Arion: A Model-driven Middleware for Minimizing Data Loss in Stream Data Storage

Nhan Nguyen, Mohammad Maifi Hasan Khan, Yusuf Albayram, Kewen Wang, Swapna Gokhale
Department of Computer Science and Engineering, University of Connecticut
Email: {nhan.q.nguyen, maifi.khan, yusuf.albayram, kewen.wang, swapna.gokhale}@uconn.edu

***Abstract*—In large-scale data stream management systems, sampling rate of different sensors can change quickly in response to changed execution environment. However, such changes can cause significant load imbalance on the back-end servers, leading towards performance degradation and data loss. To address this challenge, in this paper, we present a model-driven middleware service (i.e., Arion) that uses a two-step approach to minimize data loss. Specifically, Arion constructs models and algorithms for overload prediction for heterogeneous systems (where different streams can have different sampling rates and message sizes) leveraging limited execution traces from homogeneous systems (where each stream has the same sampling rate and message size). Subsequently, when an overload condition is predicted (or detected), Arion first leverages the a priori constructed models to identify the streams (if any) that can be split into multiple substreams to scale up the performance and minimize data loss without allocating additional servers. If the software based solution turns out to be inadequate, in the second stage, the system allocates additional servers and redirects streams to stabilize the system leveraging the models. Extensive evaluation on a 6 node cluster using Apache Cassandra for various scenarios shows that our approach can predict the potential overload condition with high accuracy (81.9%) while minimizing data loss and the number of additional servers significantly.**

***Keywords*-Cassandra; Data Stream Management; Middleware**

I. Introduction

With significant advancement in wired and wireless sensor technologies and wide adoption of Internet connectivity, researchers are exploring increasingly complex and large-scale applications such as battlefield monitoring, smart grid monitoring, infrastructure and asset management, and IoT applications [1], just to name a few. Due to the nature of these applications, design of efficient data stream management systems is becoming increasingly important to ensure the reliability and performance of such systems. However, in these systems, rapid changes in the execution environment can require to vary the sampling rates of different groups of sensors deployed in the field, causing significant load imbalance on the data stream management systems, leading towards performance degradation and data loss [2].

Among various approaches that attempted to address this (or similar) problem in the past, common strategies include resource allocation based on execution time estimation [3], future resource requirement prediction based on past execution behavior [2], and optimization techniques for resource provisioning targeting specific objectives (e.g., QoS requirements, SLA) [4]. Load shedding technique [5] is another common approach that is used to mitigate overload conditions in systems where data streams are processed on the fly [6], [7]. However, this is ill suited especially for systems where data needs to be stored for future analyses [8], [9].

While prior efforts present different mechanisms to predict/detect overload conditions and allocate resources accordingly, they often either allocate additional servers or drop packets to deal with the overload conditions. However, we would like to argue that these approaches are often not optimal and lead to additional hardware resources and/or data loss even when it is not needed. Specifically, in our investigation, we identify that the interaction between the software and hardware can be quite complex, and it is often the software that is causing the bottleneck, and should be addressed first before resorting to hardware based solutions to ensure efficient resource utilization.

Towards that, in this paper we present Arion, a model-driven middleware service that integrates a software based approach with a hardware based solution to minimize data loss while minimizing the number of additional servers. Specifically, in the first step, Arion leverages a priori models developed based on limited execution traces from homogeneous systems (where each stream has the same sampling rate) to predict potential overload conditions in heterogeneous systems (where different streams have different sampling rates), and attempts to address the overload condition by selectively splitting streams first. If splitting streams turns out to be inadequate, at the second stage, it allocates additional servers to further minimize potential data loss. Our extensive evaluation on a 6 node cluster demonstrates that the integrated approach can reduce the data loss rate significantly ($< 3\%$) while reducing the number of additional servers compared to the hardware based solution.

II. Design of the Middleware Service

In large-scale streaming applications (e.g., IoT applications), data is often generated by heterogeneous sensors and is replicated across multiple servers for reliability. In such systems, assuming that a given replicated stream storage system with replication factor of r and n different streams is currently stable, one of the key questions that needs to be answered in real-time is "Will the system become overloaded if the number of streams and/or sampling rates change?" Once a system experiences such changes in execution conditions and the current system is inadequate to handle the changed load, it needs to adapt quickly to mitigate the situation.

2159-6190/17 $31.00 © 2017 IEEE
DOI 10.1109/CLOUD.2017.14

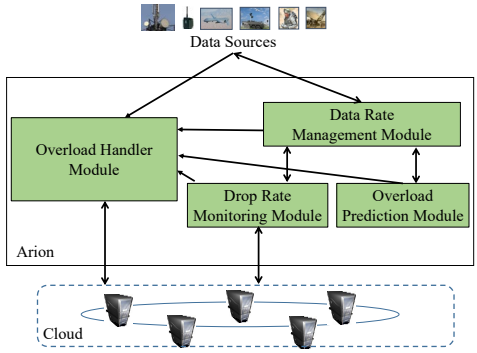

Fig. 1: System architecture.

As the execution condition can change quickly in such systems, any middleware service targeting to minimize data loss should satisfy two key requirements. *First,* it is imperative for the service to be able to predict potential overload conditions in advance so that the system can act in time to mitigate the possible negative effects of such overload conditions (e.g., minimize data loss). *Second,* in addition to be able to predict the overload condition, the middleware service needs to be able to automatically react to the changes to minimize data loss while minimizing the resource overhead.

To address these challenges, in this paper, we use a combination of proactive and reactive approach to minimize data loss caused by sudden changes in execution conditions. Specifically, the proactive approach uses a priori constructed models to predict potential overload conditions. However, as the prediction models may mispredict, we complement the proactive approach with a reactive approach that runs a process in the background monitoring the run time condition to detect overload conditions. Finally, if an overload condition is predicted (or detected), the system uses a two-step solution to minimize the negative effect of the overload condition.

To demonstrate our approach, we built a middleware service (i.e., Arion) as shown in Figure 1. In this framework, the sensor streams are simulated where the sampling rate can be changed in real-time to generate different workload conditions during execution. In Arion, we have four modules, namely, the *data rate management module*, the *overload prediction module*, the *drop rate monitoring module*, and the *overload handler module*.

In this framework, we assume that the *data rate management module* is aware of the changes in sampling rates in the system and triggers the *overload prediction module* whenever a change happens. The *overload prediction module* applies the trained models to predict whether the new combination of streams, sampling rate and message size will cause overload or not, and triggers the *overload handler module* if needed. However, as the model is not perfect and may fail to identify a potential overload condition (i.e., false negatives), the *drop rate monitoring module* is used to monitor the system to see whether there are any dropped messages, which can trigger the *overload handler module* as well. The *overload handler module* uses a two-step approach to minimize data loss. Specifically, first it leverages the constructed models to identify the streams (if any) that can be split into multiple substreams to scale up the performance and minimize data loss without actually allocating additional servers. If the software based solution turns out to be inadequate, in the second stage, the system resorts to traditional resource allocation scheme where it allocates additional servers and redirects streams to stabilize the system. During normal operation, the *overload handler module* does not change any data streams and therefore works just as a data forwarding module.

# Streams	1, 2, 4, 8, 10
Message Size (byte)	1, 100, 300, 500, 1000
Replication Factor	1, 2, 3

TABLE I: Number of streams, message sizes, and replication factor of the system used in the experiments.

The details of our approach are presented below.

A. Constructing Models for Overload Prediction

In this work, for modeling purpose, we consider two scenarios. In the first scenario, all the streams have the same sampling rate $rate$ and message size msg_size, which we refer to as the "homogeneous" system. In the second scenario (the more realistic one), different data sources have different sampling rates and message sizes, which we refer to as the "heterogeneous" system. To keep the training phase tractable, we first build prediction models for homogeneous systems, and then apply the constructed models to predict the performance of heterogeneous systems.

To generate the models for predicting the system capacity for a given load condition, in this work, we consider three variables that can change during run-time (i.e., number of streams, sampling rate, and message size for each stream), and a fourth variable that is usually set at the beginning (i.e., the replication factor). As each variable can have a large number of possible values, leading to an exponential number of combinations, for tractability, we only choose a subset of the combinations and use that information to build models that can be used to predict the potential overload condition in real-time.

To collect the training data, we run experiments with the values listed in Table I. All experiments presented in this paper were performed on a cluster of 6 nodes, where each node is equipped with Intel E5-2440 CPU and 32GB of RAM memory. The network bandwidth is 1 Gbps. We used Apache Cassandra[1] version 2.1.8, and its corresponding client Datastax Java Driver[2] version 2.1.8. We used a ramp up method where we ran each experiment for a predefined amount of time (τ minutes) to collect data. In our case, we ran experiments with different amounts of time (3 minutes, 10 minutes, 30 minutes) and determined that 3-minute run was enough to capture the characteristics of the system. In each experiment, n clients (i.e., # streams in Table I) push data to the storage system concurrently where each client pushes data to an assigned row in the database table. To determine the maximum data

[1]http://cassandra.apache.org/
[2]https://github.com/datastax/java-driver

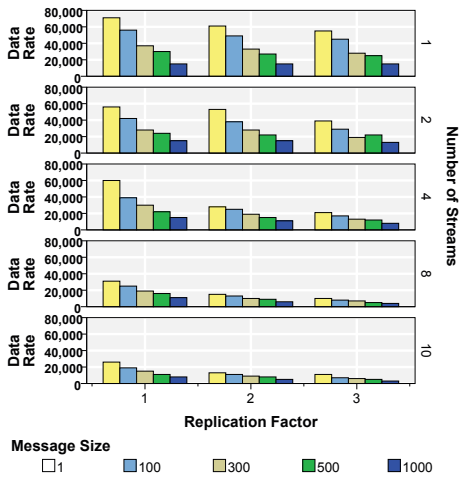

Fig. 2: Maximum data rate for different number of streams. The data rate is in messages/second, the message size is in byte.

rate that the system can handle from n streams, we varied data rate, the value of msg_size (message size in Table I), and the value of r (replication factor in Table I). At the end of each run, we checked how many messages were stored in the database. If client j pushes data at a rate of $rate_j$ for τ minutes, the total number of messages expected in the storage system is $\Sigma_{msg} = rate_j * \tau * 60$ (messages). If the total number of messages (Ω_{msg}) in the corresponding row of that client in the storage system is smaller than Σ_{msg}, we know that the storage system is not able to handle data rate greater than or equals to $rate_j$. The drop rate of the system in that case will be $\frac{\Sigma_{msg}-\Omega_{msg}}{\Sigma_{msg}}(\%)$. Over multiple iterations of increasing $rate_j$, we determine the maximum data rate for a combination of condition that can be handled by the system. We repeated each experiment 5 times and set the maximum data rate equal to the average of those 5 runs. Figure 2 shows the maximum data rate that the system can handle for different number of streams, message sizes, and replication factor. Each sub-figure (from top to bottom) is generated for a specific number of streams. In each sub-figure, each plot (from left to right) is generated for a different replication factor. In each plot, each bar shows the corresponding maximum data rate for a specific message size. From the figure, we can see that, as we increase the number of streams, the maximum data rate reduces significantly (e.g., 45,000 messages/second for a single stream compared to 7,000 messages/second for 10 streams with a message size of 100 bytes). For each value of replication factor, when we increase the message size (the left bar to the right bar in each plot), the data rate also reduces quickly.

In our training set, the total number of combinations we consider is 75 (e.g., 5 different values for the number of streams, 5 different values for message size, and 3 different values for replication factor). Based on the collected data, we used linear regression to train the model that is used to predict the maximum data rate in our system as follows.

$$log(predicted\_rate) = \theta_1 \#_{streams} + \theta_2 log(msg\_size) + \theta_3 r + \epsilon \quad (1)$$

The input to the model is the number of streams $\#_{streams}$, their message size msg_size, and the replication factor r. The unit of $predicted\_rate$ is messages/second, the unit of msg_size is byte. Due to the variability of $predicted\_rate$ and msg_size, their data was log-transformed in order to approximative a normal distribution. In this work, we used the least-square approach to fit the performance model (1). The model parameters are θ_1 = -0.145, θ_2 = -0.324, θ_3 = -0.141, and ϵ = 11.811. The p-values for the three independent variables are all significant at 0.0001 level. The goodness-of-fit metric (R-squared) for the overall model is 0.8583, and significant at 0.0001 level. These results indicate that the model was effective as 85% of the variability of the dependent variable was explained by the independent variables. However, we acknowledge that the high value of R-squared value should be interpreted with caution as the number of data points in the analysis is not large.

B. Overload Prediction Leveraging the Model

We use the constructed model to predict whether the system is likely going to be overloaded for a given execution condition during runtime for homogeneous and heterogeneous systems as follows.

1) Overload Prediction for Homogeneous Systems: In case of a homogeneous system, we simply use the information regarding the number of streams, their message sizes, and the replication factor of the back-end storage to predict the maximum data rate that the system can handle using model (1). We then compare the predicted rate (i.e., $predicted\_rate$) against the actual data rate of the streams (i.e., $real\_rate$) and conclude that the system is going to be overloaded if $predicted\_rate < real\_rate$.

2) Overload Prediction for Heterogeneous Systems: In case of a heterogeneous system, we determine potential overload condition by leveraging model (1) as follows.

If there are n heterogeneous streams with the set of data rates $\mathcal{D}$ (stream i has data rate $d_i \in \mathcal{D}$, i = 1,...,n) and average message size *avg_size*, we first model them as n homogeneous streams with message size *avg_size*. Next, we use model (1) to predict the maximum data rate (i.e., $predicted\_rate$) that the system can handle for these n homogeneous streams. We then find the maximum data rate of $\mathcal{D}$ (i.e., d messages/second), calculate the average rate avg_rate, the standard deviation δ for the data rate of n heterogeneous streams, the difference $\Delta_{rate} = |avg\_rate - \delta|$, and use them to determine whether these n heterogeneous streams may cause overload or not.

Towards that, we define the overload prediction function f as follows.

$$f = \begin{cases} \text{not overloaded,} & \text{if } \frac{d*avg\_rate}{predicted\_rate^2} \leq 1 \text{ (a)} \\ \text{overloaded,} & \text{if } \frac{d*avg\_rate}{predicted\_rate^2} > 1 \text{ (b)} \\ & \text{or } \Delta_{rate} > predicted\_rate \text{ (c)} \\ \text{undetermined,} & \text{otherwise. (d)} \end{cases}$$

In the above equation, cases (a), (b) and (c) can be explained as follows.

Case (a): $avg\_rate \leq predicted\_rate$ and $d_i \leq predicted\_rate\ \forall d_i \in \mathcal{D}$, i = 1,...,n
Case (b) and (c): $avg\_rate > predicted\_rate$ and $d_i > predicted\_rate\ \forall d_i \in \mathcal{D}$, i = 1,...,n
If the value of f is undetermined (Case (d)) or $\Delta_{rate} \leq predicted\_rate$, we need to perform additional checks to ensure that the system is not overloaded. This is due to the fact that the average value of a group can be much smaller than the value of one/more individual element(s) in the group due to skewed distribution, causing Arion to mistakenly determine that a group of streams does not cause overload while some individual stream(s) in the group with large data rate(s) and/or message size(s) can in fact cause overload.
To address this, first, we group all streams which have data rate greater than avg_rate into one group T and sort all the streams in T in non-increasing order of data rate. Assuming that this group has p elements, next we divide them into p different subgroups where group sg_i includes the first i streams from T ($1 \leq |sg_i| \leq p$, i = 1,..,p). Group sg_i (i = 1,..,p) is viewed as a group of i heterogeneous streams, and Arion performs overload prediction for each subgroup separately by following the steps described above (i.e., by considering the above cases). If any subgroup sg_i (i = 1,..,p) is found to cause overload, then the middleware concludes that the original n heterogeneous streams will cause overload as the load of n streams will be higher than the load of the subgroup sg_i.

C. Overload Detection in Real-time

As the prediction models are not perfect and may miss potential overload condition, in addition to using the prediction models, Arion uses a drop rate monitoring module as shown in Figure 1 for detecting data loss in real-time. While early signs of potential overload may be detected by monitoring utilization of certain system resources (e.g., CPU utilization, memory utilization), this is often error-prone and hard to tune the right threshold. Hence, to avoid the possibility of misprediction, in this work, we periodically check the information stored in the database to determine whether data is being dropped or not. In particular, we define a message as a (key, value) pair where key is a unique message ID and value is the value of the message. Message IDs in our experiments are consecutive integers starting from 1. As such, if the current message ID in the storage is x for a client with data rate λ, then if there is no data loss, the message ID will be $x + \lambda{*}t - 1$ after one second interval (t = 1 second). However, if the ID turns out to be smaller than $x + \lambda{*}t - 1$, we know that the server has dropped data, and the overload handler module is triggered.

D. Handling of Overload Condition

Arion uses a two-step approach towards handling the overload condition as follows.

1) Stream Splitting: The first step is inspired based on the observation that software is often the bottleneck which causes the inefficient utilization of hardware resources. In such cases, it is important to address the limitations of the software first before allocating additional hardware resources to address the

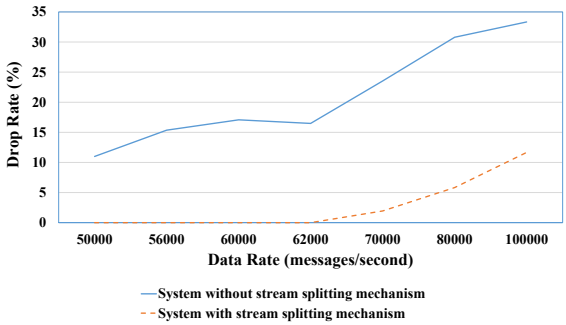

Fig. 3: Drop rate of the system with and without stream splitting mechanism. Replication factor = 3, message size = 100 bytes.

overload condition. For instance, in our study, we observed that the resource utilization with more than one stream is higher compared to a single stream when the system starts losing data, indicating that, in case of a single stream, the system is losing data even when the hardware resources are underutilized. In Figure 2 we can see that the maximum rate per stream for different number of streams that the storage system can handle is not linear, and the total rate generated by two separate streams that can be handled by the system is greater than the maximum rate of a single stream. For example, when replication factor is 3 and message size is 100 bytes, the maximum total rate for two streams is 76,000 messages/second compared to the maximum rate of 45,000 messages/second for a single stream. This phenomenon indicates that, in case of a single stream, the way a stream is handled by Cassandra is causing the bottleneck rather than the hardware itself.
Based on this observation, we investigated the idea of splitting a stream into multiple streams so that the system can achieve better performance by exploiting the way multiple streams are handled by Cassandra. The effectiveness of the stream splitting mechanism for a single stream is demonstrated in Figure 3 where we changed the data rate of one stream from 50,000 to 100,000 (messages/second) and ran each experiment for 3 minutes and recorded the drop rate. We then split one stream into two streams which push data into the same row at the new rate. For example, if the original stream has rate of 60,000 messages per second, we split it into two streams with rate of 30,000 messages per second each. The drop rate of the split streams was then recorded and compared against the drop rate of the original stream. Figure 3 shows that the splitting mechanism helps to reduce the drop rate to 0% in most cases and reduced by more than 6 times in other cases.
While the idea of splitting stream is simple, however, it is not obvious which streams should be split and into how many substreams it should be split. Furthermore, after splitting, each substream can either push data into different rows of the same table or can push data into the same row. The first approach requires us to change the storage schema as it requires us to have a new mechanism to manage the row keys. For example, if a stream is stored in a row with key k, if we split it into two streams, we can store them in rows with key $k:1$ and $k:2$. However, with this scheme, an application (e.g., data analytic application) that uses the data from that stream will

Algorithm 1 *StreamSplit(j)*

1: *isoverloaded* ← **true**
2: Sort the streams based on non-increasing order of $rate_i$ of stream i in S //S: list of streams for server j
3: **while** *isoverloaded* **do**
4: Pick and remove the first stream from S
5: Apply the model to split the picked stream
6: Update the value of *isoverloaded*
7: **if not** *isoverloaded* **then**
8: **return false**//the system is not overloaded
9: **end if**
10: **end while**
11: **return true**//the system is still overloaded

now need to read data from two different rows (i.e., $k:1$ and $k:2$), merge them, and sort them based on their timestamps to get back the original stream.

To avoid that effect on the data analytic applications, if a stream has data rate of *rate*, we split that stream into q substreams where each substream has a rate of *rate*/q. Substream i will include messages with IDs $id+i$, $id+i+q$, $id+i+2q$, ..., where id is the ID of the message, i = 0,..,q - 1. For example, if we split a stream with starting ID 0 into 2 substreams, substream 0 will contain messages with IDs $< 0, 2, 4, 6, ... >$ and substream 1 will contain messages with IDs $< 1, 3, 5, 7, ... >$.

The procedure to split streams is illustrated in Algorithm 1. The run time of the algorithm was in the order of seconds in our experiment (e.g., 10 seconds in the worst case). At step 5 in Algorithm 1, once we split a stream into q substreams, we have n - 1 heterogeneous streams and q substreams. Currently, the optimum value for q is determined empirically by running experiments. In our case, the value of q is set to 2. Next, we apply the same method described in Section II-B2 to predict the performance of these n + q - 1 heterogeneous streams. However, we treat these q substreams as q homogeneous streams where each has the data rate as follows.

$$split\_rate = \sigma(rate/q) \quad (2)$$

Here σ is the parameter representing the performance overhead of the system when we push q substreams into the same row in the data storage system. In our case, the value of σ is 0.8. The value is calculated empirically by comparing q split streams against q homogeneous streams. For example, if we split a stream into 2 substreams, we can determine the maximum rate for the two split streams. We then compare the maximum rate for two substreams against the maximum rate of two homogeneous streams. In Algorithm 1, Arion tries different ways to split the streams to address the overload condition. However, if the overload condition persists even after trying all feasible combinations, Arion uses the stream redirection approach as explained next.

2) Stream Redirection: Once Arion determines that the splitting stream is not going to resolve the overload condition completely, it redirects certain streams to additional servers

Algorithm 2 *ResourceAllocation(n, m)*

//n: number of streams that need to be redirected; m: number of idle servers

1: Sort the streams based on non-increasing order of $\frac{rate*msg\_size}{60}$
2: $M \leftarrow 1$ //number of additional servers so far
3: **for** *i*=1 **to** *n* **do**
4: *assigned* ← **false**
5: **for** *j*=1 **to** *M* **do**
6: **if** server *j* can handle stream *i* **then**
7: add stream *i* to the list of streams for server *j*
8: *assigned* ← **true**
9: break
10: **end if**
11: **end for**
12: **if not** *assigned* **then**
13: **for** *j*=1 **to** *M* **do**
14: add stream *i* to the list of streams for server *j*
15: *isoverloaded* = *StreamSplit(j)*
16: **if** *isoverloaded* **then**
17: remove stream *i* from the list of streams for server *j*
18: **else**
19: break
20: **end if**
21: **end for**
22: **if** *isoverloaded* & $M < m$ **then**
23: wake up an idle server and add stream *i* to the list of streams for that server
24: $M \leftarrow M + 1$
25: **end if**
26: **end if**
27: **end for**

to minimize data loss while minimizing the number of new servers. This part of the solution is an extension of our own prior work [2] where we integrated stream splitting with the stream redirection to minimize the resource need.

Briefly, given a list of m idle servers which are powered on, and a list of n streams with data rate $rate_1$, $rate_2$, ..., $rate_n$, the goal is to find the number of servers M ($M \leq m$) and M-partition $S_1 \cup ... \cup S_M$ of the set $\{1, .., n\}$ such that the set of streams S_i can be handled by server i ($1 \leq i \leq M$).

To minimize the number of additional servers in the above problem, we apply the algorithm presented in Algorithm 2. Briefly, Arion first checks whether server j can handle stream i by applying the method described in Section II-B (line 6 of Algorithm 2). If the model predicts that the server will be overloaded, the stream splitting mechanism is invoked (line 15 of Algorithm 2). However, if the stream splitting fails to mitigate the overload condition, additional servers are allocated as needed (line 23 in Algorithm 2), which is done based on the first fit decreasing approach for the bin packing problem [10]. The main idea is to sort the streams that need

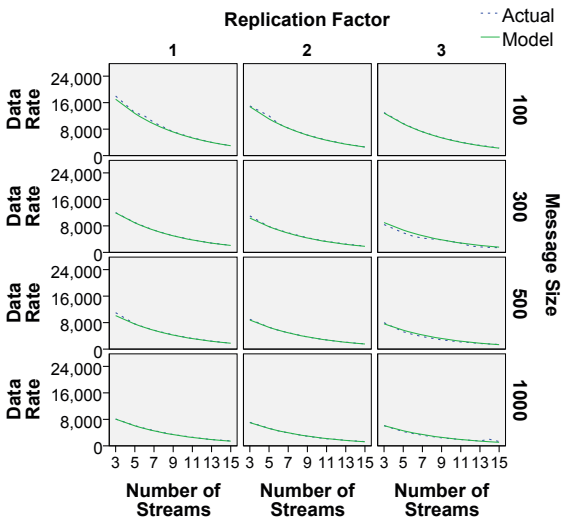

Fig. 4: Predicted value and actual value for maximum data rate. The data rate is in messages/second, the message size is in byte.

to be redirected in non-increasing order of data rate and then assign them to the available servers recursively. Please see [2] for details.

III. Evaluation

In this work, we simulated multiple streams with different data rates and message sizes, and tested the prediction accuracy of our model along with performance improvement in terms of resource allocation and data loss rate. We used Apache Cassandra as the back-end storage platform to store the data which is a distributed storage platform widely used in the industry and by the research community [8], [11]. The overload handler module in Figure 1 either forwards the data which it receives from the data sources to the storage system, splits data into multiple streams or redirects streams if needed. In particular, the overload handler module is a Cassandra client which is built based on Datastax Driver. To split a stream into multiple substreams, it generates new processes with the corresponding data rate to push data to Cassandra. Each data source is handled by a corresponding Cassandra client.

A. Accuracy of the Model for Homogeneous Systems

To test the accuracy of model (1), we compared the predicted maximum data rate for a given execution condition against the maximum data rate that can be handled by the system which was determined through experiments. As we want to avoid the cases of false negatives (e.g., erroneously missing a potential overload condition), if the predicted value is found to be less than or equal to the actual system capacity then we consider the output of the model to be correct. Otherwise, we consider the output of the model to be incorrect as the predicted value will cause the system to continue beyond the system capacity and lose data.

The number of streams in the test case is set to 3, 5, 6, 7, 9, 11, 12, 13, 14, 15. Figure 4 shows the values predicted by model (1) and the actual values for the maximum data rate for different values of replication factor and message sizes. In Figure 4, the results of the model for different replication factors are represented from left to right. Columns 1, 2, and 3 represent the result when the system has replication factor of 1, 2, and 3 respectively. The results for different message sizes are shown from top to bottom. Rows 1, 2, 3, and 4 show the results when the streams have message size of 100 bytes, 300 bytes, 500 bytes, and 1,000 bytes respectively. The accuracy of the model is 84.9% where *accuracy = (true positive + true negative)/(true positive + true negative + false positive + false negative)*.

# streams	4, 6, 8, 10
Message size (byte)	100, 500
Replication factor	1, 2, 3

TABLE II: Number of streams, message sizes, and replication factor of the system used in the test phase for heterogeneous systems.

	Predicted Overload	Predicted Not Overload
Overload	80	20
Not Overload	15.7	84.3

TABLE III: Rate of false positives and false negatives for heterogeneous systems.

B. Accuracy of the Model for Heterogeneous Systems

We use the same model to predict the potential overload condition for heterogeneous systems. In particular, we test the system by varying the number of streams, message size, and replication factor as shown in Table II, which gives us a total of 24 combinations. For each combination, we generate three different sets of data rate for the n streams to cover different cases as explained in Section II-B2. The data rates were selected to ensure that the overload happens in several of these cases. Out of these 72 test cases, there were 40 cases where the system was overloaded, which allowed us to test the rate of false positive and false negative for the overload prediction mechanism. In our work, false positive refers to the scenario where Arion determines a condition as potential overload when it is actually not overloaded. On the other hand, false negative refers to the scenario where Arion determines a condition to be not overloaded when it is in fact overloaded. In each case, we used the maximum data rate predicted by the model to determine whether the system is overloaded or not. Table III shows the rate of false positive and false negative when the system is heterogeneous. The true positive rate of the model in case of heterogeneous system is 80%, which is slightly smaller compared to the true positive rate of the model when we apply it to homogeneous systems (84.9%) due to the variance of data rates. The rate of false positive and false negative of the model are 15.7% and 20%, respectively. The accuracy of the overload prediction mechanism is 81.9%.

To test against more realistic load condition, we also tested the model by generating data streams based on normal and heavy-tail distributions (Pareto distribution). For each distribution, we considered 3 cases where we generated 24 points for each case (Table II) such that the generated points in each case follow that particular distribution. Table V shows the information

Case	# Streams		Data Rate (messages/second)	Stream Redirection
1	8	Original Rate	6000 1000 **8000** **10000** 2000 **10000** 5000 **30000**	No
		New Rate	6000 1000 **4000-4000** **5000-5000** 2000 **5000-5000** 5000 **15000-15000**	
2	10	Original Rate	4000 4000 4000 4000 4000 **10000** **10000** **10000** **10000** **10000**	No
		New Rate	4000 4000 4000 4000 4000 **5000-5000** **5000-5000** **5000-5000** **5000-5000** **5000-5000**	
3	8	Original Rate	**16000** 1000 **18000** 1000 **20000** **10000** 5000 5000	Yes
		New Rate	**8000-8000** 1000 **9000-9000** 1000 **10000-10000** **5000-5000** 5000 5000	
4	10	Original Rate	8000 9000 **10000** 9000 9000 9000 **10000** 8000 8000 8000	Yes
		New Rate	8000 9000 **5000-5000** 9000 9000 9000 **5000-5000** 8000 8000 8000	

TABLE IV: Data rate without ("Original Rate") and with stream splitting mechanism ("New Rate"). A number in bold font in row "Original Rate" ("New Rate") indicates the data rate of a stream before (after) splitting the stream.

Distribution	Normal	Heavy tailed
Case 1	μ = 9,000, σ = 20	α = 1
Case 2	μ = 9,000, σ = 50	α = 2
Case 3	μ = 8,000, σ = 80	α = 3

TABLE V: Distribution parameters.

Distribution	Case	False positive	False Negative
Normal	1	16.8	19.8
Normal	2	15.9	19
Normal	3	17.6	21
Heavy tailed	1	19.9	22
Heavy tailed	2	18.8	21.2
Heavy tailed	3	17.9	20.8

TABLE VI: Rate of false positives and false negatives.

regarding distributions and Table VI shows the rate of false positive and false negative for our model.

C. Effectiveness of the Overload Handling Mechanism

If the prediction model misses potential overload condition, the drop rate monitoring module detects such cases in real-time with 100% accuracy and triggers the overload handler module. However, in such cases, the system suffers initial data loss, which the prediction module tries to prevent and does successfully in more than 80% cases.

Table IV shows 4 cases (selected from 72 cases described in the previous subsection) where Arion applied the stream splitting mechanism to avoid or reduce the number of dropped messages. In Table VII, columns 2 - 4 show the drop rate of the system without and with data loss minimization mechanism for different number of streams. In the second case, the stream splitting mechanism helps to reduce the drop rate to less than 1% (37.49% to 0.5%) without using any additional servers. However, in the last two cases, although the stream splitting mechanism can reduce the drop rate significantly, it is still high, implying that the software solution is not adequate alone in these scenarios. In such cases, the stream redirection mechanism is used to redirect streams selectively to additional servers. The fourth column shows the drop rate when Arion uses the stream splitting and the stream redirection mechanism together, which is about 3% or less.

Next, we present the number of additional servers that is needed when Arion uses only the stream redirection mechanism compared to when it uses the two mechanisms together

Test Case	No Data Loss Minimization Scheme	Stream Splitting	Stream Splitting & Redirection	Stream Redirection	Stream Splitting & Redirection
1	26.27	1.03	-	2	0
2	37.49	0.5	-	2	0
3	74.261	5.09	2.77	2	2
4	39.19	33.74	2.39	4	2

TABLE VII: Drop rate (columns 2 - 4) and the number of additional servers (columns 5 - 6) with different data loss minimization schemes for the test cases shown in Table IV. The unit of drop rate is %.

(columns 5 - 6 in Table VII). In the first two cases, Arion does not need to allocate any additional server when it uses both schemes together and needs two additional servers in the last two cases. None the less, in one of the last two cases, the combined approach requires fewer number of servers compared to the stream redirection mechanism alone, and needs the same number of servers in another case, indicating the effectiveness of the combined approach.

IV. Discussion

The main idea of our work is to address situations where the software causes the bottleneck instead of the hardware. In particular, when the performance does not improve as the workload continues to increase while the system resources remain underutilized, we assume that the software is causing the bottleneck. In such cases, we propose splitting the workload into multiple smaller workloads to increase the parallelism in the system to leverage the unused processing capacity of the hardware in a novel way. We strongly believe that the idea of splitting streams (or similar ideas in other contexts) will work as well.

Finally, we do realize that a typical data center often hosts hundreds to thousands of servers. However, in a distributed key-value storage system with replication factor of r, a node only manages data assigned to it and data of its replica nodes (r - 1 nodes). Hence, even if there are 10,000 nodes in a cluster, a particular node is only affected when the streams it is responsible for are changed in some way (e.g., increase in sampling rate). Based on this observation, a large scale stream processing system can be viewed as a group of multiple small-scale clusters where each of these small clusters is responsible for managing only a limited number of streams. As the performance of the storage system for a particular stream only depends on the small number of nodes responsible for

handling that particular stream instead of all the nodes in the cluster, we used 6 nodes to simulate such a small cluster and evaluated the system by varying the replication factor from 1 to 3 [12], number of streams, sampling rates, and message size.

V. Related Work

Numerous prior efforts looked into the problem of resource allocation and overload management for distributed and stream processing systems. For instance, one of the prior efforts investigated minimum spanning tree based algorithms to discover and allocate resources to meet real-time constraints [14]. Although this work assumes that data arrives continuously and the volume of data is large, it is not designed to handle the sudden changes in workload conditions without incurring initial loss of critical data which is addressed in this paper. Researchers also looked at distributed algorithms for reallocating system resources (i.e., CPU) based on its utilization to maximize the quality of the results in stream processing systems [13].

In addition to investigating the challenge of resource allocation, research community has also looked into the problem of building efficient data management systems for supporting real-time sensor data streams [6]. For example, active warehousing techniques for systems that perform frequent joins between streams and persistent disk relations has been studied in the past [15]. Another work looked at the technique of submodular maximization to summarize a large stream of data "on-the-fly" to achieve high utility value with less computation cost [7]. In addition to these approaches, load shedding technique, which drops excess load to avoid overload condition, has been effectively applied to data stream processing systems in prior efforts [5]. Other notable efforts investigated techniques such as the use of dedicated storage management system for stream processing [16] and summarization of data streams [17].

In contrast to prior efforts, the presented work integrates the task of resource allocation with model-driven stream splitting mechanism, and develops models that can be applied to selectively split and redirect streams to minimize data loss while minimizing the resource need as well, which has not been attempted before.

VI. Conclusion

In this paper, we present Arion, a model-driven middleware service that leverages a priori models developed based on limited execution traces from homogeneous systems to predict potential overload conditions in heterogeneous systems, and attempts to address the overload condition by selectively splitting streams and allocating additional servers as needed. Extensive evaluation on a 6 node cluster demonstrates the superiority of the approach compared to prior efforts. As the solution can be implemented as a middleware service, the presented framework can be easily extended for different data streaming platforms, enabling context-aware management of data acquisition in safety-critical IoT applications.

VII. Acknowledgment

This material is based upon work supported by the Air Force Office of Scientific Research award number FA 9550-15-1-0184 under the DDDAS program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agency.

References

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013.

[2] N. Nguyen and M. M. Hasan Khan, "A Closed-loop Context Aware Data Acquisition and Resource Allocation Framework for Dynamic Data Driven Applications Systems (DDDAS) on the Cloud," *J. Syst. Softw.*, vol. 109, no. C, pp. 88–105, Nov. 2015.

[3] J. Melendez and S. Majumdar, "Matchmaking with limited knowledge of resources on clouds and grids," in *Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, July 2010.

[4] R. Buyya, S. K. Garg, and R. N. Calheiros, "SLA-oriented Resource Provisioning for Cloud Computing: Challenges, Architecture, and Solutions," in *Proceedings of the 2011 International Conference on Cloud and Service Computing*, ser. CSC '11.

[5] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, "Load Shedding in a Data Stream Manager," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03. VLDB Endowment, 2003, pp. 309–320.

[6] S. Nittel, "Real-time Sensor Data Streams," *SIGSPATIAL Special*, vol. 7, no. 2, pp. 22–28, Sep. 2015.

[7] A. Badanidiyuru, B. Mirzasoleiman, A. Karbasi, and A. Krause, "Streaming Submodular Maximization: Massive Data Summarization on the Fly," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14.

[8] N. Nguyen and M. M. H. Khan, "Performance analysis of a fault-tolerant exact motif mining algorithm on the cloud," in *2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC)*, Dec 2013, pp. 1–9.

[9] B. Muhsin, A. Sampath, and T. Gruber, "Systems and methods for storing, analyzing, retrieving and displaying streaming medical data," Sep. 22 2015, uS Patent 9,142,117.

[10] D. S. Johnson, "Near-optimal bin packing algorithms," 1973, phd thesis.

[11] J. S. Van der Veen, B. Van der Waaij, and R. J. Meijer, "Sensor data storage performance: Sql or nosql, physical or virtual," in *5th IEEE Cloud Computing (CLOUD) 2012*.

[12] "Cassandra Data Replication," https://docs.datastax.com.

[13] A. Tang, Z. Liu, C. Xia, and L. Zhang, *Distributed Resource Allocation for Stream Data Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 91–100.

[14] L. Chen and G. Agrawal, "Resource Allocation in a Middleware for Streaming Data," in *Proceedings of the 2Nd Workshop on Middleware for Grid Computing*. NY, USA: ACM, 2004.

[15] A. Chakraborty and A. Singh, "A Partition-based Approach to Support Streaming Updates over Persistent Data in an Active Datawarehouse," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*.

[16] I. Botan, G. Alonso, P. M. Fischer, D. Kossmann, and N. Tatbul, "Flexible and Scalable Storage Management for Data-intensive Stream Processing," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '09.

[17] R. Chiky and G. Hébrail, "Summarizing Distributed Data Streams for Storage in Data Warehouses," in *Proceedings of the 10th International Conference on Data Warehousing and Knowledge Discovery*, ser. DaWaK '08.