



CIM Query Language Specification

3 **DSP0202**
4 ***Pending***

Status: Second Preliminary -

5 Copyright © 2000-2006 Distributed Management Task Force, Inc. (DMTF). All rights
6 reserved.

7 DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability.
8 Members and non-members may reproduce DMTF specifications and documents for uses consistent with this purpose, provided that correct
9 attribution is given. As DMTF specifications may be revised from time to time, the particular version and release date should always be
10 noted.

11 Implementation of certain elements of this standard or proposed standard may be subject to third party patent rights, including provisional
12 patent rights (herein "patent rights"). DMTF makes no representations to users of the standard as to the existence of such rights, and is not
13 responsible to recognize, disclose, or identify any or all such third party patent right, owners or claimants, nor for any incomplete or
14 inaccurate identification or disclosure of such rights, owners or claimants. DMTF shall have no liability to any party, in any manner or
15 circumstance, under any legal theory whatsoever, for failure to recognize, disclose, or identify any such third party patent rights, or for such
16 party's reliance on the standard or incorporation thereof in its product, protocols or testing procedures. DMTF shall have no liability to any
17 party implementing such standard, whether such implementation is foreseeable or not, nor to any patent owner or claimant, and shall have
18 no liability or responsibility for costs or losses incurred if a standard is withdrawn or modified after publication, and shall be indemnified
19 and held harmless by any party implementing the standard from any and all claims of infringement by a patent owner for such
20 implementations.

21
22 For information about patents held by third-parties which have notified the DMTF that, in their opinion, such patent may relate to or impact
23 implementations of DMTF standards, visit <http://www.dmtf.org/about/policies/disclosures.php>.

24

CIM Query Language Specification

25
26
27 **Version 1.0.0h Second Preliminary - Pending**
28 **March 22, 2006**

29

Abstract

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

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

37

38 Table of Contents

39	Table of Contents.....	i
40	1 Introduction and Overview	1
41	2 Background Materials	2
42	3 Terminology	3
43	4 Requirements and Concepts	4
44	5 CIM Query Language (CQL)	6
45	5.1. CQL Introduction.....	6
46	5.2. Identifying the CIM Query Language	7
47	5.3. The Query Language Type Lattice	8
48	5.4. Query Language BNF.....	10
49	5.4.1. Reserved Words.....	11
50	5.4.2. String Literals.....	12
51	5.4.3. Identifiers	12
52	5.4.4. Class Paths	13
53	5.4.5. Property Names.....	13
54	5.4.6. Numeric Literals	14
55	5.4.7. Expressions	15
56	5.4.8. Sort Specification.....	23
57	5.4.9. Select List.....	24
58	5.4.10. From Criteria.....	25
59	5.4.11. The Select Statement	26
60	5.5. Considerations of the Constructs in the BNF.....	27
61	5.5.1. Property Identification.....	27
62	5.5.2. Arrays	28
63	5.5.3. Embedded Objects	28
64	5.5.4. Symbolic Constants	29
65	5.5.5. Computation and Types.....	29
66	5.5.6. Comparisons.....	30
67	5.5.7. Comparisons of Array and Scalar.....	32
68	5.6. Query Language Functions.....	33
69	5.6.1. Aggregation Functions	33
70	5.6.2. Numeric Functions	33
71	5.6.3. String Functions.....	34
72	5.6.4. Instance Functions	34
73	5.6.5. Path Functions.....	35
74	5.6.6. Datetime Functions.....	35
75	5.7. Query Considerations	37
76	5.8. Query Errors	38
77	5.9. Query Functional Description	39
78	6 CIM Query Template Language.....	42
79	6.1. Pre-processor Examples.....	43
80	7 Examples.....	45
81	7.1. Discovery examples	45

CIM Query Language Specification

82	7.2. Event detection examples	51
83	7.3. Policy examples	54
84	Appendix A: Change History	56
85	Appendix B: Dependencies and References	59
86	Appendix B.1: Dependencies	59
87	Appendix B.2: References	59
88	Appendix C: Acknowledgements	60
89	Appendix D: Regular Expression BNF	61
90	Appendix D.1: Basic Like Regular Expressions	61
91	Appendix D.2: Full Like Extended Regular Expressions	62
92	Appendix E: Datetime Operations and BNF	63
93	Appendix E.1: Datetime Operations	63
94	Appendix E.2: Datetime BNF	66
95		
96		
97		

98 **1 Introduction and Overview**

99 CIM and WBEM support a query mechanism that is used to select sets of properties from
100 CIM object instances. Query support is available in some operations defined by the CIM
101 Operations Specification over HTTP [11] and some CIM classes within the Event [14] and
102 Policy [15] Models. Query definitions allow a WBEM client to specify the nature and the
103 number of instances that are selected and what information is returned from those instances.
104 This enables a WBEM managed environment to place less burden on the network
105 infrastructure. The precise mechanics for delivering query requests and receiving query
106 results are specified as a part of the CIM Operations over HTTP Specification [11].

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

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

122 **2 Background Materials**

123 CIM's query design is based on concepts from both ISO/IEC's Structured Query Language
124 [12] (SQL-92) and W3C's XML-Query [13]. Basic understanding of the use of relational
125 databases is required. However specific knowledge of these other works is not required in
126 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.

128 4 Requirements and Concepts

129 The CIM Query Language has been widely anticipated and exploited in the CIM Operations
130 over HTTP Specification, by the CIM Events Model [14], and by the CIM Policy Model
131 [15]. The language defines the desired instance-level data ranging over a certain set of
132 objects to be returned as the result of an ExecuteQuery CIM operation. Also, it defines the
133 conditions and data for Indications returned as a result of one of the following:

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

137 Query semantics **MUST** include instance property projection (e.g., a SQL SELECT clause)
138 and a range (e.g., a SQL FROM clause) and **MAY** include predicate logic (e.g., a SQL
139 Where clause). This support (defined specifically using the keywords Select-From-Where)
140 was included in a preliminary version of the CIM Query specification, called the WBEM
141 Query Language (WQL), and implemented in various code bases, although the preliminary
142 specification was never released. It is important to maintain these keywords and concepts
143 (unless a critical performance or operational error is found), in order to prevent unnecessary
144 code churn.

145 As noted above, instance property projection **MUST** be supported. This is a mechanism to
146 select particular properties from a class to be included in a query response or Indication
147 object. The projection may include "static" entries that can be used for tagging the response
148 and/or Indication object. (These requirements are provided by the specific or array class-
149 property-identifier and select-string-literal constructs, respectively.) In addition, the CIM
150 Query Language **MUST**:

- 151 • Support the ability to project meta-data such as instance name and instance class into
152 a response (see the OBJECTPATH() and CLASSPATH() methods, respectively).
- 153 • Support query of class versioning information (see the query of CLASSQUALIFIER
154 data).
- 155 • Define and support a mechanism for querying class inheritance/hierarchy in a query
156 predicate (provided using the ISA operator).
- 157 • Support the ability to query all data types as well as the entries of an array, since CIM
158 defines arrays of simple data types as valid class properties.

CIM Query Language Specification

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

- 163 • The ability to set a returned property value (such as an Indication Priority which
164 could be overridden by a customer)
- 165 • The ability to specify a constant value set of properties to be returned
- 166 • Support accessing property values of an EMBEDDEDOBJECT

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

171
172 CQL MUST support polymorphism. This means, if a query is issued against a base class, all
173 derived class instances will be considered as well. For instance, consider:

```
174  
175     SELECT *  
176     FROM CIM_Indication
```

177
178 This would match all instances of derived classes of CIM_Indication.

179 5 CIM Query Language (CQL)

180 5.1. CQL Introduction

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

```
183
184     SELECT <select-list>
185     FROM <class list>
186     WHERE <selection expression>
```

187

188 Where:

- 189 • A <select-list> is a comma-separated list of:
 - 190 ○ CIM property names (optionally qualified by their class name) related to the
 - 191 individual classes specified in the FROM clause. The asterisk (*) can be used
 - 192 to specify ALL the properties of a class. The resultant column is named by
 - 193 the property name, this may be modified using the keyword AS followed by a
 - 194 new name.
 - 195 ○ Literals, named via the keyword AS followed by a name.
 - 196 ○ Function results, named via the keyword AS followed by a name.
- 197 • The <class-list> is a comma-separated list of class names.
- 198 • A <selection expression> specifies the criteria by which results are selected. It is
- 199 limited to relatively simple property comparisons.

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

- 203 • the ability to process arrays via indices,
- 204 • the ability to query the properties of EMBEDDEDOBJECTS, and
- 205 the ability to traverse associations (based on the values of their REF
- 206 properties).

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

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

214 5.2. Identifying the CIM Query Language

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

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

220 The <language id> MUST include a unique, (in the context of the identified organization),
221 name for the query language.

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

223 5.3. The Query Language Type Lattice

224 The CQL type system incorporates the type system of the CIM Infrastructure Specification
225 [1][11], but also extends that type system, as follows:

226 For every class *C*, there is an "object of *C*" type, whose values may be either

- 227 • instances of *C* (including instances of any subclasses of *C*), or
- 228 • the class *C* itself, or one of *C*'s subclasses.

229 Note that classes arise as CQL values only when they appear as embedded objects, and that
230 support for embedded objects is an optional feature of CQL. CQL implementations that do
231 not support embedded objects may consider the values for "object of *C*" to be limited to
232 instances of *C* (including instances of any subclasses of *C*).

233 The "object of *C*" types recapitulate the CIM class hierarchy, in that, if *C1* is a superclass of
234 *C2*, then "object of *C1*" is a supertype of "object of *C2*".

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

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

237 There is an "unsigned integer" type that is a supertype of uint8, uint16, uint32, and uint64.

238 There is a "signed integer" type that is a supertype of sint8, sint16, sint32, and sint64.

239 There is an "integer" type that is a supertype of unsigned integer and signed integer.

240 There is a "real" type that is a supertype of real32 and real64.

241 There is a "numeric" type that is a supertype of integer and real.

242 CIM defines a "datetime" type, which contains either timestamp or interval values. Note that
243 timestamp and interval are not defined as explicit types within CIM, but are defined by
244 Appendix E: Datetime Operations and BNF. A timestamp with the year field set to 0000 is
245 interpreted as the year 1 BCE. A year field set to 0001 is interpreted as the year 1 CE.

246 There is a "string" type that is the CIM datatype string. It contains a sequence of Unicode [4]
247 characters. The range of allowed code points is the same as the CIM datatype string. The
248 encoding form is defined by the specification that is using CQL.

249 There is a "char16" type that is the CIM datatype char16. It contains one Unicode [4]
250 character. The range of allowed code points is the same as the CIM datatype char16. The
251 encoding form is defined by the specification that is using CQL.

252 The CIM Infrastructure Specification [1] also defines a system of array types, which is
253 similarly extended. That is, every non-array type, *T*, in the CQL type lattice has a
254 corresponding array type, array of *T*. The structure of the array type lattice exactly matches

CIM Query Language Specification

255 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
256 supertype of array of T_2 if and only if T_1 is a supertype of T_2 .

257 CQL expressions are assigned types according to the rules that accompany the grammar,
258 below. Any CQL construct which has been assigned a particular type is said also to "have"
259 all the supertypes of that type. E.g., an expression which has been assigned type "object of
260 CIM_ManagedElement" also "has" type "object".

261 5.4. Query Language BNF

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

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

CIM Query Language Specification

272 Notes:

- 273 1. ABNF is NOT case-sensitive.
- 274 2. UNICODE-CHAR is a Unicode [4] character. The range of allowed codepoints is the
275 same as the range for the char16 datatype in the “CIM Query Type Lattice” section.
276 UNICODE-S1 is a subset of UNICODE-CHAR where the characters from the US-
277 ASCII range {U+0000...U+007F} are limited to the set S1, where S1 = {U+005F,
278 U+0041...U+005A, U+0061...U+007A} [This is alphabetic, plus underscore]. The
279 encoding form of UNICODE-CHAR is defined by the specification that is using
280 CQL.
- 281 3. The CQL string (i.e. the entire string, beyond just string literals) uses Unicode [4]
282 characters. The encoding of the CQL string is the same as the encoding of
283 UNICODE-CHAR.

284
285 In the following BNF, **bold** text marks a Basic Query component and *italicized* text marks
286 components not in the Basic Query feature.

287
288 The grammar for all features is defined as follows. As much as possible, this grammar is
289 constructed to be LALR(1)-parsable.

290 5.4.1. Reserved Words

291	AND = “AND”
292	ANY = "ANY"
293	AS = "AS"
294	ASC = "ASC"
295	BY = "BY"
296	CLASSQUALIFIER = "CLASSQUALIFIER"
297	DESC = "DESC"
298	DISTINCT = "DISTINCT"
299	EVERY = "EVERY"
300	FALSE = "FALSE"
301	FIRST = "FIRST"
302	FROM = "FROM"
303	IN = "IN"
304	IS = "IS"
305	ISA = "ISA"
306	LIKE = "LIKE"
307	NOT = "NOT"
308	NULL = "NULL"
309	OR = “OR”
310	ORDER = "ORDER"
311	PROPERTYQUALIFIER = "PROPERTYQUALIFIER"
312	SATISFIES = "SATISFIES"
313	SELECT = "SELECT"
314	TRUE = "TRUE"

315 **WHERE = "WHERE"**

316 **5.4.2. String Literals**

317 **single-quote = '''**

318

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

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

322 **char-escape = "\", ("\" | single-quote | "b" | "t" | "n" | "f" | "r" | ("u", 4*4(hex-digit)**
 323 **)**
 324 **| ("U", 8*8(hex-digit))**

325 The escape characters directly following the initial backslash are case
 326 sensitive, even though ABNF is case insensitive. The meaning of these
 327 escape characters is:

328 **** - Backslash (U+005C)

329 **\'** - Single Quote (U+0027)

330 **\b** - Backspace (U+0008)

331 **\t** - Horizontal Tab (U+0009)

332 **\n** - Line Feed (U+000A)

333 **\f** - Form Feed (U+000C)

334 **\r** - Carriage Return (U+000D)

335 **\u<hex>** - One Unicode character, with <hex> being exactly 4 hexadecimal
 336 digits in any lexical case, to be interpreted as a Unicode [4] code point.

337 Note: the hexadecimal value is not in an encoded form, but is given as a code
 338 point.

339 **\U<hex>** - One Unicode character, with <hex> being exactly 8 hexadecimal
 340 digits in any lexical case, to be interpreted as a Unicode [4] code point.

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

344

345 Note: The escaping of double quotes is not necessary within a literal string,
 346 since only single quotes can be used to delimit string literals. If the entire
 347 CQL string is put into an environment that uses double quotes to delimit that
 348 string (e.g. as a default value for properties in the MOF), then that
 349 environment must define the escape rules for double quotes.

350 **5.4.3. Identifiers**

351 **identifier-start = UNICODE-S1**

352

353 **identifier-subsequent = identifier-start | DECIMAL-DIGIT**

354

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

356 **5.4.4. Class Paths**

357 **class-name = identifier**

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

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

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

363 **5.4.5. Property Names**

364 **property-scope = class-path "::"**

365 The scoping operator "::" provides a class within which the property name
366 identifier is interpreted. Generally, the class of the property is sufficient.
367 However, if a property of a class is covered by another property, having the
368 same name, that belongs to a subclass, then the "::" syntax is required to
369 access the covered property when in the scope of the covering subclass.
370 Details on how to determine which property to use are in Section 5.4.1.

371

372 **5.4.6. Numeric Literals**

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

375

376 **sign = "+" | "-"**

377

378 **binary-digit = "0" | "1"**

379

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

381

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

382 **decimal-digit = binary-digit | "2" | "3" | "4" | "5"**
 383 **| "6" | "7" | "8" | "9"**

384

385 **hex-digit = decimal-digit**
 386 **| "A" | "B" | "C" | "D" | "E" | "F"**

387

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

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

389

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

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

391

392 **decimal-value = [sign] unsigned-integer**

393

394 **exact-numeric = unsigned-integer "." [unsigned-integer] |**
 395 **"." unsigned-integer**

396

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

398

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

399 **5.4.7. Expressions**

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

401

402 **literal = literal-string**

403 A literal-string has string type.

404 | **decimal-value**

405 A decimal-value has integer type

406 | **binary-value**

407 A binary-value has integer type

408 | **hex-digit-value**

409 A hex-digit-value has integer type

410 | **real-value**

411 A real-value has real type

412 | **TRUE | FALSE**

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

415

416 **arg-list = "*" | ([*DISTINCT*] expr)**

417

418 **chain = literal**

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

420 | **"(" expr ")"**

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

422	identifier
423	The identifier is interpreted as one of the following:
424	<ul style="list-style-type: none"> • 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 <i>C</i>, then the identifier refers to an instance of <i>C</i>, and has type object of <i>C</i>; • Otherwise, if the identifier matches the name of a class <i>C</i> that appears in a FROM criterion without a class-alias, then the identifier refers to an instance of <i>C</i>, and has type object of <i>C</i>; • 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. <p style="margin-left: 2em;">If query feature "Embedded Properties" is supported then the ability to directly access properties of the embedded instance shall be supported.</p> <ul style="list-style-type: none"> • Otherwise, is the query is invalid. <p>If type is Array, then this form without a following “[“ is equivalent to “Identifier [*]”, and only “=” and “<” comparisons are allowed.</p>
425	
426	
427	
428	
429	
430	
431	
432	
433	
434	
435	
436	
437	
438	
439	
440	
441	
442	
443	
444	
445	
446	property-scope identifier
447	Property-scope declares that the identifier identifies a property exposed by
448	the property-scope classname, (see 5.5.1 Property Identification below.) The
449	type of the property is taken as the type of this production.
450	<i>chain</i> CLASSQUALIFIER <i>identifier</i>
451	chain MUST be of type object of <i>C</i> for some class <i>C</i> . This production refers
452	to a qualifier on that class, and the type of the expression is the type of that
453	qualifier. If the class does not expose a qualifier with this name, the
454	qualifier's default value applies.
455	identifier "#" literal-string
456	identifier MUST unambiguously identify a property, (see 5.5.1 Property
457	Identification below.). The type of the property is taken as the type of this
458	production. This production forms a symbolic constant based on the
459	VALUES and VALUEMAP qualifiers; see 5.5.4 Symbolic Constants, below.
460	

CIM Query Language Specification

461		identifier "(" arg-list ")"
462		identifier MUST be the name of a query language function. See 5.6 Query
463		Language Functions for type rules of function calls. For Basic Query, only
464		the numeric, string, instance, path, pathname, and datetime functions shall be
465		supported. Note in particular that this syntax does NOT describe the
466		invocation of a method defined on a CIM class.
467		chain "." [property-scope] identifier
468		Chain MUST have type object of <i>C</i> for some class <i>C</i> .
469		Identifier MUST be the name of a property. For details on the selection of
470		the identified see 5.5.1 Property Identification below below. The type of this
471		production is the type of the property.
472		For Basic Query, chain is restricted to be a class name or class-alias bound in
473		the FROM clause, i.e., Basic Query does not support extraction of properties
474		from embedded objects.
475		identifier₁ PROPERTYQUALIFIER identifier₂
476		This production refers to a property qualifier. Identifier ₁ MUST
477		unambiguously identify a property, (see 5.5.1 Property Identification below),
478		and the type of the expression is the type of that qualifier. If the property
479		doesn't expose a qualifier with this name, the qualifier's default value applies.
480		chain "." [property-scope] identifier₁
481		PROPERTYQUALIFIER identifier₂
482		chain, property-scope (if present), and identifier ₁ together identify a property,
483		as described in 5.5.1 Property Identification below. This production refers to
484		the value of a property qualifier from that property, and the type of the
485		expression is the type of that qualifier. If the property doesn't expose a
486		qualifier with this name, the qualifier's default value applies.
487		For Basic Query, chain is restricted to be a class name or class-alias bound in
488		the FROM clause, i.e., Basic Query does not support extraction of properties
489		from embedded objects.
490		chain "." [property-scope] identifier
491		"#" literal-string
492		chain, property-scope (if present), and identifier together identify a property,
493		as described in 5.5.1 Property Identification below. This production forms a
494		symbolic constant based on the VALUES and VALUEMAP qualifiers; see
495		5.5.4 Symbolic Constants, below. The type of this expression is the type of
496		the identified property.
497		For Basic Query, chain is restricted to be a class name or class-alias bound in
498		the FROM clause, i.e., Basic Query does not support extraction of properties
499		from embedded objects.
500		chain "[" array-index-list "]"
501		chain MUST have type array of <i>T</i> . If array-index-list comprises just a single
502		expr, then this production has type <i>T</i> ; otherwise, the production has type
503		array of <i>T</i> .

CIM Query Language Specification

504	concat = chain
505	The type of the chain is taken as the type of this production.
506	concat " " chain
507	concat and chain MUST have string or char16 type, and the result has string
508	type.
509	
510	factor = concat
511	The type of the concat is taken as the type of this production.
512	("+" "-") concat
513	When this production is used, concat MUST have numeric type, which will
514	be the type of the production
515	If concat is NULL, then the production evaluates to NULL.
516	term = factor
517	The type of the factor is taken as the type of this production.
518	term "*" factor
519	If term and factor both have numeric types, the production has numeric type.
520	If term has a numeric type, and factor has datetime type and evaluates to an
521	interval, then the production has datetime type and will produce an interval
522	value.
523	If term has datetime type and evaluates to an interval, and factor has a
524	numeric type, then the production has datetime type and will produce an
525	interval value. The rules for operations with datetime type operands are
526	defined in Appendix E.1: Datetime Operations.
527	If term or factor is NULL, then the production evaluates to NULL.
528	No other type combinations are allowed.
529	term "/" factor
530	If term and factor both have numeric types, the production has numeric type.
531	If term has a datetime type and evaluates to an interval, and factor has a
532	numeric type, the production has datetime type and will produce an interval
533	value. The rules for operations with datetime type operands are defined in
534	Appendix E.1: Datetime Operations.
535	If term or factor is NULL, then the production evaluates to NULL.
536	No other type combinations are allowed.
537	arith = term
538	The type of the term is taken as the type of this production.
539	arith ("+" "-") term
540	If arith and term both have numeric type, the result has numeric type.

CIM Query Language Specification

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

543 No other type combinations are allowed.

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

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

547 **value-symbol = "#" literal-string**

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

550 **arith-or-value-symbol = arith | value-symbol**

551

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

553

CIM Query Language Specification

554	comp = arith
555	The type of the arith is taken as the type of this production.
556	arith IS [NOT] NULL
557	This production has type Boolean.
558	arith comp-op arith
559	This production has type Boolean for all cases in which it applies. See 5.5.6
560	Comparisons for more detailed description of comparisons.
561	If either arith is NULL, then the production evaluates to NULL.
562	chain comp-op value-symbol
563	The left-hand-side MUST be a property reference, and that property is used
564	as the context for the value-symbol, see 5.5.4 Symbolic Constants below.
565	This production has type Boolean for all cases in which it applies. See 5.5.6
566	Comparisons for more detailed description of comparisons.
567	If chain or the value-symbol is NULL, then the production evaluates to
568	NULL.
569	value-symbol comp-op chain
570	The right-hand-side MUST be a property reference and that property is used
571	as the context for the value-symbol, see 5.5.4 Symbolic Constants below.
572	This production has type Boolean for all cases in which it applies. See 5.5.6
573	Comparisons for more detailed description of comparisons.
574	If chain or the value-symbol is NULL, then the production evaluates to
575	NULL.
576	arith ISA identifier
577	The left-hand-side MUST be either an instance, or a property containing an
578	EMBEDDEDOBJECT or EMBEDDEDINSTANCE. The right-hand-side
579	MUST be the name of a class or a class-alias.
580	The ISA tests whether the left-hand-side is of the class or a subclass of the
581	class named by the right-hand-side identifier.
582	If arith is NULL, then the production evaluates to NULL.
583	The production has Boolean type.
584	/ arith LIKE literal-string
585	arith MUST have string or char16 type; the result has Boolean type.
586	If arith is NULL, then the production evaluates to NULL.
587	The Basic Query feature only includes the Like features described in:
588	Appendix D.1: Basic Like Regular Expressions.
589	<i>arith LIKE arith</i>
590	Both sides of the LIKE comparison must have string or char16 type; the
591	result has Boolean type. The LIKE comparison allows a string or char16 to
592	be tested by pattern-matching, using special characters in the pattern on the
593	right-hand-side. See Appendix D.2: Full Like Extended Regular Expressions

CIM Query Language Specification

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

595 / *array-comp*

596

597 **expr-factor = comp**

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

599 | **NOT comp**

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

601 The following table defines the result of the NOT expression:

<i>comp</i>	<i>NOT comp</i>
TRUE	FALSE
FALSE	TRUE
NULL	NULL

602 **expr-term = expr-factor**

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

604 | **expr-term AND expr-factor**

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

607 The following table defines the result of the AND expression:

<i>expr-term</i>	<i>expr-factor</i>	<i>expr-term AND expr-factor</i>
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
TRUE	NULL	NULL
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE
FALSE	NULL	FALSE

608 **expr = expr-term**

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

CIM Query Language Specification

610 | **expr OR expr-term**

611 | expr and expr-term must both have Boolean type; the production has Boolean
612 | type.

613 | The following table defines the result of the OR expression:

<i>expr-term</i>	<i>expr-factor</i>	<i>expr-term OR expr-factor</i>
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	NULL	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE
FALSE	NULL	NULL

614

615 | **array-index = expr**

616 | expr MUST have unsigned integer type.

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

620 | / *expr* **".."** [*expr*]

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

623 | / **".."** *expr*

624 | expr MUST have unsigned integer type.

625 | **array-index-list = array-index** *(", " *array-index*)

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

627 | / **"*"**

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

629 | / **""**

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

632

CIM Query Language Specification

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

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

639 / *arith-or-value-symbol comp-op* (*ANY* | *EVERY*) *arith*

640 This production acts like the preceding one, except that the array value
641 appears on the right-hand side.

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

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

649 5.4.8. Sort Specification

650 *sort-spec* = *expr* (*ASC* | *DESC*)

651 The specified *expr* MUST be defined in the *SELECT* clause. Note that
652 properties resulting from the specification of a *star-expr* as the *selected-entry*
653 can be subject to sorting. *NULL* values are considered "higher" than all
654 other values. If the *ORDER BY* clause does not completely order the
655 instances of the result set, instances with duplicate values in sorting
656 properties will be displayed in an arbitrary order.

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

658 **5.4.9. Select List**

659 **star-expr = "*"**

660 This production refers to all the properties exposed by all classes defined in
 661 the from-criteria. This includes uncovered properties of superclasses of the
 662 from-criteria classes. Properties of subclasses of the from-criteria classes are
 663 NOT included.

664 Covered properties (i.e. properties of the same name that are not overridden
 665 MAY be explicitly referenced by using the scoping operator "::" in the expr
 666 of the selected-entry production.

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

672 **chain "." [property-scope] "*" |**

673 chain MUST have type object of C for some class C. If property scope is not
 674 present, this production refers to all the properties exposed by C, including
 675 those of C's superclasses. Properties of subclasses of C are NOT included in
 676 the set. If property-scope is present and identifies some class S, it must be the
 677 same class as, or be a superclass of, class C; this production refers to all the
 678 properties exposed by S, including those of S's superclasses. Properties of
 679 subclasses of S are NOT included in the set. The property list produced does
 680 NOT vary over the query.

681 **selected-entry = expr [AS identifier]**

682 expr may have any primitive, reference, or array type, and defines the type of
 683 the column defined by this production. If the type of the expression is an
 684 object type, then the corresponding result column MUST have string type,
 685 and be populated with string representations of the values.

686 The set of column names in the query result MUST NOT contain duplicates.
 687 To avoid duplicated column names in the query result, the "AS identifier"
 688 clause is used to explicitly specify a name. If the "AS identifier" clause is not
 689 present, then the selected entry MUST be a property reference, and the expr
 690 itself (minus any white space) is taken as the name of the corresponding
 691 result column. Note that this means that Basic Query allows only properties
 692 in the select-list.

693 If there is more than one entry in the FROM list, then each selected entry that
 694 is a property reference MUST be a chain expression starting with either a
 695 class name or an alias that is included in the FROM list.

696

697 **star-expr**

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

CIM Query Language Specification

701 If there is more than one entry in the FROM list, then each star-expr MUST
702 be a chain expression starting with either a class name or an alias that is
703 included in the FROM list.

704

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

706 If the select-list contains any aggregating expressions, then all items in the
707 select-list MUST be aggregating expressions.

708 5.4.10. From Criteria

709 *subquery = select-statement*

710

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

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

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

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

723 | *(" subquery ") identifier*

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

728

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

762 **5.5. Considerations of the Constructs in the BNF**

763 The CIM Query Language does not currently define "data change" operations (INSERT,
764 DELETE or UPDATE). These may be added at a later time, but are not currently required.
765 Today, these operations are supported by invoking individual operations defined in the CIM
766 Operations over HTTP Specification [11].

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

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

772 **5.5.1. Property Identification**

773 A CIM class may expose more than one property with a given name, but it is not permitted to
774 define more than one property with a particular name. This can happen if a base class defines
775 a property with the same name as a property defined in a derived class without overriding the
776 base class property. The scoping operator, "::", is used to provide an explicit context for
777 resolving identifiers to properties.
778

779 The general syntax by which a property is identified is:

780 *[chain "."] [property-scope] identifier*, where chain MUST have type object of C.

781

782 Property names identify properties relative to a class context. Given a class context C, the
783 search for the property begins at C and selects a property defined on C whose name matches
784 the identifier, if there is one; if C does not define a property with this name, then the search
785 continues with C's direct superclass, and so on. If no property is located with this search,
786 then the property reference is invalid.
787

CIM Query Language Specification

788 The class context is determined according to the following rules:

- 789 • If property-scope is present, then it declares the class context *C*.
 - 790 ○ If the scoped identifier does NOT name a property exposed by *C*, then the query
791 is invalid.
 - 792 ○ If chain is NOT present, *C* MUST be the same as, a superclass of, or a subclass
793 of, exactly one entry in the FROM list. In this case, chain is inferred to refer to794 instances produced by that FROM list entry. - 795 ○ If chain is present, and it has type object of *D*, for some *D*, then *C* MUST be the
796 same as, or a superclass of, or a subclass of *D*. - 797 ○ If chain is present, and it does NOT have type object of *D* for some *D*, then chain
798 MUST have type object. - 799 ○ If the value of the chain expression is NOT of class *C*, (or subclasses of *C*), then
800 the application of the property produces NULL.
- 801 • Otherwise, if chain "." is present, then chain MUST be of type object of *C*, and *C* is the
- 802 class context.
- 803 • Finally if neither are present, then the identifier must be declared in at most one of the
- 804 classes named in the FROM list.
- 805 • Otherwise the context cannot be determined, and the query is invalid.
- 806

807 **5.5.2. Arrays**

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

811 **5.5.3. Embedded Objects**

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

819
820 The actual type of an EMBEDDEDOBJECT is not known until an instance is selected. This can
821 lead to situations in which the type of a projected result cannot be determined in advance of
822 the query's execution, and, indeed, may vary even within the execution of a single query.
823 This affects the resolution of properties of the embedded object. To remove ambiguity,
824 queries that concern themselves with properties of embedded objects MUST use the scoping
825 operator (":") to scope those properties. A CQL implementation MUST reject any query
826 which involves expressions whose type cannot be determined.

827

CIM Query Language Specification

828 For example, the following would be permitted since both properties of SourceInstance were
829 provided a scope. The DeviceID property would be returned as NULL when SourceInstance
830 is a CIM_PhysicalElement.

```
831  
832     SELECT SourceInstance.CIM_LogicalDevice::DeviceID,  
833            SourceInstance.CIM_ManagedSystemElement::OperationalStatus  
834 FROM CIM_InstIndication  
835 WHERE SourceInstance ISA CIM_LogicalDevice  
836 OR     SourceInstance ISA CIM_PhysicalElement
```

837 5.5.4. Symbolic Constants

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

842
843 If the property does not also expose a VALUEMAP qualifier, then the property MUST have
844 integer type, and the index of the literal-string among the VALUES qualifier's value is taken
845 as the value of this production. If, conversely, the property does also expose a VALUEMAP
846 qualifier, then the value for this production will be based on the value in the VALUEMAP
847 array corresponding to the selected value of the VALUES array, as follows: (1) if the
848 property has type string, then the VALUEMAP entry itself is the value of the production;
849 otherwise, (2) the property MUST have integer type, the VALUEMAP entry MUST NOT
850 include the sequence "..", and the VALUEMAP entry is converted into an integer of the
851 appropriate type. E.g., CIM_FCPort.OperationalStatus#'OK' is equivalent to the constant 2,
852 and CIM_FCPort.OperationalStatus#'Predictive Failure' is equivalent to 5.

853
854 If the expression on one side of a comparison identifies exactly one property, then the #
855 syntax MAY be used in a standalone form on the opposite side of the comparison. The
856 identified property becomes the defining context of the symbolic constant. For example:

```
857     CIM_FCPort.OperationalStatus[3] > #'OK'  
858 is equivalent to  
859     CIM_FCPort.OperationalStatus[3] > CIM_FCPort.OperationalStatus #'OK'
```

860
861 If a class name is used to qualify a symbolic constant, that class does not need to be related to
862 any class in the query. For example the following query is valid even though
863 CIM_LogicalDevice has nothing to do with the query:

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

866 5.5.5. Computation and Types

867 The use of arithmetic operators causes numeric types to be "widened" as necessary to
868 minimize the loss of precision. Unless both operands are unsigned, addition, subtraction, and
869 multiplication among integer types results in sint64. If both operands are unsigned, then the

870 result is uint64. Otherwise (i.e., all cases of division, as well as addition, subtraction, or
871 multiplication involving at least one non-integer type), all arithmetic operations produce
872 real64 type. If an overflow or underflow occurs, an error is returned.
873 Arithmetic and comparisons on datetime types are defined in Appendix E: Datetime
874 Operations and BNF

875 **5.5.6. Comparisons**

876 Comparison is supported between all numeric types. When comparisons are made between
877 different numeric types, comparison is performed using the type with the greater precision.
878

879 Comparison between strings and between char16 values is supported, and is done case-
880 sensitively on a unicode character basis. A comparison between a string and a char16 is
881 accomplished by treating the char16 value as a single-character string. For string and char16
882 comparison and sort operations, the Default Collation Algorithm as defined in ISO/IEC
883 14651 [21] and Unicode Technical Report #10 [20] MUST be applied. Unicode character
884 based comparison is done as follows:
885

886 The "=" and "<>" operators MUST use the string identity matching rules defined in W3C
887 "Character Model for the World Wide Web 1.0: Normalization" [7], section 4 "String
888 Identity Matching".
889

890 The following rules apply to comparison between strings and char16 values using the "<",
891 and ">" operators:
892

893 1) For Basic Query, these operators MUST behave as if the normalization defined in
894 "Character Model for the World Wide Web 1.0: Normalization" [7], section 4 "String
895 Identity Matching", was applied and then the comparison was performed on the resulting
896 strings. The strings are compared from the beginning, on a Unicode character basis. Each
897 character is compared based on its Unicode codepoint order. The first character found to be
898 different determines the result of the comparison. If the strings are of different lengths, but
899 are otherwise equal, then the longer string is greater than the shorter string. Note: for
900 implementations that use the UTF-8 or UTF-32 as the encoding, the binary order of the
901 encoded characters matches the Unicode codepoint order. For UTF-16, the binary order of
902 the supplementary characters does not match their Unicode codepoint order. For more
903 information, refer to section 2.5 of "The Unicode Standard" [5].
904

905 2) For the Full Unicode feature, these operators MUST behave as if the normalization
906 defined in [7], section 4 "String Identity Matching", was applied and then the default
907 collation order defined in the Unicode Collation Algorithm [8] was used on the resulting
908 strings. Note that this collation order accomodates most languages, without having to take
909 any locales into account.
910
911
912

CIM Query Language Specification

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

915

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

918

919 Reference comparison is performed via a process of comparing certain components of the
920 references. The components to be compared are the namespace type, namespace handle, and
921 model path, as defined by the CIM Infrastructure Specification [1]. Two references are
922 considered to be equal if all of the following conditions are true:

- 923 • For the model path, all of the following conditions must be true: There must be the
924 same number of key property name/value pairs. For each key property name/value
925 pair in one reference, exactly one matching key property name/value pair must be
926 found in the other reference. The order of the key property name/value pairs does not
927 affect the comparison. Comparison is done case-insensitively for key property
928 names. Key property values are compared according to their type, as defined in
929 section 5.4.6, Comparisons.
- 930 • For all components except the model path, the comparison is done case-insensitively.

931

932 Note: the implementation MAY perform reference comparisons using alternative, but
933 equivalent, paths or representations.

934

935 Comparison of classes **REQUIRES** that the ClassName is the same and that the properties
936 and property types defined by this class and by each superclass in the classes hierarchy
937 compare equal. The comparison of class names, property names and property types is done
938 case-insensitively. The set of qualifiers defined on each class **MUST** be the same and
939 evaluate to the same values.

940

941 Comparison of instances **REQUIRES** that the instances be of the same class, and that all
942 property values either compare equal or are both null. The comparison of the property values
943 is done case-sensitively.

944

945 For comparison between an array property and a non-array property, please refer to section
946 5.5.7 (Comparisons of Array and Scalar). Note that this type of comparison shall be
947 supported if query feature "Array Range" is supported.

948

949 For comparison between arrays, comparison of complete arrays shall be supported in Basic
950 Query. Comparison of parts of arrays shall be supported in query feature Array Range. The
951 ArrayType governs how matches are made. There are three types: Bag, Ordered, and
952 Indexed. If one of the arrays is a Bag, then comparison rules for Bags are used. As defined
953 in DSP0004 [1], a bag is an unordered multiset. Two arrays of ArrayType "Bag" are equal if
954 and only if the number of elements is equal and if it is possible to find a permutation for one
955 of the arrays so that for an element-by-element comparison, all elements of the compared
956 arrays are equal. Equality for Bag-type arrays MAY be tested by sorting both arrays and then
957 doing an element-by-element comparison. For comparison of Ordered and Indexed, an

958 element-by-element comparison is performed. Arrays which have different numbers of
959 elements do not compare equal.
960
961 Other than the cases described in this section, comparisons among disparate types are not
962 part of CQL.

963 **5.5.7. Comparisons of Array and Scalar**

964 This section only applies to comparison operations between array properties and non-array
965 properties, as part of query features “Array Range” and “Satisfies Array”. A comparison
966 between an array property and a non-array property is illegal if neither “EVERY” nor
967 “ANY” keyword is used. If multiple elements of an array property are compared, the
968 operation evaluates to TRUE if and only if the specified comparison is TRUE for all the
969 indicated Array Range. Here are a few examples of the use of array processing:
970

- 971 • "EVERY CIM_LogicalDevice.OperationalStatus[*] <> 2" is TRUE if and only if
972 every value of the OperationalStatus array is not 2
- 973 • "EVERY CIM_LogicalDevice.OperationalStatus[*] = 2" is TRUE if and only if all of
974 the values of OperationalStatus are 2
- 975 • "EVERY CIM_LogicalDevice.OperationalStatus[*] < 2" is TRUE if and only if all of
976 the values of OperationalStatus are less than 2
- 977 • "ANY CIM_LogicalDevice.OperationalStatus[*] > 2" is TRUE if and only if any the
978 values of the OperationalStatus array are greater than 2
- 979 • "ANY CIM_LogicalDevice.OperationalStatus[*] <> 2" is TRUE if and only if any of
980 the values of the OperationalStatus array are NOT 2
- 981 • "NOT EVERY CIM_LogicalDevice.OperationalStatus[*] = 2" is TRUE if and only if
982 any of the values of the OperationalStatus array are <> 2
- 983 • "CIM_LogicalDevice.OperationalStatus[0] = 2" is TRUE if the first value of the array
984 is set to 2
- 985 • "EVERY CIM_LogicalDevice.OperationalStatus[0..3] > 2" is TRUE if the first 4
986 values of the OperationalStatus array are each greater than 2
- 987 • "ANY stat IN CIM_LogicalDevice.OperationalStatus[*] SATISFIES (stat=3 OR stat
988 > 5)" is TRUE if any value of the OperationalStatus array is equal to 3 or greater than
989 5

990 5.6. Query Language Functions

991 This section describes the functions available for CIM Query Language.

992

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

995 5.6.1. Aggregation Functions

996 These functions are only valid within the select-list. If the select-list contains any
997 aggregating expressions, then all items in the select-list **MUST** be aggregating expressions.
998 In this case, the result set contains one row and the aggregating expressions operate on the
999 rows determined by the **WHERE** clause. An aggregating expression is an expression with at
1000 least one aggregation function, where any properties are used only in the expression
1001 representing the argument of an aggregation function.
1002

1003 **COUNT([DISTINCT] expr)**: Counts the number of rows for which the argument is non-
1004 NULL. If **DISTINCT** is specified, then **COUNT** counts the number of different non-NULL
1005 values the argument assumes. The set of rows which **COUNT** considers is affected by
1006 **FIRST** or **DISTINCT** on the select-statement. The result type is uint64.
1007

1008 **COUNT(*)**: **COUNT(*)** is a special function returning the number of rows the query selects.
1009 The value returned by **COUNT** is affected by **FIRST** or **DISTINCT** on the select-statement.
1010 The result type is uint64.
1011

1012 **MIN(expr)**

1013 **MAX(expr)**

1014 **SUM(expr)**: These functions all act analogously to the like-named SQL functions. The
1015 argument to each function must have numeric type; the result is of the same type as the
1016 argument. The result type is the same as the type of expr.
1017

1018 **MEAN(expr)**

1019 **MEDIAN(expr)**: These functions compute the mean and median, respectively, of the
1020 distribution represented by the non-NULL values the arguments assumes. The result type for
1021 **MEAN** is real64. The result type for **MEDIAN** is the type of expr.

1022 5.6.2. Numeric Functions

1023 **DATETIMETOMICROSECOND(expr)**: The argument **MUST** have datetime type, and
1024 the result has type uint64. If the argument is a timestamp, it is converted to the number of
1025 microseconds since 00:00:00.000000UTC on 1/1/0000; otherwise (i.e., if the argument is an
1026 interval), it is converted to microseconds.

1027 If expr computes to a time before 00:00:00.000000UTC on 1/1/0000 the result is an
1028 arithmetic underflow error. If expr computes to a time after 23:59:59.999999 UTC on

1029 12/31/9999, the result is an arithmetic overflow error. In either case, the query will result in
1030 an error.

1031

1032 **STRINGTOUINT(expr):** The argument **MUST** have string or char16 type and must be a
1033 binary-value, hex-digit-value, decimal-value, or real-value in the range of 0 to $2^{64}-1$. The
1034 result has type uint64. The fractional portion of any real-value is discarded.

1035

1036 **STRINGTOSINT(expr):** The argument **MUST** have string or char16 type and must be a
1037 binary-value, hex-digit-value, decimal-value, or real-value in the range of -2^{63} to $2^{63}-1$. The
1038 result has type sint64. The fractional portion of any real-value is discarded.

1039

1040 **STRINGTOREAL(expr):** The argument **MUST** have string type and must be a binary-
1041 value, hex-digit-value, decimal-value, or real-value. The result has type real64.

1042 **5.6.3. String Functions**

1043 **UPPERCASE(expr):** The argument **MUST** have string or char16 type, and the result has
1044 string type. This function canonicalizes its argument by converting all lowercase characters
1045 to uppercase. For Basic Query, this function converts lowercase characters in the US-ASCII
1046 range (U+0000...U+007F) to uppercase. Characters outside of the US-ASCII range are not
1047 changed. For the Full Unicode feature, this function performs Case Mapping, as defined in
1048 the Unicode standard [5], on all characters.

1049

1050 **NUMERICTOSTRING(expr):** The argument **MUST** have numeric type, and the result has
1051 string type. This function constructs a string representation of its argument, using the
1052 following rules:

- 1053 • If the argument is of one of the integer types, it is represented using decimal radix.
1054 Positive numbers do not have a plus sign, and negative numbers have a preceding
1055 minus sign.
- 1056 • If the argument is of one of the real types, it is represented using decimal mantissa. If
1057 an exponent is needed, it uses decimal radix and follows after an upper case "E", and
1058 does not have any leading zeros. If the mantissa has more than one digit, the decimal
1059 point is always after the first digit. Positive mantissas and exponents do not have a
1060 plus sign, and negative mantissas and exponents have a preceding minus sign.
- 1061 • If the argument has a value of zero, it is represented as the single character "0".

1062

1063 **REFERENCETOSTRING(expr):** The argument **MUST** have reference type, and the
1064 result has string type. This function returns an object path string based exclusively on the
1065 information in the input reference. Canonicalization **MAY** be accomplished by using the
1066 Path Functions.

1067 **5.6.4. Instance Functions**

1068 These functions operate on objects, references or strings whose contents is a WBEM-URI, as
1069 defined in the WBEM URI Mapping Specification, DSP0207 [2].

1070

CIM Query Language Specification

1071 **INSTANCEOF([expr]):** The argument **MUST** be an instance, an embedded instance, an
1072 embedded object, a reference to an instance, or a string containing a WBEM-URI to an
1073 instance. If the argument is of type embedded object, it **MUST** represent an instance and
1074 **MUST** be scoped using the property-scope syntax. In all cases using valid input, if the
1075 instance is of type C, the result of this function is an embedded instance of type C. In all
1076 other cases, the query is invalid

1077

1078 **5.6.5. Path Functions**

1079 These functions operate on objects, references or strings whose contents is a WBEM-URI, as
1080 defined in the WBEM URI Mapping Specification, DSP0207 [2].

1081

1082 **CLASSPATH([expr]):** The argument **MUST** be an object, a reference, or a string
1083 containing a WBEM-URI. The result of this function is of type reference. If the argument is
1084 of type reference or string and it refers to a class, the result of this function refers to that
1085 class. If the argument is of type reference or string and it refers to an instance, the result of
1086 this function refers to the creation class of that instance. If the argument is of type object, it
1087 **MUST** be an instance value that is **NOT** an Indication or an embedded instance and the result
1088 of this function refers to the creation class of that instance. In all other cases, the query is
1089 invalid. Whether or not the class or instance referenced by the argument exists, does not
1090 matter for the successful execution of the function. The function does not add any missing
1091 components to the namespace path of the resulting reference.

1092

1093 **OBJECTPATH([expr]):** The argument **MUST** be an object, a reference, or a string
1094 containing a WBEM-URI. The result of this function is of type reference. If the argument is
1095 of type reference or string and it refers to a class, the result of this function refers to that
1096 class. If the argument is of type reference or string and it refers to an instance, the result of
1097 this function refers to that instance. If the argument is of type object, it **MUST** be an instance
1098 value that is **NOT** an Indication or an embedded instance and the result of this function refers
1099 to that instance. In all other cases, the query is invalid. Whether or not the class or instance
1100 referenced by the argument exists, does not matter for the successful execution of the
1101 function. The function does not add any missing components to the namespace path of the
1102 resulting reference.

1103

1104 **5.6.6. Datetime Functions**

1105 **CURRENTDATETIME():** Returns the "current" datetime as determined by the
1106 implementation.

1107

1108 **DATETIME(expr):** The argument **MUST** be of type string, and at runtime **MUST** take on a
1109 25-character value conformant with a datetime specification (either timestamp or interval).
1110 The result has datetime type.

1111

CIM Query Language Specification

1112 **MICROSECONDTOTIMESTAMP(expr):** The argument **MUST** be of an integer type,
1113 and the result has datetime type. The argument will be interpreted as a number of
1114 microseconds since 00:00:00.000000UTC on 1/1/0000, and the result will be a timestamp.

1115

1116 **MICROSECONDTOINTERVAL(expr):** The argument **MUST** be of an integer type, and
1117 the result has interval (datetime) type. The argument will be interpreted as a number of
1118 microseconds, and the result will be an interval.

1119

1120 5.7. Query Considerations

1121 The result of a query is a table that contains a set of zero or more rows that contain the
1122 columns defined in the select-list.. This table is not stored beyond the execution of a
1123 particular invocation of the query. These instances have the following additional
1124 characteristics:

- 1125 • Each column has a type and a distinct name.
- 1126 • Each classname in the FROM list is considered by query as a table that has one row
1127 for each class instance and where the properties of the class are mapped to columns of
1128 the table.
- 1129 • Subqueries are considered by query to produce tables.
- 1130 • On the relation to classes, instances, and properties.
 - 1131 1. Each table MAY be considered as a class. However, it is NOT required to
1132 conform to the definition a CIM class.
 - 1133 2. Each row MAY be considered as an instance. However, it is NOT required to
1134 conform to the definition a CIM instance.
 - 1135 3. Each column MAY be considered a property that conforms to the definition of
1136 a CIM Property.
- 1137 • A query may be specified as part of a class definition, (such as CIM_IndicationFilter,
1138 CIM_QueryCondition, and CIM_MethodAction.) The implementation of the class is
1139 responsible for processing query specified in instances of that class For example,
1140 CIM_IndicationFilter subclasses constrain the select-list to produce entries that
1141 conform to the CIM_Indication subclass that is used in the FROM clause. The results
1142 are then typically delivered by the CIM_ListenerDestination subclass as instances of
1143 the named CIM_Indication subclass.

1144 5.8. Query Errors

1145 When processing a query (either by a CIM Server or a provider), it is legitimate to reject the
1146 query. The following errors are defined in the CIM Operations Specification for Exec
1147 Query:

- 1148 • CIM_ERR_ACCESS_DENIED
- 1149 • CIM_ERR_NOT_SUPPORTED
- 1150 • CIM_ERR_INVALID_NAMESPACE
- 1151 • CIM_ERR_INVALID_PARAMETER (including missing, duplicate, unrecognized or
1152 otherwise incorrect parameters)
- 1153 • CIM_ERR_QUERY_LANGUAGE_NOT_SUPPORTED (the requested query
1154 language is not recognized)
- 1155 • CIM_ERR_INVALID_QUERY (the query is not a valid query in the specified query
1156 language; i.e., a syntax or semantic error occurred. For CQL, this error is also
1157 returned if the language is correct, but the query features used by the query are not
1158 supported)
- 1159 • CIM_ERR_FAILED (some other unspecified error occurred)

1160 If a Query is implemented as part of a Class, then the Provider of the class is responsible for
1161 error handling and for appropriately passing errors back to the client of the class or its
1162 instances. For instance, if the class supports a string property named Query and the string is
1163 a constant, then the implementation must assure that the string is correct. Note that in this
1164 case, the implementation may be completely hard-coded. If the property is set by a CIM
1165 Client, then the implementation is responsible for checking the validity of the query when the
1166 property is set. If invalid, the CIM operation used to set the property MUST return
1167 CIM_ERR_INVALID_QUERY if an ExecuteQuery or CIM_ERR_FAILED for all others.

1168 In the future, the CIM_Error class will be used to expand on the errors defined above.

1169 **5.9. Query Functional Description**

1170 CIM environments vary greatly in terms of processing capabilities, and required
 1171 functionality. The CIM Query Language can be segmented based on functionality, with the
 1172 assumption that a reduction in functionality is equivalent to reduced processing requirements.

1173 The following table defines the "Basic Features" required for CQL support and a set of
 1174 optional CQL processing features that MAY be provided by a component. Discovery of
 1175 these features is enabled via the CQLFeatures enumeration property of the QueryCapabilities
 1176 class. Each optional feature MUST be fully supported before it is advertised as being
 1177 supported.

1178 The table also tracks the status of each feature. A status of "Final" means that the feature
 1179 has at least two independent implementations and that all issues have been resolved in a
 1180 manner consistent with DMTF policy. Otherwise, the status will be marked as
 1181 "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: <ul style="list-style-type: none"> • MUST support at least two from-specifiers. • Support for more than two from-specifiers IS NOT part of Simple Join. 	Basic Query	Experimental

CIM Query Language Specification

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 <ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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

CIM Query Language Specification

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

1182 **Table 1: Query Features**

1183 If a query includes clauses or constructs not supported by the infrastructure, the error
1184 `CIM_ERR_INVALID_QUERY` MUST be returned on a request made via `ExecuteQuery` or
1185 `CIM_ERR_FAILED` for all other CIM operations.

1186 6 CIM Query Template Language

1187 This section defines a separate and optional pre-processing facility that supports the
1188 conversion of CQL template strings into CQL strings.

1189 The pre-processing facility parses the input string from left to right for pre-processor tokens.
1190 Each pre-processor token represents a pre-processor variable named by identifier.

- 1191 • The pre-processor recognizes a backslash, (\) as an escape character when the next
1192 character is a single-quote (') (U+0027)

1193 **Note:** The escaping of double quotes is not necessary within a literal string,
1194 since only single quotes can be used to delimit string literals. If the entire pre-
1195 processor string is put into an environment that uses double quotes to delimit
1196 that string (e.g. as a default value for properties in the MOF), then that
1197 environment must define the escape rules for double quotes.

- 1198 • If a non-escaped single-quote is encountered, detection of pre-processor tokens is
1199 disabled until the first character after a corresponding non-escaped single-quote.
- 1200 • While detection is enabled, the sequence "\$"identifier"\$" is recognized as a pre-
1201 processor token.
- 1202 • For each pre-processor token encountered, the pre-processor makes a string
1203 substitution for that token and resumes parsing with the first character after the
1204 replaced token.

1205 The string substitution replaces the token with the value of the pre-processor variable as
1206 defined to the pre-processing facility. The value of the pre-processor variable must be a
1207 string value. Note that any occurrences of the sequence "\$"identifier"\$" in that string value
1208 will not be replaced. The mapping of a pre-processor variable to a value is not specified here
1209 and must be specified where this facility is used.

1210 Pre-processor tokens are semantically unrelated to the identifiers of the CQL query itself.

1211 Unquoted \$'s may not appear in the query template except as part of pre-processing tokens.

1212 Following the convention detailed in section 5.2 on identifying a query language, the string
1213 “DMTF:CQLT” will identify the CIM Query Template language to represent the use of this
1214 pre-processing capability for CQL.

1215 6.1. Pre-processor Examples

1216 1.) Define a template for retrieving instances of the class identified by the variable
1217 *targetClassName*.

1218 Assuming the value of *targetClassName* is "CIM_StorageExtent", the CQL pre-processor
1219 would translate the string

1220 SELECT *

1221 FROM \$targetClassName\$

1222 into

1223 SELECT *

1224 FROM CIM_StorageExtent

1225 2.) Define a template for requesting account information about the entity identified by the
1226 variable *UserID*.

1227 Assuming the value of *UserID* is "guest", the CQL pre-processor would translate the string

1228 SELECT *

1229 FROM CIM_Account

1230 WHERE UserID = \$UserID\$

1231 into

1232 SELECT *

1233 FROM UserID = 'guest'

1234 3.) Define a template that allows the filter condition to be restricted based on the value of the
1235 variable *whereClause*.

1236 Assuming the value of *whereClause* is "WHERE UPSSttyPath = '/dev/ttyOp1' AND
1237 MonitorEventID = 20", the CQL pre-processor would translate the string

1238 SELECT DetectionTime,

1239 SystemIPAddress,

CIM Query Language Specification

1243

1240 PerceivedSeverity,

1241 MonitorEventID,

1242 UPSttyPath

FROM Acme_UPSAlertIndication

1244 \$whereClause\$

1245 into

1246 SELECT DetectionTime,

1247 SystemIPAddress,

1248 PerceivedSeverity,

1249 MonitorEventID,

1250 UPSttyPath

1251 FROM Acme_UPSAlertIndication

1252 WHERE UPSttyPath = '/dev/ttyOp1' AND MonitorEventID = 20

1253 7 Examples

1254 This section provides a number of sample queries to illustrate the use of the Query language.

1255 7.1. Discovery examples

1256
1257 1. Get the object path, ElementName and Caption for all StorageExtents

1258 Required Features: Basic Query, Extended Select List

1259
1260 SELECT OBJECTPATH(CIM_StorageExtent) AS Path,
1261 ElementName, Caption
1262 FROM CIM_StorageExtent

1263 A set of instances would be returned with three properties: the object path of the
1264 instance, as well as the ElementName and Caption properties.

1265 2. Select all LogicalDevices on a particular ComputerSystem that have an
1266 OperationalStatus not equal to "OK" (value = 2), and return their object paths
1267 and OperationalStatus.

1268 Required Features: Basic Query, Extended Select List, Complex Join,
1269 Array Range

1270
1271 SELECT OBJECTPATH(CIM_LogicalDevice) AS Path,
1272 CIM_LogicalDevice.OperationalStatus[*]
1273 FROM CIM_LogicalDevice,
1274 CIM_ComputerSystem,
1275 CIM_SystemDevice
1276 WHERE CIM_ComputerSystem.ElementName = 'MySystemName'
1277 AND CIM_SystemDevice.GroupComponent =
1278 OBJECTPATH(CIM_ComputerSystem)
1279 AND CIM_SystemDevice.PartComponent =
1280 OBJECTPATH(CIM_LogicalDevice)
1281 AND ANY CIM_LogicalDevice.OperationalStatus[*] <> 2)

1282 A set of instances would be returned, each with the following properties: a string
1283 containing the object path of the instance of CIM_LogicalDevice and the
1284 OperationalStatus array property.

CIM Query Language Specification

- 1285 3. Get all StorageExtent and MediaAccessDevice instances. Note that the
1286 projection is limited to instances that are either CIM_StorageExtent or
1287 CIM_MediaAccessDevice, however only properties of CIM_LogicalDevice
1288 and its superclasses are returned.

1289 Required Features: Basic Query

```
1290 SELECT *  
1291 FROM CIM_LogicalDevice  
1292 WHERE CIM_LogicalDevice ISA CIM_StorageExtent OR  
1293 CIM_LogicalDevice ISA CIM_MediaAccessDevice
```

1294 A set of instances would be returned with a complete select-list as defined by
1295 CIM_LogicalDevice.

- 1296 4. List all ComputerSystems and the object paths of any instances dependent on
1297 the system as described by the Dependency association.

1298 Required Features: Basic Query, Extended Select List, Complex Join

```
1299  
1300 SELECT CIM_ComputerSystem.*,  
1301         OBJECTPATH(CIM_ManagedElement) AS MEObjectName  
1302 FROM CIM_ComputerSystem,  
1303         CIM_ManagedElement,  
1304         CIM_Dependency  
1305 WHERE CIM_Dependency.Antecedent =  
1306         OBJECTPATH(CIM_ComputerSystem)  
1307         AND CIM_Dependency.Dependent =  
1308         OBJECTPATH(CIM_ManagedElement)
```

1309 This query returns a set of instances defined by the references of the Dependency
1310 association's instances. The instances that are created contain all the properties of
1311 CIM_ComputerSystem and a string representing the related/associated
1312 ManagedElement's object path.

1313

CIM Query Language Specification

1314 5. Traverse from a resource (CIM_ComputerSystem) to the
1315 CIM_BaseMetricValue instances associated through the CIM_MetricForME
1316 association. The resource instance is known by its keys, and there are many
1317 BaseMetricValue objects associated with it (>10000), and the selection criteria
1318 is such that only a handful of them matches.

1319 Required Features: Basic Query, Extended Select List, Complex Join

```
1320  
1321 SELECT OBJECTPATH(CIM_ComputerSystem) AS CSOBJECTPATH,  
1322     CIM_BaseMetricValue.*  
1323 FROM CIM_ComputerSystem,  
1324     CIM_BaseMetricValue,  
1325     CIM_MetricForME  
1326 WHERE CIM_ComputerSystem.Name = 'MySystem1'  
1327     AND CIM_BaseMetricValue.TimeStamp >  
1328     DATETIME('200407101000*****+300')  
1329     AND CIM_BaseMetricValue.TimeStamp <  
1330     DATETIME('200407101030*****+300')  
1331     AND CIM_BaseMetricValue.Duration =  
1332     DATETIME('000000000005*****:000')  
1333     AND CIM_MetricForME.Antecedent =  
1334     OBJECTPATH(CIM_ComputerSystem)  
1335     AND CIM_MetricForME.Dependent =  
1336     OBJECTPATH(CIM_BaseMetricValue)
```

1337 As in #4, this query returns a set of instances defined by the query's join. The
1338 instances that are returned contain all properties of CIM_BaseMetricValue and
1339 the associated ComputerSystem's object path.

1340 The query in this example is very selective: Only 6 instances are returned, where
1341 the combined number of instances in the classes selected from can be in the tens
1342 of thousands. This shows that it is essential that these instances never be
1343 enumerated or "walked" in the implementation of the query engine, since this
1344 would likely result in huge computational penalties. It is critical to appropriately
1345 break down the query to the different providers involved.

CIM Query Language Specification

- 1346 6. Display all the Settings for a particular CIM_ManagedSystemElement in a
1347 Composite Setting that is associated with the MSE.

1348 Required Features: Basic Query, Complex Join

```
1349  
1350 SELECT SD.*  
1351 FROM CIM_SettingData CSD,  
1352      CIM_SettingData SD,  
1353      CIM_ManagedSystemElement MSE,  
1354      CIM_ElementSettingData ESD,  
1355      CIM_ConcreteComponent CC  
1356 WHERE OBJECTPATH(MSE) = 'some desired key'  
1357      AND ESD.ManagedElement = OBJECTPATH(MSE)  
1358      AND ESD.SettingData = OBJECTPATH(CSD)  
1359      AND CC.GroupComponent = OBJECTPATH(CSD)  
1360      AND CC.PartComponent = OBJECTPATH(SD)
```

1361 A set of instances would be returned (which meet the association criteria) with
1362 properties as specified by CIM_SettingData.

CIM Query Language Specification

1363 7. Get a storage array's LUN masking and mapping for a failed FCPort. This
1364 query uses aliasing in the FROM clause and a series of sub-queries. The use of
1365 nested subqueries guides the query engine through a step-wise process that is
1366 similar to one that would be used by a client executing a series of CIM intrinsic
1367 operations. Use of subqueries is recommended to limit the complexity of
1368 otherwise very large joins. The principle advantage over the series of intrinsic
1369 operations is that the query is a single operation that only returns the final
1370 results.

1371 Required Features: Basic Query, Extended Select List, Complex Join,
1372 Subquery, Array Element
1373

```
1374 SELECT OBJECTPATH(pms) AS PrivilegeMgmtServiceInst,  
1375     Oh AS StorageHardwareIDInst, Op AS AuthorizedPrivilegeInst,  
1376     Ov AS StorageVolumeInst  
1377 FROM CIM_HostedService hs,  
1378     CIM_PrivilegeManagementService pms,  
1379     ( SELECT OBJECTPATH(cs) AS Oc, O.Op, O.Oh, O.Ov  
1380       FROM CIM_ComputerSystem cs, CIM_SystemDevice sd,  
1381         ( SELECT OBJECTPATH(v) AS Ov, P.Op, P.Oh  
1382           FROM CIM_AuthorizedTarget t,  
1383             CIM_StorageVolume v,  
1384             ( SELECT OBJECTPATH(p) AS Op,  
1385               OBJECTPATH(hi) AS Oh  
1386               FROM CIM_StorageHardwareID hi, CIM_AuthorizedPrivilege p,  
1387                 CIM_AuthorizedSubject s,  
1388                 ( SELECT SourceInstance.  
1389                   CIM_FCPort ::PermanentAddress  
1390                   FROM CIM_InstModification  
1391                   WHERE SourceInstance ISA CIM_FCPort  
1392                   AND ANY  
1393                     SourceInstance.CIM_FCPort::OperationalStatus[*]  
1394                     <> #'OK'  
1395                 ) fc  
1396               WHERE fc.PermanentAddress = hi.StorageID  
1397                 AND s.PrivilegedElement = OBJECTPATH(hi)  
1398                 AND s.Privilege = OBJECTPATH(p)  
1399             ) P  
1400           WHERE t.Privilege = P.Op AND t.TargetElement = OBJECTPATH(v)  
1401         ) O  
1402       WHERE sd.PartComponent = Ov  
1403         AND sd.GroupComponent = OBJECTPATH(cs)  
1404     ) C  
1405 WHERE hs.Antecedent = Oc AND hs.Dependent = OBJECTPATH(pms)  
1406
```

CIM Query Language Specification

1407 Without the use of subqueries, but keeping the same color codes to relate to the
1408 subqueries of the above query, an equivalent query can be expressed as:

```
1409  
1410 SELECT OBJECTPATH(pms) AS PrivilegeMgmtServiceInst,  
1411         OBJECTPATH(hi) AS StorageHardwareIDInst,  
1412         OBJECTPATH(p) AS AuthorizedPrivilegeInst,  
1413         OBJECTPATH(v) AS StorageVolumeInst  
1414 FROM   CIM_InstModification im,  
1415        CIM_StorageHardwareID hi,  
1416        CIM_AuthorizedSubject s,  
1417        CIM_AuthorizedPrivilege p,  
1418        CIM_AuthorizedTarget t,  
1419        CIM_StorageVolume v,  
1420        CIM_SystemDevice sd,  
1421        CIM_ComputerSystem cs,  
1422        CIM_HostedService hs,  
1423        CIM_PrivilegeManagementService pms  
1424 WHERE im.SourceInstance ISA CIM_FCPort  
1425        AND ANY im.SourceInstance.CIM_FCPort::OperationalStatus[*] <> #'OK'  
1426        AND im.SourceInstance.CIM_FCPort::PermanentAddress = hi.StorageID  
1427        AND s.PrivilegedElement = OBJECTPATH(hi)  
1428        AND s.Privilege = OBJECTPATH(p)  
1429        AND t.Privilege = OBJECTPATH(p)  
1430        AND t.TargetElement = OBJECTPATH(v)  
1431        AND sd.PartComponent = OBJECTPATH(v)  
1432        AND sd.GroupComponent = OBJECTPATH(cs)  
1433        AND hs.Antecedent = OBJECTPATH(cs)  
1434        AND hs.Dependent = OBJECTPATH(pms)
```

1436 The primary difference is that without the use of subqueries, the query
1437 implementation would have to determine how to optimize this query to avoid an
1438 uncorrelated join across all of the instances belonging to the 10 classes named in
1439 the 'FROM' clause. This level of analysis is beyond the capability of most
1440 expected implementations.

1441
1442 8. Example of mathematical aggregation function

1443 Required Features: Basic Query, Extended Select List, Aggregation, Result Set
1444 Operations, Subquery

```
1445  
1446 SELECT DISTINCT OBJECTPATH(sv) AS VolumePath,  
1447         (sv.BlockSize * sv.NumberOfBlocks) AS Size  
1448 FROM   CIM_StorageVolume sv,  
1449        (SELECT MAX(v.BlockSize*v.NumberOfBlocks) AS Maxbytes  
1450         FROM CIM_StorageVolume v) mv  
1451 WHERE (sv.BlockSize * sv.NumberOfBlocks) = mv.Maxbytes
```

1452 7.2. Event detection examples

1453
1454 1. As regards query in Indication processing, the following examples are taken
1455 from storage management requirements.

1456
1457 Required Features: Basic Query
1458
1459 SELECT *
1460 FROM CIM_InstCreation
1461 WHERE SourceInstance ISA CIM_FCPort

1462 Using the lifecycle indication classes, this query would be stored in the Query string
1463 property of an instance of IndicationFilter and its delivery defined by an
1464 IndicationSubscription association to a ListenerDestination (please see the CIM Event
1465 Model [14]). An InstCreation notification would be delivered any time that an
1466 FCPort was created. The notification would consist of a single instance with a select-
1467 list as defined by the CIM_InstCreation class.

1468 2. As above, this query would be stored in the Query string property of an
1469 instance of IndicationFilter, and its delivery defined by an
1470 IndicationSubscription association. An InstModification notification would be
1471 delivered any time that an FCPort was modified and its first array property had
1472 changed. The notification would consist of a single instance with a select-list
1473 as defined by the CIM_InstModification class.

1474 Required Features: Basic Query, Embedded Properties
1475
1476 SELECT *
1477 FROM CIM_InstModification
1478 WHERE SourceInstance ISA CIM_FCPort
1479 AND PreviousInstance ISA CIM_FCPort
1480 AND SourceInstance.CIM_FCPort::OperationalStatus[0] <>
1481 PreviousInstance.CIM_FCPort::OperationalStatus[0]

CIM Query Language Specification

1482
1483 3. Send an Indication consisting of DetectionTime, SystemIPAddress,
1484 PerceivedSeverity, MonitorEventID and UPSttyPath properties, whenever
1485 MonitorEventID = 20 occurs on device /dev/ttyOp1.

1486 Required Features: Basic Query

1487
1488 SELECT DetectionTime,
1489 SystemIPAddress,
1490 PerceivedSeverity,
1491 MonitorEventID,
1492 UPSttyPath
1493 FROM Acme_UPSAlertIndication
1494 WHERE UPSttyPath = '/dev/ttyOp1'
1495 AND MonitorEventID = 20

1496
1497 4. Building on the previous example, in order to facilitate auditing and
1498 maintenance, the IT department requires that all Indications are "tagged" with
1499 an ID that identifies the filter condition that the Indication satisfied.

1500 Required Features: Basic Query, Extended Select List

1501
1502 SELECT DetectionTime,
1503 SystemIPAddress,
1504 PerceivedSeverity,
1505 MonitorEventID,
1506 UPSttyPath,
1507 'HP12345' AS FilterID
1508 FROM Acme_UPSAlertIndication
1509 WHERE UPSttyPath = '/dev/ttyOp1'
1510 AND MonitorEventID = 20
1511
1512

CIM Query Language Specification

- 1513 5. Continuing the example above, to ensure prompt processing of this type of
1514 Indication, define a CustomSeverity and set it to "Critical".

1515

1516 Required Features: Basic Query, Extended Select List

1517

```
1518 SELECT DetectionTime,  
1519         SystemIPAddress,  
1520         PerceivedSeverity,  
1521         'Critical' AS CustomSeverity,  
1522         MonitorEventID,  
1523         UPSttyPath,  
1524         'HP12345' AS FilterID  
1525 FROM Acme_UPSAlertIndication  
1526 WHERE UPSttyPath = '/dev/tty0p1'  
1527        AND MonitorEventID = 20  
1528  
1529
```

1528

1529

1530

6. Locate sick System/LogicalDevice combinations

1531

Required Features: Basic Query, Satisfies Array, Complex Join

1532

```
1533 SELECT s.Name, d.Name  
1534 FROM CIM_System s, CIM_SystemDevice sd, CIM_LogicalDevice d  
1535 WHERE OBJECTPATH(s) = sd.GroupComponent  
1536        AND OBJECTPATH(d) = sd.PartComponent  
1537        AND ANY i IN s.OperationalStatus[*] SATISFIES  
1538             (i = #'Non-Recoverable Error' OR i = #'Degraded')  
1539        AND ANY j in d.OperationalStaus[*] SATISFIES (j = #'Degraded' )  
1540
```

1540

1541

7. Locate creation of an export relationship for a FileShare

1542

Required Features: Basic Query

1543

```
1544 SELECT  
1545     InstanceOf(  
1546         SourceInstance.CIM_SharedElement::SameElement)  
1547         AS FileShare  
1548 FROM CIM_InstCreation  
1549 WHERE SourceInstance ISA CIM_SharedElement  
1550        AND InstanceOf(SourceInstance.CIM_SharedElement::SameElement)  
1551        ISA CIM_FileShare  
1552
```

1552

1553 7.3. Policy examples

1554

1555 For policy, identify a StoragePool that is low on space and allocate more space to it. In this
 1556 example, there are two underlying StoragePools to draw space from. The preferred one is a
 1557 free pool. The other is only used if the free pool can not satisfy the need.

1558

- 1559 1. This first query is used in a QueryCondition with QueryResultName set to
 1560 "PR_Needy". The query selects a StoragePool that is low on space.
 1561 Evaluation results in zero or more PR_Needy instances that are used by a
 1562 related MethodAction.

1563

1564 Required Features: Basic Query, Extended Select List, Complex Join, Embedded
 1565 Properties

1566

```

1567    SELECT OBJECTPATH(IM.SourceInstance) AS NeedySPPath
1568    FROM CIM_InstModification AS IM,
1569        CIM_PolicyRule AS PR,
1570        CIM_PolicySetAppliesToElement AS PSATE
1571    WHERE IM.SourceInstance ISA CIM_StoragePool
1572        AND PR.Name = 'AllocateMoreSpace'
1573        AND OBJECTPATH(PR) = PSATE.PolicySet
1574        AND OBJECTPATH(IM.SourceInstance) = PSATE.ManagedElement
1575        AND 100 * (IM.SourcInstance. CIM_StoragePool::RemainingManagedSpace /
1576            IM.SourcInstance. CIM_StoragePool::TotalManagedSpace) < 10
1577        AND IM.SourcInstance. CIM_StoragePool::RemainingManagedSpace <>
1578            IM.PreviousInstance. CIM_StoragePool::RemainingManagedSpace
1579
```

CIM Query Language Specification

1580 2. This next query is used in MethodAction to invoke a
1581 CreateOrModifyStoragePool method. It uses PR_Needy instances produced by
1582 the previous QueryCondition. The InstMethodCall results of the call are
1583 named by the property InstMethodCallName set to "PR_ModifySP".

1584 Required Features: Basic Query, Extended Select List, Complex Join

```
1585  
1586 SELECT OBJECTPATH(SCS) || '.CreateOrModifyStoragePool'  
1587         AS MethodName,  
1588        QCR.NeedySPPath AS Pool,  
1589        QCR.NeedySPPath.Size + (QCR.TotalManagedSpace / 10) AS Size,  
1590        OBJECTPATH(SP) AS InPools  
1591 FROM PR_Needy AS QCR,  
1592        CIM_ServiceAffectsElement AS SAE,  
1593        CIM_StorageConfigurationService AS SCS,  
1594        CIM_StoragePool AS SP,  
1595        CIM_AllocatedFromStoragePool AS AFSP  
1596 WHERE QCR.NeedySPPath = SAE.AffectedElement  
1597        AND OBJECTPATH(SCS) = SAE.AffectingElement  
1598        AND SP.ElementName = 'FreePool'  
1599        AND QCR.NeedySPPath = AFSP.Antecedent  
1600        AND OBJECTPATH(SP) = AFSP.Dependent
```

1601
1602 3. Use the results of the previous MethodActionResults as input to a second
1603 MethodAction to take action on an error. It also calls
1604 CreateOrModifyStoragePool.

1605
1606 Required Features: Basic Query, Extended Select List, Complex Join, Array Range,
1607 Embedded Properties

```
1608  
1609 SELECT MAR.MethodName,  
1610        MAR.MethodParameters.Pool,  
1611        MAR.MethodParameters.Size,  
1612        OBJECTPATH(SP) AS InPools  
1613 FROM PR_ModifySP MAR,  
1614        StoragePool SP,  
1615        AllocatedFromStoragePool AFSP  
1616 WHERE MAR.ResultValue <> '0'  
1617        AND SP.ElementName = 'SafetyPool'  
1618        AND MAR.MethodParameters ISA __MethodParameters  
1619        AND MAR.MethodParameters.__MethodParameters::Pool = AFSP.Antecedent  
1620        AND OBJECTPATH(SP) = AFSP.Dependent
```

1621 **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 <code>_KEY</code> is a complete instance path and that a property value of "*" indicates all properties + <code>_KEY</code> , <code>_CLASS</code> and <code>_VERSION</code> , 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: <ul style="list-style-type: none"> - 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. <code>_KEY</code> renamed to <code>_OBJECTPATH</code> . <code>_CLASS</code> renamed to <code>_CLASSPATH</code> . <code>_VERSION</code> 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

CIM Query Language Specification

<p>Do not include Array ANY/EVERY processing Make consistent with ABNF: IETF RFC 2234, http://www.ietf.org/rfc/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</p> <ul style="list-style-type: none">• Bag: set match• Ordered: element by element match to maxsize of both arrays.• Indexed: element by element match to maxsize of both arrays. <p>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-Criteria. Add semantics for ANY/EVERY/SATISFIES as proposed by Jeff. Select classname.* returns only properties defined in named class or its superclasses</p>
<p>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 prerequisite column Clarified errors Clarified string definition Removed Truth values from arithmetic expressions Clarified Count Clarified Regular Expression use by Basic and Regular Expression Like.</p>
<p>Version .13 – May 2004, Updates to cover review comments Simplify Basic Like</p>

CIM Query Language Specification

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

1622 **Appendix B: Dependencies and References**

1623 **Appendix B.1: *Dependencies***

- 1624 [1] DMTF [2004] Distributed Management Task Force: CIM Infrastructure
1625 Specification, DSP0004.pdf, version 2.3,
1626 http://www.dmtf.org/standards/published_documents.
- 1627 [2] DMTF [2004] Distributed Management Task Force: WBEM URI Specification,
1628 DSP0207.pdf, version 1.0,
1629 http://www.dmtf.org/standards/published_documents.
- 1630 [3] Augmented BNF for Syntax Specifications: ABNF, RFC 2234, Nov 1997,
1631 <http://www.faqs.org/rfcs/rfc2234.html>.
- 1632 [4] In this document, the term Unicode refers to the Universal Character Set (UCS),
1633 defined jointly by the Unicode Standard [5] and ISO/IEC 10646 [6].
- 1634 [5] The Unicode Consortium, "The Unicode Standard, Version 4.1", ISBN 0-321-
1635 18578-1, as updated from time to time by the publication of new minor versions.
1636 See <http://www.unicode.org/unicode/standard/versions> for the latest version and
1637 additional information on versions of the standard and of the Unicode Character
1638 Database.
- 1639 [6] ISO/IEC 10646:2003, "Information technology – Universal Multiple-Octet Coded
1640 Character Set (UCS)" as, from time to time, amended replaced by a new edition or
1641 expanded by the addition of new parts. See <http://www.iso.org> for the latest version.
- 1642 [7] W3C Working Draft "Character Model for the World Wide Web 1.0:
1643 Normalization", February 24, 2004, <http://www.w3.org/TR/charmod-norm/>
- 1644 [8] The Unicode Consortium, "Unicode Collation Algorithm (Unicode Technical
1645 Standard #10)". - as, from time to time, amended, replaced by a new edition or
1646 expanded by the addition of new parts. See <http://www.unicode.org/reports/tr10> for
1647 the latest version.
- 1648 [9] The Unicode Consortium, "Unicode Regular Expressions (Unicode Technical
1649 Standard #18)". - as, from time to time, amended, replaced by a new edition or
1650 expanded by the addition of new parts. See <http://www.unicode.org/reports/tr18> for
1651 the latest version.
- 1652 [10] See "XQuery 1.0 and XPath 2.0 Functions and Operators", section 7.6.1 Regular
1653 Expression Syntax. The latest version is at <http://www.w3.org/TR/xpath-functions>.

1654 **Appendix B.2: *References***

- 1655 [11] DMTF [2003] Distributed Management Task Force: CIM Operations over HTTP
1656 Specification, DSP0200, version 1.2,
1657 <http://www.dmtf.org/standards/documents/WBEM/DSP200.html>.

CIM Query Language Specification

- 1658 [12] ISO/IEC [1992] ISO/IEC 9075:1992, Database Language SQL- July 30, 1992. See
1659 <http://www.iso.org> for the latest version.
- 1660 [13] W3C [2001] World-Wide Web Consortium: XML-Query,
1661 <http://www.w3.org/XML/Query>.
- 1662 [14] DMTF [2002] Distributed Management Task Force: CIM Event Model V2.9
1663 (Final), http://www.dmtf.org/standards/cim/cim_schema_v29.
- 1664 [15] DMTF [2002] Distributed Management Task Force: CIM Policy Model V2.9
1665 (Final), http://www.dmtf.org/standards/cim/cim_schema_v29.
- 1666 [16] UTF-8, a transformation format of ISO 10646,
1667 <http://www.ietf.org/rfc/rfc3629.txt?number=3629>.
- 1668 [17] RFC 1034: DOMAIN NAMES - CONCEPTS AND FACILITIES,
1669 <http://www.ietf.org/rfc/rfc1034.txt?number=1034>.
- 1670 [18] RFC 1123: Requirements for Internet Hosts -- Application and Support,
1671 <http://www.ietf.org/rfc/rfc1123.txt?number=1123>.
- 1672 [19] DMTF [2002] Distributed Management Task Force: Specification for the
1673 Representation of CIM in XML, DSP0201, version 2.1
1674 <http://www.dmtf.org/standards/documents/WBEM/DSP201.html>.
- 1675 [20] UNICODE [2005] Unicode, Inc.: Unicode Technical Standard #10: Unicode
1676 Collation Algorithm, <http://www.unicode.org/unicode/reports/tr10/>
- 1677 [21] ISO/IEC 14651[2000], Information technology – International string ordering and
1678 comparison – Method for comparing character strings and description of the
1679 common template tailorable ordering
1680

1681 **Appendix C: Acknowledgements**

1682 The primary authors of this specification are George Ericson of EMC Corporation, Jeff
1683 Piazza of AppIQ, Inc. and Andrea Westerinen of Cisco Systems, Inc. The document is
1684 based on an original WBEM Query Language Specification submitted by Patrick
1685 Thompson of Microsoft.

1686 Significant editing contributions were made by Andreas Maier, Oliver Benke and others
1687 of IBM.

1688 **Appendix D: Regular Expression BNF**

1689 The Regular Expression grammar below uses Augmented BNF (ABNF) [3] with the
1690 following exceptions.

- 1691 1. Rules separated by a bar (|) represent choices. (Instead of using a slash (/) as
1692 defined in ABNF).
- 1693 2. Ranges of alphabetic characters or numeric values are specified using two
1694 periods (..) placed between the beginning and ending values of the range.
1695 (Instead of using the minus sign (-) as defined in ABNF).
- 1696 3. The rules defined in this syntax are meant to be assembled into a complete
1697 query by assuming whitespace characters between them, except where noted
1698 otherwise. (ABNF requires explicit specification of whitespace.)
- 1699 4. The comma (,) is used to explicitly designate concatenation of rules.
1700 (Instead of implicit concatenation of rules as specified by ABNF.)

1701 Note:

- 1702 1. ABNF is NOT case-sensitive.
- 1703 2. The rules above apply to the ABNF used here and NOT to the resultant Regular
1704 Expression used in Full or Basic Like. In particular, except where noted, white
1705 space is significant within the resultant Regular Expression.

1706
1707 The grammar is defined in two sections. The first is used to construct Regular
1708 Expressions used by the Basic Like feature. The second, Extended Regular Expressions
1709 is used to create Regular Expressions used by the Regular Expression Like feature. Both
1710 are defined as follows:
1711

1712 **Appendix D.1: *Basic Like Regular Expressions***

1713 Basic Like Regular Expressions is a subset of the XQuery Regular Expression syntax as
1714 defined in Regular Expressions [10].

1715
1716 Note: Basic Like Regular Expressions complies with levels RL1.1 and RL 1.7 of
1717 Unicode Regular Expressions Level 1 [9], which is a subset of the XQuery Regular
1718 Expression [10] compliance to Unicode Regular Expressions Level 1 [9].

1719
1720 **blre-ordinary-char= UNICODE-CHAR**
1721 A character, other than a metacharacter excluded from the Char
1722 production of XQuery Regular Expressions [10].

1723	
1724	blre-escaped-char = char-escape SingleCharEsc
1725	An escaped character. The char-escape is defined in the String Literals
1726	section. The SingleCharEsc is defined in XQuery Regular Expressions
1727	[10]. The "/u" and "/U" syntax of char-escape replaces the character
1728	reference syntax defined in XQuery Regular Expressions [10].
1729	
1730	Note: the char-escape includes escape sequences that may not be
1731	supported by XQuery. The CQL processor may need to convert these
1732	escape sequences to a form that is compatible with XQuery.
1733	
1734	blre-single-char = "." blre-ordinary-char blre-escaped-char
1735	Single character regular expression. The '.' meta-character matches any
1736	character except the newline character (\u000A).
1737	
1738	blre-multi-char = blre-single-char,"*"
1739	Matches multiple occurrences of a single character
1740	
1741	blre-expression = *(blre-single-char blre-multi-char)
1742	Basic Like regular expression

1743 **Appendix D.2: Full Like Extended Regular** 1744 **Expressions**

1745 Full Like Regular Expressions is conformant with the XQuery Regular Expression syntax
 1746 as defined in Regular Expressions [10], with the following exceptions:

- 1747 1) The Unicode characters allowed in the expression are defined by UNICODE-
 1748 CHAR in the Query Language BNF section.
- 1749
- 1750 2) The escape sequences of char-escape in the String Literals section may be used
 1751 in addition to the escape sequences in SingleCharEsc in XQuery Regular
 1752 Expressions [10]. The "/u" and "/U" syntax of char-escape replaces the character
 1753 reference syntax defined in XQuery Regular Expressions [10]. Note: the char-
 1754 escape includes escape sequences that may not be supported by XQuery. The
 1755 CQL processor may need to convert these escape sequences to a form that is
 1756 compatible with XQuery.
- 1757
- 1758 3) None of the flags defined in section 7.6.1.1 of XQuery Regular Expressions
 1759 [10] are supported, and the expression matching behaves as if all the flags have
 1760 the default values.

1761 **Appendix E: Datetime Operations and BNF**

1762 The operations on datetime and the datetime BNF described in this appendix will
1763 ultimately be incorporated into some other DMTF specification and references to this
1764 appendix should be updated to refer to the incorporating specification.

1765 **Appendix E.1: *Datetime Operations***

1766 The following operations are defined on datetime types:

1767 1. Arithmetic operations:

- 1768 § Adding or subtracting an interval to or from an interval results in an
1769 interval
- 1770 § Adding or subtracting an interval to or from a timestamp results in a
1771 timestamp
- 1772 § Subtracting a timestamp from a timestamp results in an interval
- 1773 § Multiplying an interval with a numeric or vice versa results in an
1774 interval
- 1775 § Dividing an interval by a numeric results in an interval

1776 Other arithmetic operations are NOT defined.

1777 2. Comparison operations:

- 1778 § Testing for equality or inequality of two timestamps or two intervals
1779 results in a boolean
- 1780 § Testing for the ordering relation (<, <=, >, >=) of two timestamps or
1781 two intervals results in a boolean

1782 Other comparison operations are NOT defined.

1783

1784 Note that comparison between a timestamp and an interval, and vice versa, is not
1785 defined.

1786

1787 Specifications using the definition of these operations (for instance, query languages)
1788 SHOULD define how undefined operations are handled.

1789

1790 Any operations on datetime types in an expression MUST be handled as if the following
1791 sequential steps were performed:

1792

- 1793 1. Each datetime value is converted into a range of microsecond values, as
1794 follows:
 - 1795 • The lower bound of the range is calculated from the datetime
1796 value, with any asterisks replaced by their minimum value,

CIM Query Language Specification

1797 • the upper bound of the range is calculated from the datetime
1798 value, with any asterisks replaced by their maximum value,
1799 • the basis value for timestamps is the oldest valid value (i.e. 0
1800 microseconds corresponds to 00:00.000000 in the timezone
1801 with datetime offset +720, on January 1st in the year 1 BCE,
1802 using the proleptic Gregorian calendar). Note that this
1803 definition implicitly performs timestamp normalization. Note
1804 that 1 BCE is the year before 1 CE.
1805 2. The expression is evaluated, using the following rules for any datetime
1806 ranges:
1807
1808 Definitions:
1809 T(x, y) is the microsecond range for a timestamp with the
1810 lower bound x and the upper bound y
1811 I(x, y) is the microsecond range for an interval with the lower
1812 bound x and the upper bound y
1813 D(x, y) is the microsecond range for a datetime (timestamp or
1814 interval) with the lower bound x and the upper bound y
1815
1816 Rules:
1817 I(a, b) + I(c, d) := I(a+c, b+d)
1818 I(a, b) - I(c, d) := I(a-d, b-c)
1819 T(a, b) + I(c, d) := T(a+c, b+d)
1820 T(a, b) - I(c, d) := T(a-d, b-c)
1821 T(a, b) - T(c, d) := I(a-d, b-c)
1822 I(a, b) * c := I(a*c, b*c)
1823 I(a, b) / c := I(a/c, b/c)
1824
1825 D(a, b) < D(c, d) := true if b < c, false
1826 if a >= d, otherwise NULL (uncertain)
1827 D(a, b) <= D(c, d) := true if b <= c,
1828 false if a > d, otherwise NULL (uncertain)
1829 D(a, b) > D(c, d) := true if a > d, false
1830 if b <= c, otherwise NULL (uncertain)
1831 D(a, b) >= D(c, d) := true if a >= d,
1832 false if b < c, otherwise NULL (uncertain)
1833 D(a, b) = D(c, d) := true if a = b = c =
1834 d, false if b < c OR a > d, otherwise NULL
1835 (uncertain)
1836 D(a, b) <> D(c, d) := true if b < c OR a >
1837 d, false if a = b = c = d, otherwise NULL
1838 (uncertain)
1839
1840 These rules follow the well known mathematical interval arithmetic. An
1841 informational link to a definition of mathematical interval arithmetic is
1842 http://en.wikipedia.org/wiki/Interval_arithmetic.

CIM Query Language Specification

1843
1844 Note that mathematical interval arithmetic is commutative and associative
1845 for addition and multiplication, like ordinary arithmetic.
1846
1847 Note that mathematical interval arithmetic mandates the use of three-state
1848 logic for the result of comparison operations, using a special value called
1849 "uncertain" to represent that a decision cannot be made. The special value
1850 of "uncertain" is mapped to the NULL value in datetime comparison
1851 operations.
1852 3. Overflow and underflow condition checking is performed on the result of
1853 the expression, as follows:
1854

- For timestamp results:
1855
 - A timestamp older than the oldest valid value in the timezone
1856 of the result produces an arithmetic underflow condition
 - A timestamp newer than the newest valid value in the timezone
1857 of the result produces an arithmetic overflow condition
- For interval results:
1858
 - A negative interval produces an arithmetic underflow condition
 - A positive interval greater than the largest valid value produces
1859 an arithmetic overflow condition

1864 Specifications using the definition of these operations (for instance, query languages)
1865 SHOULD define how these conditions are handled.
1866

- 1867 4. If the result of the expression is again a datetime type, the microsecond
1868 range gets converted into a valid datetime value such that the set of
1869 asterisks (if any) determines a range that matches the actual result range,
1870 or encloses it as closely as possible. The GMT timezone MUST be used
1871 for any timestamp results.

1872
1873 Note that for most fields, asterisks can be used only with the granularity of
1874 the entire field.
1875

1876 Examples:

1877
1878 "20051003110000.000000+000" + "00000000002233.000000:000"
1879 evaluates to "20051003112233.000000+000"
1880 "20051003110000.*****+000" + "00000000002233.000000:000"
1881 evaluates to "20051003112233.*****+000"
1882 "20051003110000.*****+000" + "00000000002233.00000*:000"
1883 evaluates to "200510031122**.******+000"
1884 "20051003110000.*****+000" + "00000000002233.*****:000"
1885 evaluates to "200510031122**.******+000"
1886 "20051003110000.*****+000" + "00000000005959.*****:000"
1887 evaluates to "20051003*****.******+000"
1888 "20051003110000.*****+000" + "000000000022**.******:000"
1889 evaluates to "2005100311****.******+000"
1890 "20051003112233.000000+000" - "00000000002233.000000:000"
1891 evaluates to "20051003110000.000000+000"
1892 "20051003112233.*****+000" - "00000000002233.000000:000"

CIM Query Language Specification

```
1893         evaluates to "20051003110000.*****+000"
1894 "20051003112233.*****+000" - "00000000002233.00000*:000"
1895         evaluates to "20051003110000.*****+000"
1896 "20051003112233.*****+000" - "00000000002232.*****:000"
1897         evaluates to "200510031100**.*****+000"
1898 "20051003112233.*****+000" - "00000000002233.*****:000"
1899         evaluates to "20051003*****.*****+000"
1900 "20051003060000.000000-300" + "00000000002233.000000:000"
1901         evaluates to "20051003112233.000000+000"
1902 "20051003060000.*****-300" + "00000000002233.000000:000"
1903         evaluates to "20051003112233.*****+000"
1904 "000000000011**.*****:000" * 60
1905         evaluates to "0000000011****.*****:000"
1906 60 times adding up "000000000011**.*****:000"
1907         evaluates to "0000000011****.*****:000"
1908 "20051003112233.000000+000" = "20051003112233.000000+000"
1909         evaluates to true
1910 "20051003122233.000000+060" = "20051003112233.000000+000"
1911         evaluates to true
1912 "20051003112233.*****+000" = "20051003112233.*****+000"
1913         evaluates to NULL (uncertain)
1914 "20051003112233.*****+000" = "200510031122**.*****+000"
1915         evaluates to NULL (uncertain)
1916 "20051003112233.*****+000" = "20051003112234.*****+000"
1917         evaluates to false
1918 "20051003112233.*****+000" < "20051003112234.*****+000"
1919         evaluates to true
1920 "20051003112233.5*****+000" < "20051003112233.*****+000"
1921         evaluates to NULL (uncertain)
```

1922 **Appendix E.2: *Datetime BNF***

1923 The URI grammar below uses Augmented BNF (ABNF) [3] with the following
1924 exceptions.

- 1925 1. Rules separated by a bar (|) represent choices. (Instead of using a slash (/) as
1926 defined in ABNF).
- 1927 2. Ranges of alphabetic characters or numeric values are specified using two
1928 periods (..) placed between the beginning and ending values of the range.
1929 (Instead of using the minus sign (-) as defined in ABNF).
- 1930 3. The rules defined in this syntax are meant to be assembled into a complete
1931 query by assuming whitespace characters between them, except where noted
1932 otherwise. (ABNF requires explicit specification of whitespace.)
- 1933 4. The comma (,) is used to explicitly designate concatenation of rules.
1934 (Instead of implicit concatenation of rules as specified by ABNF.)

1935 Note: ABNF is NOT case-sensitive.

1936

1937 The grammar is defined as follows:

```
1938 dt-decimal-digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

1939

CIM Query Language Specification

1940	dt-single-quote = ""
1941	
1942	dt-two-time-digits = (2*2(dt-decimal-digit)) ("**")
1943	
1944	dt-microsecond-digits = 6*6(dt-decimal-digit)
1945	5*5(dt-decimal-digit), ("*")
1946	4*4(dt-decimal-digit), ("**")
1947	3*3(dt-decimal-digit), ("***")
1948	2*2(dt-decimal-digit), ("****")
1949	1*1(dt-decimal-digit), ("*****")
1950	("*****")
1951	See the CIM Infrastructure Specification [1] for a detailed description of the use of interval-specification.
1952	
1953	dt-timestamp-specification = dt-single-quote,
1954	(
1955	((4*4(dt-decimal-digits) "*****"), "*****",
1956	".", ("*****"), ("+" "-"), 3*3(dt-decimal-digit))
1957	years: A timestamp with the year field set to 0000 is interpreted as the
1958	year 1 BCE. A year field set to 0001 is interpreted as the year 1 CE.
1959	(6*6(dt-decimal-digits), dt-two-time-digits, "*****",
1960	".", ("*****"), ("+" "-"), 3*3(dt-decimal-digit))
1961	months
1962	(8*8(dt-decimal-digits), dt-two-time-digits, "****",
1963	".", ("*****"), ("+" "-"), 3*3(dt-decimal-digit))
1964	days
1965	(10*10(dt-decimal-digits), dt-two-time-digits, "***",
1966	".", ("*****"), ("+" "-"), 3*3(dt-decimal-digit))
1967	minutes
1968	(12*12(dt-decimal-digits), dt-two-time-digits,
1969	".", ("*****"), ("+" "-"), 3*3(dt-decimal-digit))
1970	seconds
1971	(14*14(dt-decimal-digits),
1972	".", (dt-microsecond-digits), ("+" "-"), 3*3(dt-decimal-digit))
1973	microseconds
1974), dt-single-quote
1975	See the CIM Infrastructure Specification [11] for a detailed description of the use of interval-specification.
1976	

CIM Query Language Specification

1977	dt-interval-specification = dt-single-quote,
1978	((14*14(")", ".", ("*****"), (":"), 3*3(dt-decimal-digit))
1979	nothing
1980	(8*8(dt-decimal-digit) ("*****")), ("*****"),
1981	(".", ("*****"), (":"), 3*3(dt-decimal-digit))
1982	days
1983	(8*8(dt-decimal-digits), dt-two-time-digits, "****",
1984	(".", ("*****"), (":"), 3*3(dt-decimal-digit))
1985	hours
1986	(10*10(dt-decimal-digits), dt-two-time-digits, "***",
1987	(".", ("*****"), (":"), 3*3(dt-decimal-digit))
1988	minutes
1989	(12*12(dt-decimal-digits), dt-two-time-digits,
1990	(".", ("*****"), (":"), 3*3(dt-decimal-digit))
1991	seconds
1992	(14*14(dt-decimal-digits),
1993	(".", (dt-microsecond-digits), (":"), 3*3(dt-decimal-digit))
1994	microseconds
1995), dt-single-quote
1996	See the CIM Infrastructure Specification [11] for a detailed description
1997	of the use of interval-specification.