basic Query

   ```
    SELECT EventTime,
      AlertingManagedElement,
      PerceivedSeverity,
      ProbableCause
   FROM CIM_AlertIndication
   WHERE AlertingManagedElement = '/dev/tty0p1'
      AND ProbableCause = 20
   ```

4) Building on the previous example, in order to facilitate auditing and maintenance, the IT department requires that all Indications be "tagged" with an ID that identifies the filter condition that the Indication satisfied.

   Required Features: Basic Query, Extended Select List

   ```
    SELECT EventTime,
      AlertingManagedElement,
      PerceivedSeverity,
   ```

```
   ProbableCause,
   'HP12345' AS FilterID
FROM CIM_AlertIndication
WHERE AlertingManagedElement = '/dev/tty0p1'
   AND ProbableCause = 20
```

5) Continuing the previous example, to ensure prompt processing of this type of Indication, define a CustomSeverity and set it to "Critical".

   Required Features: Basic Query, Extended Select List

```
SELECT EventTime,
   AlertingManagedElement,
   1 = PerceivedSeverity,
   'Critical' = OtherSeverity,
   ProbableCause,
   'HP12345' AS FilterID
FROM CIM_AlertIndication
WHERE AlertingManagedElement = '/dev/tty0p1'
   AND ProbableCause = 20
```

6) Locate sick System/LogicalDevice combinations.

   Required Features: Basic Query, Satisfies Array, Complex Join, Conversion Utilities

```
SELECT s.Name, d.Name
FROM CIM_System s, CIM_SystemDevice sd, CIM_LogicalDevice d
WHERE OBJECTPATH(s) = sd.GroupComponent
   AND OBJECTPATH(d) = sd.PartComponent
    AND ANY i IN s.OperationalStatus[*] SATISFIES
        (i = #'Non-Recoverable Error' OR i=#'Degraded')
   AND ANY j in d.OperationalStaus[*] SATISFIES (j =#'Degraded' )
```

7) Locate creation of an export relationship for a `CIM_FileShare`.

   Required Features: Basic Query, PropertyScope, Conversion Utilities

```
SELECT
   InstanceOf(
      SourceInstance.CIM_SharedElement::SameElement)
        AS FileShare
FROM CIM_InstCreation
WHERE SourceInstance ISA CIM_SharedElement
   AND InstanceOf(SourceInstance.CIM_SharedElement::SameElement)
      ISA CIM_FileShare
```

A.3 Policy Examples

For policy, identify a StoragePool that is low on space and allocate more space to it. In this example, there are two underlying StoragePools to draw space from. The preferred one is a free pool. The other is used only if the free pool cannot satisfy the need.

1) The following query is used in a `CIM_QueryCondition` with `QueryResultName` set to "`PR_Needy`". The query selects a `CIM_StoragePool` that is low on space. Evaluation results in zero or more `PR_Needy` instances that are used by a related `CIM_MethodAction`.

Required Features: Basic Query, Extended Select List, Complex Join, Embedded Properties, Conversion Utilities, Property Scope

```
SELECT OBJECTPATH(IM.SourceInstance) AS NeedySPPath
FROM CIM_InstModification AS IM,
  CIM_PolicyRule AS PR,
  CIM_PolicySetAppliesToElement AS PSATE
WHERE IM.SourceInstance ISA CIM_StoragePool
  AND PR.Name = 'AllocateMoreSpace'
  AND  OBJECTPATH(PR) = PSATE.PolicySet
  AND  OBJECTPATH(IM.SourceInstance) = PSATE.ManagedElement
  AND  100 * (IM.SourcInstance.
CIM_StoragePool::RemainingManagedSpace / IM.SourcInstance.
CIM_StoragePool::TotalManagedSpace) < 10
  AND IM.SourcInstance. CIM_StoragePool::RemainingManagedSpace <>
    IM.PreviousInstance. CIM_StoragePool::RemainingManagedSpace
```

2) The following query is used in MethodAction to invoke a `CreateOrModifyStoragePool` method. It uses `PR_Needy` instances produced by the previous `CIM_QueryCondition`. The `CIM_InstMethodCall` results of the call are named by the property `InstMethodCallName` set to "`PR_ModifySP`".

Required Features: Basic Query, Extended Select List, Complex Join, Conversion Utilities

```
SELECT OBJECTPATH(SCS) || '.CreateOrModifyStoragePool'
      AS MethodName,
   QCR.NeedySPPath AS Pool,
   QCR.NeedySPPath.Size + (QCR.TotalManagedSpace / 10) AS Size,
   OBJECTPATH(SP) AS InPools
FROM PR_Needy AS QCR,
   CIM_ServiceAffectsElement AS SAE,
   CIM_StorageConfigurationService AS SCS,
   CIM_StoragePool AS SP,
   CIM_AllocatedFromStoragePool AS AFSP
WHERE QCR.NeedySPPath = SAE.AffectedElement
   AND  OBJECTPATH(SCS) = SAE.AffectingElement
   AND  SP.ElementName = 'FreePool'
   AND  QCR.NeedySPPath = AFSP.Antecedent
   AND  OBJECTPATH(SP) = AFSP.Dependent
```

3) Use the results of the previous `CIM_MethodActionResults` as input to a second `CIM_MethodAction` to take action on an error. It also calls the `CreateOrModifyStoragePool` method.

Required Features: Basic Query, Extended Select List, Complex Join, Array Range, Embedded Properties, Conversion Utilities

```
SELECT MAR.MethodName,
      MAR.MethodParameters.Pool,
      MAR.MethodParameters.Size,
        OBJECTPATH(SP) AS InPools
FROM PR_ModifySP MAR,
      StoragePool SP,
      AllocatedFromStoragePool AFSP
WHERE MAR.ResultValue <> '0'
   AND  SP.ElementName = 'SafetyPool'
   AND  MAR.MethodParameters ISA __MethodParameters
   AND  MAR.MethodParameters.__MethodParameters::Pool =
      AFSP.Antecedent
   AND  OBJECTPATH(SP) = AFSP.Dependent
```

Annex B CQL BNF (normative)

This document uses Augmented BNF (ABNF) with the following exceptions:

- Rules separated by a bar (`|`) shall represent choices (instead of using a slash (`/`) as defined in ABNF).
- Ranges of alphabetic characters or numeric `VALUES` shall be specified using two periods (`..`) between the beginning and ending `VALUES` of the range (instead of using the minus sign (`-`) as defined in ABNF).
- The rules defined in this syntax should be assembled into a complete query by assuming whitespace characters between them, except where noted otherwise. (ABNF requires explicit specification of whitespace.)
- The comma (`,`) shall explicitly designate `concate`nation of rules with all intervening whitespace removed (instead of implicit `concate`nation of rules as specified by ABNF).

NOTE 1: ABNF is not case sensitive.

NOTE 2: The BNF used here and not to the resultant Regular Expression used in Full or Basic Like. In particular, except where noted, whitespace is significant within the resultant Regular Expression.

NOTE 3: UNICODE-CHAR is a Unicode character. The range of allowed code points is the same as the range for the char16 datatype in 6.2. UNICODE-S1 is a subset of UNICODE-CHAR in which the characters from the US-ASCII range (U+0000…U+007F) are limited to the set S1, where S1 = {U+005F, U+0041…U+005A, U+0061…U+007A}. (This is alphabetic, plus underscore.) The encoding form of UNICODE-CHAR is defined by the specification that is using CQL.

NOTE 4: The CQL string (that is, the entire string, beyond just string literals) uses Unicode characters. The encoding of the CQL string is the same as the encoding of UNICODE-CHAR.

Annex C Regular Expressions (normative)

This annex describes the Regular Expression grammar used by CQL.

The grammar is defined in two sections. The first (C.2) is used to construct Regular Expressions used by the Basic Like feature. The second (C.2) is used to create Regular Expressions used by the Regular Expression Like feature.

The Regular Expression grammar described in this annex uses the BNF conventions described in 0

C.1 Basic Like Regular Expressions

Basic Like Regular Expressions is a subset of the XQuery Regular Expression syntax as defined in Regular Expressions.

NOTE: Basic Like Regular Expressions complies with levels RL1.1 and RL 1.7 of Unicode Regular Expressions Level 1, which is a subset of the XQuery Regular Expressions compliance to Unicode Regular Expressions Level 1.

```
blre-ordinary-char= UNICODE-CHAR
```

A character, other than a metacharacter, excluded from the Char production of XQuery Regular Expressions.

```
blre-escaped-char = char-escape | SingleCharEsc
```

An escaped character. The char-escape is defined in **Error! Reference source not found.**. The SingleCharEsc is defined in XQuery Regular Expressions. The "/u" and "/U" syntax of char-escape replaces the character reference syntax defined in XQuery Regular Expressions.

NOTE: The char-escape includes escape sequences that may not be supported by XQuery. The CQL processor may need to convert these escape sequences to a form that is compatible with XQuery.

```
blre-single-char = "." | blre-ordinary-char| blre-escaped-char
```

Single character regular expression. The '.' meta-character matches any character except the newline character (\u000A).

```
blre-multi-char = blre-single-char,"*"
```

Matches multiple occurrences of a single character.

```
blre-expression = *(blre-single-char | blre-multi-char)
```

Basic Like regular expression.

C.2 Full Like Extended Regular Expressions

Full Like Regular Expressions is conformant with the XQuery Regular Expression syntax as defined in Regular Expressions, with the following exceptions:

1) The Unicode characters allowed in the expression are defined by UNICODE-CHAR in the Query Language BNF section.
2) The escape sequences of char-escape in **Error! Reference source not found.** may be used in addition to the escape sequences in SingleCharEsc in XQuery Regular Expressions. The "/u"

and "/U" syntax of char-escape replaces the character reference syntax defined in XQuery Regular Expressions.

NOTE: The char-escape includes escape sequences that may not be supported by XQuery. The CQL processor may need to convert these escape sequences to a form that is compatible with XQuery.

3) None of the flags defined in section 7.6.1.1 of XQuery Regular Expressions are supported, and the expression matching behaves as if all the flags have the default VALUES .

Annex D Datetime Operations and BNF (normative)

The operations on `datetime` and the `datetime` BNF described in this annex will ultimately be incorporated into some other DMTF specification, and references to this annex should be updated to refer to the incorporating specification.

D.1 Datetime Operations

The following operations are defined on `datetime` types:

1) Arithmetic operations:
   - Adding or subtracting an interval to or from an interval results in an interval.
   - Adding or subtracting an interval to or from a timestamp results in a timestamp.
   - Subtracting a timestamp from a timestamp results in an interval.
   - Multiplying an interval with a numeric or vice versa results in an interval.
   - Dividing an interval by a numeric value results in an interval.

   Other arithmetic operations are not defined.

2) Comparison operations:
   - Testing for equality or inequality of two timestamps or two intervals results in a `Boolean` value.
   - Testing for the ordering relation (<, <=, >, >=) of two timestamps or two intervals results in a `Boolean` value.

   Other comparison operations are not defined.

   Note that comparison between a timestamp and an interval, and vice versa, is not defined.

Specifications using the definition of these operations (for instance, query language specifications) should define how undefined operations are handled.

Any operations on `datetime` types in an expression shall be handled as if the following sequential steps were performed:

1) Each `datetime` value is converted into a range of microsecond VALUES , as follows:
   - The lower bound of the range is calculated from the `datetime` value, with any asterisks replaced by their minimum value.
   - The upper bound of the range is calculated from the `datetime` value, with any asterisks replaced by their maximum value.
   - The basis value for timestamps is the oldest valid value (that is, 0 microseconds corresponds to 00:00.000000 in the timezone with `datetime` offset +720, on January 1 in the year 1 BCE, using the proleptic Gregorian calendar). Note that this definition implicitly performs timestamp normalization. Note that 1 BCE is the year before 1 CE.

2) The expression is evaluated, using the following rules for any `datetime` ranges:

- **Definitions**
  - `T(x, y)` is the microsecond range for a timestamp with the lower bound x and the upper bound y.
  - `I(x, y)` is the microsecond range for an interval with the lower bound x and the upper bound y.
  - `D(x, y)` is the microsecond range for a `datetime` (timestamp or interval) with the lower bound x and the upper bound y.
- **Rules**

```
I(a, b) + I(c, d)   :=  I(a+c, b+d)
I(a, b) - I(c, d)   :=  I(a-d, b-c)
T(a, b) + I(c, d)   :=  T(a+c, b+d)
T(a, b) - I(c, d)   :=  T(a-d, b-c)
T(a, b) - T(c, d)   :=  I(a-d, b-c)
I(a, b) * c         :=  I(a*c, b*c)
I(a, b) / c         :=  I(a/c, b/c)

D(a, b) <  D(c, d)  :=  true if b < c, false if a >= d, otherwise
NULL (uncertain)
D(a, b) <= D(c, d)  :=  true if b <= c, false if a > d, otherwise
NULL (uncertain)
D(a, b) >  D(c, d)  :=  true if a > d, false if b <= c, otherwise
NULL (uncertain)
D(a, b) >= D(c, d)  :=  true if a >= d, false if b < c, otherwise
NULL (uncertain)
D(a, b) =  D(c, d)  :=  true if a = b = c = d, false if b < c OR a
> d, otherwise NULL (uncertain)
D(a, b) <> D(c, d)  :=  true if b < c OR a > d, false if a = b = c
= d, otherwise NULL (uncertain)
```

These rules follow the well known mathematical interval arithmetic. An informational link to a definition of mathematical interval arithmetic is http://en.wikipedia.org/wiki/Interval_arithmetic.

Mathematical interval arithmetic is commutative and associative for addition and multiplication, like ordinary arithmetic.

Mathematical interval arithmetic mandates the use of three-state logic for the result of comparison operations, using a special value called "uncertain" to represent that a decision cannot be made. The special value of "uncertain" is mapped to the `NULL` value in `datetime` comparison operations.

3) Overflow and underflow condition checking is performed on the result of the expression, as follows:

- **For timestamp results**
  - A timestamp older than the oldest valid value in the time zone of the result produces an arithmetic underflow condition.
  - A timestamp newer than the newest valid value in the time zone of the result produces an arithmetic overflow condition.

- **For interval results**
  - A negative interval produces an arithmetic underflow condition.
  - A positive interval greater than the largest valid value produces an arithmetic overflow condition.

Specifications using the definition of these operations (for instance, query languages) should define how these conditions are handled.

4) If the result of the expression is again a `datetime` type, the microsecond range gets converted into a valid `datetime` value such that the set of asterisks (if any) determines a range that matches the actual result range, or encloses it as closely as possible. The GMT time zone must be used for any timestamp results.

   For most fields, asterisks can be used only with the granularity of the entire field.

   EXAMPLE: The following are `datetime` examples:

```
"20051003110000.000000+000" + "00000000002233.000000:000"
  evaluates to  "20051003112233.000000+000"
"20051003110000.******+000" + "00000000002233.000000:000"
  evaluates to  "20051003112233.******+000"
"20051003110000.******+000" + "00000000002233.00000*:000"
  evaluates to  "200510031122**.******+000"
"20051003110000.******+000" + "00000000002233.******:000"
  evaluates to  "200510031122**.******+000"
"20051003110000.******+000" + "00000000005959.******:000"
  evaluates to  "20051003******.******+000"
"20051003110000.******+000" + "000000000022**.******:000"
  evaluates to  "2005100311****.******+000"
"20051003112233.000000+000" - "00000000002233.000000:000"
  evaluates to  "20051003110000.000000+000"
"20051003112233.******+000" - "00000000002233.000000:000"
  evaluates to  "20051003110000.******+000"
"20051003112233.******+000" - "00000000002233.00000*:000"
  evaluates to  "20051003110000.******+000"
"20051003112233.******+000" - "00000000002232.******:000"
  evaluates to  "200510031100**.******+000"
"20051003112233.******+000" - "00000000002233.******:000"
  evaluates to  "20051003******.******+000"
"20051003060000.000000-300" + "00000000002233.000000:000"
  evaluates to  "20051003112233.000000+000"
"20051003060000.******-300" + "00000000002233.000000:000"
  evaluates to  "20051003112233.******+000"
"000000000011**.******:000" * 60
  evaluates to  "0000000011****.******:000"
60 times adding up "000000000011**.******:000"
  evaluates to  "0000000011****.******:000"
"20051003112233.000000+000" = "20051003112233.000000+000"
  evaluates to true
"20051003122233.000000+060" = "20051003112233.000000+000"
  evaluates to true
"20051003112233.******+000" = "20051003112233.******+000"
  evaluates to NULL (uncertain)
```

```
"20051003112233.******+000" = "200510031122**.******+000"
  evaluates to NULL (uncertain)
"20051003112233.******+000" = "20051003112234.******+000"
  evaluates to false
"20051003112233.******+000" < "20051003112234.******+000"
  evaluates to true
"20051003112233.5*****+000" < "20051003112233.******+000"
  evaluates to NULL (uncertain)
```

D.2 Datetime BNF (Normative)

The Datatime grammar below uses BNF as defined in 0.

```
dt-decimal-digit  =  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
"9"
```

Allowed decimal digits

```
dt-single-quote = "'"
```

A single Quote

```
dt-two-time-digits = (2*2(dt-decimal-digit)) | ("**")
```

A two digit field for time. This field either contains two valid decimal digits or it contains asterisks ("*"), which signifies that those digits are not significant.

```
dt-microsecond-digits = 6*6(dt-decimal-digit)
                                  |   5*5(dt-decimal-digit), ("*")
                                  |   4*4(dt-decimal-digit), ("**")
                                  |   3*3(dt-decimal-digit), ("***")
                                  |   2*2(dt-decimal-digit), ("****")
                                  |   1*1(dt-decimal-digit), ("*****")
                                  |   ("******")
```

A six character field for microseconds. The lower order digits may be specified as a field of asterisks ("*"), which signifies that those lower order digits is not significant.

```
dt-datetime = dt-single-quote,
             (dt_timestamp-specification | dt_interval_specification),
              dt-single-quote
```

See the CIM Infrastructure Specification (DSP0004) for a detailed description of the use of a `datetime` .

A timestamp with the year field set to 0000 is interpreted as the year 1 BCE. A year field set to 0001 is interpreted as the year 1 CE.

```
dt-timestamp-specificaton = (14*14("*")
                 ".", ("******"), ("+"|"-"), 3*3(dt-decimal-digit))
```

A wildcard timestamp specification

```
|     (4*4(dt-decimal-digits), "**********",
                ".", ("******"), ("+"|"-"), 3*3(dt-decimal-digit))
```

A timestamp specifying a precision of years

```
|      (6*6(dt-decimal-digits), dt-two-time-digits, "******",
                ".", ("******"), ("+"|"-"), 3*3(dt-decimal-digit))
```

A timestamp specifying a precision of months

```
|     (8*8(dt-decimal-digits), dt-two-time-digits, "****",
                 ".", ("******"), ("+"|"-"), 3*3(dt-decimal-digit))
```

A timestamp specifying a precision of days

```
|     (10*10(dt-decimal-digits), dt-two-time-digits,"**",
                ".", ("******"), ("+"|"-"), 3*3(dt-decimal-digit))
```

A timestamp specifying a precision of minutes

```
|     (12*12(dt-decimal-digits), dt-two-time-digits,
              ".", ("******"), ("+"|"-"), 3*3(dt-decimal-digit))
```

A timestamp specifying a precision of seconds

```
|     (14*14(dt-decimal-digits),
              ".", (dt-microsecond-digits), ("+"|"-"), 3*3(dt-decimal-
digit))
```

A timestamp specifying a precision of microseconds

```
dt-interval-specification = ( (14*14("*"), ".", ("******"), (":"),
3*3("0"))
```

A wildcard interval specification

```
|     (8*8(dt-decimal-digit) | ("********")), ("******"),
              ".", ("******"), (":"), 3*3(dt-decimal-digit))
```

An interval specifying a precision of days

```
|     (8*8(dt-decimal-digits), dt-two-time-digits, "****",
              ".", ("******"), (":"), 3*3(dt-decimal-digit))
```

An interval specifying a precision of hours

```
|     (10*10(dt-decimal-digits), dt-two-time-digits,"**",
                 ".", ("******"), (":"), 3*3(dt-decimal-digit))
```

An interval specifying a precision of minutes

```
|     (12*12(dt-decimal-digits), dt-two-time-digits,
              ".", ("******"), (":"), 3*3(dt-decimal-digit))
```

An interval specifying a precision of seconds

```
|     (14*14(dt-decimal-digits),
            ".", (dt-microsecond-digits), (":"), 3*3(dt-decimal-digit))
```

An interval specifying a precision of microseconds

Annex E Additional Query Language Features (normative)

Section 6.4 defines the features required for CQL support. The following subsections each describe additional CQL features.

The features described in this section are all experimental and may change as a result of implementation experience.

E.1 Simple Join

The `from-criteria` production from Basic Query is enhanced to support a join of two elements defined by `from-specifier`.

```
from-criteria  = <as specified in Basic Query>
|  from-specifier "," from-specifier
```

E.2 Complex Join

The `from-criteria` production from Simple Join is enhanced as follows.

The `FROM` clause shall support a join of more than two `from-specifiers`.

```
from-criteria  =  <as specified in Simple Join>
|  from-specifier *("," from-specifier)
```

E.3 Subquery

The `from-specifier` production from Basic Query is enhanced as follows.

```
subquery = select-statement
```

A subquery is defined by a `select-statement`

```
from-specifier  =  <as specified in Basic Query>
|   "(" subquery ")" identifier
```

This production defines `identifier` as a name by which the rows returned by the subquery are identified. The subquery is self-defined. There is no correlation between identifiers used within the `select-statement` of the subquery and those used within the `select-statement` containing the subquery.

E.4 Result Set Operations

The DISTINCT and FIRST operators shall be supported and the ORDER BY clause shall be supported. The Extended Select List feature is a prerequisite.

```
sort-spec = expr ( ASC | DESC )
```

The specified `expr` shall be defined in the SELECT clause. Properties resulting from the specification of a star-`expr` as the `selected-entry` can be subject to sorting. `NULL VALUES` are considered "higher" than all other `VALUES` . If the ORDER BY clause does not completely order the instances of the result set, instances with duplicate `VALUES` in sorting properties will be displayed in an arbitrary order.

```
sort-spec-list = sort-spec *( "," sort-spec )
```

The sort-spec entries are listed in order of sorting preference; the first entry on the list defines the primary sort order of the query result.

```
select-statement = <as defined in Basic Query>

| SELECT [FIRST unsigned-integer][DISTINCT] select-list
   FROM from-criteria
   [WHERE search-condition]
   [ORDER BY sort-spec-list]
```

If DISTINCT is used, all but one of each set of duplicate rows will be eliminated from the result set. Two instances are considered duplicates of one another if and only if the `VALUES` of all of the properties (including those of embedded instances), are equal after the projection operation has been executed. When determining duplicates, two `NULL VALUES` are considered equal.

If FIRST is used, the result set will contain only the first n rows. Typically, this clause is used with ORDER BY to define a specific and repeatable sort order of the results and then to define the number of instances to return. Note that the sort order for string or char16 data is defined by the rules for operator “=”, operator “<”, and operator “>” in the Comparison section. Note that if DISTINCT is also specified, the duplicate entries are eliminated before the FIRST *n* instances are determined. If *n* instances do not exist, then all the available instances are returned and the query completes normally.

```
arg-list = <as defined in Basic Query>

|         DISTINCT expr
```

This production specifies that only distinct `VALUES` selected by the `expr`ession shall be used as input to the query language function.

E.5 Extended Select List

The `select-list`:

- shall support functions
- shall support `CLASSQUALIFIER` and `PROPERTYQUALIFIER`
- shall support the `AS` construct for property aliasing

An underlying assumption is that function support in the Select list requires the `AS` construct to provide a column name.

```
selected-entry  =  <as defined by Basic Query>
|         expr AS identifier
```

To avoid duplicated column names in the query result, the "`AS identifier`" clause is used to explicitly specify a name. If the "`AS identifier`" clause is not present, then the selected entry shall be a property reference and the `expr` itself (minus any whitespace) is taken as the name of the corresponding result column.

The "`AS identifier`" clause shall be used for all forms of `expr` that do not resolve to a property name. This includes the use of functions and all forms of arithmetic `expr`essions.

```
chain = <as defined in Basic Query>
|  identifier1 PROPERTYQUALIFIER identifier2
```

This production refers to a property qualifier. `identifier1` shall unambiguously identify a property (see 7.1.1), and the type of the `expr`ession is the type of that qualifier. If the property does not expose a qualifier with this name, the qualifier's default value applies.

```
|  chain "."  identifier "#" literal-string
```

`chain`, and `identifier1` together identify a property, as described in 7.1.1. This production refers to the value of a property qualifier from that property, and the type of the `expr`ession is the type of that qualifier. If the property does not expose a qualifier with this name, the qualifier's default value applies.

```
|  chain CLASSQUALIFIER identifier
```

`chain` shall be of type object of *C* for some class *C*. This production refers to a qualifier on that class, and the type of the `expr`ession is the type of that qualifier. If the class does not expose a qualifier with this name, the qualifier's default value applies.

E.6 Embedded Properties

The query shall support the ability to reference the properties of the embedded instance.

```
chain = <as specified in Basic Query>
|  chain "."  identifier
```

`chain` shall have a type of object of *C* for some class *C*.

`identifier` shall be the name of a property. For details on the selection of the identified property, see 7.1.1. The type of this production is the type of the property.

```
|  chain "." identifier "#" literal-string
```

chain, and identifier together identify a property, as described in 7.1.1. This production forms a symbolic constant based on the VALUES and VALUEMAP qualifiers (see 7.1.4). The type of this expression is the type of the identified property.

E.7 Aggregations

The query shall support aggregation functions in the select clause. The Extended Select List feature is a prerequisite, (see E.5.) The Conversion Utilities feature is a prerequisite, (see E.14.)

These functions are valid only within the select-list. If the select-list contains any aggregating expressions, then all items in the select-list shall be aggregating expressions. In this case, the result set contains one row and the aggregating expressions operate on the rows determined by the WHERE clause. An aggregating expression is an expression with at least one aggregation function, in which any properties are used only in the expression representing the argument of an aggregation function.

- **COUNT([DISTINCT] expr)**

  This function counts the number of rows for which the argument is non-NULL. If DISTINCT is specified, then COUNT counts the number of different non-NULL VALUES that the argument assumes. The set of rows that COUNT considers is affected by including FIRST or DISTINCT on the select-statement. The result type is uint64.

- **COUNT(*)**

  COUNT(*) is a special function returning the number of rows the query selects. The value returned by COUNT is affected by including FIRST or DISTINCT on the select-statement. The result type is uint64.

- **MIN(expr), MAX(expr), SUM(expr)**

  These functions all act analogously to the like-named SQL functions. The argument to each function must have a numeric type; the result is of the same type as the argument. The result type is the same as the type of expr.

- **MEAN(expr), MEDIAN(expr)**

  These functions compute the mean and median, respectively, of the distribution represented by the non-NULL VALUES that the arguments assume. The result type for MEAN is real64. The result type for MEDIAN is the type of expr.

E.8 Regular Expression Like

The WHERE clause shall support for the like-predicate with the capabilities defined in C.2.

```
comp = <as specified in Basic Query>
|  arith LIKE arith
```

Both sides of the LIKE comparison must have a string or char16 type; the result has a Boolean type. The LIKE comparison allows a string or char16 to be tested by pattern-matching, using special characters in the pattern on the right side (see C.2).

If either arith is NULL, then the production evaluates to NULL.

E.9 Array Range

The query shall support the full range of `array-index`-list productions in order to compare Array properties with Non-Array properties as described in 5.6 or in order to compare parts of arrays.

The WHERE clause shall support the array-`comp` production.

```
array-index-list = <as defined in Basic Query>
|            array-index "," array-index
```

The `array-index`-list specifies one or more elements of an array.

```
|            "*"
```

This `array-index`-list refers to all the elements of the array.

```
|            ""
```

This `array-index`-list refers to none of the array elements. *x*`[]` is an empty array with the same type as *x*, for any *x* with array type.

```
comp = <as defined in Basic Query>
|         array-comp
```

Add array comparison.

```
arith-or-value-symbol = arith | value-symbol
array-comp = ( ANY | EVERY ) arith comp-op arith-or-value-symbol
```

arith shall have type array of *T*. Each element of `arith` 's value will be compared to the value of the `arith-or-value-symbol`. If ANY is specified, the results of these comparisons are combined as if by OR; if EVERY is specified, the results are combined as if by AND.

```
|  arith-or-value-symbol comp-op ( ANY | EVERY ) arith
```

This production acts like the preceding one, except that the array value represented by arith appears on the right side.

```
array-index = <as defined in Basic Query>
|   expr ".." [ expr ]
```

Both `expr` VALUES shall have an unsigned integer type. The `..` notation is used to specify ranges of indices within an array.

```
|   ".." expr
```

`expr` shall have an unsigned integer type.

E.10 Satisfies Array

The WHERE clause shall support the SATISFIES clause. This feature extends the Array Range feature.

```
array-comp = <as defined in Array Range>
|   (ANY | EVERY ) identifier IN expr SATISFIES "(" comp ")"
```

The SATISFIES construct makes `identifier` available as a name whose scope is the included `comp`. `expr` shall have type array of *T*, in which case `identifier` will have type *T* within `comp`. `identifier` shall not be the same as any name established by the `from-criteria` and shall not be the same as any name established by any surrounding SATISFIES clauses.

E.11 Foreign Namespace Support

The query shall support references to namespaces other than the one in which the query is executed.

Following is an additional production for class-path

```
class-path  = <As defined in Basic Query>
|             literal-string "." class-name
```

If specified, `literal-string` shall conform to the format of the namespacePath production defined in the *WBEM URI Specification v1.0* (DSP0207).

E.12 Arithmetic Expression

The query must support arithmetic `expressions` using +, -, *, and /.

```
factor = concat
|          ("+" | "-") concat
```

When this production is used, `concat` shall have a numeric type, which will be the type of the production.

If `concat` is `NULL`, then the production evaluates to `NULL`.

```
term = <as defined in Basic Query>
|          term "*" factor
```

If `term` and `factor` both have numeric types, the production has a numeric type.

If `term` has a numeric type, and `factor` has a `datetime` type and evaluates to an interval, then the production has `datetime` type and will produce an interval value.

If `term` has `datetime` type and evaluates to an interval, and `factor` has a numeric type, then the production has a `datetime` type and will produce an interval value. The rules for operations with `datetime` type operands are defined in D.1.

If `term` or `factor` is `NULL`, then the production evaluates to `NULL`.

No other type combinations are allowed.

```
|          term "/" factor
```

If `term` and `factor` both have numeric types, the production has a numeric type.

If `term` has a `datetime` type and evaluates to an interval, and `factor` has a numeric type, the production has a `datetime` type and will produce an interval value. The rules for operations with `datetime` type operands are defined in D.1.

If `term` or `factor` is `NULL`, then the production evaluates to `NULL`.

No other type combinations are allowed.

```
arith = <as defined in Basic Query>
|         arith ("+" | "-") term
```

If arith and `term` both have a numeric type, the result has a numeric type.

If arith and `term` have `datetime` types, then refer to D.1 for a definition of the operation.

No other type combinations are allowed.

If `arith` contains multiple occurrences of arithmetic operators, normal mathematical precedence rules apply.

If `arith` or `term` is `NULL`, then this production evaluates to `NULL`.

E.13 Full Unicode

The comparison operators shall behave as if the normalization defined in *Character Model for the World Wide Web 1.0: Normalization*, section 4 "String Identity Matching", was applied and then the default collation order defined in the Unicode Collation Algorithm was used on the resulting strings. Note that this collation order accommodates most languages, without having to take any locales into account.

In CQL Basic Query the Unicode Collation Algorithm is not required, see 7.1.6.

E.14 Conversion Utilities

This feature adds support for conversion utilities. Use of conversion utilities in a `select-list` depends on the Extended Select List feature.

```
chain = <As defined in Basic Query>
|  identifier "(" arg-list ")"
```

`identifier` shall be the name of a query language function. See 7.2 for type rules of function calls. The `numeric`, `string`, `instance`, `path`, `pathname`, and `datetime` functions shall be supported.

NOTE: This syntax does not describe the invocation of a method defined on a CIM class.

E.15 Property Scope

The following production defines how properties are scoped by their defining class in CQL.

```
property-scope = class-path "::"
```

The scoping operator "::" provides a class within which the property name identifier is interpreted. Generally, the class of the property is sufficient. However, if a property of a class is covered by another property that has the same name and that belongs to a subclass, then the

"::" syntax is required to access the covered property when it is in the scope of the covering subclass. Details on how to determine which property to use are provided in 7.1.1.

```
chain = <As defined in Basic Query>
|  property-scope identifier
```

`Property-scope` declares that the `identifier` identifies a property exposed by the `property-scope classname` (see 7.1.1). The type of the property is taken as the type of this production.

```
|  chain "." property-scope identifier
```

`chain` shall have a type of object of *C* for some class *C*.

`identifier` shall be the name of a property. For details on the selection of the identified property, see 7.1.1. The type of this production is the type of the property.

For Basic Query, `chain` is restricted to a single class name or class alias bound in the `FROM` clause because Basic Query does not support extraction of properties from embedded objects.

```
|  chain "." property-scope identifier "#" literal-string
```

`chain`, `property-scope`, and `identifier` together identify a property, as described in 7.1.1. This production forms a symbolic constant based on the `VALUES` and `VALUEMAP` qualifiers (see 7.1.4). The type of this expression is the type of the identified property.

For Basic Query, `chain` is restricted to a single class name or class alias bound in the FROM clause because Basic Query does not support extraction of properties from embedded objects.

```
star-expr = <As in Basic Query>
|  chain "." property-scope "*"
```

`chain` shall have type object of *C* for some class *C*. `property-scope` identifies some class *S*. *S* shall be the same class as, or be a superclass of, class *C*; this production refers to all the properties exposed by *S*, including those of *S*'s superclasses. Properties of subclasses of *S* are not included in the set. The property list produced does not vary over the query.

Annex F CIM Query Template Language (normative)

This section defines an experimental, separate and optional pre-processing facility that supports the conversion of CQL template strings into CQL strings.

The CQLT processing facility parses the input string from left to right for CQLT tokens. Each CQLT token represents a CQLT variable named by `identifier`.

- The CQLT processor recognizes a backslash (`\`) as an escape character when the next character is a single-quote (`'`) (U+0027).

  NOTE: The escaping of double quotes is not necessary within a literal string, because only single quotes can be used to delimit string literals. If the entire CQLT string is put into an environment that uses double quotes to delimit a string (for example, as a default value for properties in the MOF), then that environment must define the escape rules for double quotes.

- If a non-escaped single-quote is encountered, detection of CQLT tokens is disabled until the first character after a corresponding non-escaped single-quote.

- While detection is enabled, the sequence "$"`identifier`"$" is recognized as a CQLT token.
- For each CQLT token encountered, the CQLT processor makes a string substitution for that token and resumes parsing with the first character after the replaced token.

The string substitution replaces the token with the value of the CQLT variable as defined to the pre-processing facility. The value of the CQLT variable must be a string value. Note that any occurrences of the sequence "$"`identifier`"$" in that string value will not be replaced. The mapping of a CQLT variable to a value is not specified here and must be specified where this facility is used.

CQLT tokens are semantically unrelated to the `identifiers` of the CQL query itself.

Unquoted $'s may not appear in the query template except as part of pre-processing tokens.

Following the convention on identifying a query language detailed in 6.1, the string "`DMTF:CQLT`" will identify the CIM Query Template Language to represent the use of this pre-processing capability for CQL.

7.5 CQLT Examples

Following are three examples which show the input to a query template processor and the output from that processor.

1) Define a template for retrieving instances of the class identified by the *targetClassName* variable.

   Assuming the value of *targetClassName* is "`CIM_StorageExtent`", the CQLT would translate the following string:

```
    SELECT *
    FROM $targetClassName$
into
    SELECT *
    FROM CIM_StorageExtent
```

2) Define a template for requesting account information about the entity identified by the *UserID* variable.

   Assuming the value of UserID is "guest", the CQLT processor would translate the string

```
    SELECT *
    FROM CIM_Account
     WHERE UserID = $UserID$
into
    SELECT *
    FROM UserID = 'guest'
```

3) Define a template that allows the filter condition of a CIM_IndicationFilter to be restricted to a particular provider and with a selectable level of severity.

   Assuming the value of TemplateVariable[0] = 'AcmeWidgets' AND TemplateVariable[1] = '2', the CQLT processor would translate the string

```
        SELECT *
    FROM CIM_AlertIndication
    WHERE ProviderName = $TemplateVariable[0]$
    AND PerceivedSeverity > StringToUINT($TemplateVariable[1]$)
into
        SELECT *
    FROM CIM_AlertIndication
    WHERE ProviderName = 'AcmeWidgets'
    AND PerceivedSeverity > '2'
```

Annex G Acknowledgements(informative)

This document is based on an original WBEM Query Language Specification submitted by Patrick Thompson of Microsoft.

The authors wish to acknowledge the following people.

Authors:

- George Ericson – EMC Corporation
- Jeff Piazza – Hewlett-Packard Company
- Andrea Westerinen – Microsoft Corporation

Contributors:

- Andreas Maier – IBM Corporation
- Oliver Benke – IBM Corporation
- Lee Vantine – EMC Corporation
- Aaron Merkin – IBM Corporation
- Brian Lucier – IBM Corporation
- Dave Sudlik – IBM Corporation
- Asad Faizi – Microsoft Corporation
- Jim Davis – WBEM Solutions, Inc.
-

Annex H Bibliography (informative)

This section contains a list of the external references and dependencies for this specification that are not otherwise listed in the Normative References section (2).

DMTF, DSP0200, *CIM Operations over HTTP Specification* v1.3, draft

In this document, the `term` Unicode refers to the Universal Character Set (UCS), defined jointly by the Unicode Standard and ISO/IEC 10646.

The Unicode Consortium, *The Unicode Standard, version 4.1*, ISBN 0-321-18578-1, as updated from time to time by the publication of new minor versions. See http://www.unicode.org/unicode/standard/versions for the latest version and additional information on versions of the standard and of the Unicode Character Database.

ISO/IEC 10646:2003, *Information Technology – Universal Multiple-Octet Coded Character Set (UCS)* as, from time to time, amended replaced by a new edition or expanded by the addition of new parts. See http://www.iso.org for the latest version.

W3C, Working Draft, *Character Model for the World Wide Web 1.0: Normalization*, February 24, 2004,

The Unicode Consortium, Unicode Collation Algorithm (Unicode Technical Standard #10), as, from time to time, amended, replaced by a new edition, or expanded by the addition of new parts. See Unicode Consortium Web site for the latest version.

The Unicode Consortium, *Unicode Regular Expressions (Unicode Technical Standard #18)* as, from time to time, amended, replaced by a new edition, or expanded by the addition of new parts.

XQuery 1.0 and XPath 2.0 Functions and Operators, section 7.6.1 Regular Expression Syntax.

Annex I Change Log (informative)

Version	Date	Author	Description
0.1	2002/10		Initial release of the CIM Query Language definition. Document is based on work in the WBEM Interoperability Working Group and the original WBEM Query Language proposed and documented in 2000.
0.2	2002/11		Corrected one example in Section 5 and acknowledged that more examples/use cases need to be provided.
0.3	2003/01		Updates to the CIM Query Language BNF based on email feedback from Dan Nuffer; Completion of Section 3.2; Addition of information regarding what is returned by specific query examples in Section 5.
0.4	2003/01		Clarified requirement for ISA function as mechanism to query class inheritance/hierarchy, and added check for a class' Version qualifier.
0.5	2003/09		Updated much of the text previously missing, defined additional examples, clarified the text of the examples to indicate that "query-specific" instances are returned, clarified that _KEY is a complete instance path and that a property value of "*" indicates all properties + _KEY, _CLASS and _VERSION, added a section on naming of the returned "query row instances" (3.2), corrected the BNF rules, cleaned up many of the comments ("//") in the BNF, and added many capabilities to the BNF and/or corrected BNF errors. The ability to specify aliases and subqueries was also added at this time.
0.6	2003/09		Updated internal document version number, corrected example that still included the BETWEEN construct, and defined requirement for properties to be returned in the order specified in the SELECT clause.
0.7	2003/10		Updated internal document version number and made clarification changes and minor corrections to the text and BNF. Specifically, the following changes were made: - `CIM_ERR_NOT_SUPPORTED` is ambiguous, used `CIM_Error` instead - Added ability to reference a **specific-class-property-identifier** in select-string-literal - Added [("."property-identifier)*] to **specific-class-property-identifier**, deleted embedded_object in the property-identifier definition, and deleted the embedded_object definition – To allow arbitrary depth of embedding in class_property_identifier - Moved "alias" from class-list in the `from-criteria` to the individual class-names in class-list Eliminated recursive definition of sort-spec-list, and defined a "sort-spec" entry
0.8/0.9	2004/01		Updated internal document version numbers and made many changes simplifying and clarifying the text and BNF, based on Interop and DMTF member review feedback. Also, added an Acknowledgements Section.
0.10	2004/02		Many updates to deal with member comments. _KEY renamed to _OBJECTPATH. _CLASS renamed to _CLASSPATH. _VERSION eliminated.

Version	Date	Author	Description
			Extended BNF to added support for Character and Arithmetic operations. Added Symbolic constants.
0.11	2004/03	George M Ericson	Updates to cover review comments Clarified CQL Feature: Remove 'MAY NOT' clauses Isolate complex Array processing from Basic Do not include Array ANY/EVERY processing Make consistent with ABNF: IETF RFC 2234. With several exceptions called out. Isolated URI BNF to appendix. Expectation that this will move into WBEM URI spec and to reference RFC2396, or equivalent. Added ANY/EVERY/ SATISFIES syntax to clarify Array element references. Add use case for CREATEARRAY. "For MethodAction…" Clarified descriptions for DISTINCT and FIRST Agreed to include LIKE Posix API as optional feature. Simple LIKE functionality is defined as a Posix subset, described in chapter 3.3 Many editorial changes Allow White Space between "." period operator. Added "," operator to BNF to make explicit when White Space is not allowed. Make clear that Query does NOT execute intrinsic methods Agree to capitalize all keywords. However, note that these are not case sensitive. Added production for parenthesization in arithmetic-expression. Switched from properties for Path elements to using Path functions. Removed all references to Qualifying Class. Remove references to new errors. These can not be introduced with this revision. Add language that covers comparison between arrays for • Bag: set match • Ordered: element by element match to maxsize of both arrays. • Indexed: element by element match to maxsize of both arrays. Added Scoping: The incorporating identifier MAY be named in an ISA comparison-predicate of the WHERE clause. This serves to specify the class of the embedded object as used in the `select-list` and the containing `boolean`-primary of the `search-condition`. A different class MAY be compared to in different `boolean`-primaries. The outermost ISA class in a class-hierarchy that compares TRUE scopes the properties that MAY be referenced in the `select-list`. Add ISA back into the spec. Implementation casts object paths to internal REFs and compare based on the internal form. The implementation should know alternative, equivalent forms of NamespacePath and treat them all as equal. Do not allow use of LIKE on result of OBJECTPATH( ). Only support =, <>.

Version	Date	Author	Description
			Add capability to make case in-sensitive comparisons. Add UpperCase function. Created and added table of conversions. Added arithmetic-expression Added Scopingclass function Added use-case examples. Defined QueryResult subclass usage A reference is represented as an Object Path. A property that is a reference MAY be named in the Select-Critera. Add semantics for ANY/EVERY/SATISFIES as proposed by Jeff. Select classname.* returns only properties defined in named class or its superclasses
0.12	2004/04	George M Ericson	Updates to cover review comments Made Scopingclass be ScopingType function Clarify that Path_functions are part of the basic functions Clarified prerequiste column Clarified errors Clarified string definition Removed Truth VALUES from arithmetic expressions Clarified Count Clarified Regular Expression use by Basic and Regular Expression Like.
0.13	2004/05	George M Ericson	Updates to cover review comments Simplify Basic Like Clarify conversion table Man y corrections
0.14		George M Ericson	Review resolutions
0.15		George M Ericson	More review resolutions. Accepted by Interop pending resolution of set of issues
0.16		George M Ericson	Resolution resulted in conversion to compilable BNF. This is a significant revision.
0.17		George M Ericson	Resolution of issues after conversion.
0.18		George M Ericson	(Company Review Version, Version 1.0.0 Prelim) Clarify that Timestamp 0 is 1 BCE; Remove notes from text.
1.0.0f	2005/12/15	George M Ericson	Applied CRs WIPCR00251.001, WIPCR00231.009
1.0.0f	2006/02/02	George M Ericson	Applied CRs WIPCR00255.002, WIPCR00242.007, WIPCR00240.002
1.0.0f	2006/02/06	George M Ericson	Applied CRs WIPCR00270.000.htm
1.0.0f	2006/02/08	George M Ericson	Applied CRs WIPCR00272.002.htm, WIPCR00268.001.htm
1.0.0g	2006/02/10	George M Ericson	Applied CRs WIPCR00261.002.htm, WIPCR00247.006.htm
1.0.0g	2006/02/15	George M Ericson	Fixed typo wrt closing parenthesis after char-escape
1.0.0g	2006/02/16	George M Ericson	Applied CRs WIPCR00245.008.htm, WIPCR00269.001.htm, WIPCR00271.002.htm
1.0.0g	2006/02/27	George M Ericson	Applied CRs WIPCR00266.001.htm, WIPCR00268.001.htm, WIPCR00265.001.htm, WIPCR00264.000.htm, WIPCR00263.000.htm, WIPCR00262.000.htm, WIPCR00254.003.htm, WIPCR00248.001.htm
1.0.0g	2006/03/16	George M Ericson	Applied CRs WIPCR00280.000.htm, WIPCR00282.000.htm Updated reference numbers
1.0.0h	2006/03/22	George M Ericson	Ballot version of the spec
1.0.0i	2007/04/12	George M Ericson	cPubs ISO template update

Version	Date	Author	Description
1.0.0j	2007/05/01	George M Ericson	cPubs ISO template update + cPubs comment resolutions
1.0.0k	2007/06/02	George M Ericson	Added cPubs requested section lead-in paragraphs Moved each non-basic feature to separate feature specific Annexes. Applied WIPCR00398.003
1.0.0l	2007/07/23	George M Ericson	Applied: WIPCR00415.001: Move all experimental features to separate clauses in Annex. This CR also finishes making updates to implement cPubs recommendations. WIPCR00418.002: Create Conversion Utilities and Property Scope features. WIPCR00420.002: - Make Result Set Operations feature be dependent on Extended Select List feature. - Restrict select-list of Basic Query to allow only properties of at most one class in the From list. - Restrict Chain in Basic Query to not allow an embedded property. - Add boolean to type lattice. WIP_CQLCR00002.002: - Reflect change from ExecuteQuery to OpenQueryInstances throughout specification. - State that properties that are reserved words must be qualified by ClassName. - Update all examples to reflect accumulated updates.
1.0.0m	2007/08/13	George M Ericson	Updates to cover review comments from Final ballot: Tracked as issues #110, #111, and #113. (#112 was deferred to v1.1) - Issue #110 and #111: Updates to section (now 7.4) Query Errors. - Issue #113: Move was section 7 to now Annex F CIM Query Template Language and mark as experimental - Add new section 7 CIM Query Language Considerations in front of prior subsections 6.5 and clarify that contents apply to both section 6 for basic and Annex E for extended features.