Quantitative Time Calculations using Matrix Mathematics

The writer of this paper assumes the reader is familiar with Matrix Data types and their use in computer science and mathematics. The following terms are used loosely and are to be defined as follows:

“set” or “dataset” – A set of data which is scalar in form and possibly vector in dimensions

In many real world cases data in a set can be related to a specific date and time. Normally when one is performing calculations or regressions on data in a set the date and time is used for little more than filtering the data retrieved into an acceptable range. Furthermore the precision to the unit of time from which the data was retrieved is lost using this method, in other words once factored for a value can be reverse factored back to the date and time from which it came.

The problem this paper attempts to address is the storage and calculations of values within the set, the precision of those values and the ability to have “reversed” factoring back to a specific unit of time.

The initial hurdle of this problem comes when one wishes to store data from a set. The data should be able to be stored in a form which is both logical and meaningful for the person. In other words data storage should also allow for logical traversal and indexing based on date and time values.

The storage will need to be agnostic of month and year and provide methods of obtaining totals, averages, means, and other mathematical relationships between the data in the set.

Normally a person would perform a calculation to obtain the required results for a sum in a method like so:

Let sum = 0, d = 0

For every scalarValue in vectorDimension(d)

sum = sum + scalarValue

return sum

A person would perform a calculation to obtain the required results for an average in a method like so :

Let sum = 0, values = 0, d = 0

For every scalarValue in vectorDimension(d)

sum = sum + scalarValue

values = values + 1

return sum/values

From the above two examples I will outline a few major problems:

1. scalarValues from any of the vectorDimensions have no correlation to the specific unit of time from which it was retrieved.
2. Average requires sum but using any the above methods the data “count” is never used in the sum operation and thus would be meaningless to the sum function. For this reason a dependant variable is introduced to correlate the correct average from the sum.
3. Once a scalarValue has been calculated from a vectorDimension is cannot be reverse factored to a specific time unit from the vectorDimension.

The solution to these problems also comes with the added benefit of additional performance. It is called the DateMatrix. The DateMatrix is used like other math matrices.

Let us outline the storage mechanism for the DateMatrix and how it complies with the specifications required for Quantitative Time Calculations using Matrix Mathematics.

Let DateMatrix = ( Array<dataType>[31,24,60,60] )

Where dataType = “the type of the data in the DateMatrix e.g. int, double”

The indices of this matrix stand for the following:

31 – The maximum number of days in any given month

24 – The maximum number of hours in any given day

60 – The maximum number of minutes in any given hour

60 – The maximum number of seconds in any given minute

Load data from a set into the matrix using the following procedure:

Let set contain a column which is the DateTimeKey at ordinal 0

Let set contain a column which is the scalarValue at ordinal 1

Let DateTime = set.value(0)

Let Value = set.value(1)

Let DateMatrix[DateTime.Day – 1, DateTime.Hour, DateTime.Minute, DateTime.Second] = Value

Once the DateMatrix is loaded with the data from set you now have the individual scalarValues from each dimension in a specific time slice of the DateMatrix.

I call this version of the DateMatrix a 4 Dimensional View of a DateMatrix.

Using this 4 Dimensional View once can very quickly obtain the sum, mean, average or other mathematical computation of a specific time value or a range.

For example to obtain the specific value for a specific time slice you only need to logically think of its location in the matrix.

To return the value for the 7th day of the month at the 12 hour in the day at the 3rd minute of the hour at the 2nd second of the minute ( 7th of month at 12:03:02 )

One can use the following logic:

Let value = DateMatrix[6,12,3,2]

Return value

Days are offset by 1 to ensure that months with 31 days are able to fit into the matrix.

To obtain the average of the 7th day of the month at the 12 hour in the day at the 3rd minute of the hour ( 7th of month at 12:03 )

Once can use the following logic:

Let dayKey = 6

Let hourKey = 12

Let minuteKey = 3

Let secondsInMinute = 60

Let currentSecond = 0

Let averageValue = 0

While( currentSecond < secondsInMinute)

averageValue += DateMatrix[dayKey,hourKey,minuteKey,currentSecond]

currentSecond = currentSecond + 1

return averageValue / secondsInMinute

One can also perform a dimensional fold on the matrix to reduce the number of dimensions to 3. I call this version of the DateMatrix a 3 dimensional DateMatrix.

Let DateMatrix = ( Array<dataType>[7,24,60] )

Where dataType = “the type of the data in the DateMatrix e.g. int, double”

The indices of this matrix stand for the following:

7 – The maximum number of days in any given week

24 – The maximum number of hours in any given day

60 – The maximum number of minutes in any given hour

To fold / reduce a 4 dimensional DateMatrix into the 3rd dimension one can use the following logic:

Let 4D = (4 Dimensional DateMatrix)

Let 3D = (3 Dimensional DateMatrix)

Let dayEnd = 4D.GetUpperBound(0)+1

Let hourEnd = 4D.GetUpperBound(1) +1

Let minuteEnd = 4D.GetUpperBound(2) +1

Let secondEnd = 4D.GetUpperBound(3) +1

Let secondsInMinute = 60

Let minutesInHour = 60

Let minuteTotal = 0;

For( d=0 To dayEnd)

{

For(h=0 To hourEnd)

{

For(m=0 To minuteEnd)

{

minuteTotal = 0

For(s=0 To secondEnd)

{

Let DateTimeKey = new DateTime(year,month,d,h,m,s)

minuteTotal = minuteTotal + 4D[d,h,m,s]

}

3D[DateTimeKey.DayOfWeek, h, m] = minuteTotal / secondsInMinute

}

}

}

Now 3D which is a 3 dimensional DateMatrix has the total values for all of the days of the month in columns allocated by day of the week in which the day of the month fell on.

From here if you want averages you must divide each value by the occurrences of the day in month.

The matrix can even be folded down 1 dimension further into a 2 dimensional matrix which simply sums the minuteTotal to the Hour. The format of this matrix and the procedure in which it can be calculated from a 3 dimensional DateMatrix is below:

Let DateMatrix = ( Array<dataType>[7,24] )

Where dataType = “the type of the data in the DateMatrix e.g. int, double”

The indices of this matrix stand for the following:

7 – The maximum number of days in any given week

24 – The maximum number of hours in any given day

To fold / reduce a 4 dimensional DateMatrix into the 3rd dimension one can use the following logic:

Let 3D = (3 Dimensional DateMatrix)

Let 2D = (2 Dimensional DateMatrix)

Let dayEnd = 4D.GetUpperBound(0)+1

Let hourEnd = 4D.GetUpperBound(1) +1

Let minuteEnd = 4D.GetUpperBound(2) +1

Let minutesInHour = 60

Let hourTotal = 0;

For( d=0 To dayEnd)

{

For(h=0 To hourEnd)

{

hourTotal = 0

For(m=0 To minuteEnd)

{

Let DateTimeKey = new DateTime(year,month,d,h,m)

hourTotal = hourTotal + 3D[d,h,m]

}

2D[DateTimeKey.DayOfWeek, h] = hourTotal / minutesInHour

}

}

Now 2D which is a 2 dimensional DateMatrix has the total values for all of the days of the month in columns allocated by day of the week in which the day of the month fell on with all of the minute data summed to the hour index.

Depending on the values which were in 3D, 2D either is average or total data at this point.

Reverse factoring can be performed by using 4D in relation to 3D and 2D and vice versa.

For instance one can very easily relate the 2D value to the specific time values for any second in the 4D values by performing a looking for the required index.

Because I have chose to use the array notation to define the DateMatrix the concept is language agnostic so long as the language supports multi dimensional arrays. Serialization of the DateMatrix from one architecture to another maintains precision and meaning across multiple platforms allowing for high interoperability.

Because none of the original data is ever lost and because 4D, 3D and 2D may exist at the same time additional performance is gained because values are cached and calculations in a dimension only need to occur once and can be referenced back to for use in regressions without further calculation.

[References]

…..

[Thanks]

…..

[Notes]

…..