Decreasing the computation time that's used for saving data in the DEMKit simulator

Birte Brunt
University of Twente
PO Box 217, 7500 AE Enschede
The Netherlands
b.w.brunt@student.utwente.nl

ABSTRACT

The DEMKit simulator used too much computation time for saving data instead of calculating and controlling actions. This paper focusses on how the DEMKit simulator can be improved so that the computation time used for saving data is reduced. Related work suggested that other databases perform worse for the DEMKit simulator, so the implementation was adapted to improve the simulator. A new approach for saving data locally was implemented, together with an optimization of the maximum buffer size. The computation time for saving data was reduced to 23% of the original time. The optimal buffer size appeared to be 40.000 datapoints, unless more real time updates are preferred.

Keywords

Optimization, Timeseries Database, Smart Grids, Simulator

1. INTRODUCTION

The University of Twente developed a simulation tool to simulate smart grids, the DEMKit simulator [9]. This DEMKit simulator is, among others, used by one of the departments of this University, namely the Computer Architecture for Embedded Systems (CAES) group [8]. With the future in mind, they are trying to optimize the energy balance in smart grids in order to make it more accessible to use sustainable energy to a greater extent.

Up to now, communication between the power generator and the power consumer has always been one way. The information went from power generating units and utilities to the consumers. This was fine as long as the power was generated at a central point and was distributed from there to the consumers. With sustainable energy, consumers can also generate power by using decentralized methods, such as solar panels or windmills. When the line between consumer and generator fades, grids have to support two way communication so that every part of the network can receive and send information. Smart grids support this function and are therefore necessary to implement sustainable energy to a greater extent. [6]

To optimize these energy balances in smart grids, the CAES group runs various simulations. Each simulation contains different compositions of different households and each household consists of multiple devices, which all have their own components. So, to get an accurate analysis about what happened during a simulation, a lot of data needs to be saved about all these different elements. When saving the data generated during these simulations takes a long time, this process becomes unnecessarily time-consuming. Unfortunately, that is exactly the problem the CAES group is struggling with at the moment.

In order to solve this problem we have to ask ourselves the question: "How can the DEMKit simulator be improved so that the computation time used for saving data is reduced?". Broadly, there are two sub-questions that result from the main research question, namely: "Can the computation time for saving data be reduced by using another database?" and "Can the computation time for saving data be reduced by adapting the implementation of the DEMKit simulator?"

2. BACKGROUND

2.1 State of the art

The DEMKit simulator is an application which is written in Python. During a simulation, a loop of three different steps is running, namely the *preTick* the *timeTick* and the *logValue*. The simulator uses a synchronized clock with a fixed discrete time step to simulate the time progress. For each timestamp, one iteration of the loop runs. [2]

The three different phases all have their own tasks. The *preTick* phase makes sure all entities update their current state, all the calculations and controlling of actions take place within the *timeTick* phase and the *logValue* phase logs all data. This last phase is the most important one when talking about saving data.

The *logValue* phase consists of two parts. The first thing (Part 1 in Figure 1) this function does is saving all the statistics about the different components and the overall statistics locally. The function goes over every single component of the simulation and adds its statistics one by one to a shared string. Thereafter (Part 2 in Figure 1), it has to be decided whether this data has to be saved in the database yet or that it can be done in a later stage in the simulation. This decision depends on how much data is being stored locally at that moment. When the amount of data is more than the initialized buffer can handle, the data has to be written to the database and the shared string is emptied. If not, the locally saved data passes on to the next iteration of the loop. There is also the possibility to force writing data to the database after each iteration.

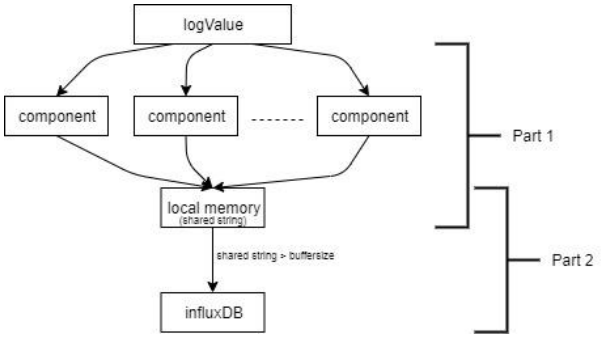


Figure 1: logValue phase

2.2 Related work

The DEMKit simulator uses an Influxdb [5] as database. This is a timeseries database which means that it is optimized for saving lots of values indexed by time. The CAES group is using Grafana [3] as an interface to analyze all the data saved in the database. Grafana takes queries as input to show the right values in one of the many visualizations offered. The combination of the database and the visualization makes it easy to save and analyze data generated by the DEMKit simulator.

According to Di Martino et al.[7], timeseries databases (in particular Influxdb) perform better at ingestion, retrieval and disk performances. The only test where the Influxdb was not the best performer (but was still working), was at the execution time of queries by non-temporal attribute filtering. For the DEMKit simulator, this test is not relevant, as it only uses temporal attributes. In addition, the only test that is truly relevant for the computation time used during a simulation, is the ingestion performance. This shows that it is a reasonable choice to use a timeseries database for the DEMKit simulator.

But Influxdb is not the only timeseries database out there. DB-Engines [1] has made a ranking of different timeseries databases based on popularity. Of the 32 databases, the Influxdb is in the first place. This comparison does not directly say anything about the performances of these databases. Yet, the Influxdb is four times as popular as the second most popular database, which is probably not without any reason.

Influxdata wrote some tutorial code [4] to get used to python in combination with the Influxdb. In this code there is one main client object and different classes which extend this class. These children are different examples on how to use the parent class. There is one important difference between the code of Influxdata and the code of the DEMKit simulator, namely the way of saving data locally. In the DEMKit simulator there is only the possibility to save this data in one big string. Influxdata on the other hand has two possibilities, namely by a list of JSON objects or as a list of string objects. These lists are being translated to one big string just before communication with the database takes place.

3. METHOD

Since related work suggests that other databases perform worse for the DEMKit simulator, we focus on adapting the implementation of the DEMKit simulator rather than searching for another database.

The following steps need to be taken:

1. Finding the bottlenecks in the existing implementation.
2. Searching for new approaches to achieve the same result, but with less computation time.
3. Integrating these new approaches into the old implementation.
4. Testing whether the new implementation takes less computation time.

To *find the bottlenecks* in the existing implementation we investigate the time needed for the different parts in the logValue phase. This is done by using the datetime module. With this module, the current timestamp can be obtained. Subtracting the timestamps before and after the execution of a part of the code gives us the time that was needed for this part during one iteration. The aggregate time of all the time intervals for the same part of the code, is the time that was needed for this part during the entire simulation. The parts that take the most time are seen as the bottlenecks.

To *search for new approaches* we explore the Influxdata's implementations for more specific objectives than before. If this does not give the desired outcome, more sources should be checked to see if a solution can be found.

To *integrate these new approaches* we adapt the implementation of the DEMKit simulator according to the findings of the step before.

To *test* whether the new implementation takes less computation time, we run the same simulation with the old implementation as well as with the new implementation. The computation time is measured in the same way as mentioned in the first step.

To perform steps one and four we use two simulations. These are already written by the CAES group and are included in the example simulations of the DEMKit simulator. One of them is a simulation about just one household (demohouse), this one is suitable to find the bottlenecks. To compare the two implementations, we also use the simulation about a street of households (demostreet). In this way, we can see the result in a smaller and a bigger simulation.

4. ANALYSIS

The logValue phase is broken down to seven different parts. The total time of these different parts are measured and printed at the end of each simulation. The results in Table 1 are an average of three runs of the demohouse simulation. The results per run can be found in Appendix A.

Function	Time in sec
Log devices	15,920182
Log meters	4,027849
Log controllers	11,358850
Log flow	0,022013
Networkmaster	0,017027
Log overall stats	5,682247
Time to write to database	8,054594

Table 1: Average computation time per function of the logValue phase, running demohouse

As shown in the Table 1, logging statistics about the devices and the controllers is taking the most time and can be seen as the bottlenecks of the current simulator. Taking a closer look at these functions, it becomes clear that the main functionality of these functions is to save data locally. Or in other words, extending the shared string takes too much computation time.

5. NEW IMPLEMENTATION

To improve the old implementation, a new approach for saving data locally should be implemented. As noticed before, Influxdata saves local data as a list of strings or JSON objects instead of one big string. When taking a closer look at this implementation, the difference with the DEMKit simulator code is in the used operator. The DEMKit simulator uses the plus operator to extend the string, where Influxdata's implementation uses the append operator to add new values to

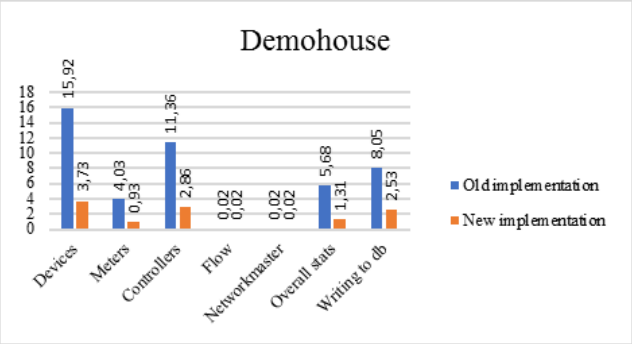
the list. The difference between these two is that the plus operator has to create a new object to save the result, whereas the append operator adapts the first given object. So instead of saving both strings again, the append operator only saves the additional string once more. Because the first string becomes larger each time a new part of the data is added, it is highly beneficial when this string does not need to be saved over and over again. In order to reduce the computation time, the shared string is replaced by a list of strings, so that the append operator can be used instead of the plus operator.

During the implementation of this new approach, another aspect showed up, namely the buffer size. This buffer size determines how many times, and how much data is sent to the database. When the buffer size decreases, more queries need to be sent to the database. When the buffer size increases, the queries become larger and the lists that need to be parsed into strings are longer. To see if it makes a difference in the overall time, several buffer sizes are tested. The buffer size cannot increase endlessly when using the default maximum body size (25.000.000 bytes) for a query.

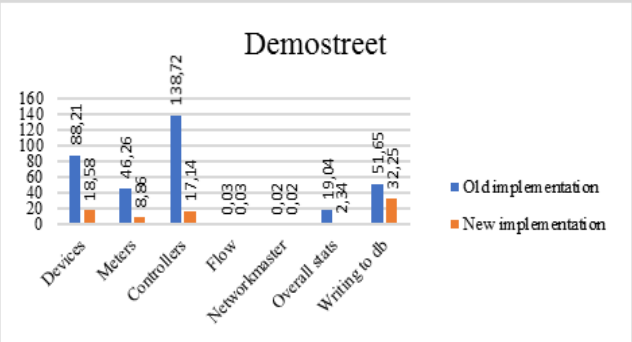
6. RESULTS

Both the simulations demohouse and demostreet have simulated a period of seven days. The demohouse simulation was simulating just one household, the demostreet simulation was simulating ten households.

In Graph 1 and Graph 2, the results of the new implementation with a buffer size of 100.000 datapoints are shown. These are the rounded averages of three runs of the simulations. In Appendices B, C and D all individual results are shown.

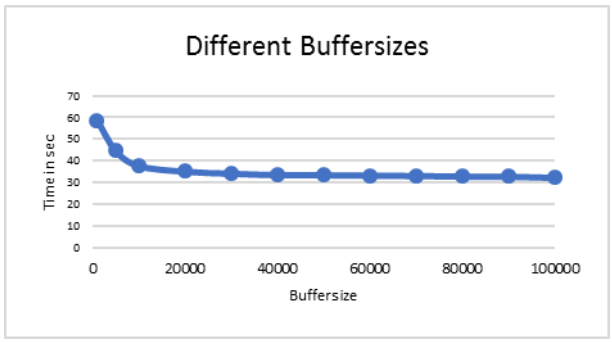


Graph 1: Average computation time of the old and the new implementation, running demohouse



Graph 2: Average computation time of the old and the new implementation, running demostreet

In Graph 3, the results of the new implementation with different buffer sizes are shown. The computation time on the y-axis is the computation time needed to write data to the database since this in the only function affected by the buffer size. In Appendix E all individual results are shown.



Graph 3: Computation time needed for writing to the database with different buffer sizes, running demostreet

7. DISCUSSION

As shown in the first two graphs, the computation time for saving data decreased significantly. Not only the functions that were defined as the bottlenecks decreased in time, but most other functions in the logValue phase as well. Based on the results in Graph 2, the total computation time for saving data is reduced to only 23% of the initial computation time. For the functions that were defined as the bottlenecks, these percentages are even more consequential. The computation time for logging devices is reduced to 21% and the computation time for logging controllers is reduced to 12%.

The networkmaster is the only part that did not decrease in computation time. This can be easily clarified, since this is the only part that does not save data locally and therefore does not benefit from the new approach to save data.

The different buffer sizes also affect the computation time for saving data. The part where data is written to the database decreases in computation time when the buffer size increases. These factors are exponentially related with a horizontal asymptote around the 30 seconds. At a buffer size of 40.000 datapoints, a computation time of 33,508435 seconds is already obtained. Increasing the buffer size even more does not have significant beneficial effect anymore and only results in less real time updates. The optimal buffer size therefore is around 40.000 datapoints, unless more real time updates are preferred.

Further research on this topic should start at reducing the founded asymptote of 30 seconds for writing data to the database. This is now the function that takes the most computation time for saving data, namely 41%.

8. CONCLUSION

So to get back to the research question, we succeeded in adapting the implementation of the DEMKit simulator in such a way that the new implementation takes only 23% of the initial computation time for saving data. The optimal buffer size to use is around the 40.000 datapoints. A new database was not the best solution since the Influxdb is already a good functioning database.

For future research, writing the data to the database is thought to play an important role. For now, the minimal time this

takes is around 30 seconds, what comes down to 41% of the total computation time needed for saving data. Different versions of the Influxdb can be compared to see if this affects the computation time.

9. REFERENCES

[1] DB-Engines. *DB-Engines Ranking of Time Series DBMS*, Jan. 2020. Accessed on: Jan. 26, 2020. [Online] Available: https://db-engines.com/en/ranking/time+series+dbms

[2] G. Hoogsteen, J. L. Hurink and G. J. M. Smit, "DEMKit: a Decentralized Energy Management Simulation and Demonstration Toolkit," *2019 IEEE PES Innovative Smart Grid Technologies Europe (ISGT-Europe)*, Bucharest, Romania, 2019, pp. 1-5.

[3] Grafana Labs. *The open observability platform*, 2020. Accessed on: Jan. 26, 2020. [Online] Available: https://grafana.com/

[4] InfluxData. *Python client for InfluxDB*, Dec. 5, 2019. Accessed on: Jan. 26, 2020. [Online] Available: https://github.com/influxdata/influxdb-python

[5] InfluxData. *Real-time visibility into stacks, sensors and systems*, 2020. Accessed on: Jan. 26, 2020. [Online] Available: https://www.influxdata.com/

[6] P. Bansal and A. Singh, "Smart metering in smart grid framework: A review," *2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, Waknaghat, 2016, pp. 174-176.

[7] S. Di Martino, L. Fiadone, A. Peron, A. Riccabone and V. N. Vitale, "Industrial Internet of Things: Persistence for Time Series with NoSQL Databases," *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, Napoli, Italy, 2019, pp. 340-345.

[8] University of Twente. *Computer Architecture for Embedded Systems*, May 28, 2019. Accessed on: Jan. 26, 2020 [Online] Available: https://www.utwente.nl/en/eemcs/caes/

[9] University of Twente. *DEMKIT*, Apr. 5, 2019. Accessed on: Jan. 26, 2020 [Online] Available: https://www.utwente.nl/en/eemcs/energy/demkit/

10. APPENDICES

A. Old implementation, demohouse

Function	Time in sec per run			Average time in sec
Log devices	15,579139	16,178368	16,003038	15,920182
Log meters	3,925075	4,102371	4,056101	4,027849
Log controllers	11,169033	11,527038	11,380480	11,358850
Log flow	0,022032	0,019988	0,024020	0,022013
Network master	0,016010	0,015012	0,020058	0,017027
Log overall stats	5,535880	5,786798	5,724063	5,682247
Time to write to database	7,834503	8,136058	8,193222	8,054594
Total execution time of the simulation	69,459577	71,895333	71,421037	70,925316

B. New implementation, demohouse

Function	Time in sec per run			Average time in sec
Log devices	3,836833	3,701561	3,636695	3,725030
Log meters	0,933531	0,922622	0,931586	0,929246
Log controllers	2,947401	2,789468	2,829963	2,855611
Log flow	0,016043	0,013050	0,018023	0,015705
Network master	0,017976	0,019011	0,017992	0,018326
Log overall stats	1,347923	1,313996	1,253081	1,305000
Time to write to database	2,519025	2,587528	2,488507	2,531687
Total execution time of the simulation	38,069910	37,222012	36,863541	37,385154

C. Old implementation, demostreet

Function	Time in sec per run			Average time in sec
Log devices	84,608669	88,712221	91,318820	88,213237
Log meters	44,416374	47,112617	47,264375	46,264455
Log controllers	136,378303	140,033254	139,750894	138,720817
Log flow	0,024020	0,040020	0,031021	0,031687
Network master	0,024017	0,021017	0,028025	0,024521
Log overall stats	18,705274	19,325533	19,075414	19,035407
Time to write to database	50,196540	53,448676	51,308209	51,651142
Total execution time of the simulation	477,595220	495,843843	490,949723	488,129595

D. New implementation, demostreet

Function	Time in sec per run Buffer size of 100.000 datapoints			Average time in sec
Log devices	18,167656	18,507509	19,055039	18,576735
Log meters	8,707016	8,827776	9,044233	8,859675
Log controllers	16,747438	17,093942	17,566794	17,136058
Log flow	0,023989	0,024015	0,027022	0,025008
Network master	0,026022	0,017989	0,030022	0,024678
Log overall stats	2,271807	2,305772	2,428685	2,335421
Time to write to database	31,907216	31,926355	32,920332	32,251301
Total execution time of the simulation	207,113820	210,468600	215,209788	210,930736

E. New implementation with different buffer sizes, demostreet

Function	Buffer = 1.000 datapoints	Buffer = 5.000 datapoints	Buffer = 10.000 datapoints	Buffer = 20.000 datapoints
Time to write to database	58,766383	44,639182	37,671154	35,088432

Function	Buffer = 30.000 datapoints	Buffer = 40.000 datapoints	Buffer = 50.000 datapoints	Buffer = 60.000 datapoints
Time to write to database	34,178852	33,508435	33,475678	33,274320

Function	Buffer = 70.000 datapoints	Buffer = 80.000 datapoints	Buffer = 90.000 datapoints	Buffer = 100.000 datapoints
Time to write to database	33,128551	32,837697	32,766992	32,251301