

CUBE 4.1.4 – Cube Derived Metrics

Usage and syntax documentation

February 2, 2013

The Scalasca Development Team
scalasca@fz-juelich.de

Copyright

Copyright © 1998–2012 Forschungszentrum Jülich GmbH, Germany

Copyright © 2003–2008 University of Tennessee, Knoxville, USA

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the names of Forschungszentrum Jülich GmbH or the University of Tennessee, Knoxville, nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Contents

Copyright	iii
1 Introduction into Cube Derived Metrics	1
1.1 Introduction	1
1.2 Creation of a derived metric in Cube	1
1.3 The three kinds of derived metrics	2
2 Syntax of CubePL	5
2.1 Syntax step by step	5
2.2 Grammar of CubePL	12
3 Reference of CubePL	13
3.1 Supported calls in CubePL	13
4 Examples of CubePL expressions	15
4.1 Simple Examples	15
4.2 Complex Examples	15

1 Introduction into Cube Derived Metrics

1.1 Introduction

With Cube 4.1.0 derived metrics are introduced and provide a powerful tool for performance data processing.

Unlike normal metrics of the cube, derived metrics do not store any data inside of the cube report, but derive their value according to an expression, formulated using CubePL (Cube Processing Language) syntax ([‘Syntax of CubePL’](#)).

Cube supports different kinds ([‘The three kinds of derived metrics’](#)) of derived metrics. Derived metric can be a child metric of any metric of cube or other derived metrics.

Derived metrics are naturally metrics with the value `DOUBLE`

1.2 Creation of a derived metric in Cube

Derived metrics play the very same role as a normal metric in cube, therefore one proceeds with creation of derived metrics as usual. The only difference is the specification of the CubePL expression.

1.2.1 Using the C writer

To create a derived metric in a C program, one calls `cube_metric_set_expression(...)` subsequently to `cube_def_met(...)`. Here is a code example:

```
cube_metric * met = cube_def_met(cube,
                                "Derived metric",
                                "derivedmetric1",
                                "DOUBLE",
                                "sec",
                                "",
                                "https://doc.web.org/metrics.html#derived",
                                "Average execution time",
                                NULL,
                                CUBE_METRIC_PREDERIVED_EXCLUSIVE
```

```
);  
cube_metric_set_expression(met, "metric::time(i)/metric::visits(e)");
```

1.2.2 Using the C++ library

To create a derived metric in C++ programs, one specifies the expression as the last parameter of the metric definition. This parameter can be omitted, in this case the derived metric returns always 0.

Here is a code example:

```
Metric * met = cube.def_met(  
    "Derived metric",  
    "derivedmetric1",  
    "DOUBLE",  
    "sec",  
    "",  
    "https://doc.web.org/metrics.html#derived",  
    "Average execution time",  
    NULL,  
    CUBE_METRIC_PREDERIVED_EXCLUSIVE,  
    "metric::time(i)/metric::visits(e)"  
);
```

CUBE_METRIC_PREDERIVED_EXCLUSIVE defines the kind of the derived metric (see [The three kinds of derived metrics](#) below).

Note, that the CubePL expression should be valid at the time of execution. It means, if there is a reference to metric, this metric should be defined before.

1.3 The three kinds of derived metrics

Cube provides an API to calculate different inclusive and exclusive values aggregated over different dimensions.

In the context of derived metrics it is sometimes useful, when evaluation of the expression is done after the aggregation of the metrics in the expression, sometimes it is needed to aggregate values of the derived metrics.

Therefor Cube provides three kinds of derived metrics:

1. **Prederived exclusive metric** - metric, which value of which expression is observed as "being stored" inside of cube report and it behaves as a usual exclusive (along call tree) metric.

One specifies this type of metric using constant

```
CUBE_METRIC_PREDERIVED_EXCLUSIVE
```

2. **Prederived inclusive metric** - metric, which value of which expression is observed as "being stored" inside of cube report and it behaves as a usual inclusive (along call tree) metric.

One specifies this type of metric using constant

`CUBE_METRIC_PREDERIVED_INCLUSIVE`

3. **Postderived metric** - metric, which expression evaluated only after metrics, which were used inside of the expression, get aggregated according to the aggregation context (along system tree or along call tree).

If expression doesn't contain references to another metrics, no aggregation is done. In this case the expression gets calculated and its value returned.

One specifies this type of metric using constant

`CUBE_METRIC_POSTDERIVE`

2 Syntax of CubePL

2.1 Syntax step by step

This chapter introduced the CubePL syntax of the most common programming structures.

2.1.1 Expressions

2.1.1.1 Constants

Any constant expression is an `expression`. eg. `123.0`, `"someString"`

2.1.1.2 Arithmetical Expressions

Common mathematical notation of an arithmetical expression is valid in CubePL. Any arithmetical expression is an `expression`. eg. `123.0 + 23.6`, `sin(23)`

2.1.1.3 Boolean Expressions

Common literal notation of a boolean expression is valid in CubePL. Any boolean expression is an `expression`.

```
[expression] or|and|xor [expression]
```

or

```
not [expression]
```

Any `expression` can be a term of a boolean expression. Non zero value is observed as `TRUE`, zero value is `FALSE`.

2.1.1.4 Function call expressions

Function calls have the following syntax:

```
name ( [expression ] )
```

2.1.2 Lambda function (In-place function definition)

To define a function in-place, one uses the following syntax:

```
{
  [statement];
  [statement];
  ...
  return [expression];
}
```

In-place definition of a function is an `expression`. It means, it can appear everywhere, where one can use an `expression`.

2.1.3 Control structures

Control structures like *if-else* or *while* are `statements`.

2.1.3.1 Condition IF-ELSE

One can execute series of statements under a condition using the short form of the `if` statement:

```
if ( condition )
{
  [statement];
  [statement];
  ...
}
```

or full form with `else` :

```
if ( condition )
{
  [statement];
  [statement];
  ...
}
else
{
  [statement];
  [statement];
  ...
}
```

`condition` is a `boolean expression`

2.1.3.2 Loop WHILE

One can execute a series of statements as long as a condition is true using the `while` statement:

```
while ( condition )
{
    [statement];
    [statement];
    ...
}
```

Sequence of statements will be executed till condition is not fulfilled, max 1000000000 times.

2.1.4 Variables

CubePL allows to work with memory, by using variables. All variables are multi-typed:

- In string context value of a numeric variable is presented in string format.
- In numerical context a string variable is converted to its numerical representation, if possible. Otherwise it is 0.

All variables are arrays. Indexless access to the variable assumes mutually index value 0.

2.1.4.1 User defined Variables

The user can define a variable using the syntax:

```
${ name } = [expression];
```

or

```
${ name }[index] = [expression];
```

Currently its name is a fixed string of characters. In later versions of CubePL it will allow any expression.

Example of a string context :

```
${name} seq "STRING"
```

Example of a numeric context :

```
${name} >= 0.34
```

Example of an array access to the variable :

```
${name}[ ${i} ] >= 0.34
```

Access to the variable is an expression.

Index for the access to the value is also an expression.

2.1.4.2 Predefined Variables

Cube provides a set of predefined variables for every calculation, which value depends on the current calculation context:

Predefined variable	Explanation
cube::#mirrors	Number of mirrors in cube file
cube::#metrics	Number of metrics in cube file
cube::#root::metrics	Number of root metrics in cube file
cube::#regions	Number of regions in cube file
cube::#callpaths	Number of call paths in cube file
cube::#root::callpaths	Number of root call paths
cube::#threads	Nuber of threads in cube file
cube::#threads::void	Number of void threads in cube file
cube::#threads::nonvoid	Number of nonvoid threads in cube file
cube::filename	Name of the cube file
calculation::metric::unit::name	Unique name of the current metric
calculation::metric::disp::name	Display name of the current metric
calculation::metric::url	URL of the documentation of the current metric
calculation::metric::description	Description of the current metric
calculation::metric::dtype	Data type of the current metric
calculation::metric::uom	Unit of measurement of the current metric
calculation::metric::expression	CubePL expression of the current metric
calculation::metric::id	ID of the current metric
calculation::metric::#children	Number of children in the current metric
calculation::callpath::mod	Module of the current call path
calculation::callpath::line	File line of the current call path
calculation::callpath::id	ID of the current call path
calculation::callpath::#children	Number of children in the current call path

<code>calculation::region::name</code>	Name of the region of the current call path. In fact it is a an array with the names of regions of all parents of the current call path. Index 0 corresponds to the current call path, index 1 to its parent, index 2 to parent of the parent and so on. One can apply <code>sizeof(calculation::region::name)</code> to get the size of the array.
<code>calculation::region::url</code>	URL with the description of the region of the current call path
<code>calculation::region::description</code>	Description of the region of the current call path
<code>calculation::region::mod</code>	Module of the region of the current call path
<code>calculation::region::begin::line</code>	Begin line of the region of the current call path
<code>calculation::region::end::line</code>	End line of the region of the current call path
<code>calculation::region::id</code>	ID of the region of the current call path
<code>calculation::sysres::name</code>	Name of the current system resource (Machine, Node, Process or Thread)
<code>calculation::sysres::id</code>	ID (rank) of the current system resource
<code>calculation::sysres::kind</code>	The kind of the system resource: <ul style="list-style-type: none"> • MACHINE • NODE • PROCESS • THREAD
<code>calculation::sysres::rank</code>	The rank of the process or thread. This variable is set only, if the kind of the system resource <code>PROCESS</code> or <code>THREAD</code>
<code>calculation::thread::void</code>	Set to <code>true</code> (1) if the thread is voided, otherwise to <code>false</code> (0)
<code>calculation::process::void</code>	Set to <code>true</code> (1) if the process is voided, otherwise to <code>false</code> (0)

2.1.5 Different ways to refer an another metric

2.1.5.1 Context sensitive reference to another metric

To use values of another metric in the same calculation context, one uses syntax:

```
metric::<[uniq_name]( modifier, modifier )
```

or

```
metric::<[uniq_name]( modifier)
```

or

```
metric::<[uniq_name]()
```

There are three version of this call:

1. with two arguments (call path and system);
2. with one argument (call path);
3. with no argument(an arguments takes as '*').

`modifier` specifies flavor of the calculation: *i* - inclusive, *e* - exclusive, * - same like in calculation context.

Metric reference is an expression.

2.1.5.2 Context insensitive reference to another metric

To use values of another metric in the some fixed calculation context (e.g. aggregated over threads), one uses syntax:

```
metric::
```

or

```
metric::
```

or

```
metric::
```

There are three version of this call:

1. with two arguments (call path and system);
2. with one argument (call path);

3. with no argument(an arguments takes as '*').

modifier specifies flavor of the calculation: *i* - inclusive, *e* - exclusive, * - same like in calculation context.

Metric reference is an expression.

2.1.5.3 Definition of an encapsulated calculation within CubePL expression using metrics of Cube.

One special mechanism of CubePL processing engine allows some level of calculation separation. A derived metric can be created within another CubePL expression. Such "ghost" metric gets its name and properties and exists inside of the cube object as a casual metric. Only difference to the casual metric is in the fact that ghost metric is not visible in GUI and tools and is not stored inside of the metric tree of the cube file.

One can refer such metric as a casual metric using metric references (see '[Context sensitive reference to another metric](#)' and '[Context insensitive reference to another metric](#)').

Example for definition of such metric :

```
cube::metric::prederived::nvisitors(e)
<<
{
  ${return}=0;
  if ( ${calculation::process::void} != 1)
  {
    if (metric::visits()>0 )
    {
      ${return} = 1;
    };
  };
  return ${return};
}
>>
${visitors} = metric::fixed::nvisitors(e);
```

where

```
cube::metric::prederived::nvisitors(e)
```

gives a type (prederives, exclusive) and unique name (nvisitors) of this metric. Unique name is used then later to refer to this metric via

```
${visitors} = metric::fixed::nvisitors(e);
```

There are kinds of metrics, which can be defined on such manner:

1. cube::metric::prederived::name(e) - An exclusive prederived metric with the name name;

2. `cube::metric::prederived::name(i)` - An inclusive prederived metric with the name `name`;
3. `cube::metric::postderived::name` - A postderived metric with the name `name`.

Notice that once the metric with some unique name created it exists whole lifetime of the cube object. Therefore one can refer to some somewhere previously defined ghost metric from any following it CubePL expressions.

Ghost Metric definition is a `statement`.

2.2 Grammar of CubePL

Here will be a full grammar of CubePL expressions (later).

3 Reference of CubePL

3.1 Supported calls in CubePL

Here is the table of currently supported calls in CubePL:

Operation	Explanation
123.34	a numerical constant
"STRING"	a string constant
-x	negative value of x
()	grouping (priority)
	absolute value
+ - / *	arithmetical operations
A ^x	A in power of x
> < >= <= != ==	numerical comparison
and or xor not	boolean operations
eq	string comparison
seq	case insensitive string comparison
==~ /expression/	matching of regular <i>expression</i>
sin(x) cos(x) asin(x) acos(x) tan(x) cot(x) atan(x) acot(x)	trigonometrical functions
ln(x)	natural logarithm of x
exp(x)	natural exponent of x
sqrt(x)	square root value of x
random(x)	random number from 0 to x
pos(x)	returns x, if x>, or 0
neg(x)	returns x, if x<0, or 0
sgn(x)	returns -1, if x<0, 0 if x is 0 and 1 is x>0
abs(x)	returns absolut value
ceil(x)	returns smallest integer value, which is bigger than x
floor(x)	returns biggest integer value, which is smaller than x
min(x,y)	returns smaller value of x and y
max(x,y)	returns bigger value of x and y
uppercase(x)	returns uppercase version of x
lowercase(x)	returns lowercase version of x

<code>\${var}</code>	0th element of the variable with name <i>var</i>
<code>\${var}[x]</code>	x-th element of the variable with name <i>var</i>
<code>sizeof(var)</code>	number of elements in the variable <i>var</i>
<code>defined(var)</code>	returns true (1), if the variable <i>var</i> is defined, otherwise it returns false (0)
<code>metric::[uniq_name](i e *, i e *)</code>	context sensitive value of metric <i>uniq_name</i>
<code>metric::fixed::[uniq_name](i e *, i e *)</code>	context independent value of metric <i>uniq_name</i>

4 Examples of CubePL expressions

4.1 Simple Examples

1. Calculation of an arithmetical expression

```
123.4 + 234 - ( 23 ) ^ 2
```

2. Calculation of an arithmetical expression with different functions

```
sin(23 + ln(12))
```

4.2 Complex Examples

4.2.1 Different made-up expressions

1. Definition of a constant function

```
{ return 24; }
```

2. Definition of a more complex function

```
{ return sin ( { return 1; } ); }
```

3. Definition of a function with an access to one variable

```
{ ${a}=123; return ${a}; }
```

4. Definition of a function with a control structure

```
{  
  ${a}=metric::visits();  
  ${b}=0;  
  if ( ${a}>100 )  
    { ${b}=metric::time(); };  
  return ${b};  
}
```

5. Definition of a function with a loop structure

```
{  
  ${a}=0; ${b}=0;  
  while ( ${a}<123 )  
  {  
    ${b}=${b}+metric::time();  
    ${a}=${a}+1;  
  };  
}
```

```
return ${b};
}
```

6. Definition of a function with a access to predefined variable

```
{
  ${a}=0;
  if (${calculation::region::name} =~ /^MPI_/)
  {
    ${a}=metric::time();
  };
  return ${a};
}
```

4.2.2 Special metrics of Scalasca

1. Calculation of an average runtime of a call path (Kenobi metric, postderived)

```
metric::time(i)/metric::visits(e)
```

2. Calculation of the time, spend in MPI synchronization calls

```
{
  if (
    (${calculation::region::description} eq "MPI")
    and
    (
      (${calculation::region::name} seq "mpi_barrier" )
      or
      (${calculation::region::name} seq "mpi_win_post" )
      or
      (${calculation::region::name} seq "mpi_win_wait" )
      or
      (${calculation::region::name} seq "mpi_win_start" )
      or
      (${calculation::region::name} seq "mpi_win_complete" )
      or
      (${calculation::region::name} seq "mpi_win_fence" )
      or
      (${calculation::region::name} seq "mpi_win_lock" )
      or
      (${calculation::region::name} seq "mpi_win_unlock" )
    ))
  {
    ${a}=metric::time();
  }
  else
  {
    ${a}=0;
  };
  return ${a};
}
```

3. Calculation of the Computational load imbalance (single participant)

```
cube::metric::prederived::nvisitors(e)
<<{  ${return}=0;
    if ( ${calculation::process::void} != 1)
    {
        if (metric::visits()>0 )
        {
            ${return} = 1;
        };
    };
    return ${return};
}
>>;

${devidor}=0;
while (($i < sizeof(calculation::region::name))
    and (($is_parallel) != 0) )
    )
{
    if ( ${calculation::region::name}[$i] seq "$omp parallel")
    {
        ${is_parallel}=1;
    };
};
if ( ${is_parallel} != 0)
{
    ${devidor}=${cube::#threads::nonvoid};
}
else
{
    ${devidor}=${cube::#processes::nonvoid};
};
${myvalue}=0;
${visitors} = metric::fixed::nvisitors(e);
if (${visitors} != ${calculation::#threads::nonvoid})
{
    if ( ${calculation::process::void} != 1)
    {
        if (${visitors} == 1)
        {
            if ( metric::visits() != 0 )
            {
                {
                    ${sum}= metric::fixed::execution(e)
                        -metric::fixed::mpi(e)
                        -metric::fixed::omp_time(e);
                    ${myvalue} =
                        (${sum}/${devidor}) * (${calculation::#threads::nonvoid} - 1);
                };
            };
        };
    };
    return ${myvalue};
}
```