Travel Light - State Shedding for Efficient Operator Migration

Espen Volnes,
Thomas Plagemann
University of Oslo
{espenvol,plageman}@ifi.uio.no

Boris Koldehofe
University of Groningen
b.koldehofe@rug.nl

Vera Goebel
University of Oslo
goebel@ifi.uio.no

ABSTRACT

Operator migration is a crucial concept to adapt event processing systems to dynamic changes. When the placement of a stateful operator changes, the operator state must be migrated to the new host. However, operator state size and time constraints can make it impossible to migrate the operator without severe Quality of Service (QoS) degradation. As a relief, we propose to perform state shedding in such a situation. The core idea of state shedding is to partition the operator state, assign a utility to each partial state, and use the utility and size of each partial state to identify the most useful partial states that can be migrated in a given time frame. Thus, state shedding can maintain a substantially higher QoS with a lower impact on query results than state-of-the-art solutions targeting consistent state at the old and new host. In this paper, we define this novel approach and in a simulation environment evaluate state shedding in migration scenarios with pattern-matching queries.

ACM Reference Format:
Espen Volnes,, Thomas Plagemann, Boris Koldehofe, and Vera Goebel. 2022. Travel Light - State Shedding for Efficient Operator Migration. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27-July 1, 2022, Copenhagen, Denmark*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3524860.3539638

1 INTRODUCTION

Stream and event processing systems are of fundamental importance and an integral part of Big Data systems. They support important requirements of Big Data applications to integrate and analyze in real-time high volume data streams which can stem from many distinct and highly distributed data sources. Current stream and event processing systems operate in a highly distributed manner, i.e., the operators in charge of analyzing data streams can be flexibly executed on systems resources in the cloud, at the edge, or even on connected (mobile) devices. This way they can support application requirements regarding Quality of Service (QoS).

System and application dynamics, like bursty input rates, resource contention, and mobility can lead to reduced QoS and require adapting the way operators are executed. Two established methods to react to such changes are *operator migration* [18] and *load shedding* [3, 17, 20]. In operator migration, the placement of operators on resources is changed by migrating one or several operators from their current host (further on referred to as old host) to a new host, which is better suited to meet the required QoS. Load shedding allows reacting to temporary overload situations, by dropping tuples in the input stream or dropping some state of the operator to ensure the operator can process fresh data tuples timely.

Both approaches are effective to deal with overload situations, but they also impose a cost for the distributed operator execution and therefore need to be carefully designed and applied. Operator migration allows changing the resources and this way also the performance, e.g., the processing rate for executing and operator or communication delays for input tuples. Operator migration requires (1) to set up a new resource, the new host of the operator, (2) transmit state from the old host to the new host, and (3) coordinate the handover between the old and new host. As such operator migration can consume temporally redundant resources and increase delays until the new host becomes operational. Load shedding reduces the time to react to overload situations, but dropping tuples and state reduce the accuracy of the results produced by the operators. For longer periods of overload situations, load shedding may therefore be costly in terms of ensured accuracy.

Adapting distributed operator execution approaches mostly treat these two mechanisms as alternatives performed in isolation. We propose and study the combined use of migration and load shedding mechanisms. In particular, we propose to apply state shedding in the course of operator migration to counteract unexpected long delays during operator migrations. State-of-the-art methods aim to atomically transfer the entire operator state based on good estimates of the transfer cost. Contrary, in this paper, we observe that the combination of state shedding and migration is promising for operator migration to better adapt to unexpected situations. We propose to counteract abrupt changes, such

ACM ISBN 978-1-4503-9308-9/22/06.
https://doi.org/10.1145/3524860.3539638

as reduced bandwidth and increased transmission latencies, by transferring only the most necessary state. This requires appropriate online migration procedures to prioritize the partial state to ensure a high utility in terms of accuracy and imposed migration and execution delays.

In this paper, we contribute to (1) a novel concept of combining state shedding and operator migration by maximizing the utility of partially migrated state, and (2) an analysis including a first empirical evaluation that illustrates possible advantages of utility-based load shedding in the context of two real-world data sets: the Citi Bike data set [2] and a bus GPS data set from Dublin [1].

2 BACKGROUND

In this section, we introduce background on distributed operator execution, operator migration and load shedding.

2.1 Distributed Operator Execution

In stream and event processing systems, the logic and the computational functions to analyze and transform data streams are given in form of operators, e.g., filter, join, grouping, and pattern detection operators. The operators are commonly organized in a data flow graph, called the *operator graph*. The operator graph models dependencies between operators and data sources in receiving and producing *tuples* from/to specific *streams*. The operators are executed on hosts of the distributed infrastructure. They can also be dynamically migrated between hosts to meet the performance requirements of the application or react to other changes, such as failures. It is important to note that during the execution of an operator on a host, state is built up while performing processing steps on the received input tuples. Such state can be modeled in the form of (1) tuples in input and output queues and (2) so-called partial states [17, 20], which correspond to intermediate results needed to produce output tuples. When adapting the operator execution, e.g., performing migrations or load shedding, managing the operator state is highly important for the resulting accuracy and consistency.

2.2 Operator migration

Operator migration is a mechanism for exchanging the hosts engaged in the distributed operator execution. It requires organizing the state transfer between the old and new host and reorganizing the flow of data streams, also named *data stream management*. A major objective of current operator migration procedures is to ensure consistency, i.e., to ensure the migration of the entire state completes and the resulting migration has no impact on the operator results.

Approaches for performing operator migration can be classified according to their stream management during the state transfer, i.e., in a *single track* or *parallel track* [18]. In single-track migration, the tuples of upstream operators are buffered (at the upstream node, new host, or old host). Therefore, the migration procedure results in a temporary downtime during the handover between the new and old host until all upstream tuples and operator state are transferred consistently.

Parallel-track migration algorithms are able to migrate state without operator downtime by upstream nodes sending tuples to the old and new host [18]. Either the old host continues its executions until the state transfer has been completed or the old host gradually moves state to the new host. These algorithms require temporary duplication of input streams and good connectivity. Under high system dynamics, e.g., slow communication links and drastically reduced bandwidth, these mechanisms can significantly reduce the performance of the distributed operator execution.

2.3 Load shedding

Load shedding is an established mechanism for operator execution to react to overload situations, e.g., as originally proposed for the data stream management system Aurora [3, 16]. In overload scenarios, part of the workload for an operator is dropped to stabilize the system. Most of the literature describes solutions where input tuples are dropped [6–10, 12, 13, 16]. For aggregation operators, the goal is to minimize the relative error of the calculated aggregate. For join operators, the goal is to drop the tuples that eventually join with the fewest tuples. Another method is to drop windows [17] internally, which reduces the number of produced aggregates instead of reducing the aggregates' accuracy. In pattern-matching operators, dropping input tuples is likely to distort the results completely, because individual tuples can determine whether a sequence fulfills a pattern or not. In such cases, a different state-based load shedding mechanism that drops partial states from the operator is a better option. A partial match might or might not result in an output complex event. If the likelihood of the partial match in producing output is low, the entire sequence of tuples might be dropped. This is done for the pattern-matching operator in a few recent works [5, 14, 19, 20]. As a result of load shedding, the consistency may be invalidated, but the accuracy and utility of the query may remain high.

3 PROBLEM STATEMENT

All state-of-the-art operator migration approaches aim to establish a consistent state at the new host. Unforeseen network conditions can prevent a timely transmission of the entire state between the old and new host. Consequently, operators can experience unexpected freeze times before the operator execution can be resumed. This is an inherent limitation of single-track operator migration algorithms.

Figure 1 shows a VANET scenario executing with three roadside base stations running applications and collecting data from passing vehicles. The red base station has a

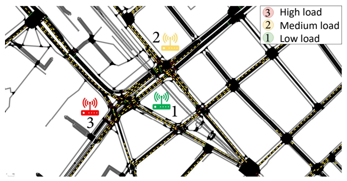


Figure 1: Example VANET scenario

critically high load and needs to reduce it by moving some operators to the green node that has sufficient capacity. In this scenario, the operators deployed on the colored nodes execute operators for detecting collisions, bottlenecks and other traffic situations which need to be timely reported to traffic participants to properly act. Clearly, freezing the operator execution can lead to the situation where traffic participants cannot react while the operator execution is suspended during an unexpected long migration.

Therefore, a better strategy, which we study in this paper, is to limit the effect of delayed migrations by transmitting the most relevant state until the time the operator needs to be resumed. With the help of state shedding, the old host can decide on the most relevant partial state to be transmitted yielding the highest utility for the application, e.g., to react to a possible dangerous traffic event. In this paper, we address the following research questions (RQ):

- RQ1: How to partition operator state in such a way that each partial state is useful for further processing?
- RQ2: How to determine the utility of partial states?
- RQ3: How to select the partial states that can be sent in a given time frame and provide the highest accumulated utility?
- RQ4: How do different approaches for operator migration with state shedding perform?

In the next section, we present the overall approach of operator migration with state shedding. RQ1 and RQ2 are addressed in Section 5. RQ3 is addressed in Section 6 and RQ4 is answered in Section 7.

4 APPROACH

In this section, we present the overall approach that combines operator migration with state shedding. It comprises six steps illustrated in Figure 2 and Algorithm 1. (1) a monitor detects an overload situation or a network problem, which triggers (2) the placement module to determine the new placement and the maximum migration time, and triggers (3) the migration module to extract the current operator state S and to partition S into i partial states, i.e, $S1$ to $S10$. Each partial state is the smallest useful unit for resuming the operator at the new host. (4) The state shedding function determines the utility of each partial state u_i , and (5) selects the most useful partial states, i.e., $S8$, $S5$, $S1$, and $S9$, migrates them to the new host, and drops the remaining partial states. The final Step (6) is to resume the operator at the new host.

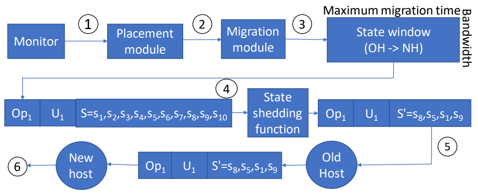


Figure 2: Migration with state shedding in six steps

Algorithm 1 Operator migration from old host (oh) to new host (nh) with state shedding. Abbreviations: operator (op), partial states (p_states), available bandwidth (bw), latency (lat), shedded state (s_state), state shedding function (ls)

```
 1: $trigger\_migration \leftarrow monitor(load, resources)$
 2: $migration\_time \leftarrow calculate\_migration\_time(oh, nh, op)$
 3: $p\_states \leftarrow partition(state)$
 4: $p\_states\_util[S_i, U_i] \leftarrow calculate\_utils(p\_states[S_i])$
5a: $c \leftarrow calculate\_c(oh, nh, op, bw(oh, nh), lat(oh, nh))$
5b: $s\_state \leftarrow ls(start\_time, max\_time, p\_states, bw(oh, nh)lat(oh, nh))$
5c: $migrate(oh, nh, shed\_state, start\_time)$
 6: $resume\_operator(nh)$
```

The optimization problem of selecting the most useful partial states to migrate during the maximum migration time can be reduced to solving the knapsack problem. The objective function is to maximize the utility of the operator's partial states, each with utility u_i and size s_i, subject to a limited capacity c that represents the maximum amount of data that can be sent during the migration.

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} u_i \\ \text{s.t.} \quad & \sum_{j} s_j < c \end{aligned} \tag{1}$$

The success of the solution depends highly on the specific operator semantics which determine how to partition the state and assign utility.

5 STATE PARTITIONING

This section explores how to partition the operator state into partial states of limited size and determine their utility (RQ1 and RQ2). We identify three common stateful operators that differ significantly in how their state manifests: aggregation, join and pattern-matching operators (see example queries in Figure 3). Based on the established design procedures of these operators, we want to analyze how state needs to be represented in order to be be ready for partitioning, and its impact on utility.

An aggregation operator such as in Q1 can record the state as partial aggregates (Figure 3a) that are updated for each tuple that is processed. If it uses a sliding window, it will update multiple aggregates for each tuple. Alternatively, all received tuples can be stored until the end of the window, and the tuples are aggregated (Figure 3b). However, the latter method requires substantially more storage space and can increase the delay of the aggregation.

```
Q1: Sliding window 10 seconds        Q2: Pattern A+B
    Jump every 1 second                  Define A as speed > max(A.speed)
    Group by id                          Group by S1.id

Q3: Pattern A+B                      Q4: Join S2
    Define A as T.temp > 45              on S1.id = S2.s1
    B as H.humidity < 25
```

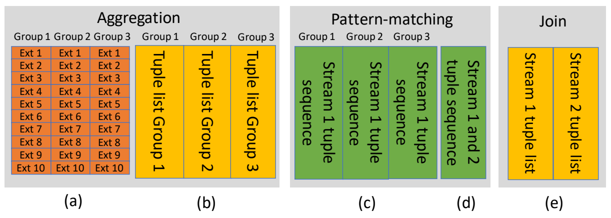


Figure 3: Internal state of operators

A pattern-matching operator looks for particular sequences of tuples that indicate a higher-level event. The stored tuples in the sequences might vary in size and the length of the partial matches may vary. A pattern-matching operator such as in Q2 looks for patterns in a single stream and groups the patterns by a key, leading to an internal state of a tuple sequence for each group (Figure 3c). If the pattern-matching operator looks in multiple streams, such as in Q3, it is only able to keep one sequence in the internal state at the time, because a group is defined for one stream only (Figure 3d). If a query joins and does pattern-matching with groups in the same query, the query must first join the streams as in Q4 before matching patterns.

A join operator such as in Q4 might store the internal state as tuples in a window and evict tuples when the window jumps (Figure 3e). It may keep cached matches on filter predicates to match new tuples to stored tuples faster, using some lookup mechanism. Tuples often vary in size, especially tuples from two different streams that are being joined. Even within the same stream, some attributes, e.g., text attributes, can vary in size.

A state shedding function can drop random state, but there is a strong incentive to keep the most important states. What this means depends on the type of operator that is being assessed. The utility of a partial state is not trivial to define or calculate. It is an operator-specific function that depends also on the type of application that is executed.

In traditional aggregation queries, the shedding of input tuples reduces the accuracy of the results produced output, but keeps the number of produced tuples the same. As such, the goal has traditionally been to minimize the relative error in results [4]. On the other hand, the number of produced tuples may be reduced when shedding tuples in a join operator, sequences in a pattern-matching operator or window extents in an aggregation operator. The accuracy of the produced tuples is retained, but the accuracy of the query is reduced.

A tuple for a join operator has utility if it joins with other tuples. Therefore, the overall goal is to maximize the number of tuples that are produced by the join operator. For the pattern-matching operator, the goal is the same. Either a match completes and produces a complex event, or it expires and never completes. A partial match has no utility until tuples are produced. For an aggregation operator, this is different. A window extent can be considered as just a few integers that indicate the start and stop of the window extent, the count of tuples in the window, and the current aggregate. An incoming tuple triggers an increment of the count in every window extent, and the aggregate is updated.

6 PARTIAL STATE SELECTION

This section discusses how to select the partial states to migrate (RQ3). The knapsack problem can be solved in a few ways, where the greedy approach of sending partial states in a descending order of utility density is the easiest way, but it can not guarantee to achieve the optimum. If all partial states have an identical size, such as for the aggregation operator in Figure 3a, the greedy solution will perform exactly as the optimum. However, with variable partial state sizes in Figure 3b, c, d and e, the state shedding solution might result in a significantly higher utility than the greedy solution. In some cases, the maximum migration time is uncertain, and therefore, the best effort provided with the greedy method might perform reasonably well.

However, if the maximum migration time is short or the state size varies significantly, finding the optimum or near-optimum might yield a significantly higher utility than the greedy solution. Since the knapsack problem is NP-hard, brute-forcing is unfeasible for even small input sizes. However, the knapsack problem has a polynomial-time approximation scheme that uses dynamic programming to find the optimum. This solution has a run-time and space complexity of $O(n^2 \cdot m)$, where n is the total number of partial states, m is the highest utility of the partial states, and $n \cdot m$ is the highest possible sum of utilities. A simplification can be done with the fully polynomial-time approximation scheme algorithm that scales down the utility with factor $\theta > 0$ to reduce the number of iterations. This reduces the run-time and space complexity significantly to $O(n^2 \lfloor \frac{m}{\theta} \rfloor)$.

These algorithms might still have a significant run-time and memory usage, which might be unfeasible if the migration must occur quickly. If we design the utility functions such that the utility values depend on each other, we can create a heuristic that binds the complexity of the optimal search without compromising the accuracy: (1) each partial state i gets assigned a utility through $U(i)$, (2) the utilities are updated as $U'(i) = \frac{U(i)}{m} \cdot 100$. Each partial state gets a utility between 0 and 100, depending on the most important partial state. Since the maximum utility is capped at 100 for each partial state, and the inner loop iterates through maximum $\sum_{i=1}^{n} u_i$, the worst case number iterations is $n \cdot 100$, which leads to a run-time and space complexity of $O(n^2)$.

7 ANALYSIS

This section studies a practical application of the state shedding technique to compare the different approaches for partial state selection (RQ4). We apply a scenario similar to the VANET scenario from Section 3 with two real-world VANET data sets: (1) a data set from Citi Bike [2] that describes bike trips of users, and (b) a data set containing GPS readings from buses in Dublin [1]. We describe three queries in Listing 1, Q_{pm1} uses the Citi Bike data set and Q_{pm2} and Q_{pm3} use the bus data set. The configurations result in different state characteristics: Q_{pm1} produces many partial states; Q_{pm2} only produces eight partial states; and Q_{pm3} produces many partial states.

Listing 1: Queries used in the simulations

Q_{pm1}
```
SEQ(A+B)
GROUP BY BikeTrip.bikeid
DEFINE B AS BikeTrip.end_station_id == 3116
WITHIN 1 day
```

Q_{pm2}
```
SEQ(A+B)
GROUP BY BusRecord.operator
DEFINE B AS BusRecord.block_id == 67002
WITHIN 1 hour
```

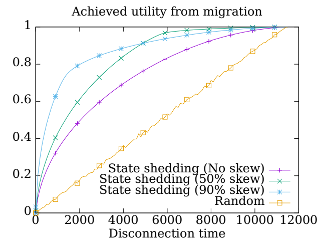
Q_{pm3}
```
SEQ(A+B)
GROUP BY BusRecord.vehicle_id
DEFINE B AS BusRecord.block_id == 67002
WITHIN 1 hour
```

We simulate a connectivity issue scenario where the old host is about to lose connection or experience heavily degraded link connection with the upstream and downstream nodes, and has to complete the migration by a certain deadline. The connectivity failure can be due to imminent node failure or network disconnection. The deadline is estimated by the migration decision model Figure 2, and the data it uses to predict this time is assumed to be collected throughout normal execution. The old host attempts to migrate the operator state using the state shedding function, and after the scheduled state is sent, it continues sending the remaining states as long as possible.

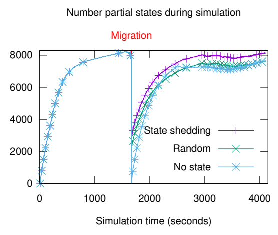
7.1 Simulations

Simulations are performed using the distributed stream processing simulator DCEP-Sim [15]. The simulator implementation incorporates a small-scale stream processing engine and builds upon the well-established discrete-event network simulator ns-3 [11]. Four nodes are deployed in DCEP-Sim: one upstream node, one downstream node, the old and new host. The upstream node produces tuples and sends them to the query that produces tuples for the downstream node. During the simulation, the old host migrates state to the new host, and the utility achieved from the migration is measured and analyzed.

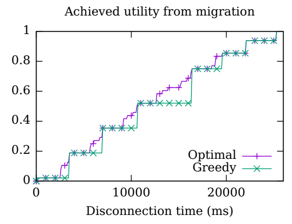
We do three runs, one for each query in Listing 1. The first run compares optimal partial state selection with the random ordering, i.e., without prioritization, where partial states are sent until the old host disconnects. Three utility distributions are used: one balanced and two skewed distributions. In the first distribution, each partial state gets a random utility between 0 and 100. In the second distribution, the random utility is assigned and for 10% of the partial states, i.e., those with utility above 90, we introduce skew by multiplying the utility by ten. In the third distribution, utility values above 50 are multiplied by ten. State shedding is expected to have a more significant effect on skewed utility distributions. The random scenario is only executed with the balanced distribution, but with a high number of partial states, the random case without skew has a very similar utility result as the random case with skew. The second and third runs compare the optimal to the greedy solution, without any utility skew. These runs aim to show how the size and number of partial states affect the achieved utility.



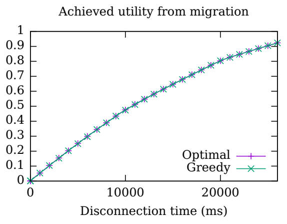
(a) Q_{pm1} with multiple utility distributions



(b) Size of internal state during simulation using Q_{pm1}



(c) Q_{pm2} with few big partial states



(d) Q_{pm3} with many small partial states

Figure 4: Simulation results where (a) and (b) use Q_{pm1}, (c) uses Q_{pm2}, and (d) uses Q_{pm3}

7.2 Results

The utility obtained with the Citi Bike data set are shown in Figure 4a. For the state shedding technique, the utility achieved depends on the utility distribution of the partial states. In the case with skew for 10% of the partial states, 80% of the total utility is reached when the disconnection time is 2 s, less than 20% of the time it takes to send all partial states. When the skew is 50% of the partial states, it takes 3.6 s to send partial states with 80% of the total utility. At 5.5 s disconnection time, the 50% skew reaches >95% of total utility. Even without skew, there is a clear advantage to using state shedding.

Figure 4b illustrates the internal state of the query throughout the simulation, including during and after the migration. The query reaches a bit over 8000 partial states. The migration goes on until the old host is disconnected, at which point, the remaining partial states are dropped. The configurations migrate approximately the same number of tuples, but a different number of partial states. Since the utility is random, the state shedding scheme drops fewer partial states than the random case to achieve a higher utility.

Figure 4c and 4d compare the optimal and greedy solutions. The utility is a function of the state size—the bigger the partial state is, the higher the utility is. The optimal solution performs visibly better than the greedy solution in Figure 4c, but not in Figure 4d. Figure 4c illustrates a case with few big partial states of varying size and Figure 4d is based on many small partial states of similar size. The main difference between the two runs is the size of the partial states compared to the disconnection time. This suggests that the benefit of optimization increases with the proportion of the partial state size variance to the total capacity.

8 CONCLUSION AND FUTURE WORK

This work presents the first investigation of the opportunities and challenges of state shedding for operator migration. It is grounded in the insight that the triggers for operator migration, i.e., overload or network problems, can be limiting factors to successfully performing operator migration with state-of-the-art solutions. Instead of the prevailing all-or-nothing solutions, we propose to perform state shedding to migrate the most useful partial state under the given situation. Both partitioning operator state and estimating the utility of partial state depend on the particular operator. To maximize the aggregated utility of the migrated partial states, we present a solution to the given optimization problem with complexity $O(n^2)$. The simulation-based comparison of this optimal solution with the greedy approach reveals that the distribution of the partial size and the number of partial states play an important role. With few larger partial states the optimal solution outperforms greedy and with many partial states of similar size, they perform almost identically. Further simulation experiments confirm the intuition that the larger the skew in the distribution of the utility of partial states, the faster the aggregated utility at the new host increases. Random selection of partial states to migrate will in cases with utility skew and disconnection before all state is migrated result in lower utility achieved than when using state shedding.

To thoroughly investigate the full potential of state shedding, future work will address novel solutions for utility estimation, e.g., to consider application requirements and to use statistics about previous upstream data, as well as to combine state shedding with scheduling of operator migration, e.g., to delay migration.

Acknowledgments: This work was supported by the Parrot project (Research Council of Norway, 311197).

REFERENCES

[1] 2013. Dublin bus GSP data from Dublin city council insight project. https://data.smartdublin.ie/dataset/dublin-bus-gps-sample-data-from-dublin-city-council-insight-project.

[2] 2018. Citi Bike trip data set. https://s3.amazonaws.com/tripdata/201810-citibike-tripdata.csv.zip.

[3] Daniel Abadi, Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, C Erwin, Eduardo Galvez, M Hatoun, Anurag Maskey, Alex Rasin, et al. 2003. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 666–666.

[4] Brian Babcock, Mayur Datar, and Rajeev Motwani. 2004. Load shedding for aggregation queries over data streams. In *Proceedings. 20th international conference on data engineering*. IEEE, 350–361.

[5] Koral Chapnik, Ilya Kolchinsky, and Assaf Schuster. 2021. DARLING: data-aware load shedding in complex event processing systems. *Proceedings of the VLDB Endowment* 15, 3 (2021), 541–554.

[6] Yun Chi, Philip S Yu, Haixun Wang, and Richard R Muntz. 2005. Loadstar: A load shedding scheme for classifying data streams. In *Proceedings of the 2005 siam international conference on data mining*. SIAM, 346–357.

[7] Bugra Gedik, Kun-Lung Wu, and S Yu Philip. 2008. Efficient construction of compact shedding filters for data stream processing. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 396–405.

[8] Bugra Gedik, Kun-Lung Wu, S Yu Philip, and Ling Liu. 2007. A load shedding framework and optimizations for m-way windowed stream joins. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 536–545.

[9] Buğra Gedik, Kun-Lung Wu, Philip S Yu, and Ling Liu. 2005. Adaptive load shedding for windowed stream joins. In *Proceedings of the 14th ACM international conference on Information and knowledge management*. 171–178.

[10] Wilhelm Kleiminger, Evangelia Kalyvianaki, and Peter Pietzuch. 2011. Balancing load in stream processing with the cloud. In *2011 IEEE 27th International Conference on Data Engineering Workshops*. IEEE, 16–21.

[11] George F Riley and Thomas R Henderson. 2010. The ns-3 network simulator. In *Modeling and tools for network simulation*. Springer, 15–34.

[12] Nicoló Rivetti, Yann Busnel, and Leonardo Querzoni. 2016. Load-aware shedding in stream processing systems. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. 61–68.

[13] Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. 2020. hSPICE: state-aware event shedding in complex event processing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. 109–120.

[14] Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. 2020. State-Aware Load Shedding from Input Event Streams in Complex Event Processing. *IEEE Transactions on Big Data* (2020).

[15] Fabrice Starks, Thomas Peter Plagemann, and Stein Kristiansen. 2017. DCEP-SIM: an open simulation framework for distributed CEP. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. 180–190.

[16] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load shedding in a data stream manager. In *Proceedings 2003 vldb conference*. Elsevier, 309–320.

[17] Nesime Tatbul and Stan Zdonik. 2006. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, Vol. 6. 799–810.

[18] Espen Volnes, Thomas Plagemann, and Vera Goebel. 2022. To Migrate or not to Migrate: An Analysis of Operator Migration in Distributed Stream Processing. *arXiv preprint arXiv:2203.03501* (2022).

[19] Bo Zhao. 2018. Complex event processing under constrained resources by state-based load shedding. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1699–1703.

[20] Bo Zhao, Nguyen Quoc Viet Hung, and Matthias Weidlich. 2020. Load shedding for complex event processing: Input-based and state-based techniques. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1093–1104.