CIM Query Language Specification

DSP0202 ***Status: Second Preliminary - Pending***

CIM Query Language Specification

Version 1.0.0h Second Preliminary - Pending
March 22, 2006

Abstract

The DMTF Common Information Model (CIM) utilizes 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

Table of Contents

1 Introduction and Overview

CIM and 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 Specification over HTTP [11] and some CIM classes within the Event [14] and Policy [15] Models. 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 of the CIM Operations over HTTP Specification [11].

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 enables CIM service implementations, especially on 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 processing point further along. It is important to note that the query template language can be used to support the query engine, but is not part of the formal query language itself.

2 Background Materials

CIM's query design is based on concepts from both ISO/IEC's Structured Query Language [12] (SQL-92) and W3C's XML-Query [13]. 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.

3 Terminology

Term	Definition
CIM	Common Information Model, an object-oriented definition of a managed enterprise or Internet environment.
CIM Indications	A CIM class hierarchy, starting at CIM_Indication, which defines the data in various types of management notifications.
CIM service	A service that provides access to CIM object instances.
From-criteria	A definition of the range of data over which a query is conducted
Query	An act of asking for specific data / For purposes of this document, a query will specify 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), plus other processing options.
Search-condition	A specification of the criteria/conditions that select data to be returned in a query result.
Select-list	A definition of the specific data to be returned in a query result.
SQL	Structured Query Language [12] (SQL-92).
WBEM protocol	A protocol specified by DMTF for accessing a CIM service over the internet. One of these is defined by the CIM Operations over HTTP specification [11].
WBEM service	A CIM service that supports WBEM protocol interfaces.
XML-Query	XML-based Query Language from W3C.

4 Requirements and Concepts

The CIM Query Language has been widely anticipated and exploited in the CIM Operations over HTTP Specification, by the CIM Events Model [14], and by the CIM Policy Model [15]. The language defines the desired instance-level data ranging over a certain set of objects to be returned as the result of an ExecuteQuery CIM operation. Also, it defines the conditions and data for Indications returned as a result of one of the following:

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

Query semantics MUST include instance property projection (e.g., a SQL SELECT clause) and a range (e.g., a SQL FROM clause) and MAY include predicate logic (e.g., a SQL Where clause). This support (defined specifically using the keywords Select-From-Where) was included in a preliminary version of the CIM Query 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.

As noted above, instance property projection MUST be supported. This 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 and/or Indication object. (These requirements are provided by the specific or array class-property-identifier and select-string-literal constructs, respectively.) In addition, the CIM Query Language MUST:

- Support the ability to project meta-data such as instance name and instance class into a response (see the OBJECTPATH() and CLASSPATH() methods, respectively).
- Support query of class versioning information (see the query of CLASSQUALIFIER data).
- Define and support a mechanism for querying class inheritance/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, since 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 must be defined in the basic query language in order to have a consistent BNF and query engine. These requirements are:

- 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 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 MUST support polymorphism. This means, if a query is issued against a base class, all derived class instances will be considered as well. For instance, consider:

```
SELECT *
FROM CIM_Indication
```

This would match all instances of derived classes of CIM_Indication.

5 CIM Query Language (CQL)

5.1. CQL Introduction

In its simplest form, the CIM Query Language 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:
  - 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, this may be modified using the keyword AS followed by a new name.
  - Literals, named via the keyword AS followed by a name.
  - Function results, named via the keyword AS 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:

- the ability to process arrays via indices,
- the ability to query the properties of EMBEDDEDOBJECTs, and
  the ability to traverse associations (based on the values of their REF properties).

Queries are used to define the operation of some CIM classes, (e.g. CIM_IndicationFilter, CIM_MethodAction and CIM_QueryCondition). If using CIM Operations, and if supported, a client MAY issue a query via the ExecuteQuery operation (see the CIM Operations over HTTP specification [11].

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.

5.2. Identifying the CIM Query Language

In order to ensure uniqueness, valid values for query-language SHOULD conform to the following syntax: <organization id>":"<language id>.

<organization id> MUST NOT include a colon (":") and MUST include a copyrighted, trademarked or otherwise unique name that is owned by the entity that had defined query language. For DMTF defined query languages, the <organization id> is "DMTF".

The <language id> MUST 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.

5.3. The Query Language Type Lattice

The CQL type system incorporates the type system of the CIM Infrastructure Specification [1][11], but 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*), or
- the class *C* itself, or one of *C*'s subclasses.

Note that 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".

There is an "object" type that is a supertype of "object of C" type, for all classes, C.

There is a "reference" type that is a supertype of "C REF" type, for all classes, C.

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.

CIM defines a "datetime" type, which contains either timestamp or interval values. Note that timestamp and interval are not defined as explicit types within CIM, but are defined by Appendix E: Datetime Operations and BNF. 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.

There is a "string" type that is the CIM datatype string. It contains a sequence of Unicode [4] 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.

There is a "char16" type that is the CIM datatype char16. It contains one Unicode [4] 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.

The CIM Infrastructure Specification [1] also defines a system of array types, which is similarly extended. That is, 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, i.e., 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 that accompany the grammar, below. Any CQL construct which has been assigned a particular type is said also to "have" all the supertypes of that type. E.g., an expression which has been assigned type "object of CIM_ManagedElement" also "has" type "object".

5.4. Query Language BNF

The CQL grammar below uses Augmented BNF (ABNF) [3] with the following exceptions.

1. Rules separated by a bar (|) represent choices. (Instead of using a slash (/) as defined in ABNF).
2. The rules defined in this syntax are meant to be assembled into a complete query by assuming whitespace characters between them. (ABNF requires explicit specification of whitespace.)
3. The comma (,) is used to explicitly designate concatenation of rules with all intervening whitespace removed. (Instead of implicit concatenation of rules as specified by ABNF.)

Notes:

1. ABNF is NOT case-sensitive.
2. UNICODE-CHAR is a Unicode [4] character. The range of allowed codepoints is the same as the range for the char16 datatype in the “CIM Query Type Lattice” section. UNICODE-S1 is a subset of UNICODE-CHAR where 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.
3. The CQL string (i.e. the entire string, beyond just string literals) uses Unicode [4] characters. The encoding of the CQL string is the same as the encoding of UNICODE-CHAR.

In the following BNF, **bold** text marks a Basic Query component and *italicized* text marks components not in the Basic Query feature.

The grammar for all features is defined as follows. As much as possible, this grammar is constructed to be LALR(1)-parsable.

5.4.1. Reserved Words

```
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"

5.4.2. String Literals

single-quote = "'"

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

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

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 case insensitive. The meaning of these escape characters is:

\\ - 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 [4] code point. Note: the hexidecimal value is not in an encoded form, but 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 [4] code point. Note: the hexidecimal value is not in an encoded form, but is given as a code point. The range of allowed code points is \u0 to \u10FFFF, unless restricted by the range of the CIM datatype char16.

Note: The escaping of double quotes is not necessary within a literal string, since 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 that string (e.g. as a default value for properties in the MOF), then that environment must define the escape rules for double quotes.

5.4.3. Identifiers

identifier-start = UNICODE-S1

identifier-subsequent = identifier-start | DECIMAL-DIGIT

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

5.4.4. Class Paths

class-name = identifier

The identifier MUST be in accordance with the definition of classname in the CIM Infrastructure Specification [1].

class-path = [*literal-string "."]* class-name

*If specified,literal-string MUST conform to the format of the **namespacePath** production defined in the WBEM URI Mapping Specification, DSP0207.*

5.4.5. Property Names

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, having the same name, that belongs to a subclass, then the "::" syntax is required to access the covered property when in the scope of the covering subclass. Details on how to determine which property to use are in Section 5.4.1.

5.4.6. Numeric Literals

The numeric literals are intended to agree with the numeric literals of MOF, as defined in the CIM Infrastructure Specification [1].

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

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

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

Since ABNF is case insensitive, this 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"
```

Since ABNF is case insensitive, this defines both upper and lower case.

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

Since ABNF is case insensitive, 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]
```

Since ABNF is case insensitive, this defines both upper and lower case.

5.4.7. Expressions

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

literal = literal-string

A literal-string has string type.

| decimal-value

A decimal-value has integer type

| binary-value

A binary-value has integer type

| hex-digit-value

A hex-digit-value has integer type

| real-value

A real-value has real type

| TRUE | FALSE

These literals have Boolean type. Since ABNF is case insensitive, this defines both upper and lower case.

arg-list = “*” | (*[DISTINCT]* **expr** *)*

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-comp, 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.
- Otherwise, 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 type object of *C*;
- Otherwise, 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 type object of *C*;
- Otherwise, 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 5.5.1 Property Identification below), 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 query feature "Embedded Properties" is supported then the ability to directly access properties of the embedded instance shall be supported.
- Otherwise, is the query is invalid.

If type is Array, then this form without a following “[“ is equivalent to “Identifier [*]”, and only “=” and “<>” comparisons are allowed.

| **property-scope identifier**

Property-scope declares that the identifier identifies a property exposed by the property-scope classname, (see 5.5.1 Property Identification below.) The type of the property is taken as the type of this production.

| *chain CLASSQUALIFIER identifier*

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

| **identifier "#" literal-string**

identifier MUST unambiguously identify a property, (see 5.5.1 Property Identification below.). 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 5.5.4 Symbolic Constants, below.

| identifier "(" arg-list ")"

identifier MUST be the name of a query language function. See 5.6 Query Language Functions for type rules of function calls. For Basic Query, only the numeric, string, instance, path, pathname, and datetime functions shall be supported. Note in particular that this syntax does NOT describe the invocation of a method defined on a CIM class.

| chain "." [property-scope] identifier

Chain MUST have type object of *C* for some class *C*.

Identifier MUST be the name of a property. For details on the selection of the identified see 5.5.1 Property Identification below below. The type of this production is the type of the property.

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

| *identifier*$_1$ *PROPERTYQUALIFIER identifier*$_2$

This production refers to a property qualifier. Identifier$_1$ MUST unambiguously identify a property, (see 5.5.1 Property Identification below), and the type of the expression is the type of that qualifier. If the property doesn't expose a qualifier with this name, the qualifier's default value applies.

| *chain "." [property-scope] identifier*$_1$ *PROPERTYQUALIFIER identifier*$_2$

chain, property-scope (if present), and identifier$_1$ together identify a property, as described in 5.5.1 Property Identification below. This production refers to the value of a property qualifier from that property, and the type of the expression is the type of that qualifier. If the property doesn't expose a qualifier with this name, the qualifier's default value applies.

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

| chain "." [property-scope] identifier "#" literal-string

chain, property-scope (if present), and identifier together identify a property, as described in 5.5.1 Property Identification below. This production forms a symbolic constant based on the VALUES and VALUEMAP qualifiers; see 5.5.4 Symbolic Constants, below. The type of this expression is the type of the identified property.

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

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

chain MUST have type array of *T*. If array-index-list comprises just a single expr, 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 MUST have string or char16 type, and the result has string type.

factor = concat

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

| ("+" | "-") concat

When this production is used, concat MUST have numeric type, which will be the type of the production

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

term = factor

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

| term "*" factor

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

If term has a numeric type, and factor has 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 datetime type and will produce an interval value. The rules for operations with datetime type operands are defined in Appendix E.1: Datetime Operations.

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 numeric type.

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

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

No other type combinations are allowed.

arith = term

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

| arith ("+" | "-") term

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

If arith and term have datetime types, then refer to Appendix E.1: Datetime Operations for a definition of the operation.

No other type combinations are allowed.

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

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

value-symbol = "#" literal-string

This is a degenerate syntax for symbolic constants, used only for direct comparison; type is determined by context. See productions for comp.

arith-or-value-symbol = arith | value-symbol

comp-op = "=" | "<>" | "<" | "<=" | ">" | ">="

comp = arith

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

| arith IS [NOT] NULL

This production has type Boolean.

| arith comp-op arith

This production has type Boolean for all cases in which it applies. See 5.5.6 Comparisons for more detailed description of comparisons.

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

| chain comp-op value-symbol

The left-hand-side MUST be a property reference, and that property is used as the context for the value-symbol, see 5.5.4 Symbolic Constants below.

This production has type Boolean for all cases in which it applies. See 5.5.6 Comparisons for 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-hand-side MUST be a property reference and that property is used as the context for the value-symbol, see 5.5.4 Symbolic Constants below.

This production has type Boolean for all cases in which it applies. See 5.5.6 Comparisons for more detailed description of comparisons.

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

| **arith ISA identifier**

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

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

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

The production has Boolean type.

| **arith LIKE literal-string**

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

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

The Basic Query feature only includes the Like features described in: Appendix D.1: Basic Like Regular Expressions.

| *arith LIKE arith*

Both sides of the LIKE comparison must have string or char16 type; the result has 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-hand-side. See Appendix D.2: Full Like Extended Regular Expressions

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

| *array-comp*

expr-factor = comp

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

| NOT comp

comp MUST have Boolean type; this production has Boolean type.

The following table defines the result of the 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 Boolean type; the production has Boolean type.

The following table defines the result of the 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 Boolean type; the production has Boolean type.

The following table defines the result of the 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 MUST have unsigned integer type.

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

| *expr ".." [expr]*

Both exprs MUST have unsigned integer type. The ".." notation is used to specify ranges of indices within an array.

| *".." expr*

expr MUST have unsigned integer type.

array-index-list = 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.

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

arith MUST 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 appears on the right-hand side.

| *(ANY | EVERY) identifier IN expr*
SATISFIES "(" comp ")"

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

5.4.8. Sort Specification

sort-spec = expr (ASC | DESC)

The specified expr MUST be defined in the SELECT clause. Note that 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 resbult set, instances with duplicate values in sorting properties will be displayed in an arbitrary order.

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

5.4.9. Select List

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 (i.e. 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 "." [property-scope] "*"

chain MUST have type object of C for some class C. If property scope is not present, 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. If property-scope is present and identifies some class S, it must 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.

selected-entry = expr *[AS identifier]*

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 expression is an object type, then the corresponding result column MUST have string type, and be populated with string representations of the values.

The set of column names in the query result MUST NOT contain duplicates. 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 MUST be a property reference, and the expr itself (minus any white space) is taken as the name of the corresponding result column. Note that this means that 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 MUST be a chain expression 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 sub-set of the columns returned.

If there is more than one entry in the FROM list, then each star-expr MUST be a chain expression 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 MUST be aggregating expressions.

5.4.10. From Criteria

subquery = select-statement

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

Each from-specifier using this production identifies a CIM class which 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.

| *"(" 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.

from-criteria = from-specifier *("," from-specifier)*

5.4.11. The Select Statement

search-condition = expr

expr MUST have Boolean type.

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

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 is 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.

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 only contain 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 define the number of instances to return. Note that the sort order for string or char16 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.

start = select-statement

5.5. 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 are not currently required. Today, these operations are supported by invoking individual operations defined in the CIM Operations over HTTP Specification [11].

CQL queries only operate against instances and their properties. They do not have the ability to query the supported schema, or invoke methods of instances. Query does support the ability to determine if an instance is a member of a CLASS via the ISA operator.

Several of the constructs in the BNF require usage information and/or additional explanation, as described below.

5.5.1. 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. This 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:

[chain "."]] [property-scope] identifier, where chain MUST 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*.
    - If the scoped identifier does NOT name a property exposed by *C*, then the query is invalid.
    - If chain is NOT present, *C* MUST 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* MUST be the same as, or 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.
- Otherwise, if chain "." is present, then chain MUST be of type object of *C*, and *C* is the class context.
- Finally if neither are present, then the identifier must be declared in at most one of the classes named in the FROM list.
- Otherwise the context cannot be determined, and the query is invalid.

5.5.2. Arrays

For properties of type Array, [*] is implicitly used if no specific array-index-list is given, so e.g. "OperationalStatus" has the same semantical meaning as "OperationalStatus[*]". For more details on Arrays, please refer to the CIM specification (DSP0004).

5.5.3. Embedded Objects

An embedded object is conveyed as a property of type string annotated only with the EMBEDDEDOBJECT qualifier. This qualifier indicates that the property's value is to be interpreted as an embedded object, but 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 which refer to string properties with the EMBEDDEDOBJECT qualifier are assigned type object. Reference 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 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 affects the resolution of properties of the embedded object. To remove ambiguity, queries that concern themselves with properties of embedded objects MUST use the scoping operator ("::") to scope those properties. A CQL implementation MUST reject any query which involves expressions whose type cannot be determined.

For example, the following would be permitted since 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
```

5.5.4. 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 MUST expose a VALUES qualifier, and the accompanying literal-string MUST match one of the strings in the VALUES qualifier's value.

If the property does not also expose a VALUEMAP qualifier, then the property MUST have 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 does also expose a VALUEMAP qualifier, then the value for this production will 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 type string, then the VALUEMAP entry itself is the value of the production; otherwise, (2) the property MUST have integer type, the VALUEMAP entry MUST NOT include the sequence "..", and the VALUEMAP entry is converted into an integer of the appropriate type. E.g., 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 equivalient 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’
```

5.5.5. 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 (i.e., all cases of division, as well as addition, subtraction, or multiplication involving at least one non-integer type), all arithmetic operations produce real64 type. If an overflow or underflow occurs, an error is returned.
Arithmetic and comparisons on datetime types are defined in Appendix E: Datetime Operations and BNF

5.5.6. 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 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 [21] and Unicode Technical Report #10 [20] MUST be applied. Unicode character based comparison is done as follows:

The "=" and '<>" operators MUST use the string identity matching rules defined in W3C "Character Model for the World Wide Web 1.0: Normalization" [7], 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 MUST behave as if the normalization defined in "Character Model for the World Wide Web 1.0: Normalization" [7], 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 codepoint 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 the UTF-8 or UTF-32 as the encoding, the binary order of the encoded characters matches the Unicode codepoint order. For UTF-16, the binary order of the supplementary characters does not match their Unicode codepoint order. For more information, refer to section 2.5 of "The Unicode Standard" [5].

2) For the Full Unicode feature, these operators MUST behave as if the normalization defined in [7], section 4 "String Identity Matching", was applied and then the default collation order defined in the Unicode Collation Algorithm [8] was used on the resulting strings. Note that this collation order accomodates most languages, without having to take any locales into account.

Comparison between datetime types is supported and is defined in Appendix E: Datetime Operations and BNF.

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

Reference comparison is performed via 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 the CIM Infrastructure Specification [1]. 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 done case-insensitively for key property names. Key property values are compared according to their type, as defined in section 5.4.6, Comparisons.
- For all components except the model path, the comparison is done case-insensitively.

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 done case-insensitively. The set of qualifiers defined on each class MUST 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 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, please refer to section 5.5.7 (Comparisons of Array and Scalar). Note that this type of comparison shall be supported if query feature "Array Range" is supported.

For comparison between arrays, comparison of complete arrays shall be supported in Basic Query. Comparison of parts of arrays shall be supported in query feature Array Range. The ArrayType governs how matches are made. There are three types: Bag, Ordered, and Indexed. If one of the arrays is a Bag, then comparison rules for Bags are used. As defined in DSP0004 [1], 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, an element-by-element comparison is performed. Arrays which have different numbers of elements do not compare equal.

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

5.5.7. Comparisons of Array and Scalar

This section only applies to comparison operations between array properties and non-array properties, as part of query features “Array Range” and “Satisfies Array”. A comparison between an array property and a non-array property is illegal if neither “EVERY” nor “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. Here are a few examples of the use of 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 4 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

5.6. Query Language Functions

This section describes the functions available for CIM Query Language.

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

5.6.1. Aggregation Functions

These functions are only valid within the select-list. If the select-list contains any aggregating expressions, then all items in the select-list MUST 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, where any properties are used only in the expression representing the argument of an aggregation function.

COUNT([DISTINCT] expr): 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 the argument assumes. The set of rows which COUNT considers is affected by 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 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 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 the arguments assumes. The result type for MEAN is real64. The result type for MEDIAN is the type of expr.

5.6.2. Numeric Functions

DATETIMETOMICROSECOND(expr): The argument MUST have 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 (i.e., 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 MUST have string or char16 type and must be a binary-value, hex-digit-value, decimal-value, or real-value in the range of 0 to 2^{64}-1. The result has type uint64. The fractional portion of any real-value is discarded.

STRINGTOSINT(expr): The argument MUST have string or char16 type and must be a binary-value, hex-digit-value, decimal-value, or real-value in the range of -2^{63} to 2^{63}-1. The result has type sint64. The fractional portion of any real-value is discarded.

STRINGTOREAL(expr): The argument MUST have string type and must be a binary-value, hex-digit-value, decimal-value, or real-value. The result has type real64.

5.6.3. String Functions

UPPERCASE(expr): The argument MUST have string or char16 type, and the result has 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 [5], on all characters.

NUMERICTOSTRING(expr): The argument MUST have numeric type, and the result has 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 and 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 MUST have reference type, and the result has 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.

5.6.4. Instance Functions

These functions operate on objects, references or strings whose contents is a WBEM-URI, as defined in the WBEM URI Mapping Specification, DSP0207 [2].

INSTANCEOF([expr]): The argument MUST 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 MUST represent an instance and MUST 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

5.6.5. Path Functions

These functions operate on objects, references or strings whose contents is a WBEM-URI, as defined in the WBEM URI Mapping Specification, DSP0207 [2].

CLASSPATH([expr]): The argument MUST be an object, a reference, or a string containing a WBEM-URI. The result of this function is of type reference. 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 the creation class of that instance. If the argument is of type object, it MUST 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 or not 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 MUST be an object, a reference, or a string containing a WBEM-URI. The result of this function is of type reference. 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 MUST 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 or not 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.

5.6.6. Datetime Functions

CURRENTDATETIME(): Returns the "current" datetime as determined by the implementation.

DATETIME(expr): The argument MUST be of type string, and at runtime MUST take on a 25-character value conformant with a datetime specification (either timestamp or interval). The result has datetime type.

MICROSECONDTOTIMESTAMP(expr): The argument MUST be of an integer type, and the result has datetime type. The argument will be interpreted as a number of microseconds since 00:00:00.000000UTC on 1/1/0000, and the result will be a timestamp.

MICROSECONDTOINTERVAL(expr): The argument MUST be of an integer type, and the result has interval (datetime) type. The argument will be interpreted as a number of microseconds, and the result will be an interval.

5.7. 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.
    1. Each table MAY be considered as a class. However, it is NOT required to conform to the definition a CIM class.
    2. Each row MAY be considered as an instance. However, it is NOT required to conform to the definition a CIM instance.
    3. 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 query 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.

5.8. Query Errors

When processing a query (either by a CIM Server or a provider), it is legitimate to reject the query. The following errors are defined in the CIM Operations Specification for Exec Query:

- CIM_ERR_ACCESS_DENIED
- CIM_ERR_NOT_SUPPORTED
- CIM_ERR_INVALID_NAMESPACE
- CIM_ERR_INVALID_PARAMETER (including missing, duplicate, unrecognized or otherwise incorrect parameters)
- CIM_ERR_QUERY_LANGUAGE_NOT_SUPPORTED (the requested query language is not recognized)
- CIM_ERR_INVALID_QUERY (the query is not a valid query in the specified query language; i.e., a syntax or semantic error occurred. For CQL, this error is also returned if the language is correct, but the query features used by the query are not supported)
- CIM_ERR_FAILED (some other unspecified error occurred)

If a Query is implemented as part of a Class, then the Provider of the class is responsible for error handling and for appropriately passing errors back to the client of the class or its instances. For instance, if the class supports a string property named Query and the string is a constant, then the implementation must assure that the string is correct. Note that in this case, the implementation may be completely hard-coded. If the property is set by a CIM Client, then the implementation is responsible for checking the validity of the query when the property is set. If invalid, the CIM operation used to set the property MUST return CIM_ERR_INVALID_QUERY if an ExecuteQuery or CIM_ERR_FAILED for all others.

In the future, the CIM_Error class will be used to expand on the errors defined above.

5.9. Query Functional Description

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

The following table defines the "Basic Features" required for CQL support and a set of optional CQL processing features that MAY be provided by a component. Discovery of these features is enabled via the CQLFeatures enumeration property of the QueryCapabilities class. Each optional feature MUST be fully supported before it is advertised as being supported.

The table also tracks the status of each feature. A status of "Final" means that the feature has at least two independent implementations and that all issues have been resolved in a manner consistent with DMTF policy. Otherwise, the status will be marked as "Experimental".

Query Feature	**Description**	**Prerequisite Feature(s)**	**Feature Status**
Basic Query =2	The query MUST support the syntax and processing rules designated as Basic Query in Section 5.4, “Query Language BNF”.	None	Experimental
Simple Join =3	The FROM clause has the following constraints: • MUST support at least two from-specifiers. • Support for more than two from-specifiers IS NOT part of Simple Join.	Basic Query	Experimental

Query Feature	Description	Prerequisite Feature(s)	Feature Status
Complex Join =4	The FROM clause MUST support more than two from-specifiers	Simple Join	Experimental
Subquery =5	The FROM clause MUST support subqueries	Basic Query	Experimental
Result Set Operations =6	The query MUST support the DISTINCT and FIRST operators The ORDER BY clause MUST be supported	Basic Query	Experimental
Extended Select List =7	The select-list • MUST support functions • MUST support CLASSQUALIFIER and PROPERTYQUALIFIER • MUST support the AS construct for property aliasing	Basic Query	Experimental
Embedded Properties =8	The query MUST support the ability to reference the properties of the embedded instance.	Basic Query	Experimental
Aggregations =9	The query MUST support aggregation functions.	Extended Select List	Experimental
Regular Expression Like =10	The WHERE clause MUST support for the like-predicate with the capabilities defined in Appendix D.2: Full Like Extended Regular Expressions	Basic Query	Experimental
Array Range =11	The query MUST support the full range of array-index-list productions in order to compare Array properties with Non-Array properties as described in section 5.6 or in order to compare parts of arrays. The WHERE clause MUST support the array-comp production	Basic Query	Experimental
Satisfies Array =12	• The WHERE clause MUST support the satisfies clause	Array Range	Experimental
Foreign Namespace Support =13	The query MUST support references to namespaces other than the one in which the query is executed.	Basic Query	Experimental
Arithmetic Expression =14	The query must support arithmetic expressions using +, -, *, and /.	Basic Query	Experimental

Query Feature	Description	Prerequisite Feature(s)	Feature Status
Full Unicode =15	The query must support the Unicode string processing algorithms described in this specification.	Basic Query	Experimental

Table 1: Query Features

If a query includes clauses or constructs not supported by the infrastructure, the error CIM_ERR_INVALID_QUERY MUST be returned on a request made via ExecuteQuery or CIM_ERR_FAILED for all other CIM operations.

6 CIM Query Template Language

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

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

- The pre-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, since only single quotes can be used to delimit string literals. If the entire pre-processor string is put into an environment that uses double quotes to delimit that string (e.g. 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 pre-processor 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 pre-processor token.
- For each pre-processor token encountered, the pre-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 pre-processor variable as defined to the pre-processing facility. The value of the pre-processor 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 pre-processor variable to a value is not specified here and must be specified where this facility is used.

Pre-processor 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 detailed in section 5.2 on identifying a query language, the string “DMTF:CQLT” will identify the CIM Query Template language to represent the use of this pre-processing capability for CQL.

6.1. Pre-processor Examples

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

Assuming the value of *targetClassName* is "CIM_StorageExtent", the CQL pre-processor would translate the string

```
SELECT *
FROM $targetClassName$
```

into

```
SELECT *
FROM CIM_StorageExtent
```

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

Assuming the value of *UserID* is "guest", the CQL pre-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 to be restricted based on the value of the variable *whereClause*.

Assuming the value of *whereClause* is "WHERE UPSttyPath = '/dev/ttyOp1' AND MonitorEventID = 20", the CQL pre-processor would translate the string

```
SELECT DetectionTime,
    SystemIPAddress,
```

```
    PerceivedSeverity,
    MonitorEventID,
    UPSttyPath
FROM Acme_UPSAlertIndication
$whereClause$
```

into

```
SELECT DetectionTime,
    SystemIPAddress,
    PerceivedSeverity,
    MonitorEventID,
    UPSttyPath
FROM Acme_UPSAlertIndication
WHERE UPSttyPath = '/dev/ttyOp1' AND MonitorEventID = 20
```

7 Examples

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

7.1. Discovery examples

1. Get the object path, ElementName and Caption for all StorageExtents

   Required Features: Basic Query, Extended Select List

```
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 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

```
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 StorageExtent and 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 ComputerSystems and the object paths of any instances dependent on the system as described by the Dependency association.

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

```
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 Dependency association's instances. The instances that are created contain all the properties of CIM_ComputerSystem and a string representing the related/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, and there are many BaseMetricValue objects associated with it (>10000), and the selection criteria is such that only a handful of them matches.

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

```
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 #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 ComputerSystem's object path.

The query in this example is very selective: Only 6 instances are returned, where 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, since 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

```
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 (which meet the association criteria) with properties as specified by CIM_SettingData.

7. Get a storage array's LUN masking and mapping for a failed FCPort. This query uses aliasing in the FROM clause and a series of sub-queries. 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 only returns the final results.

Required Features: Basic Query, Extended Select List, Complex Join,
Subquery, Array Element

```
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:

```
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
```

7.2. Event detection examples

1. As regards query in Indication processing, the following examples are taken from storage management requirements.

Required Features: Basic Query

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

Using the lifecycle indication classes, this query would be stored in the Query string property of an instance of IndicationFilter and its delivery defined by an IndicationSubscription association to a ListenerDestination (please see the CIM Event Model [14]). An InstCreation notification would be delivered any time that an FCPort was created. The notification would consist of a single instance with a select-list as defined by the CIM_InstCreation class.

2. As above, this query would be stored in the Query string property of an instance of IndicationFilter, and its delivery defined by an IndicationSubscription association. An InstModification notification would be delivered any time that an 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

```
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 DetectionTime, SystemIPAddress, PerceivedSeverity, MonitorEventID and UPSttyPath properties, whenever MonitorEventID = 20 occurs on device /dev/ttyOp1.

Required Features: Basic Query

```
SELECT DetectionTime,
    SystemIPAddress,
    PerceivedSeverity,
    MonitorEventID,
    UPSttyPath
FROM Acme_UPSAlertIndication
WHERE UPSttyPath = '/dev/tty0p1'
   AND MonitorEventID = 20
```

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

Required Features: Basic Query, Extended Select List

```
SELECT DetectionTime,
     SystemIPAddress,
     PerceivedSeverity,
     MonitorEventID,
     UPSttyPath,
     'HP12345' AS FilterID
FROM Acme_UPSAlertIndication
WHERE UPSttyPath = '/dev/tty0p1'
     AND MonitorEventID = 20
```

5. Continuing the example above, 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 DetectionTime,
     SystemIPAddress,
     PerceivedSeverity,
    'Critical' AS CustomSeverity,
     MonitorEventID,
     UPSttyPath,
     'HP12345' AS FilterID
FROM Acme_UPSAlertIndication
WHERE UPSttyPath = '/dev/tty0p1'
   AND MonitorEventID = 20
```

6. Locate sick System/LogicalDevice combinations

Required Features: Basic Query, Satisfies Array, Complex Join

```
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 FileShare

Required Features: Basic Query

```
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
```

7.3. 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 only used if the free pool can not satisfy the need.

1. This first query is used in a QueryCondition with QueryResultName set to "PR_Needy". The query selects a StoragePool that is low on space. Evaluation results in zero or more PR_Needy instances that are used by a related MethodAction.

Required Features: Basic Query, Extended Select List, Complex Join, Embedded Properties

```
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. This next query is used in MethodAction to invoke a CreateOrModifyStoragePool method. It uses PR_Needy instances produced by the previous QueryCondition. The InstMethodCall results of the call are named by the property InstMethodCallName set to "PR_ModifySP".

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

```
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 MethodActionResults as input to a second MethodAction to take action on an error. It also calls CreateOrModifyStoragePool.

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

```
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
```

Appendix A: Change History

Version 0.1 – October 2002, 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.

Version 0.2 – November 2002, Corrected one example in Section 5 and acknowledged that more examples/use cases need to be provided

Version 0.3 – January 2003, 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

Version 0.4 – January 2003, Clarified requirement for ISA function as mechanism to query class inheritance/hierarchy, and added check for a class' Version qualifier

Version 0.5 – September 2003, 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.

Version 0.6 – September 2003, 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.

Version 0.7 – October 2003, 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

Version 0.8 and 0.9 – January 2004, 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.

Version .10 – February 2004, Many updates to deal with member comments.
_KEY renamed to _OBJECTPATH.
_CLASS renamed to _CLASSPATH.
_VERSION eliminated.
Extended BNF to added support for Character and Arithmetic operations.
Added Symbolic constants.

Version .11 – March 2004, 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,` http://www.faqs.org/rfcs/rfc2234.html.
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 =, <>.
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

Version .12 – April 2004, 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.

Version .13 – May 2004, Updates to cover review comments
Simplify Basic Like

Clarify conversion table Many corrections
Version .14 Review resolutions
Version .15 More review resolutions. Accepted by Interop pending resolution of set of issues
Version .16 Resolution resulted in conversion to compilable BNF. This is a significant revision.
Version .17 Resolution of issues after conversion.
Version .18 (Company Review Version, Version 1.0.0 Prelim) Clarify that Timestamp 0 is 1 BCE Remove notes from text.
Draft 1.0.0f – December 15, 2005 Applied CRs WIPCR00251.001, WIPCR00231.009
Draft 1.0.0f (Prelim 2) – January 13, 2006 Applied CRs WIPCR00255.002, WIPCR00242.007, WIPCR00240.002
Draft 1.0.0f (Prelim 2) – February 2, 2006 Applied CRs WIPCR00270.000.htm
Draft 1.0.0f (Prelim 2) – February 8, 2006 Applied CRs WIPCR00272.002.htm, WIPCR00268.001.htm
Draft 1.0.0g (Prelim 2) – February 10, 2006 Applied CRs WIPCR00261.002.htm, WIPCR00247.006.htm
Draft 1.0.0g (Prelim 2) – February 15, 2006 Fixed typo wrt closing paranthesis after char-escape
Draft 1.0.0g (Prelim 2) – February 16, 2006 Applied CRs WIPCR00245.008.htm, WIPCR00269.001.htm, WIPCR00271.002.htm
Draft 1.0.0g (Prelim 2) – February 27, 2006 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
Draft 1.0.0g (Prelim 2) – March 16, 2006 Applied CRs WIPCR00280.000.htm, WIPCR00282.000.htm Updated reference numbers
Draft 1.0.0h (Prelim 2) – March 22, 2006 Ballot version of the spec

Appendix B: Dependencies and References

Appendix B.1: *Dependencies*

[1] DMTF [2004] Distributed Management Task Force: CIM Infrastructure Specification, DSP0004.pdf, version 2.3, http://www.dmtf.org/standards/published_documents.

[2] DMTF [2004] Distributed Management Task Force: WBEM URI Specification, DSP0207.pdf, version 1.0, http://www.dmtf.org/standards/published_documents.

[3] Augmented BNF for Syntax Specifications: ABNF, RFC 2234, Nov 1997, http://www.faqs.org/rfcs/rfc2234.html.

[4] In this document, the term Unicode refers to the Universal Character Set (UCS), defined jointly by the Unicode Standard [5] and ISO/IEC 10646 [6].

[5] 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.

[6] 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.

[7] W3C Working Draft "Character Model for the World Wide Web 1.0: Normalization", February 24, 2004, http://www.w3.org/TR/charmod-norm/

[8] 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 http://www.unicode.org/reports/tr10 for the latest version.

[9] The Unicode Consortium, "Unicode Regular Expressons (Unicode Technical Standard #18)". - as, from time to time, amended, replaced by a new edition or expanded by the addition of new parts. See http://www.unicode.org/reports/tr18 for the latest version.

[10] See "XQuery 1.0 and XPath 2.0 Functions and Operators", section 7.6.1 Regular Expression Syntax. The latest version is at http://www.w3.org/TR/xpath-functions.

Appendix B.2: *References*

[11] DMTF [2003] Distributed Management Task Force: CIM Operations over HTTP Specification, DSP0200, version 1.2, http://www.dmtf.org/standards/documents/WBEM/DSP200.html.

[12] ISO/IEC [1992] ISO/IEC 9075:1992, Database Language SQL- July 30, 1992. See http://www.iso.org for the latest version.
[13] W3C [2001] World-Wide Web Consortium: XML-Query, http://www.w3.org/XML/Query.
[14] DMTF [2002] Distributed Management Task Force: CIM Event Model V2.9 (Final), http://www.dmtf.org/standards/cim/cim_schema_v29.
[15] DMTF [2002] Distributed Management Task Force: CIM Policy Model V2.9 (Final), http://www.dmtf.org/standards/cim/cim_schema_v29.
[16] UTF-8, a transformation format of ISO 10646, http://www.ietf.org/rfc/rfc3629.txt?number=3629.
[17] RFC 1034: DOMAIN NAMES - CONCEPTS AND FACILITIES, http://www.ietf.org/rfc/rfc1034.txt?number=1034.
[18] RFC 1123: Requirements for Internet Hosts -- Application and Support, http://www.ietf.org/rfc/rfc1123.txt?number=1123.
[19] DMTF [2002] Distributed Management Task Force: Specification for the Representation of CIM in XML, DSP0201, version 2.1 http://www.dmtf.org/standards/documents/WBEM/DSP201.html.
[20] UNICODE [2005] Unicode, Inc.: Unicode Technical Standard #10: Unicode Collation Algorithm, http://www.unicode.org/unicode/reports/tr10/
[21] ISO/IEC 14651[2000], Information technology – International string ordering and comparison – Method for comparing character strings and description of the common template tailorable ordering

Appendix C: Acknowledgements

The primary authors of this specification are George Ericson of EMC Corporation, Jeff Piazza of AppIQ, Inc. and Andrea Westerinen of Cisco Systems, Inc. The document is based on an original WBEM Query Language Specification submitted by Patrick Thompson of Microsoft.

Significant editing contributions were made by Andreas Maier, Oliver Benke and others of IBM.

Appendix D: Regular Expression BNF

The Regular Expression grammar below uses Augmented BNF (ABNF) [3] with the following exceptions.

1. Rules separated by a bar (|) represent choices. (Instead of using a slash (/) as defined in ABNF).
2. Ranges of alphabetic characters or numeric values are specified using two periods (..) placed between the beginning and ending values of the range. (Instead of using the minus sign (-) as defined in ABNF).
3. The rules defined in this syntax are meant to be assembled into a complete query by assuming whitespace characters between them, except where noted otherwise. (ABNF requires explicit specification of whitespace.)
4. The comma (,) is used to explicitly designate concatenation of rules. (Instead of implicit concatenation of rules as specified by ABNF.)

Note:

1. ABNF is NOT case-sensitive.
2. The rules above apply to the ABNF used here and NOT to the resultant Regular Expression used in Full or Basic Like. In particular, except where noted, white space is significant within the resultant Regular Expression.

The grammar is defined in two sections. The first is used to construct Regular Expressions used by the Basic Like feature. The second, Extended Regular Expressions is used to create Regular Expressions used by the Regular Expression Like feature. Both are defined as follows:

Appendix D.1: *Basic Like Regular Expressions*

Basic Like Regular Expressions is a subset of the XQuery Regular Expression syntax as defined in Regular Expressions [10].

Note: Basic Like Regular Expressions complies with levels RL1.1 and RL 1.7 of Unicode Regular Expressions Level 1 [9], which is a subset of the XQuery Regular Expression [10] compliance to Unicode Regular Expressions Level 1 [9].

blre-ordinary-char= UNICODE-CHAR

A character, other than a metacharacter excluded from the Char production of XQuery Regular Expressions [10].

blre-escaped-char = char-escape | SingleCharEsc

An escaped character. The char-escape is defined in the String Literals section. The SingleCharEsc is defined in XQuery Regular Expressions [10]. The "/u" and "/U" syntax of char-escape replaces the character reference syntax defined in XQuery Regular Expressions [10].

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 occurences of a single character

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

Basic Like regular expression

Appendix D.2: *Full Like Extended Regular Expressions*

Full Like Regular Expressions is conformant with the XQuery Regular Expression syntax as defined in Regular Expressions [10], 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 the String Literals section may be used in addition to the escape sequences in SingleCharEsc in XQuery Regular Expressions [10]. The "/u" and "/U" syntax of char-escape replaces the character reference syntax defined in XQuery Regular Expressions [10]. 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 [10] are supported, and the expression matching behaves as if all the flags have the default values.

Appendix E: Datetime Operations and BNF

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

Appendix E.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 results in an interval

   Other arithmetic operations are NOT defined.

2. Comparison operations:
    - § Testing for equality or unequality of two timestamps or two intervals results in a boolean
    - § Testing for the ordering relation (<, <=, >, >=) of two timestamps or two intervals results in a boolean

   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 languages) SHOULD define how undefined operations are handled.

Any operations on datetime types in an expression MUST 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 (i.e. 0 microseconds corresponds to 00:00.000000 in the timezone with datetime offset +720, on January 1st 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.

Note that mathematical interval arithmetic is commutative and associative for addition and multiplication, like ordinary arithmetic.

Note that 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 timezone of the result produces an arithmetic underflow condition
        - A timestamp newer than the newest valid value in the timezone 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 timezone MUST be used for any timestamp results.

    Note that for most fields, asterisks can be used only with the granularity of the entire field.

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)
```

Appendix E.2: *Datetime BNF*

The URI grammar below uses Augmented BNF (ABNF) [3] with the following exceptions.

1. Rules separated by a bar (|) represent choices. (Instead of using a slash (/) as defined in ABNF).
2. Ranges of alphabetic characters or numeric values are specified using two periods (..) placed between the beginning and ending values of the range. (Instead of using the minus sign (-) as defined in ABNF).
3. The rules defined in this syntax are meant to be assembled into a complete query by assuming whitespace characters between them, except where noted otherwise. (ABNF requires explicit specification of whitespace.)
4. The comma (,) is used to explicitly designate concatenation of rules. (Instead of implicit concatenation of rules as specified by ABNF.)

Note: ABNF is NOT case-sensitive.

The grammar is defined as follows:

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

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

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

```
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), ("*****")
                                  |  ("******")
```

See the CIM Infrastructure Specification [1] for a detailed description of the use of interval-specification.

```
dt-timestamp-specification = dt-single-quote,
            (
             | ((4*4(dt-decimal-digits) | "****"),  "**********",
                      ".", ("******"), ("+"|"-"), 3*3(dt-decimal-digit))
```

years: 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.

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

months

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

days

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

minutes

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

seconds

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

microseconds

```
            ), dt-single-quote
```

See the CIM Infrastructure Specification [11] for a detailed description of the use of interval-specification.

```
dt-interval-specification = dt-single-quote,
        ( (14*14("*"), ".", ("******"), (":"), 3*3(dt-decimal-digit))
```
nothing

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

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

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

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

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

```
        ), dt-single-quote
```
See the CIM Infrastructure Specification [11] for a detailed description of the use of interval-specification.