On Precision-Based Timing Comparisons in CQL

# Overview

This document describes a proposed approach to supporting precision-based comparisons in CQL, especially in the presence of dates and times specified to varying levels of precision. The proposed approach detailed here is based on the timing approach developed by Rute Martins of the Joint Commission as part of ongoing conversations with the eMIG working group.

The core issue being addressed is the proper handling of temporal comparisons in the presence of varying degrees of certainty about the time at which events occur. For example, if a measure is looking for the occurrence of a particular procedure within two years of the measurement start date, but an EHR records that a qualifying procedure occurred in a given year, not the month or day of the occurrence. In this scenario, the EHR must be allowed to provide as much information as it accurately has, but must not be required to provide information that is not known, and the record will therefore contain a DateTime value, but specified only to the year precision. If the semantics for timing comparison do not take this possibility into account, the resulting comparisons may yield incorrect results.

The underlying principle of the approach is to ensure that the best answer can be given with the available data. To this end, the results of any comparison may return true, false, or unknown, depending on whether there is enough information to make an accurate determination. The general idea is that a date/time value that is only known to some specific precision, say years, should be treated as an uncertainty over the range at the comparison precision. For example, given the following comparison:

Date(2012) < Date(2014, 2, 15)

The result should be true because even though the month and day of the first date are unknown, the year, 2012, is known to be less than the year of the second date, 2014. By contrast:

Date(2015) < Date(2014, 2, 15)

The result in this case is false because the year, 2015, is not less than the year of the second date. And finally:

Date(2014) < Date(2014, 2, 15)

The result in this case is unknown (null) because the first date could be any date within the year 2014, so it could be less than the second date, but it could be greater.

# Date/Time Representation

The first step to ensuring these semantics are properly supported is to ensure that date/time values can properly represent values with less than complete precision. To this end, date/time values within CQL must support not only the ability to represent the value of each component of a date/time, but whether or not each component is present. Note that to ensure meaningful data, at least one component must be specified, and no component may be specified with a lower granularity than any unspecified component. For example, the following date/time values are all valid:

Date(2014)

Date(2014, 5)

Date(2014, 5, 15, 16, 30)

Whereas the following are all invalid:

Date(null)

Date(null, 5)

Date(2014, 5, null, 16, 30)

Effectively, this representation considers date/time values to be tuples with attributes for each component of the date/time.

# Equality Comparison

The next step is to define equality comparison for date/time values. In CQL, equality of tuples is based on equality comparison of each attribute of the tuple. This means the comparison:

A = B

Would be equivalent to the comparison of each extracted component:

year of A = year of B

and month of A = month of B

and day of A = day of B

and hour of A = hour of B

and minute of A = minute of B

and second of A = second of B

and millisecond of A = millisecond of B

These semantics give the expected behavior that two date/time values are equivalent if and only if they are known to have the same value for each component. If any component of either comparand is null, the result of the comparison is null.

However, when dealing with missing information, authors often want to know whether the values that are present are the same. To this end, CQL supports a *same as* operator:

A same as B

This operator is defined to be equivalent to the following expansion:

(A is null and B is null) or (A = B)

In other words, the operator returns true if both comparands are null, or if the comparands are equal in the strict sense defined above. For tuple values, the *same as* operator is defined in terms of *same as* comparison of each attribute of the tuples. Using this definition, for date/time values the comparison:

A same as B

Is equivalent to the following expansion:

year of A same as year of B

and month of A same as month of B

and day of A same as day of B

and hour of A same as hour of B

and minute of A same as minute of B

and second of A same as second of B

and millisecond of A same as millisecond of B

In this way, two date/time values will be *the same* if they have the same values specified for each component. For example, the following comparison:

Date(2014, 15, 2) same as Date(2014, 15, 2)

Returns true because both dates are specified to the same level of precision, and have the same values for each of those specified levels. By contrast, the following comparison:

Date(2014, 15) same as Date(2014, 15, 2)

Returns null because the dates are specified to different levels of precision.

In addition, CQL supports the ability to perform this same as comparison at different levels of precision by including the basis for the comparison in the operator. For example:

A same day as B

Indicates that the comparison should be performed to the day component, and is equivalent to the following expansion:

year of A same as year of B

and month of A same as month of B

and day of A same as day of B

Note that this is a very different operation than just extracting the day component of each value and comparing; it involves comparison of each component up to and including the specified level of precision.

# Relative Comparison

The next step involves defining the relative comparison semantics for date/time values. Similar to the way equality is defined, relative comparison for date/time values is defined in terms of comparisons of the components:

A > B

This comparison is equivalent to the following expansion:

year of A > year of B

or (year of A = year of B and month of A > month of B)

or (year of A = year of B and month of A = month of B and day of A > day of B)

or (year of A = year of B and month of A = month of B and day of A = day of B

and hour of A > hour of B)

or (year of A = year of B and month of A = month of B and day of A = day of B

and hour of A = hour of B and minute of A > minute of B)

or (year of A = year of B and month of A = month of B and day of A = day of B

and hour of A = hour of B and minute of A = minute of B and second of A > second of B)

or (year of A = year of B and month of A = month of B and day of A = day of B

and hour of A = hour of B and minute of A = minute of B and second of A = second of B

and millisecond of A > millisecond of B)

Note that because *or* is involved, this expansion produces the intuitively expected semantics for comparisons of date/time values of different granularities. For example:

Date(2012) > Date(2014, 2, 15) // Returns false

Date(2014) > Date(2014, 2, 15) // Returns null

Date(2016) > Date(2014, 2, 15) // Returns true

Next, the semantics for relative comparisons involving equality are defined by combining equality and relative comparison:

A >= B

This comparison is equivalent to:

A = B or A > B

Which again provides the intuitively expected semantics.

For the *same as* operator, CQL allows the prefixes *at least* and *at most* to be used:

A at least same day as B

A at most same day as B

These expressions are equivalent to the following expansions:

A same day as B or A > B

A same day as B or A < B

These comparison semantics for date/time values provide the foundation for consistent handling of temporal data developed in the following sections.

# Timezone Support

For simplicity, the discussion so far has omitted consideration of the time zone component of each date/time value. To support time zones, however, CQL date/time values each have a time zone that must be supplied. The time zone component does not affect the precision-based comparison semantics, but rather conversion to a common time zone is expected to be performed implicitly by an implementation prior to performing these comparisons. This ensures consistent and accurate handling of time zone information, without needing to expose the additional complexity to the CQL author.

Note that when a date/time value is generated within CQL, the time zone component can either be supplied directly, or defaulted by the system. If it is defaulted, the time zone is based on the same time zone semantics used for the date/time functions *Now()* and *Today()*, i.e. the time zone is based on the time zone of the request. In clinical decision support contexts, this is the time zone of the client making the decision support evaluation request. In quality measurement contexts, this is the time zone of the system performing the measurement evaluation. This approach, combined with the requirement that date/time comparisons be converted to this common request-based time zone ensures that date/time handling is consistent and based on the time-zone of the originating data. Note also that this requires that the time zone component be supplied correctly for all originating data produced by the clinical system.

# Determining Duration

To determine the duration between two date/time values, CQL supports a *between* operator for each date/time component. For example:

days between A and B

This expression returns the number of day boundaries crossed between A and B. If A is before B, the result will be a positive integer. If A is after B, the result will be a negative integer. And if A is the same day as B, the result will be zero.

However, to support the case where one or the other comparand in the duration operation does not specify components to the level of precision being determined, the between operator does not return a strict integer, it returns an *uncertainty*, which is defined as a range of values, similar to an interval. For example:

days between Date(2014, 1, 15) and Date(2014, 2)

The number of days between these two dates cannot be determined reliably, but a definite range of possible values can be determined. The lower bound of that range is found by determining the duration between the maximum possible value of the first comparand and the minimum possible value of the second comparand; and the upper bound is determined using the minimum possible value of the first comparand and the maximum possible value of the second:

days between Date(2014, 1, 15) and Date(2014, 2, 1) // 17 days

days between Date(2014, 1, 15) and Date(2014, 2, 28) // 44 days

Intuitively, what this means is that the number of days between January 15th, 2014 and some date in February, 2014, is no less than 17 days, but no more than 44. By incorporating this information into an uncertainty, CQL can support the intuitively expected semantics when performing timing comparisons. For example:

days between Date(2014, 1, 15) and Date(2014, 2) > 2

This comparison returns true, because the lower bound of the uncertainty, 17, is greater than 2, so no matter what the actual date of the second comparand, it would always be at least 17 days. By contrast:

days between Date(2014, 1, 15) and Date(2014, 2) > 50

This comparison returns false, because the upper bound of the uncertainty, 44, is less than 50, so no matter what the actual date of the second comparand, it would always be at most 44 days. And finally:

days between Date(2014, 1, 15) and Date(2014, 2) > 20

This comparison returns unknown (null), because the value being compared, 20, falls within the uncertainty, so no determination can be reliably made.

In order to support these semantics, CQL supports uncertainty as a new first-class data type including complete operator semantics. However, the language also defines implicit conversions between uncertainties and point values so that in general, CQL authors need not be aware that the underlying semantics involve uncertainty. In this way, the intuitively expected semantics for timing comparisons are achieved, without introducing unnecessary additional complexity to the high-level CQL syntax as understood by authors.

# Timing Phrases

Using the foundational elements described in the previous sections, the semantics for the various CQL timing phrases can now be described in detail. The general approach for each timing phrase is to translate it to an equivalent representation in terms of either a direct comparison, or a comparison involving a duration calculation.

## Same As

The *same as* timing phrase is simply defined to be equivalent to a *same as* comparison of the date/time values involved:

A starts same day as start B

This expression is equivalent to:

start of A same day as start of B

Similarly for the *at least* and *at most* comparisons:

A starts at least same day as start B

A starts at most same day as start B

These expressions are equivalent to:

start of A at least same day as start of B

start of A at most same day as start of B

## Before/After

The basic *before* and *after* timing phrases are defined in terms of direct relative comparison:

A starts before start B

A starts after start B

These expressions are equivalent to:

start of A < start of B

start of A > start of B

If the phrase involves a duration, it is defined in terms of a duration calculation and a comparison of the result:

A starts 3 days before start B

A starts 3 days after start B

These expressions are equivalent to:

days between start of A and start of B = 3

days between start of A and start of B = -3

If the phrase involves *at least* or *at most*, a relative equality comparison is used:

A starts at least 3 days before start B

A starts at most 3 days after start B

These expressions are equivalent to:

days between start of A and start of B >= -3

days between start of A and start of B <= 3

## Within

The *within* timing phrase is defined in terms of a duration calculation and an interval test of the result:

A starts within 3 days of start B

This expression is equivalent to:

days between start of A and start of B in [-3, 3]

# Interval Comparisons

In general, interval comparisons are already defined in terms of the fundamental comparison operators (=, >, <, >=, <=) so the semantics of the interval comparisons follow directly from these extended semantics.

# Uncertainty

To support the semantics described in this approach, CQL defines a new category of values called an *uncertainty*. In form, an uncertainty is similar to an *interval*, and can be defined on any *point type* that can be used to define an *interval*. An uncertainty can appear in an expression wherever a value of the point type of the uncertainty can appear. For example, in the expression:

A + B

The values of A, B, or both may be uncertainties. Intuitively, the appearance of an uncertainty in an expression means *some value between X and Y*. For simplicity, uncertainties are only allowed to be defined with *inclusive* or *closed* boundaries. The syntax for an uncertainty is:

uncertainty[17, 44]

The above expression can be read *some value between 17 and 44*.

Note that this representation of uncertainty assumes a continuous probability distribution along the range. In other words, the assumption is that there is no information about how likely the value is to be any particular value within the range.

Note that the special case of an uncertainty of width zero:

uncertainty[2, 2]

Must be treated as equivalent to the point value, *2* in this case.

## Comparison Operators

Comparison semantics for uncertainty are defined to result in the intuitively expected behavior. For example, when comparing two uncertainties for equality:

uncertainty[17, 44] = uncertainty[17, 44]

The above expression results in *null*, because the meaning of the statement is actually:

Is *some value between 17 and 44* equal to *some value between 17 and 44*?

And the intuitively correct answer to that question is, *I don’t know*. However, for cases where there is no overlap between the uncertainties, the result is *false*:

uncertainty[17, 44] = uncertainty[2, 12]

Again, the intended semantics of this statement are:

Is *some value between 17 and 44* equal to *some value between 2 and 12*?

And the correct answer is, *No*, because there is no possible value in either uncertainty range that could evaluate to *true*.

In the special case of equality comparisons of two uncertainties of width zero, the result is true:

uncertainty[2, 2] = uncertainty[2, 2]

This expression can be read:

Is *some value between 2 and 2* equal to *some value between 2 and 2*?

And the correct answer is, *Yes*. (Given that the point type is *integer*, rather than *real*).

For relative comparisons, if the uncertainty ranges of the comparands overlap in any way, the result is unknown. Otherwise, the result is based on comparison of the relevant boundaries. For example:

uncertainty[17, 44] > uncertainty[2, 12] // returns true

uncertainty[17, 44] > uncertainty[2, 20] // returns unknown

uncertainty[17, 44] > uncertainty[17, 44] // returns unknown

uncertainty[17, 44] > uncertainty[20, 50] // returns unknown

uncertainty[17, 44] > uncertainty[45, 50] // returns false

## Arithmetic Operators

In addition to comparison operators, the basic arithmetic operators are defined for uncertainty, again based on the intuitively expected semantics. For example:

uncertainty[17, 44] + uncertainty[5, 10] // returns uncertainty[22, 54]

The above expression can be read:

*some value between 17 and 44* + *some value between 5 and 10*

The result of this calculation simply adds the respective boundaries to determine what the range of possible values of this calculation would be, in this case *some value between 22 and 54*.

Similarly for multiplication:

uncertainty[17, 44] \* uncertainty[2, 4] // returns uncertainty[34, 176]

The result of this calculation multiplies the boundaries of the uncertainties to determine the range of possible values for the result, in this case *some value between 34 and 176*.

## Additional Operators

Semantics for additional operators on uncertainty can be defined in terms of these foundational operators. For example:

uncertainty[17, 44] in interval[-3, 3] // returns false

uncertainty[17, 44] in interval[1, 100] // returns true

uncertainty[17, 44] in interval[25, 50] // returns unknown

These semantics follow directly from the rules for comparison involving uncertainty defined above.

## Implicit Conversion

The final step to achieving the intended semantics for precision-based timing comparisons in CQL is to allow for implicit conversion between uncertainties and point-values. This means that anywhere an uncertainty is involved in an operation with a point-value, the point-value will be implicitly converted to an uncertainty of width zero and the uncertainty semantics defined above are then used to perform the calculation. For example:

uncertainty[17, 44] > 2

The point-value of *2* in this example is implicitly converted to an uncertainty of width zero:

uncertainty[17, 44] > uncertainty[2, 2]

This implicit conversion means that in general, the notion of uncertainty will not be visible in the resulting syntax of CQL. For example:

days between Date(2014, 1, 15) and Date(2014, 2) > 2

Even though determining the correct answer to this question involves the use of uncertainty, it is implicit in the way the operations are defined, and does not surface to the CQL authors.

# Conclusion

Extending CQL to support a simplistic notion of uncertainty that can be implicitly used to determine the most intuitively correct answer to temporal comparisons involving precision and unknown components will result in more accurate measurement and decision support without unnecessarily increasing the complexity of the language used by artifact authors.

A summary of the proposed changes to the CQL specification balloted in September 2014 follows:

1. Extend the representation of date/time values to support varying levels of precision.
2. Extend semantics of date/time comparison to account for unspecified precisions.
3. Extend the *same as* operator to support more intuitive comparison semantics in the presence of unspecified precision.
4. Within ELM, introduce the notion of *uncertainty* and define the relevant comparison operators.
5. Within ELM, Extend the semantics of the *temporal duration* operator to result in uncertainty in the presence of unspecified precision.
6. Extend the CQL to ELM translator to support implicit conversion of point-values to equivalent zero-width uncertainties.

The approach outlined above means that there is no need to change CQL syntax, or surface the underlying notion of uncertainty to CQL authors. The behavior would be entirely handled by extensions to the ELM, the CQL-ELM translator, and the target ELM implementations.