Chapter 5

Quality-Aware Load-Shedding

This chapter provides more information about the overload management techniques employed by the DISSP system. In particular it looks at the *load-shedder* component and how it uses SIC values to make semantic decisions about what tuples to discard. The goal of the load-shedding mechanism is to manage overload by discarding a portion of the input tuples. The algorithm that decides what tuples to keep and what to shed can leverage the information contained in the SIC values of tuples to implement an intelligent shedding policy.

Using SIC values it is possible to implement a *fair shedding* policy, aiming at penalising all queries in a similar fashion. In this context, fairness is defined based on quality, a situation in which all queries achieve the same SIC values for the results. Before introducing the shedding algorithm, the main component of the overload management infrastructure are presented.

In the load-shedding process, the system has first to choose how many tuples to keep for processing. This estimate is made using a simple *cost model* that takes into account the average time spent processing a generic tuple. Then, a choice has to be made about what tuples to keep and what to discard. A fair shedding algorithm is presented in Section 5.3, which leverages the SIC values of the incoming tuples to choose the set of tuples to be discarded so that all queries experience the same reduction in quality-of-service.

Load-shedding is triggered in a system that is already overloaded so that the system can keep on processing even if some of the input has to be discarded. In this scenario, where the processing node is already running out of CPU cycles, the overload management infrastructure has to calculate the correct set of tuples to discard according to a given shedding policy. The chapter ends with a description of some low-level details of the load-shedding mechanisms employed by the DISSP system.

5.1 Fairness in load-shedding

A processing node is considered overloaded when it can not process all the data it receives in input in a timely fashion. If, in a certain time interval, the amount of incoming tuples, *input rate*, is larger than the amount of tuple the system can process, *output rate*, it means that the processing capacity of that node is exceeded. When this happens, the internal queues of pending tuples increase their size. While this condition can be sustained for a short amount of time, it eventually leads to an out-of-memory failure if it is not addressed. The occurrence of an overload condition can be caused by an increase of the rate of incoming tuples, or by the change in their nature, increasing their processing cost.

The augmenting of streams with SIC values also allows to reason about the quality of tuples. This quality metric has been designed to capture the amount of information lost during the creation of a tuple, and thus its contribution to the query results. Employing the information carried by the SIC metadata allows the system to reason about the quality of the data it processes. In particular it is possible to predict what is the impact of dropping a certain tuple on the quality of the final results of a query. This allows for the implementation of *intelligent shedding policies*. Section 5.3 explains the details of a *fair shedding policy*, which tries to equalise the resulting SIC values of all queries.

In a shared infrastructure the available processing resources are divided among a large number of concurrently running queries. In this context, a *fair* system would allocate a proportional amount of resources to each query, so that, in an overload condition, the quality degradation of results is similar for all queries. Employing the SIC metric to quantify this reduction in information content in the results allows a more precise definition of fairness in a stream processing system:

Definition 5.1 (Fairness in a Stream Processing System) *An overloaded stream processing system that has to perform load-shed is considered to be fair if it discards tuples in a way that strive to minimise the difference in SIC value of results for all running queries.*

This definition does not make a difference based on the amount of resources consumed by a query, nor on the number of operators it contains. Each query is treated equally and given an amount of resources proportional to its needs. This means that each query should provide the same quality-of-service to its user. A policy giving the same amount of resources to all queries would favour small queries, while expensive queries would be heavily penalised. It would lead to a large spread between the SIC values achieved by the light and the heavy queries. The goal of a fair system, according to the above definition, is instead to equalise the SIC values of all queries, trying to reduce the differences among queries as much as possible.

5.2 Abstract Shedder Model

Figure 5.1 shows the *abstract model* of a load-shedder. It depicts the three generic components that are needed to implement an *overload management mechanism*. When tuples are received, they are staged in an *input buffer*, waiting to be processed. Periodically, the *overload detector* checks if the load of the system is acceptable or if some of the input has to be discarded. In case the node is considered overloaded, it triggers the *tuple shedder*, which is in charge of selecting and discarding a certain amount of tuples in order to overcome the overload condition.

Input Buffer

When tuples are received by a node they are stored into an *input buffer* waiting to be processed. This allows the system to continue receiving tuples from the network without blocking, even though the system is not able to consume them at the same rate. The order in which tuples are stored usually follows the FIFO model, thus preserving the natural temporal order of tuples. Every time a processing thread becomes available, it removes the oldest tuple from the input buffer and proceeds with its processing.

When the system is overloaded the size of the input buffer grows, as more tuples are admitted than the ones processed. This leads to an increase in the latency of results and eventually to the exhaustion of system resources. If the size of the input buffer starts to grow out of control, the overload detector acts by invoking the tuple shedder, so that its size can be bring back to the normal values. The presence of an input buffer also allows the system to implement a *semantic shedding policy*. Instead of simply dropping tuples as they arrive into the system, the system stores them first into the input buffer and leave the decision about what tuples to drop to the tuple shedder. This component is in

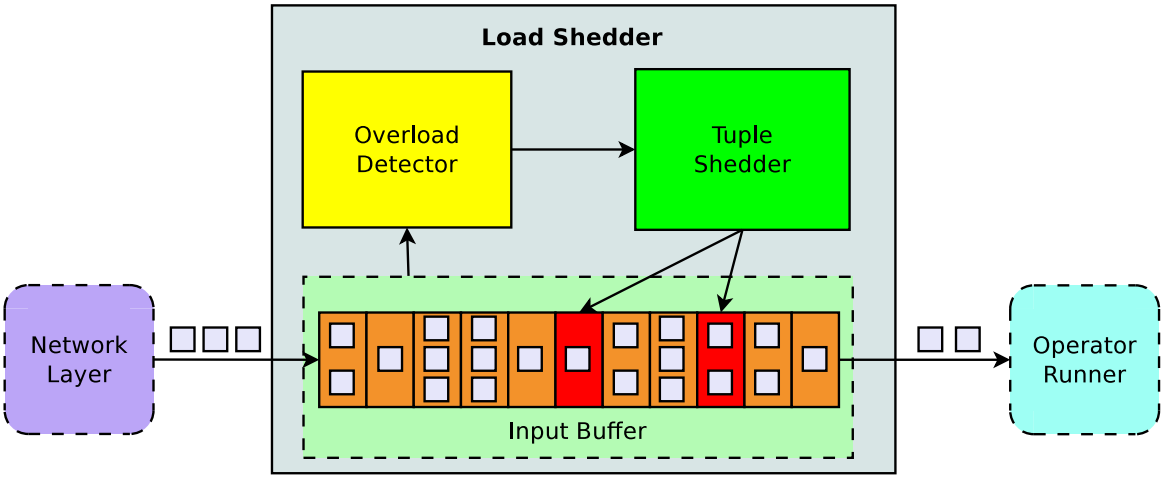

Figure 5.1: Abstract representation of a load-shedder, showing its internal conceptual model.

charge of implementing the shedding algorithm, that chooses what tuples to drop and what to keep among those awaiting processing into the input buffer.

Overload Detector

The overload detector is the component responsible for monitoring the load of the node. It detects an overload condition before it becomes critical and invokes the tuple shedder to discard a portion of the input in order to maintain the load within the operational limits of the node. Periodically, at fixed *shedding intervals* δ, it checks the size of the input buffer. The shedding interval is set a system parameter and influences the behaviour of the system. For low latency processing, it is preferable to set this to a short time interval, in the order of a few hundreds of milliseconds (i.e. 250ms). Setting the value too low, on the other hand, induce unnecessary overhead due to frequent shedder invocations.

At every shedding interval the overload detector makes a prediction about the number of tuples the system will be able to process during the next interval, which takes the name of N_{tuples}. In order to estimate this amount correctly, the overload detector employs a *tuple cost model*, as the one described in Section 5.3. The difference between the total number of tuples in the input buffer, IB_{size}, and the predicted throughput of the system in the next shedding interval, N_{tuples}, represents the number of tuples to be discarded by the tuple shedder, $N_{discard}$.

$$N_{discard} = IB_{size} - N_{tuples} \tag{5.1}$$

If $N_{discard}$ is negative, there is no need for shedding during this interval. If its value is positive instead, the tuple shedder is invoked and given this number as a parameter. The shedding policy implemented by the tuple shedder then calculates which tuples to drop among those available in the input buffer.

Tuple shedder

Once the overload detector has identified an overload condition, it invokes the tuple shedder to discard a certain amount of tuples from those awaiting processing in the input buffer. The number of tuples to be shed is calculated by the overload detector, but the choice about which tuples should be dropped is made by the tuple shedder. This inspects the input tuples and chooses the ones to shed based on a *shedding policy*.

To better understand the interactions among the different components of this abstract shedding model, let us consider again Figure 5.1. Tuples are received by the *network layer* and are placed into the *input buffer* even if the node is currently overloaded, since any load-shedding decision is made later on.

Every tuple received by the node awaits processing into the input buffer until it gets either selected for processing and passed to the *operator runner*, or it is discarded by the *tuple shedder*. At interval time, the *overload detector* decides if the amount of tuples currently in the input buffer is manageable by the operator runner or if instead some tuples should be discarded in order to avoid overload. When the overload detector estimates that N tuples should be shed, it invokes the tuple shedder passing N as a parameter. At this point the tuple shedder chooses N tuples among those currently available, and removes them from the input buffer. This load-shed decision, choosing which tuples should be dropped and which should be kept, is made based on a *load-shedding policy*, that is implemented within the tuple shedder. The simplest possible policy is the *random* one, where tuples are chosen without any reasoning.

Augmenting tuples with the SIC quality metric allows the tuple shedder to implement a more intelligent shedding policy that takes into account the amount of information captured by each tuple. The use of metadata information about the quality of tuples allows the implementation of a *semantic shedding* algorithm, that can be used to maintain certain properties among queries. The next section describes the design of a *fair shedding* algorithm, that aims at penalising all queries in a similar way.

5.3 Quality-Aware Fair Shedding

This section describes the implementation of a *fair shedding* policy that follows the principles stated in Section 5.1. The goal of this policy is to achieve fairness in the resource allocation for all queries running into the system. The policy strives to equalise the SIC values achieved by the result tuples of all queries. The tuple shedder selects the tuples to be discarded, trying to equalise the processing degradation of all queries, so that their normalised (i.e. in the [0,1] interval) resulting SIC values should be numerically close. The tuple shedder addresses overload conditions by periodically discarding batches so that the node retains a sustainable throughput of processed tuples with low queueing delays.

When the shedder is invoked by the *overload detector*, it selects N_{tuples} tuples to keep from the input buffer queue. At first, the shedder assumes that all batches must be discarded. Then, it gradually admits batches that belong to the queries that would otherwise suffer from the largest reduction in their SIC values. This process terminates when the shedder has admitted the required number of tuples as determined by the overload detector. During the shedding process, the shedder always considers the normalised SIC values of the tuples for comparisons among queries.

Section 5.3 presents the *tuple cost model* used to estimate the processing capacity of a node. The cost model is employed by the overload detector to calculate if the node is overloaded and and predicts the number of tuples that the system will be able to process before the next shedding interval. Section 5.3

presents a description in pseudocode of the algorithm employed by the fair shedding policy and provides a detailed explanation of the involved steps.

Tuple cost model

A DISSP node is overloaded when the aggregate resource demands for executing hosted operators of queries exceeds the resource capacity of the node. In particular, every DISSP node maintains an *input buffer* (IB) queue, in which all incoming tuples from other nodes or sources await processing. When the rate of tuple arrival exceeds the processing capacity of the node, the size of the IB grows and the node becomes overloaded.

To calculate the number of tuples to admit for processing, a node has to estimate the processing cost of tuples when executing different types of operators and queries. We employ a simple cost model to calculate the *average* processing time spent on a given tuple for any query in the node. We then use past resource consumption when processing tuples to estimate future resource demands.

We assume that the average tuple processing cost C_T is:

$$C_T = t/n \tag{5.2}$$

where t is the time interval elapsed between invocations of the overload detector, and n is the number of tuples processed by that node during t. C_T is then the time that the node has spent on average processing a tuple from any query. For an accurate estimation of C_T, we use the measured time interval t, instead of the fixed interval δ, that is the theoretical time interval between executions of the overload detector. In a real system, the measured time t between successive invocations of the overload detector might differ from δ. To compensate for any such differences and have an accurate estimation of C_T, we use the measured elapsed time t, rather than the fixed interval δ.

Thus, the estimated number of tuples that the system is able to process between successive invocations of the overload detector is:

$$N_{\text{tuples}} = \delta/C_T \tag{5.3}$$

where δ is the fixed time interval between shedding invocations and C_T is the average time spent processing a generic tuple. To estimate the value of N_{tuples}, we use a sliding window in order to avoid sporadic fluctuations that would lead to under- or over-prediction of the real number of tuples that a node can process. Employing a sliding windows allows for a smoother estimate that is more resilient to sudden changes of the incoming tuple rate.

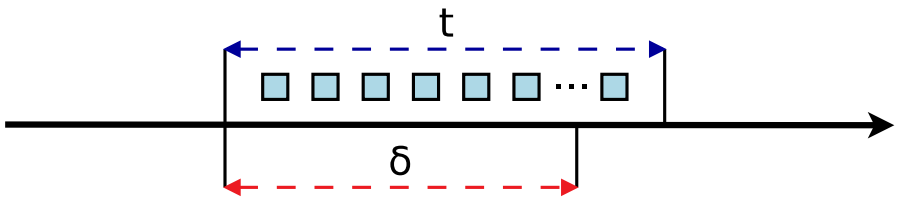

Figure 5.2: Graphical representation of the time intervals δ and t used to calculate the *average tuple cost* C_T needed to predict the *future processing capacity* N_{tuples} of a node.

Figure 5.2 shows the two time intervals, δ and t used to calculate the cost of a tuple, or the average time needed to process an arbitrary tuple. Let us assume for instance a shedding interval δ of 250ms. In reality the *overload detector* triggers with a small delay, after a time t of 300ms. During this time the system process 5000 tuples, therefore every tuple has a cost C_T of 0.06 $ms/tuple$. The system thus predicts that in the next δ interval it will have the processing capacity N_{tuples} of roughly (this value is smoothed by a sliding average) 4100 tuples.

More complex cost models have been proposed in the past, such as [WR09; JNV03]. These approaches are tailored for certain types of operators such as joins and require a number of parameters for accurate estimation. Instead, we strive for a general cost model that can be applied to any type of query or operator. Our approach assumes that the cost of processing a tuple remains fairly constant and is similar across operators. This is not always true though, some operators can require significantly more time to process different tuples, depending on their content. A more precise cost model would take into account also the semantic of the query, looking at the types of operators present. For the scope of this work, a simpler cost model has been preferred.

Quality-Aware Fair Shedding Algorithm

The algorithm used by the tuple shedder to implement the fair-shedding policy is describe using pseudocode in Algorithm 1. The main idea behind it is to start by considering all tuples as discarded. Then it estimates the reduction in SIC value experienced by each query and makes a ranking. It then iterates adding one batch of tuples at the time, choosing the one with the highest SIC contribution from the query with the lowest estimated SIC. After each step it recalculates the ranking and it keeps on saving tuples until it reaches the limit provided by the overload detector. The rest of this section provides a more formal description of the algorithm.

Initially, the *input buffer* queue is locked by the shedder (line 1), this means the system stops accepting new incoming batches and current batches in the queue are not processed by operators. While the shedder runs, new incoming batches are held temporarily in a secondary buffer, which is merged with

the *input buffer* queue before the node resumes processing. This allows the *tuple shedder* to have a snapshot of the current situation so that it can reason on a static set of pending tuples.

The *tuple shedder* then considers all batches in the *input buffer* as discarded (lines 2–8). This algorithm works at the granularity of *batches* not tuples, this greatly reduces the number of iterations and thus the overhead.

First, it updates the current result SIC value of all queries (lines 2–3). For each query that has batches in the input buffer queue, the tuple shedder calculates the new projected SIC value as if none of its batches would be processed (line 6). Then it adds all active queries (i.e. having at least one pending batch) into a priority queue sorted by increasing projected SIC value. In this way it is easy to retrieve the query having the lowest projected performance so that a batch from it can be saved.

The shedder considers all batches in the *input buffer*, it moves them to a temporary buffer and groups them according to the query they belong to while keeping an ordered list of queries and batches for each query (line 7). One by one, all batches are copied from the *input buffer* and inserted into a set of priority queues, one for each query. This builds an ordered list of batches for each query, sorted by decreasing SIC value. This is used later on, so that once the tuple shedder decides to save one more batch of a certain query, it is easy to select the one with the highest SIC value and thus the most valuable, by simply removing the head of a priority queue.

Algorithm 1: Tuple shedding with local fairness

Input : n_{tuples} tuples to keep
IB:= input buffer
Output: IB: input buffer with n_{tuples}, or less if not enough

1 lock IB
2 **foreach** *query* $\in$ *node* **do**
3 update q_{SIC} of *query*
4 **foreach** *batch* $\in$ *IB* **do**
5 find query q such that, *batch* $\in q$
6 update $\hat{q}_{SIC}$ as if *batch* is discarded
7 copy *batch* to ordered list of queries (increasing $\hat{q}_{SIC}$) and for each query keep batches in ascending SIC order
8 remove *batch* from IB
9 $n'_{\text{tuples}} = 0$
10 **while** $n'_{\text{tuples}} < n_{\text{tuples}}$ **do**
11 q := pick query with the least $\hat{q}_{SIC}$
12 b := pick batch from q with highest SIC
13 add b in IB
14 update ordered list of queries
15 n'_{tuples} += tuples in b
16 unlock IB

Then, all batches are gradually removed from the *input buffer* (line 8). This approach ensures that the shedder has to parse all batches in *input buffer* only once and at the same time builds the required priority queues for the next part of the shedding decisions. A priority queue of queries is also built, ordered by increasing SIC values, so that the head of the queue is always the query with the worst projected SIC value (line 7).

Next the shedder considers which tuples to admit from each query to ensure fairness (lines 10–15). It first selects the query that would be penalised the most by discarding all its batches (line 11), by removing the head of the query priority queue. It then selects the batch with the highest SIC value for this query (line 12). By selecting first the batch with the greatest SIC value, among those available for each query, the system maximises its projected SIC value, adding first those batches that have a greater contribution.

The shedder then moves the selected batch back into the *input buffer* for processing (line 13). Since this batch is not discarded, the shedder also updates the priority queue of queries according to their projected SIC values (line 14). The shedder repeats this process until it has accepted n_{tuples} tuples. Any remaining batches that were moved to the temporary buffer are discarded and regular tuple processing from the *input buffer* queue resumes (line 16).

5.4 DISSP Load-Shedder Implementation

This section presents some implementation details about the overload management system implemented into the DISSP prototype. It allows to better understand how the abstract components can be realised in practice and outlines some of the mechanisms that have been employed to make the system more robust.

Input buffer. It is the internal pending jobs queue of the ThreadPoolExecutor, the pool of processing threads implementing the *operator runner*. It contains a series of WorkUnits objects, bundles containing a batch of tuples and a reference to their destination operators, representing future processing units. Whenever the overload detector identifies a critical load situation, it calculates what its maximum length should be, then it triggers the tuple shedder to remove the excess tuples from the input buffer.

Overload detector. It is implemented in the CheckOverload class, a monitoring thread that performs a check on the current load of the system at every shedding interval. This thread is in charge of maintaining certain statistics about the performance of the system and to act if it detects an overload condition. It calculates the average *tuple cost*, the time needed to process one tuple, using the cost model previously explained. Based on this tuple cost it makes a prediction about the number of tuples

the system will be able to process during the next shedding interval, or the maximum number of tuples that should be present in the input buffer. If there are currently more tuples waiting to be processed in the queue, it invokes the tuple shedder that takes care of choosing which of this tuples should be dropped in order to maintain a sustainable load in the system.

Tuple shedder. It is the class implementing the load-shedding semantic of the system. It contains the algorithm that chooses which tuples to drop from the ones currently awaiting processing into the input buffer. It exposes a simple interface to the overload detector, that simply passes a parameter containing the number of tuples to keep in the buffer when calling the .loadShed(int n) method. The internal shedding policy then analyses the input buffer queue and decides which tuples to drop. Changing the class implementing the tuple shedder interface to the overload detector, it is possible to experiment with different shedding policy. In its simplest implementation it simply discards tuples at random, exploiting the SIC quality metrics though it can implement a more semantic algorithm, like the fair-shedding one presented in this chapter.

Late deserialisation. When tuples are received by the network layer, they are placed directly into the input buffer waiting for processing. At this point though, the data is still in network representation and needs to be converted into concrete objects before it can be processed. This *deserialisation* of the input tuples is costly and is delayed as much as possible, in order to avoid spending precious CPU cycles on tuples that might never be processed. The system deserialises only the minimum information needed, like the query id to which the tuples belong and their SIC value. Using this information the tuple shedder can execute its shedding policy and decide if a batch of tuples is valuable enough or it should be dropped instead. The final deserialisation happens only when the WorkUnit is selected for processing, it is the first step performed by the operator runner when it removes it from the input buffer.

Sliding windows. The load experienced by a stream processing node can vary dramatically over time. Many times though, when looking at the granularity of a few hundreds milliseconds, the number of tuples delivered to the node can be highly unconstant. It is important that the overload detector does not mistake the arrival of an isolated large batch of tuples, that can be safely kept into the input buffer and processed, with the beginning of a long term overload condition. For this reason, when implementing the tuple cost model, the system smooths all the parameters using a moving average calculated over the last N (i.e. 40) values of the metric. This provides a certain slack in the reaction of the system, that allows the distinction between a sudden load spike that can safely be ignored and a sustained change in load that needs to be addressed.

5.5 Summary

This chapter presented the load-shedder component of a stream processing system, in charge of detecting and addressing the excess of processing load by discarding a portion of the input tuples. In particular it proposed to exploit the SIC quality metric to implement an intelligent shedding policy that tries to allocate resources fairly among queries. First, it defined *fairness* in a stream processing system, based also on the SIC values achieved by queries. It then described an abstract model for the overload management system, explaining the interaction among its internal components. It then presented the algorithm implementing the *fair-shedding policy* used to equalise the achieved SIC values of all queries. It showed how a tuple cost model is needed to estimate the future sustainable throughput and thus the number of tuples to be shed, and how to choose which tuples should be dropped using the SIC metric. The chapter ended with a description of some details of the implementation of such component in the DISSP prototype.