Free University of Bolzano - Bozen
Faculty of Computer Science

Compression of Time Series Data Using Multiple Granularities

Author:
Michael Gurschler

Supervisor:
Prof. Johann Gamper

Academic Year 2010/2011

Abstract

The project of this thesis comprises the planning, implementation and testing of a compression algorithm for time series data with multiple granularities. The data which should be compressed comes from a system called SolarInspector which was developed by myself. SolarInspector collects lots of data from photo-voltaic inverters and other sensors, manages and analyzes the values and stores them regularly into a database. Since the embedded servers which collect the data have a limited hardware resources it is necessary to keep only the data which is needed. Therefore my task was to develop an algorithm which compresses the time series data as much as possible without falsifying the data to much. I used Transact SQL, which is Microsoft's proprietary extension to SQL, to develop the compression algorithm. Generally this algorithm can be used to compress time series data of any type. Moreover it is possible to use different grades of compression depending on the age and the type of the data.

A major challenge during the project was to find a way to aggregate the log values without falsifying the chart. Simply taking the average function for the aggregation would flatten the chart too much. Therefore I had to consider also the adjacent points and based on the information I got with them it was possible to choose the best aggregation function (min, max, average). Several tests at the end of the development showed that the charts resulting from the original data are nearly equivalent with the charts resulting from data which was compressed to about a quarter of the original size.

Abstract

Das Projekt dieser Diplomarbeit besteht aus der Planung, Implementierung und dem Testen eines Komprimierungs Algorithmus welcher zeitlich bezogene Daten mit verschiedenen Detailgenauigkeiten zusammenfasst. Die Daten welche komprimiert werden sollen kommen von einem System welches SolarInspector heisst und von mir entwickelt wurde. Der SolarInspector sammelt Daten von Photovoltaik Wechselrichtern und anderen Sensoren, verwaltet und analysiert die Daten und speichert Diese regelmäig in eine Datenbank. Da die verwendeten PCs nur über begrentzte Hardware Resourcen verfügen ist es notwendig dass nur die benötigten Daten gespeichert werden. Deshalb war es meine Aufgabe einen Algorithmus zu entwickeln welcher die Daten so weit wie möglich komprimiert ohne die Daten zu sehr zu verfälschen. Für die Entwicklung des Algorithmus wurde T-SQL verwendet, welches Microsofts proprietäre Erweiterung für SQL ist. Der Algorithmus kann für zeitliche Logwerte jedes Typs verwendet werden. Zudem ist es möglich verschiedene Detaillevels zu definieren, abhängig von dem Alter der Daten und dem Typ der Daten.

Die gröte Herausforderung während des Projekts war es einen Weg zu finden die Daten so zusammenzufassen dass die daraus erstellten Diagramme nicht verfälscht werden. Deshalb war es nicht möglich die Datensätze einfach durch das Ermitteln des Durchschnittwertes zu komprimieren weil es die Kurve zu sehr glätten würde. Um dieses Problem zu beheben musste ich auch die Punkte bevor und nach dem aktuellen Wert beachten, somit war es möglich für jeden Punkt das Beste Verfahren zu wählen (Minimum, Maximum, Durchschnitt). Verschiedene Tests am Ende der Entwicklungsphase haben gezeigt dass die Diagramme welche mit den originalen Daten erstellt wurden nahezu identisch mit den Diagrammen sind welche mit den Daten erstellt wurden, welche auf ein Viertel der Originalgrösse komprimiert wurden.

Abstract

Il progetto di questa tesi comprende la progettazione, realizzazione e sperimentazione di un algoritmo di compressione per dati di serie storiche con granularit multipla. I dati che devono essere compresse vengono generate da un sistema che si chiama SolarInspector, il quale stato sviluppato da me. SolarInspector raccoglie i dati da inverter fotovoltaico altri sensori, gestisce e analizza i valori e li memorizza in un database regolarmente. Dato che i server che raccolgono i dati, hanno risorse hardware limitate, necessario un sistema per salvare soltanto i dati neccesari. Quindi ho sivluppato un algoritmo che comprime i dati di serie temporali senza falsificando i propri valori. Ho usato Transact SQL, che un'estensione proprietaria di Microsoft per SQL, per sviluppare l'algoritmo di compressione. possibile impostare diversi gradi di compressione a seconda dell'et e del tipo di dati.

Una delle maggiori problemi nel corso del progetto era quello di trovare un modo per aggregare i valori di registro, senza falsificare il grafico. Prendendo la funzione di media per l'aggregrazione potrebbe appiattire troppo il grafico. Quindi ho dovuto prendere in considerazione anche i punti adiacenti e sulla base delle informazioni che ho avuto , stato possibile scegliere la funzione migliore aggregazione (min, max, media). Diversi test alla fine dello sviluppo hanno mostrato che la classifica risultante e(hangl) stata compressa a circa un quarto delle dimensioni iniziali.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Data logging and analysis is at the core of many business and scientific processes. This data analysis is used to collect information from machines, the surrounding, user behaviour and many more. Often the data is measured through sensors and used to create charts and statistics. Nowadays relational databases are used to store the data and SQL to insert or retrieve the data.

Nearly every database contains data records with timestamps which are collected regularly.

- Wheater stations collect data for the temperature, air pressure, wind, etc.
- CNC machines store the actual position and speed of the axes.
- Racing cars record the acceleration, speed, throttle, etc.
- Photo-voltaic monitoring systems collect the actual input power of the inverters.

The collected data allows the user to analyze and better understand how the system works and what could be done to improve it. But with the time the databases will become bigger and bigger. Nowadays disk space is relatively cheap but in specific situations it might not be possible to have more disk space. Moreover the database queries will slow down dramatically when the database is becoming bigger.

As the data gets older it might loose importance which means that the data might be stored at a coarser granularity. But changing the granularity of the data is not as simple as it seems. It is really important that charts which are generated with the compressed data are nearly equivalent with the charts which are generated with the original data. Therefore we decided to plan, implement and test an algorithm which is able to compress the temporal data without loosing too much details.

1.2 Problem Description

The photo-voltaic monitoring system called SolarInspector collects the data from several inverters and stores it regularly. Each data record is stored with a timestamp which indicates the point in time when this record is valid. This allows to analyze the collected data with a temporal perspective. Using temporal aggregation the user will get a summary of the data (in charts, or statistics). It allows the user to identify the trend and search for problems. Such problems might often remain invisible but with charts it is relatively simple to detect critical patterns.

But these temporal aggregations are very time consuming since they need a lot of computational power. The embedded PC's which collect the data from the inverters have only a very limited hardware (computational power and memory).

Therefore not the whole data can be kept in memory which slows the queries down dramatically. Such that it is necessary to keep the database smaller by compressing the data.

Even if a variable is logged only every 5 minutes we obtain 105120 tuples in a year. Knowing that we can imagine how much data we will have if we log 100 variables over years. In our system an inverter has about 10 to 16 variables (depending on the type) and has an average uptime of about 12 hours per day. With a log interval of 150 seconds in a facility with 4 inverters we obtain about 6 million tuples per year. With the time the queries will slow down dramatically. Moreover the size of the database increases by about 1 GB per year, which is also a big problem for our embedded server which have a relatively small hard disc and only a slow processor. Bigger photo-voltaic systems might have 50 or more inverters which generate about 12 GB (50 X 250MB) per year.

Therefore we had to build something which solves such problems. Since the data is often needed at a high detail level for a specific time it was not possible to simply increase the log interval. Because of that it is necessary to compress the data after some time. But the compression itself is a very critical task since it should remove tuples and on the other hand it should not change the charts which will be generated from the data. Deleting records or taking the average would do so, therefore it was not an acceptable solution for this particular problem.

1.3 Proposed Solution

A solution for the above problem would be to compress the data by aggregating data records. We propose in this thesis an algorithm which takes care of that. It analyzes the whole data, picks the data series which have to be compressed and aggregates them until they correspond to the specified granularity for the age of the records. In order to keep the algorithm fast we decided to first pick all data series which have to be compressed with a simple query which calculates the granularity. Such that we do not have to iterate through the whole database. As soon as that is done we have to compress the data records of the data series until they have the specified granularity. The first and the last point of a data series during one day can not be deleted because that would falsify statistics which are created with the daily uptime of the system.

An original chart which shows the input power of a photo-voltaic inverter might look like the trend shown in figure 1. The uncompressed chart here in this example has 11 tuples and our algorithm has to aggregate tuples in order to obtain a very similar chart with less tuples. The minimal and maximal points in the charts are really important but the points in the middle are less important. Therefore it is better to aggregate them first. The figure 2 shows the same trend in a chart with compressed data. The compressed data has only 7 tuples (instead of 11) and the overall trend of the chart is very similar with the chart resulting from the original data. Note that in real life there are much more tuples in the chart therefore the compressed and uncompressed charts will look even more similar then with just a small data set. We tested the quality and

the usability of our proposed compression algorithm with real data in Chapter 5.

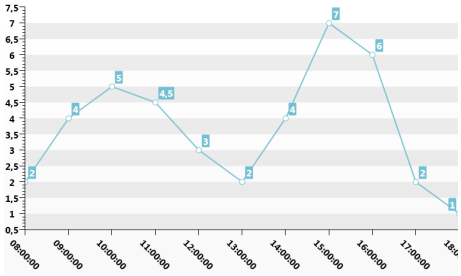

Figure 1: The uncompressed chart showing the input power of an inverter over a day (11 tuples)

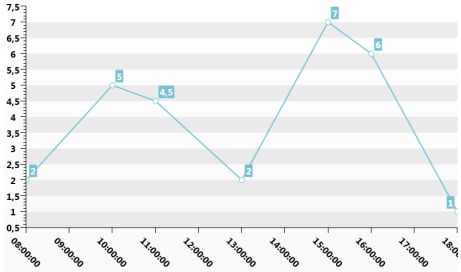

Figure 2: The compressed chart showing the input power of an inverter over a day (7 tuples)

The selection of the two data records which have to be compressed and their aggregation is a very critical task for which we planned, implemented and tested a solution in which the algorithm decides for each single pair which aggregation strategy is the best. First we order the adjacent points according to their time difference. After that we consider the trend of the data series by simply analyzing the values before and after this pair. If one of the points is a maximum or a minimum we keep that point and remove the other one. Otherwise they will be simply aggregated by taking the average. Such that it is possible to adapt the aggregation strategy based on the trend of the data.

1.4 Organization of the Thesis

This thesis contains four main chapters, this introduction, and the conclusion chapter. Each of the main chapters contains its own introduction and addresses a specific aspect.

In **Chapter 1** we introduced our motivation, the problems and our proposed solution. Moreover we guide through the organization of the thesis.

In **Chapter 2** we define the main concepts of data logging and discuss it with other works which are related to a similar topic.

In **Chapter 3** we explain the architecture of our system. Containing the network architecture, the database architecture and the logical architecture.

In **Chapter 4** we introduce the compression algorithm which we propose in this thesis. We explain the overall strategy and structure of the algorithm in his details with appropriate examples and pseudo code.

In **Chapter 5** we evaluate the solution which we proposed. First we define our setup we used for the evaluation. After that we measured the performance and the quality / usability of our compression algorithm.

2 State of the Art

Nowadays data logging is a very important process of recoding events with an automated computer program in order to provide an audit trail. These trails can either help to detect and diagnose possible problems or just to help in understanding the activity of the system during the shown timespan. The data is usually collected through sensors, analyzed and stored into a database. It is commonly used in scientific experiments and monitoring systems. In the early years of the computers the computational power and the disk space have been very expensive therefore only the important data was logged. In the last 10 years the computers became much faster and cheaper. Therefore people started to log wherever possible at a higher resolution than ever before. This data can then be analyzed in order to create statistics or charts, or simply to detect possible problems.

Common applications for data logging are weather stations which log the wind speed / direction, the temperature, the humidity, the solar radiation, etc. Another application which makes excessive use of data logging is the vehicle / crash testing. And also the monitoring of computerized numerical control (CNC) machines (position, speed, pressure) uses data logging for the analysis and better understanding.

The data logging is now changing more then ever before. The old model of a stand alone data logger which collects and monitors data is no longer up to date. Modern data loggers will not just collect data and monitor it. They analyze the data immediately and alarm the user in case of problems and create automatic reports. The loggers often send the data directly to a webserver or host their own web pages in a local network. [1]

There are different ways to store the data. The most common ones are the range temporal data logging and the timestamp data logging.

In *Range temporal data logging* the values are logged with a timestamp and another parameter which indicates for how long they are valid. This is most commonly used when the values which have to be logged do not change frequently. Such that it is possible to capture the value with just one tuple as

long as it remains unchanged. With this type of data logging it is possible to determine the exact value of the variable at each timespan. Generating charts and statistics is more expensive then with the timestamp data logging.

In *Point temporal data logging* each tuple has the value of the variable and the timestamp to which it belongs. It is most commonly used for values which are changing very frequently. The data model is very simple which makes it easy to generate charts out of the data. But it is not possible to get the exact value from a given timestamp since the values are only captured in an interval. Depending on how fast the values are changing and on the log interval it is possible to guess the value between two points.

With the increase in data which is collected, temporal aggregation has become a very important task in data analysis. Computing these temporal aggregates is usually costly and a bigger problem then just aggregating traditional data. Each tuple in the database consists of a timestamp, an interval in which it is valid (optional) and other parameters which have been logged. Such data is usually stored with multiple levels of temporal granularities such that older data is stored at a coarser granularity and more recent data is stored with a higher detail.

In the last years there has been much research on temporal aggregation using multiple granularities [2] [4] [8] [6].

A related work to this thesis is [2], which presents an effective way to aggregate temporal data with multiple granularities. In that paper they provide specialized index schemes for maintaining dynamically the temporal aggregates. Our works differs from their, because they are just maintaining the index for the temporal aggregates, we instead are compressing the existing data directly. Sure, both ways of aggregating the data have practical meanings and can be used different scenarios.

3 System Architecture

We divided the project in three parts as shown in figure 3. The embedded servers, the master server and the clients.

The *embedded servers* are to so called data loggers which collect the data from the sensors (in this case inverters via a serial connection), analyze it and store it into the local database. They run a web server on the local network which allows the clients to analyze and manage the system. The embedded servers synchronize their data regularly with the data from the master server. Such that they can be remote controlled and updated automatically.

The *master server* is running in the world wide web and manages the data from the embedded servers. It provides a web page for the clients which allows them to monitor the logged values, alarms and other things.

The *clients* can access their data over a webpage, either locally directly from the embedded server or over the internet from the master server.

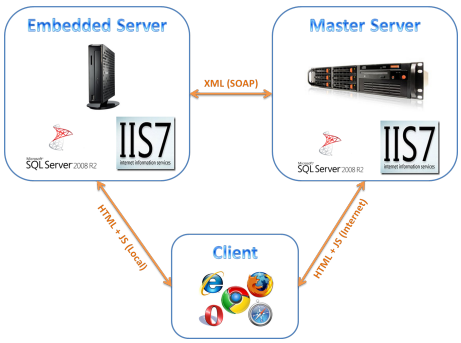

Figure 3: The structure of the system

All data is stored in a Microsoft SQL Server 2008 R2 database (on the embedded server and also on the master server). The websites (ASP.NET) are hosted in the Internet Information Server (IIS) and even the charts are created without flash, silverlight or any other plugins. In order to provide the whole functionality in the server we used the following frameworks / libraries:

- The *.NET Framework* which provides the standard classes for the development and also the special classes like the SerialPort
- *LightCore*, a lightweight dependency injection container which can also be used as a service locator
- *MathParser*, a parser which is able to solve mathematical expressions, which we used to calculate variables with the values from the inverters
- *Telerik.Web.UI*, a library which contains useful controls for the web development with ASP.NET (Charts, DataGrids, etc)

In this section we will show the main relevant tables and relations of our database.

- *Devices*: Contains all devices which are attached to the embedded server. Each device is composed by and id (PK), a name, a description, the com port on which it is attached and it's address, it's serial number, and the FK of the Type to which it belongs.

- *VariableUnits*: Contains all units the variables could be of (Hz, Volt, Ampere, Watt). Each variable-unit is composed by an id (PK), and a name

- *Variables*: Contains all variables which could be logged into the database. Each variable is composed by an id (PK), a name and a has unit (in a n:1 relation with the id of table VariableUnits)

- *VariableLogs*: Contains all log values of the variables. Each variable-log is composed by a timestamp, the device (in a n:1 relation with the id of the table Devices), the id of the string (the string is a concatenation of multiple photo-voltaic modules, an inverter can have 1 or more), the variable (in a n:1 relation with the id of the table Variables) and the value.

- *VariableLogGranularities*: Contains all granularities of the variable logs. Each variable log granularity is composed by an id (PK), the id of the variable (in a n:1 relation with the id of the table Variables), the minimal age of the tuples to fall in this granularity section and the maximal entries per hour in this granularity section.

Note that these tables are only a part of the database. There are much more tables which are not relevant for this thesis.

For the communication between the embedded server and the master server we used the Windows Communication Foundation (WCF). WCF is an API of the .NET Framework which can be used to build connected, service oriented applications with web services. For our project we used the SOAP protocol with a HTTP binding. Communication happens only one way, from the embedded server to the master server. Because of security and technical reasons it is not possible to access the embedded servers from the internet.

For the development of the compression algorithm I used Microsoft's and Sybase's proprietary extension to SQL, called Transact - SQL or even shorter T-SQL. T-SQL contains useful functions like the if statement, loops, variables and much more. Such that it is possible to manipulate the data automatically using stored procedures and functions.

4 The Compression Algorithm

4.1 Overall Strategy

For the compression of time series data using multiple granularities this thesis proposes an algorithm which is applicable for different types of log data. The main idea is to aggregate points in the time series which are the least important, meaning that they are no points of maximum or minimum and that the time distance to the next point is small. The data is first analyzed to find the data series and their days where they have to be merged. This partial result are then compressed one after the other. Thus, the main steps of the algorithm are as follows:

1. Analyze the whole data and check whether a data series has to be compressed on a specific day. Store the data series / day pairs which have to be compressed in a temporary table (T1) for further usage
2. Process the first data series / day pair in the temporary table and aggregate the least important tuples until the granularity of the data series is coarse enough
3. Repeat step 2 as long as there are data series in the temporary table which have to be compressed

The main intuition behind this strategy was to keep the amount of data with which we have to deal as small as possible. Therefore we decided to introduce the first step which picks only a few data series (the ones which have to be compressed) out of the whole data. If the whole data has to be compressed this step might not be needed but usually there is only a small subset of the data series which have to be compressed since such compression algorithms should run regularly (once a day / once a week).

4.2 Structure

Our compression algorithm is structured in different modules (as shown in figure 4 which have clear responsibilities because each type of data might require a different tuple selection or aggregation procedures. Because of this modularity it is relatively easy to adapt the algorithm.

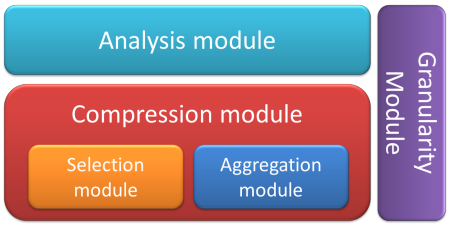

Figure 4: The structure of our algorithm

4.2.1 Analysis Module

The analysis module is responsible for the initial analysis of the data. Which means that it first of all determines the timespans where specific time series have to be compressed. This module is making use of the granularity module to obtain the granularities with which a variable has to be stored depending on the age of the data records.

1. Analyze the whole data in the database and group each data series by day
   - Get the maximal entries per day for this data series
   - Check whether the data series has to be compressed meaning that its granularity is coarse enough
2. Write all remaining data series into a new temporary table (T1) (unique rank, timestamp, number of tuples, expected number of tuples, device id, variable id, string id)
3. Call the compression module which will continue with the work

Algorithm 1 is the most important part of the analysis module. It analyzes the data series by day and writes the data series which have to be compressed in a new temporary table (T1). The information will then be used by the compression module. For the code in T-SQL see Figure 9 in the Appendix.

Algorithm 1 The algorithm which analyzes the data

```
1:  dSToCompress := New collection of data series
2:  dS := Get all data series grouped by day
3:  for i = 0 to number of dS − 1 do
4:      actualSerie := dS[i]
5:      noTuples := Get number of tuples in actualSerie]
6:      maxNoTuples := Get max. number of tuples for actualSerie
7:      if noTuples > maxNoTuples then
8:          Add actualSerie to dSToCompress
9:      end if
10: end for
11: return dSToCompress
```

Example 1: This example shows the generated output of the analysis module. Table 1 contains the raw data which should be compressed and Table 2 is the temporary table (T1) which is used by the module to add the data series which have to be compressed.

Timestamp	VariableId	DeviceId	StringId	Value
11.10.2011 09:20	1	2	-1	66
11.10.2011 09:22	1	2	-1	67
11.10.2011 09:22	2	2	-1	12
11.10.2011 09:24	1	2	-1	69
11.10.2011 09:25	2	2	-1	15
11.10.2011 09:26	1	2	-1	72
11.10.2011 09:28	1	2	-1	75
.....	.....	.....	.....	.....
11.10.2011 19:58	1	2	-1	287

Table 1: The data which was logged

Now imagine that there are much more variable logs (2012 tuples) for the variable with the id 1. Meaning that it has to be compressed. Variable 2 has only 2 logs therefore there is no need to compress it. The table 2 shows the generated output of the analysis module which is stored in the temporary table (T1).

Rank	Timestamp	Entries	MaxEntries	DeviceId	VariableId
1	11.10.2011 09:20	2012	706	2	1

Table 2: The resulting tuples in the temporary table

4.2.2 Granularity Module

The granularity module is responsible for the management of the granularities depending on the variables and the age of the data records. It will be used by the analysis and compression module. The granularities can be specified directly in this module or also in an additional table. When this module is called it will return the maximal number of data records per day depending on the given variable, the age and the length of the data series.

1. Calculate the age of the data series in days
2. Calculate the length of the data series in hours
3. Read the apropriate granularity of the given variable and age from the database
4. Verify if the resulting entries per hour are correct. If there is no matching entry in the database (ex. data is younger then the first granulation age) return the maximal value of int

5. Calculate the maximal entries for the data series and return it

Algorithm 2 returns the maximal entries / tuples for a data series which corresponds to the given data. For the code in T-SQL see Figure 10 in the Appendix.

Algorithm 2 The algorithm which determines the granularity of a data series

```
1: ageInDays := Get age of dataSerie in days
2: lengthInHours := Get length of dataSerie in hours
3: entriesPerHour := IntMaxValue
4: maxGranularityAge := 0
5: //Go through all specified granularities and pick the max. entries per hour
6: for i := 0 to numberOfGranularities − 1 do
7:    if granuarities[i].age <= ageInDays and granularities[i].age >
      maxGranularityAge and granularities[i].dataSerie = dataSerie then
8:       maxGranularityAge := granularities[i].age
9:       entriesPerHour := granularities[i].entriesPerHour
10:   end if
11: end for
12: maxEntriesInDataSerie := lenghtInHours ∗ entriesPerHour
13: return maxEntriesInDataSerie
```

Example 2: In this example we show how the granularities are retrieved from the database and what is actually returned when the call that function. The table 3 contains a test data in the granularities database. We added 2 time spans (30 days and 180 days) after which the data has to be compressed.

Min age in days	VariableId	Entries per hour
30	1	36
30	2	36
30	3	72
30	4	72
180	1	24
180	2	24
180	3	48
180	4	48

Table 3: Test data in the granularity table

With the specified granularities from table 3 we can now use the GetMaxEntriesOfDay function which returns the maximal entries per day for the given variable, startup and shutdown time. Since the entries per day depend on the age of the data we now assume that we have today the 11.10.2011.

VariableId	StartupTime	ShutdownTime	Value
1	09.09.2011 08:15	09.09.2011 18:32	360
3	09.09.2011 08:15	09.09.2011 18:32	720
1	02.01.2011 07:52	02.01.2011 19:13	264
3	02.01.2011 07:52	02.01.2011 19:13	528

Table 4: The return values of the function depending on the input

4.2.3 Compression Module

The compression module is the main part of the algorithm. It is responsible for the merging / aggregation of the data records. It receives the data series which have to be compressed and the days on which they have to be compressed. Using the granulation module it can then get the specific granularity for the given items. It makes use of the selection module which selects two adjacent points which should be merged next. Finally it uses the aggregation module which aggregates the two given points.

1. Select and delete the first data series / day pair in the temporary table (T1) which was filled by the analysis module
2. Get the next two tuples which should be aggregated with the selection module
3. Aggregate these two tuples with the aggregation module
4. Repeat step 2 until the number of tuples in the data series is less or equal the max number of tuples in this data series (information from T1)
5. Repeat step 1 until there are no data series / day pairs left for compression

Algorithm 3 is the most important part of the compression module. It goes through each data series and compresses it with the selection and aggregation module until the granularity is coarse enough. For the code in T-SQL see Figure 11 in the Appendix.

Algorithm 3 The algorithm which compresses the data series

```
1: for i := 0 to number of data series to compress - 1 do
2:    actualDS := dataSeries[i]
3:    while max number of entries ¡ number of entries do
4:       nextPair := Get next pair to compress
5:       Aggregate nextPair (AggregationModule)
6:    end while
7: end for
```

Example 3: This example shows the data before (Table 5) and after (Table 6) the compression. The granularity is set to 3 entries per hour.

Id	Timestamp	VariableId	DeviceId	StringId	Value
1	2011-10-11 08:00:00	1	2	-1	12
2	2011-10-11 08:05:10	1	2	-1	10
3	2011-10-11 08:10:02	1	2	-1	8
4	2011-10-11 08:15:40	1	2	-1	11
5	2011-10-11 08:20:20	1	2	-1	12
6	2011-10-11 08:25:10	1	2	-1	9
7	2011-10-11 08:30:01	1	2	-1	14
8	2011-10-11 08:35:05	1	2	-1	13
9	2011-10-11 08:40:22	1	2	-1	8
10	2011-10-11 08:45:11	1	2	-1	11

Table 5: Uncompressed variable log data of example 3

Id	Timestamp	VariableId	DeviceId	StringId	Value
1	2011-10-11 08:00:00	1	2	-1	12
2	2011-10-11 08:07:26	1	2	-1	9
5	2011-10-11 08:20:20	1	2	-1	12
7	2011-10-11 08:30:01	1	2	-1	14
9	2011-10-11 08:40:22	1	2	-1	8
10	2011-10-11 08:45:11	1	2	-1	11

Table 6: Compressed variable log data of example 3

The resulting compressed data which we see in table 6 was generated by the following steps:

- Aggregate tuples 4 and 5. Tuple number 5 is a maximum value therefore delete tuple 4.
- Aggregate tuples 5 and 6. Tuple number 5 is a maximum value therefore delete tuple 6.
- Aggregate tuples 2 and 3. Both are neither minimal or maximal values. Update tuple number 2 with the average values from both and delete tuple 3.
- Aggregate tuples 7 and 8. Tuple number 7 is a maximum value therefore delete tuple 8.

4.2.4 Selection Module

The selection module selects the data tuples which should be aggregated first. This module is used by the compression module which tells this module which data series should be compressed. Then this module will order all tuples in this data series according to their time difference (points with the smallest time difference go first). The result will then be stored into another temporary table (T3). Each pair consists of the id of the first tuple and the id of the second

tuple. Such that the compression module can aggregate the pairs one after the other until the granularity of the data series is coarse enough.

1. Write all tuples of the given data series on the indicated day ordered by the timestamp in a temporary table (T2)
2. Select the ids of the first and last tuples in the series
3. Order all adjacent tuples by their time difference (first and last tuples are excluded) and write their ids in a temporary table (T3)

The algorithm 4 writes the tuples of the actual data series in a new temporary table (T2) and prepares them for further use. In the next step it orders the tuple pairs according to their time difference and writes them in another temporary table (T3). Moreover it ensures that the first and last tuples are never aggregated since that might falsify the generated statistics (ex. startup time, shutdown time). For the code in T-SQL see Figure 12 in the Appendix.

Algorithm 4 The algorithm which prepares the data series for aggregation

```
1: diffTimespan :=get max value of timespan
2: firstId := −1
3: //Iterate over all tuples (except first and last) in the data series
4: for i := 1 to number of tuples in data series - 2 do
5:    //check if this pair has a smaller difference
6:    actualDifference := tuples[i + 1].time − tuples[i].time
7:    if actualDifference < diffTimespan then
8:       diffTimespan := actualDifference
9:       firstId := i
10:   end if
11: end for
12: return firstId and firstId + 1
```

Example 4: In this example we show how the tuples which have to be compressed are selected. The algorithm returns the adjacent tuples / pair with the smallest time difference. The table 7 contains the data series whichs pair with the smallest time difference should be selected.

Id	Timestamp	VariableId	DeviceId	StringId	Value
1	2011-10-11 08:00:00	1	2	-1	12
2	2011-10-11 08:05:10	1	2	-1	10
3	2011-10-11 08:10:02	1	2	-1	8
4	2011-10-11 08:15:40	1	2	-1	11
5	2011-10-11 08:20:20	1	2	-1	12
6	2011-10-11 08:25:10	1	2	-1	9
7	2011-10-11 08:30:01	1	2	-1	14
8	2011-10-11 08:35:05	1	2	-1	13
9	2011-10-11 08:40:22	1	2	-1	8
10	2011-10-11 08:45:11	1	2	-1	11

Table 7: Variable log data of example 4

FirstId	SecondId
4	5

Table 8: Output of the selection module for example 4

4.2.5 Aggregation Module

The aggregation module aggregates two given data tuples. Because of the modularity of the solution we proposed, this module can be exchanged easily. Such that it is possible to use different modules depending on the type of the data and the computational power of the database server.

1. Analyze the adjacent tuples (one tuple before the pair and one after the pair)
2. Choose the appropriate aggregation type and calculate the aggregation value
3. Update the first tuple and set the new value and timestamp
4. Delete the second tuple
5. Remove the actual pairs from the temporary tables (T2 and T3)

Algorithm 5 aggregates the two tuples with the given ids with an aggregation function according to their adjacent points. For the code in T-SQL see Figure 13 in the Appendix.

Algorithm 5 The algorithm aggregates the given tuple-pair

1: Fir :=Get first tuple of pair
2: Sec :=Get second tuple of pair
3: Pre :=Get predecessor of first
4: Suc :=Get successor of second
5: **if** ($Fir > Pre$ and $Fir > Sec$ and $Fir > Suc$) or ($Fir < Pre$ and $Fir < Sec$ and $Fir < Suc$) **then**
6: //The first entry is a min or max
7: Delete Sec from Log table
8: **else if** ($Sec > Pre$ and $Sec > Fir$ and $Sec > Suc$) or ($Sec < Pre$ and $Sec < Fir$ and $Sec < Suc$) **then**
9: //The second entry is a min or max
10: Delete Fir from Log table
11: **else**
12: //First and second are neither min or max. Aggregate them by taking the average
13: Avg :=Get average value of Fir and Se
14: $AvgDateTime$:=Get average date time of Fir and Sec
15: Update Fir in database with the new $AvgValue$ and $AvgDateTime$
16: Delete Sec from Log table
17: **end if**

Example 5: In this example we show how this algorithm behaves differently based on the input. Table 9 shows the value of previous, first, second, and next tuples and based on them the action done by the module.

Previous	**First**	**Second**	**Next**	**Action**
2	4	3	2	Delete Second
2	4	5	2	Delete First
5	4	4	2	Update First, Delete Second

Table 9: Actions of the aggregation module for example 5

5 Experimental Evaluation

5.1 Setup

In order to prove whether the algorithm we proposed in this thesis provides an acceptable result we created several tests. Such that we have been able to compare the result of test queries with data which was compressed in different granularities. Moreover we tested the performance of the solution we proposed.

For all our tests we used the following setup:

- Windows 7 x64 - core i7 860 @ 2,8Ghz - 8 GB Ram
- Microsoft SQL Server 2008 R2
- 4 inverters
- Data of a year (about 1.000.000 variable logs)
- 12 different variables (see Table 10
- Multiple granularities based on the variable and the age of the log (see Table 11)

Id (PK)	**Name**	**VariableUnit Id**
1	DailyEnergy	1
2	Frequency	2
3	GridCurrent	3
4	GridPower	1
5	GridVoltage	4
6	ILeak	3
7	InputCurrent	3
8	InputVoltage	4
9	Temperature	5
10	TotalEnergy	1
11	PowerLeak	1
12	InputPower	1

Table 10: Data in the variables table

Variable Id	Min age in days	Entries per hour
1	30	36
2	30	36
6	30	36
9	30	36
10	30	36
3	30	72
4	30	72
5	30	72
7	30	72
8	30	72
11	30	72
12	30	72
1	180	24
2	180	24
6	180	24
9	180	24
10	180	24
3	180	48
4	180	48
5	180	48
7	180	48
8	180	48
11	180	48
12	180	48

Table 11: Test data in the granularity table

5.2 Performance Evaluation

In order to prove the performance we created several scenarios. First of all we measured the runtime of the algorithm when compressing the whole data set (see Table 12).

Data	Runtime
Completely uncompressed	3 hours 48 minutes
Only half of data compressed	1 hour 23 minutes
Only last day to compress	27 seconds
No data to compress	21 seconds

Table 12: Runtime test of the algorithm

The runtime performance of our solution is acceptable. Usually such algorithms should be run regularly (every night, every week). And even if there is more data to compress the algorithm is relatively fast.

5.3 Quality / Usability Evaluation

In order to prove that the compression of the data does not affect the overall look of the chart to much we tested it by comparing the chart resulting from the original data with the one resulting from the compressed data. The figure 5 was generated with the original / uncompressed data. The figure 6 was generated with the compressed data. Both charts show the input power of an inverter over 11 hours. We see that the chart which was generated with the compressed data is very similar with the one which was generated from the original data. Only a few details are lost but the peaks are still the and the overall look is the same.

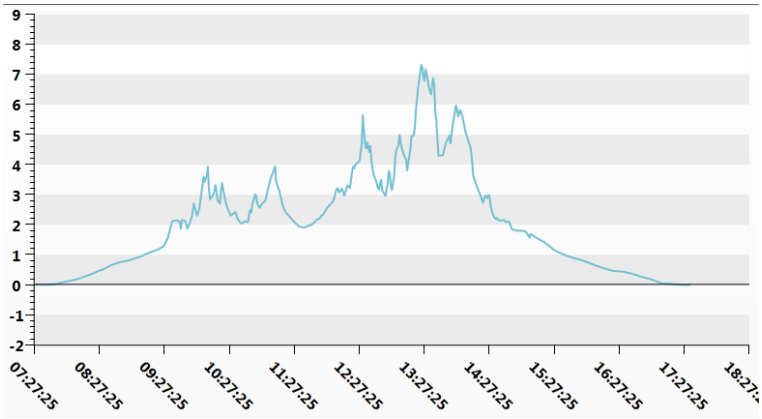

Figure 5: The day chart with the original data

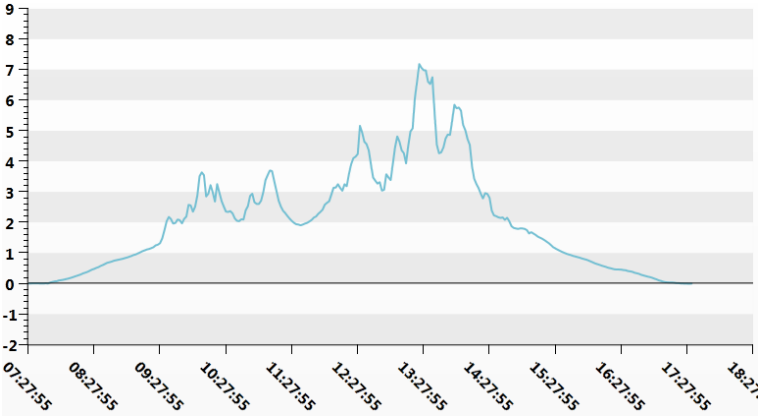

Figure 6: The day chart with the compressed data

Since our solution keeps all the maximal and minimal points we can generate charts which are grouped by day, month, or year without loosing any information. In figure 7 we see the chart which was generated with the original data and in figure 8 we see the chart which was generated with the compressed data. They show the produced energy grouped by day over a month. Both charts are equal. And this is exactly what was the target for this thesis.

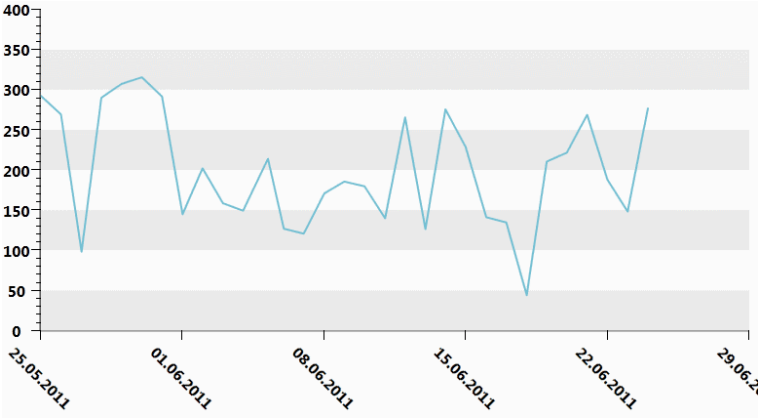

Figure 7: The chart (grouped by day) with the original data

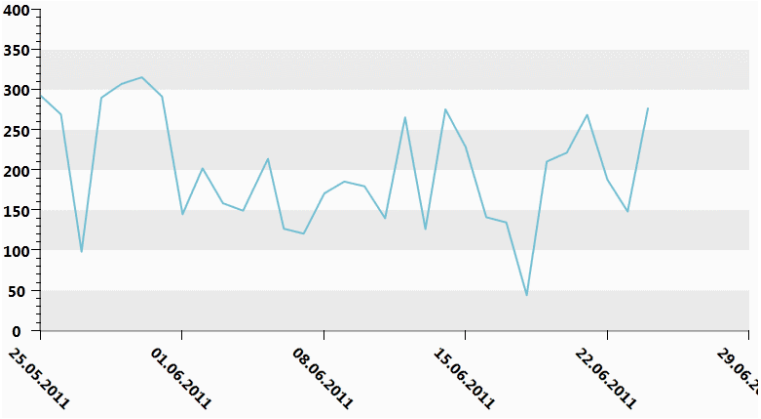

Figure 8: The chart (grouped by day) with the compressed data

6 Conclusion and Future Work

This thesis introduced the compression algorithm for time series data using multiple granularities. It showed that it is applicable for different types of data and that it is scalable for large data-sets. The main focus on the algorithm was to merge / compress the data series without changing the result too much. Such that it was necessary to not just focus on single points but we had to consider the overall trend of the curve in order to choose the right aggregation strategy. We proposed different aggregation strategies which can be used depending on the type of the data which should be compressed. Usually the aggregation strategy which also considers the adjacent points should fit best for data where the result has to be very close to reality. If it is a performance critical task where the result must not be of such a high quality then the aggregation module could be exchanged with another one.

We implemented the compression algorithm with T-SQL, Microsoft's and Sybase's extension for the SQL standard, in a Microsoft SQL Server 2008 R2 database system. But it could also be implemented in other database systems as well.

Future work could include the following tasks. First of all the aggregation strategies could be extended to not just consider the adjacent point but to consider the overall trend of the data. Another possible task would be to optimize the algorithm to obtain a better performance.

References

[1] Scott South and Adam Krumbein, *The Future of Data Acquisition: Will the Internet's Cloud-Computing Replace the Data Logger?* `http://www.stevenswater.com/articles/future_of_datalogging.aspx`

[2] Donghui Zhang, Dimitrios Gunopulos, Vassilis J. Tsotras and Bernhard Seeger, *Temporal Aggregation over Data Streams using Multiple Granularities* Computer Science Department, University of California

[3] M. R. Henzinger, S. Rajagopalan and P. Raghavan, *Computing on Data Streams*. TechReport 1998-011, DEC, 1998

[4] D. Zhang, A. Markowetz, V.J. Tsotras, B.Seeger and D. Gunopulos, *Efficient Computation of Temporal Aggregates with Range Predicates*. Proc. of PODS, 2001

[5] B.Moon, I.F. Vega Lopez and V. Immanuel, *Efficient Algorithms for Large-Scale Temporal Aggregation*. IEEE Transactions on Knowledge and Data Engineering, 2003

[6] C. Bettini, X. S. Wang and S. Jajodia, *Time Granularities in Databases, Data Mining and Temporal Reasoning*. Springer, 2000

[7] N. Kline and R.T. Snodgrass, *Computing temporal aggregates*. Proceedings of 11th International Conference on Data Engineering (ICDE95), Taiwan 1995

[8] C. Bettini, X. S. Wang and S. Jajodia, *Temporal Semantic Assumptions and Their Use in Databases*. IEEE TKDE 10(2), 1998

7 A. Appendix: Compression Algorithms in T-SQL

```
CREATE PROCEDURE [dbo].[AnalyzeData]
AS
BEGIN
SET NOCOUNT ON;
IF EXISTS(SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES WHERE
    TABLE_NAME = 'temporaryTable1') DROP TABLE temporaryTable1;
SELECT rank() OVER (ORDER BY MIN(Timestamp) + (Device_Id*1068) + (
    Variable_Id*671) + (StringId*586)) as 'Rank', MIN(Timestamp) AS
     'Timestamp', COUNT(*) AS 'Entries', [dbo].[GetMaxEntriesOfDay
    ](Variable_Id, StringId, MIN(Timestamp), MAX(Timestamp)) AS '
    MaxEntries', Device_Id, Variable_Id, StringId INTO
    temporaryTable1 FROM VariableLogs
GROUP BY Device_Id, Variable_Id, StringId, DATEPART(YEAR, Timestamp
    ), DATEPART(MONTH, Timestamp), DATEPART(DAY, Timestamp)
HAVING COUNT(*) > [dbo].[GetMaxEntriesOfDay](Variable_Id, StringId,
     MIN(Timestamp), MAX(Timestamp))
END
```

Figure 9: This procedure analyzes the log data (Analysis module)

```
CREATE FUNCTION [dbo].[GetMaxEntriesOfDay](@VariableId INT,
    @StringId INT,  @StartUpTime DATETIME, @ShutdownTime DATETIME)
RETURNS INT
AS
BEGIN
        DECLARE @AgeInDays INT
        DECLARE @LengthInHours INT
        DECLARE @EntriesPerHour INT
        SELECT @AgeInDays = [dbo].[GetAgeInDays](@StartupTime)
        SELECT @LengthInHours = DATEPART(HOUR, @ShutdownTime) -
            DATEPART(HOUR, @StartUpTime)
        IF @LengthInHours = 0
                SELECT @LengthInHours = 1;
        SELECT TOP(1) @EntriesPerHour = EntriesPerHour FROM
             VariableLogGranularities
                WHERE VariableId=@VariableId
                AND AgeInDays <= @AgeInDays
                ORDER BY AgeInDays DESC;
        IF @EntriesPerHour IS NULL
                RETURN 2147483347;
        RETURN @LengthInHours * @EntriesPerHour;
END
```

Figure 10: This function gets the maximal entries per day of the given data series (Granularity module)

```
CREATE PROCEDURE [dbo].[CompressDataSeries]
AS
BEGIN
IF EXISTS(SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES WHERE
    TABLE_NAME = 'temporaryTable1') DROP TABLE temporaryTable1;
SELECT rank() OVER (ORDER BY MIN(Timestamp) + (Device_Id*1068) + (
    Variable_Id*671) + (StringId*586)) as 'Rank', MIN(Timestamp) AS
     'Timestamp', COUNT(*) AS 'Entries', [dbo].[GetMaxEntriesOfDay
    ](Variable_Id, StringId, MIN(Timestamp), MAX(Timestamp)) AS '
    MaxEntries', Device_Id, Variable_Id, StringId INTO
    temporaryTable1 FROM VariableLogs
                        GROUP BY Device_Id, Variable_Id, StringId,
                            DATEPART(YEAR, Timestamp), DATEPART(
                            MONTH, Timestamp), DATEPART(DAY,
                            Timestamp) HAVING COUNT(*) > [dbo].[
                            GetMaxEntriesOfDay](Variable_Id,
                            StringId, MIN(Timestamp), MAX(Timestamp
                            ))
DECLARE @CurrentRank INT; DECLARE @MaxRank INT;
SELECT @CurrentRank = 1
SELECT @MaxRank = MAX(Rank) FROM temporaryTable1
WHILE @CurrentRank <= @MaxRank
BEGIN
        DECLARE @Day DATETIME; DECLARE @DeviceId INT; DECLARE
            @StringId INT;      DECLARE @VariableId INT; DECLARE
            @Entries INT; DECLARE @MaxEntries INT; DECLARE
            @PairsInTemp3 INT; DECLARE @FirstItemId INT; DECLARE
            @SecondItemId INT
        SELECT @Day = Timestamp, @DeviceId = Device_Id, @StringId =
             StringId, @VariableId = Variable_Id, @Entries =
            Entries, @MaxEntries = MaxEntries FROM temporaryTable1
           WHERE RANK = @CurrentRank;
        EXEC [dbo].[PrepareDataSeriesForAggregation] @Day=@Day,
            @DeviceId=@DeviceId, @StringId=@StringId, @VariableId=
            @VariableId
        SELECT @PairsInTemp3 = COUNT(*) FROM temporaryTable3
        WHILE @Entries > @MaxEntries
        BEGIN
        IF @PairsInTemp3 = 0
        BEGIN
                EXEC [dbo].[PrepareDataSeriesForAggregation]
                SELECT @PairsInTemp3 = COUNT(*) FROM
                    temporaryTable3
        END
        SELECT TOP(1) @FirstItemId = Id1, @SecondItemId=Id2 FROM
            temporaryTable3
        EXEC [dbo].[AggregateItems] @FirstId = @FirstItemId,
            @SecondId = @SecondItemId
        SELECT @Entries = @Entries -1;
END
SELECT @CurrentRank = @CurrentRank + 1
END
DROP TABLE temporaryTable1
END
```

Figure 11: This procedure compresses the log data (Compression module)

```
CREATE PROCEDURE [dbo].[PrepareDataSeriesForAggregation]
@Day DATE,
        @DeviceId INT,
        @StringId INT,
        @VariableId INT
AS
BEGIN
        SET NOCOUNT ON;
        DECLARE @MinId INT
        DECLARE @MaxId INT

        IF EXISTS(SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
            WHERE TABLE_NAME = 'temporaryTable2') DROP TABLE
            temporaryTable2;
                select rank() OVER (ORDER BY Timestamp) as 'Rank',
                    Id, Timestamp, Value into temporaryTable2 from
                    VariableLogs WHERE Timestamp >= @Day AND
                    Timestamp < DATEADD(DAY, 1, @Day) AND Device_Id
                     = @DeviceId AND StringId = @StringId AND
                    Variable_Id = @VariableId

        SELECT @MinId = MIN(Id), @MaxId = MAX(Id) FROM
            temporaryTable2

        IF EXISTS(SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
            WHERE TABLE_NAME = 'temporaryTable3') DROP TABLE
            temporaryTable3;
        SELECT t1.Id AS Id1, t2.Id AS Id2 INTO temporaryTable3 FROM
             temporaryTable2 t1, temporaryTable2 t2 WHERE
                    t2.Rank = t1.Rank + 1 AND t1.Id != @MinId AND t2
                        .Id != @MaxId
                ORDER BY ([dbo].[GetSeconds](t2.Timestamp) - [dbo
                    ].[GetSeconds](t1.Timestamp))
END
```

Figure 12: This procedure prepares the given data series for the aggregation (Selection module)

```
CREATE PROCEDURE [dbo].[AggregateItems]
        @FirstId INT,   @SecondId INT
AS
BEGIN
DECLARE @PreviousValue FLOAT
        DECLARE @FirstValue FLOAT
        DECLARE @SecondValue FLOAT
        DECLARE @NextValue FLOAT
        DECLARE @Rank INT
        SELECT @Rank = Rank FROM temporaryTable2 WHERE Id=@FirstId;
        SELECT TOP(1) @PreviousValue = Value FROM temporaryTable2
            WHERE Rank < @Rank ORDER BY Rank DESC;
        SELECT TOP(1) @FirstValue = Value FROM temporaryTable2
            WHERE Id=@FirstId;
        SELECT TOP(1) @SecondId = Value FROM temporaryTable2 WHERE
             Id=@SecondId;
        SELECT TOP(1) @NextValue = Value FROM temporaryTable2 WHERE
              Rank > @Rank + 1 ORDER BY Rank ASC;

        IF (@FirstValue > @PreviousValue AND @FirstValue >
            @SecondValue AND @FirstValue > @NextValue)
                OR (@FirstValue < @PreviousValue AND @FirstValue <
                    @SecondValue AND @FirstValue < @NextValue)
        BEGIN
                -- The first item is either min or max
                DELETE FROM VariableLogs WHERE Id=@SecondId;
                DELETE FROM temporaryTable2 WHERE Id=@SecondId;
                DELETE FROM temporaryTable3 WHERE Id2=@SecondId;
                RETURN;
        END

        IF (@SecondValue > @PreviousValue AND @SecondValue >
            @FirstValue AND @SecondValue > @NextValue)
                OR (@SecondValue < @PreviousValue AND @SecondValue
                    < @FirstValue AND @SecondValue < @NextValue)
        BEGIN
                -- The second item is either min or max
                DELETE FROM VariableLogs WHERE Id=@FirstId;
                DELETE FROM temporaryTable2 WHERE Id=@FirstId;
                DELETE FROM temporaryTable3 WHERE Id2=@FirstId;
                RETURN;
        END

        DECLARE @AverageDateTime DATETIME
        DECLARE @AverageValue FLOAT
        SELECT @AverageDateTime = CAST(AVG(CAST(Timestamp AS float)
            ) AS datetime) , @AverageValue = AVG(Value) FROM
            VariableLogs WHERE Id=@FirstId OR Id=@SecondId
        UPDATE VariableLogs SET Timestamp = @AverageDateTime, Value
             = @AverageValue WHERE Id = @FirstId;
        DELETE FROM VariableLogs WHERE Id=@SecondId;
        UPDATE temporaryTable2 SET Timestamp = @AverageDateTime
            WHERE Id = @FirstId;
        DELETE FROM temporaryTable2 WHERE Id=@SecondId;
        DELETE FROM temporaryTable3 WHERE Id2=@SecondId;
END
```

Figure 13: This procedure aggregates the two given tuples (Aggregation module)