GSN Cloud Storage and Processing

Yifan SUN
Section of Communication Systems, EPFL
yifan.sun@epfl.ch
Supervisors: Jean-Paul Calbimonte, Karl Aberer
Distributed Information Systems Laboratory LSIR
January 27, 2015

ABSTRACT
GSN (Global Sensor Networks) is capable of managing configurable virtual sensors through a wide range of wrappers, and is able to manage one-shot and continuous queries, even in a distributed environment with several GSN instances. However, each GSN instance runs on a single machine, and uses a relational-based data storage underneath. While in most medium-size sensor deployments this is just enough, when it comes to process very large numbers of sensor observations, and at very high incoming rates, scalability can become a problem at various stages. The project aims at integrating Spark Streaming, an extension of the core Spark API, with GSN to boost query processing of streams in a multi-node environment and achieve better scalability. We show the feasibility of our approach and demonstrate its scalability through two applications: linear segmentation and anomaly detection: discovering trend of weather data and identifying occasions when live temperature data is delivering unreasonable values.

Keywords
Global Sensor Networks, Spark Streaming

1. INTRODUCTION
Global Sensor Networks (GSN) is a middle ware aimed at fast deployment and application development for heterogeneous platforms, it supports flexible integration and discovery of sensor networks and sensor data, enables fast deployment and addition of new platforms, provides distributed querying, filtering, and combination of sensor data[1]. GSN instances can work distributed on different machines, however, when there are large numbers of sensor observations at very high incoming rates, it becomes hard for GSN to process the incoming streams in real time, i.e. scalability is the main issue for GSN.

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. The essential idea of Spark Streaming is to treat streaming computations as a series of deterministic batch computations on small time intervals[2]. The input data received during each interval is stored across the cluster, processed via deterministic parallel operations, and the result or intermediate states are stored in resilient distributed data sets (RDDs), an efficient storage abstraction that avoids replication by using lineage for fault recovery[3].

The goal of this project is to integrate GSN and Spark in order to achieve scalability, enabling large number of observed streaming data placed in distributed machines to be processed consistently in real time. In order to show the scalability effects of Spark on GSN, we devise two applications. The first is linear segmentation, which is quite practical in clustering and classification of time series sensor data. Another is anomaly detection, the automatic identification of irregular data which do not conform to other patterns in our data set, thus can be used to detect failures of sensors or unusual weather.

In the parallelization experiment, we obtain air temperature data from 2011 to 2014 of sensors placed at seven Wannengrat weather stations in Davos, Switzerland from GSN website[1]. Then we conduct a live processing on the data for both segmentation and abnormal points detection.

The remainder of this report is structured as follows. Section 2 and Section 3 focus on our approaches for linear segmentation and abnormal data detection. Section 4 describes our implementation with Spark, Section 5 presents and discusses results of anomaly detection as well as processing speed of GSN. Whereas Section 6 draws final remarks and proposes potential amendments.

2. LINEAR SEGMENTATION
In the first example, we parse data into segments that maximally approximates the trend of data using Piece-wise Linear Representation. Intuitively Piece-wise Linear Representation refers to the approximation of a time series T, of length n, with K straight lines. This preprocessing could be useful in many senses. For example, in order to automatically detect irregular weather data with sudden rise/drop or being unusually high/low, piece-wise linear approximation can support fast similarly comparison and change point detection. Second, after segmentation, we can use only two coordinates to approximate groups of data with same trend, which tremendously compressed the volume of data and requires less storage.

According to Keogh, Eamonn et al, there are three major approaches for linear segmentation[4]:

- Sliding window: Given a predefined max error, a segment keeps incorporate new data until the linear ap-

[1]http://montblanc.slf.ch:22001/data.html

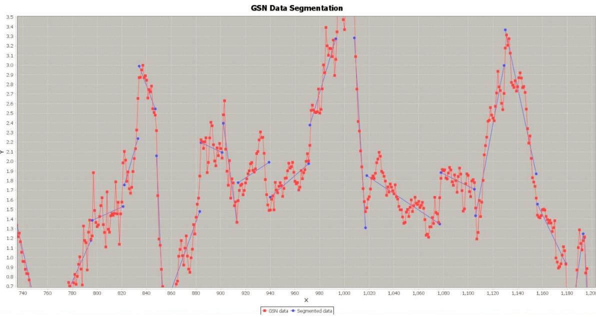

Figure 1: GSN Data Segmentation

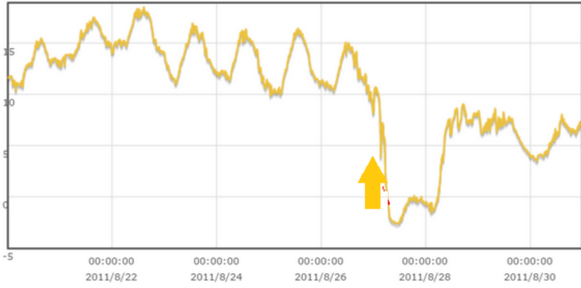

Figure 2: Sudden temperature drop example

proximation exceeds the error bound. The procedure repeats from the last data not included in the newly approximated segment. Sliding Windows algorithm is able to deal with on-line data, but behaves poorly due to its inability to look ahead.

- Bottom-up: Start from the finest possible approximation, then iteratively merge the lowest cost pair if not exceeding the predefined max error. It achieves better results but cannot deal with online data.
- Top-Down: Works similar to Bottom-Up but starts from one segment to finest.

To work with online data and retain the superiority of Bottom-Up, we adopted the Sliding Window and Bottom-up (SWAB) algorithm proposed by Keogh, Eamonn et al[4]. Implementation of the algorithm consists of two parts:

Determine error bound. As explained above, both Sliding Window algorithm and Bottom-Up algorithm need a parameter max_error as termination criterion. Choosing an appropriate parameter largely decides the effect of linear approximation, i.e. with a large max_error, each segment is likely to enroll more data and gives a very coarse approximation, whereas a small value tends to produce an over-fragmented approximation. As there is no error bound that works precisely for all segments, we are looking for one that correctly determines the order of magnitude for different data.

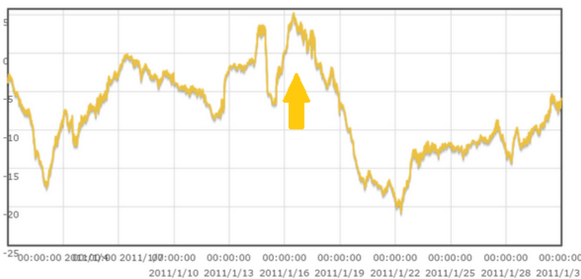

Figure 3: Continuous temperature drop example

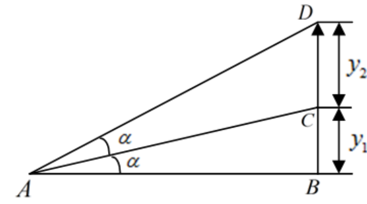

Figure 4: Illustration of comparison

The method we adopt is as below: for each time interval, we calculate a max_error by first randomly choosing a start point, with its successor being end point. Then for the interval between start point and end point, calculate the average linear regression errors for the whole interval as well as first three fourth of it, denoted as error_whole_average and error_threefourth_average respectively. If error_whole_average is twice larger than error_threefourth_average, break the procedure and record the previous error as max_error. Otherwise increase end point and repeat until the interval incorporates one third of data in a day. This procedure is conducted three times and the average is used as max_error.

With the max_error obtained above, we then fill the buffer with segments obtained using Sliding Window algorithm. The buffer size is chosen such that there is enough space to accommodate about 5 or 6 segments. We then do a refinement for these segments by applying Bottom-Up, start by separating data in the buffer into groups of two, and repeatedly merge two successive segments with lowest approximation error until exceeding the error bound. The leftmost segment is output and stored in a Map, with its key being the start index of this segment, and its value being a list storing the coordinates of start and end points of this segment. For example, say the leftmost segment starts at (1, 1.1) and ends at (10, 9.9), after linear regression, the approximated values at 1 and 10 are (1, 1) and (10, 10). The item is then stored in the Map as <1, [1, 1, 10, 10]>. The Map structure is used later for fast positioning in the Abnormal Data Detection. The segment is then removed from buffer, and new segment is incorporated using Sliding Window. This procedure is repeated until no new data streaming in, then all the left segments are output and reported.

Figure 1 displays some resulted segments samples. The red dots stand for original data points, and blue lines indicate the linear approximation for these data. As can been seen, the algorithm correctly provides satisfactory match with the temperature data.

3. ANOMALY DETECTION

Table 1: Missing data example

timestamp	value
2011-08-18T 10:50:00.000+0100	13
2011-08-18T 11:00:00.000+0100	13.52
2011-08-18T 11:50:00.000+0100	10.71
2011-08-18T 12:00:00.000+0100	13.08

Table 2: Suspected abnormal data for sensor wannengrat1 in 2011

start time	end time	difference	angle
2011-07-08 02:00	2011-07-08 06:30	-0.272	-15.222
2011-07-08 06:40	2011-07-08 08:10	0.629	30.776
2011-07-13 17:30	2011-07-13 19:40	-0.314	-16.802
2011-08-03 11:50	2011-08-03 14:20	-0.476	-25.636
2011-08-03 14:30	2011-08-03 17:20	0.286,	16.003
2011-08-18 11:00	2011-08-18 11:30	-0.069	-3.966
2011-08-27 00:10	2011-08-27 05:20	-0.182	-10.268
2011-08-27 05:30	2011-08-27 07:20	-0.408	-22.653
2011-08-28 06:10	2011-08-28 11:00	0.164	9.182
2011-09-06 05:40	2011-09-06 09:10	0.226	12.571
2011-09-11 15:20	2011-09-12 01:10	-0.179	-10.072
2011-09-20 18:50	2011-09-20 22:20	0.428	23.361
2011-10-07 01:30	2011-10-07 03:20	-0.463	-24.492

Online data sources exhibit various types of errors. The errors that we are concerned can be classified into two types:

- Connectivity errors: It refers to the inability to get any data, the reason may due to server faults and network faults. Such failures result in missing data or irregular rate of receiving data, as shown in Table 1 on 2011/08/18.
- Semantic errors: It refers to unreasonable values of data, which could result in erroneous sensor or unusual weather. According to the duration of such errors, we can divide them into sudden rise/drop, as on 2011/8/27 in Figure 2, and continuous rise/drop, as from 2011/1/16 in Figure 3.

Our work aims to provide general solutions for both Connectivity and semantic problems. For typical supervised classification problem, labeled data is required in order to learn a classifier, which is a difficult task in our case. Since our goal is to simply detect abnormal trends of temperature data, we can assume the data is normal most of the time, and is noisy because it is real world data. Consequently the detection of anomaly data can be done by comparing slopes of data segments with corresponding periods in a successive of neighboring days.

The comparison is described as below: For each segment obtained using SWAB algorithm, look for segments belonging to the same periods of time for a successive of 20 neighboring days. Then calculate the slope for each corresponding segment with restored data and return the average slope for comparison. Since all segments do not have same lengths, there exists two possibilities for finding corresponding segments:

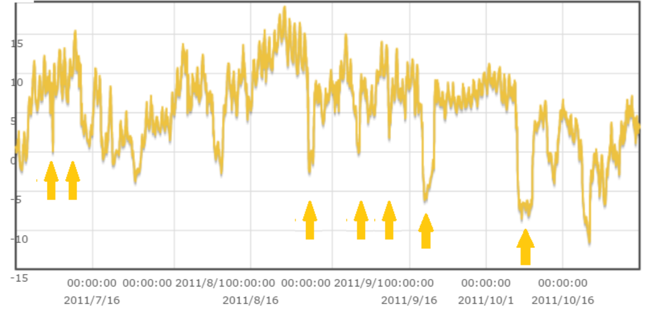

Figure 5: Detected abnormal weather

- The corresponding start and end points belong to a same segment, say T. In this situation, the slope of T is obtained as result.
- The corresponding start and end points belong to multiple segments, say T_1, ...,T_n. In this situation, we use the restored approximated data in these intervals to reconstruct a linear regression and obtain a new slope.

Take an example for illustration: wannengrat sensors report data every 10 minutes, resulting in 144 data in a day. Say now we want to check whether the segment <2000, 10.0, 2050, 15.0> (coordinates of start and end points) is irregular or not. We first obtain the slope of this segment as 0.1. Then we iteratively search for same periods in 20 neighboring days. We begin by searching for segments with start index being 2000 - 144*10 = 560 and end index being 2050 - 144*10 = 610. There could be two possibilities of the found results:

1) If the segment containing 560 and 610 happens to be <550, y1, 620, y2>, then the slope for this segment can be easily obtained as (y2 - y1)/(620 - 550).
2) If the data is spanned in several segments, for example <550, y1, 600, y2>, <601, y3, 620, y4>, then we use all the approximated data indexed from 560 to 610 for linear regression and return the resulted slope.

Next we iteratively increment indexes of start and end point by 144 until accumulating 20 days. After obtaining the average slope and the slope of the segment, the intuition is to compare the cosine similarity between two vectors. However, as can be seen from Figure 4, the same angle between vectors doesn't necessarily indicate same vertical distances (in Figure 4, BC and CD apparently have different lengths despite same α). Consequently, when the slopes of both vector are large (as with AC and AD), the difference y_2 has to be significant large in order to be reported as abnormal, whereas vectors with small slopes (as AC and AB) need only a relatively small distance y_1. The unfairness can be solved by using:

$$ratio = \frac{CD}{AB} = \frac{AB * slope_{AD} - AB * slope_{AC}}{AB}$$

$$= slope_{AD} - slope_{AC}$$

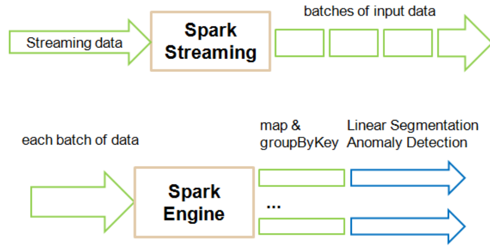

Figure 6: Workflow of system

Table 2 presents the results of suspected abnormal data for sensor wannengrat_wan1 in from July to October in 2011. We check the time stamp for each incoming data, and store -40 for missing data. This explains the sharp changes on 2011/08/18. The original data for the suspected time is depicted in Figure 5, as can been seen, the suspected data generated by the method is quite reasonable. One remaining problem is that our method is quite sensitive to our predefined threshold, i.e. larger value yields fewer but more accurate results whereas smaller value is more strict in detecting irregular data, which however may result in over fitting.

4. SPARK INTEGRATION

To execute the program, Spark Streaming receives the data stream from GSN, divide it into one second batches and store them in Sparks memory as RDDs (see Figure 6).

```
JavaReceiverInputDStream<String> inputdata
= jssc.socketTextStream("localhost", 9999);
```

It then invoke RDD transformations map and groupByKey to preprocess the RDDs and group data according to their sources. These transformations will produce a new RDD with the updated groups. Each DStream in the program thus turns into a sequence of RDDs.

```
inputdata.mapToPair(
 new PairFunction<String,String,String>(){
 public Tuple2<String,String> call(String t){
   // emit <name, string> pair;
  }
}).groupByKey();
```

After some preprocessing (e.g. checking indexes and data completion) for each group, We then parallelize Linear Segmentation and Anomaly Detection on each RDD by calling foreach() function.

```
foreachRDD(
 new Function<JavaRDD<String>,Void>() {
  public Void call(JavaRDD<String> v1){
   v1.foreach(
    new VoidFunction<String>(){
     public void call(String t){
      // do segmentation and validation;
     }
    });
   return null;
  }
});
```

Table 3: Speed Up and Efficiency

#cores	2	3	4	5	6	7
speed up	1.424	1.556	1.746	1.894	1.948	1.957
Efficiency	0.712	0.518	0.436	0.379	0.325	0.279

Spark is built on top of Mesos[5], a "cluster operating system" that lets multiple parallel applications share a cluster in a fine-grained manner and provides an API for applications to launch tasks on a cluster. In our experiment since the workload of data is not too large, we are sufficient to handle them on local machine with multiple cores. This is specified when we submit jobs in the Spark shell:

```
./bin/spark-submit --name "SparkStraming"
--master local[8] myApp.jar
```

In order to achieve better scalability, we can increase the size of data sets and deploy Spark on clusters.

5. RESULTS AND ANALYSIS

The parallelization experiment setup is as follows: there are 25 datasets, each containing one year's weather data from one wannengrat sensor. We send all these data to Spark concurrently with sending rate 2 milliseconds. We installed Spark on local machine with a maximum of 8 cores, and conduct experiments with 2, 4 and 8 cores 10 times respectively and record the average execution time.

Figure 7 shows the frequencies of dates with suspected abnormal weather, only dates with frequencies above 3 are depicted. We assume the higher frequency a day is suspected, the more likely that the weather that day is abnormal. As shown in the figure, July 8, 2011 and August 27, 2011 are suspected with highest frequencies. We present the original data of sensor wannengrat_wan1 of July and August in Figure 8. As can be seen, there are sudden drops of temperature of over 10 degrees on both days which are quite unusual as compared with other summer days. This demonstrates the feasibility of our method.

In order to verify the scalability of our system, we use different number of cores on local machine for experiment, i.e. 2 cores, 4 cores and 8 cores. Since one core is allocated to each receiver in Spark, and in our case there is one socket stream, using 2 cores is basically sequential processing. For each number of cores, we run the execution for 10 times and record the average execution time for comparison. The average execution time is shown in Figure 9.

With the actual execution time, we then calculate the speed up and efficiency for different number of cores, as recorded in table 3. As can be seen from the results, with more cores, we are achieving shorter execution time, however, the speed up ratio is not following the ideal ratio, i.e. increase proportional to number of cores. By applying Amdahl's law, we can obtain the parallel fraction for our system:

$$f = (1 - \frac{T(P)}{T(1)})/(1 - \frac{1}{P}) = 0.54$$

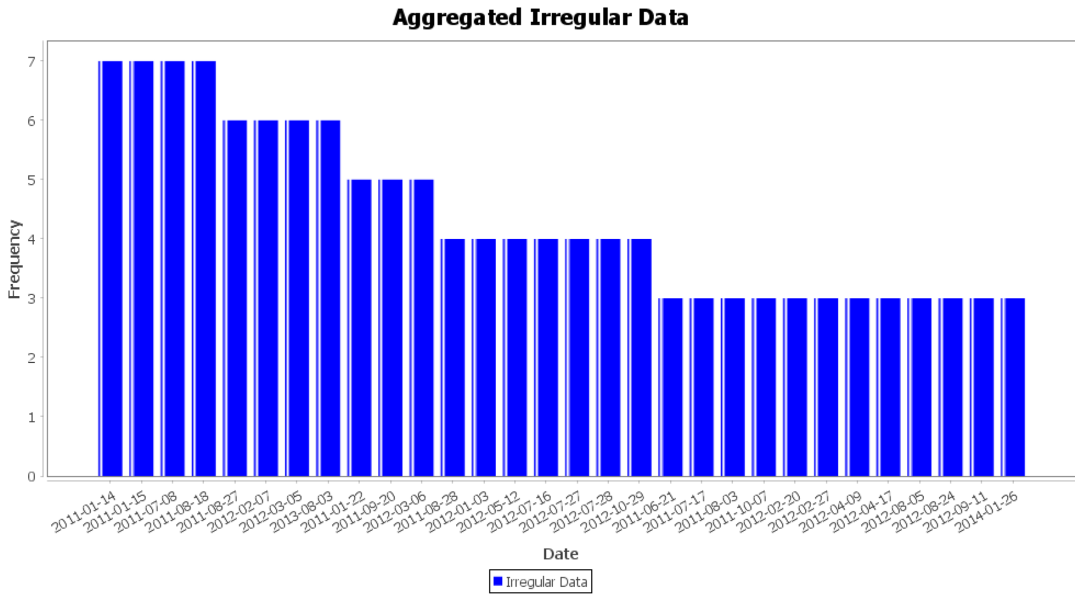

Figure 7: Aggregated Frequencies of Abnormal Weather

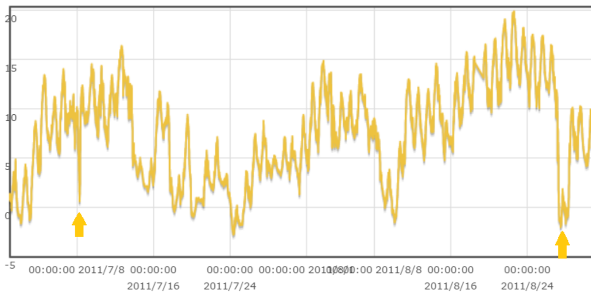

Figure 8: Sample of Abnormal Weather

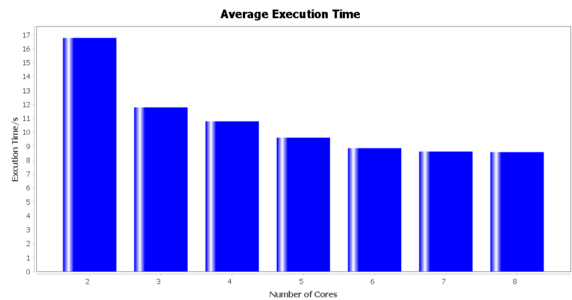

Figure 9: Comparison of Total Execution Time

As scalability performance requires a high degree of parallelization, i.e. f being close to 1. However, in our case, we only achieved f of 0.5. According to Amdahl's law, this should conform to the line marked in red in Figure 10.

The reason why the parallel fraction is so small is that the non-parallelization time accounts for a large portion of total time. We examined the average execution time for the foreach function in each batch, which consists of approximately 10 to 50 milliseconds on processing, and approximately 2 to 10 milliseconds on scheduling. In this case the scheduling can take up to a quarter of the execution time, and thus decreases the parallel fraction. Since Spark is designed for high-throughput processing of massive amounts of data on potentially large clusters. In our case of a single machine running on modest input data size, we are not winning much in the execution time against scheduling time. To overcome this constraint, future work requires testing with larger quantity of data and delegating tasks to clusters running on the Mesos or YARN cluster managers to achieve better scalability.

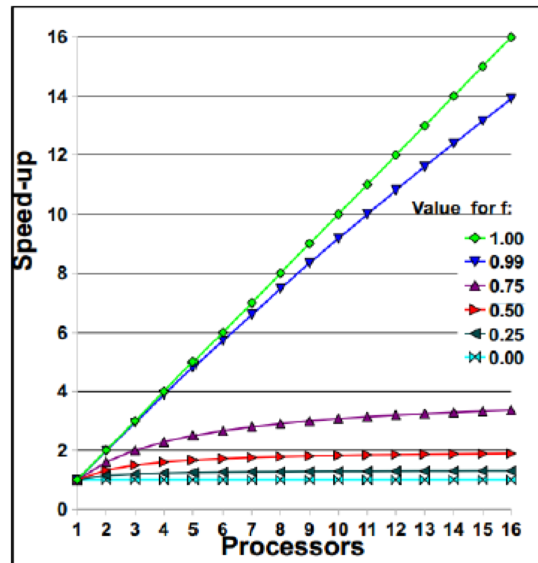

Figure 10: Amdahl's law

6. CONCLUSION

In this project, we presented the integration of Global Sensor Network, a middle ware managing configurable virtual sensors through a wide range of wrappers, together with Spark Streaming, a stream programming model for large clusters that provides consistency and scalability. We demonstrate its feasibility through two examples: linear segmentation and abnormal data detection. Due to the small quantity of data, we conduct experiments on local machine with 2 to 8 cores, and achieved a maximum of 2 times of speed up in processing time with multiple cores, thus accomplished our goal of real-time processing.

In the experiment with our small amount of data, we notice that the scheduling time can take up a quarter of the execution time for each batch, thus leading to small parallel fraction and non-satisfactory scalability result. In the future, we are interested in process large quantity of data, and deploy Spark on clusters running on the Mesos or YARN cluster managers in order to achieve better scalability.

7. REFERENCES

[1] Aberer, Karl, Manfred Hauswirth, and Ali Salehi. *A middleware for fast and flexible sensor network deployment* Proceedings of the 32nd international conference on Very large data bases. VLDB Endowment, 2006.

[2] https://spark.apache.org/docs/latest/streaming-programming-guide.html

[3] Zaharia, Matei, et al. *Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters* Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing. USENIX Association, 2012.

[4] Keogh, Eamonn, et al. *Segmenting time series: A survey and novel approach* Data mining in time series databases 57 (2004): 1-22.

[5] Hindman, Benjamin, et al. *A common substrate for cluster computing* Workshop on Hot Topics in Cloud Computing (HotCloud). Vol. 2009. 2009.