Chapter 7

Conclusion

Overload is a common condition for stream processing systems. The correct provisioning of the processing infrastructure is difficult due to to the unpredictable volatility of input rates and the costs associated with the purchase or rental of a large enough computing platform. Furthermore, it could be in the interest of the user to purposetely underprovision the system, accepting approximated results in return for a reduction of the operational cost.

This thesis presented a novel quality metric to augment data streams called Source Information Content, that allows for an efficient and flexible management of overload. The SIC metric aims at capturing the failure that occurs during the processing of query and its impact of the quality of the delivered results. This metric allows the user to reason about the quality of the delivered results and the system to implement intelligent policies for overload management.

A prototype stream processing system called DISSP has been developed to test the possibilities offered by the SIC metric. It was designed to withstand constant overload, exploiting the SIC metric to gracefully degrade the quality of results. A particular focus was given to the efficiency of load-shedding, following a design that allows for the flexible implementation of semantic shedding policies based on the SIC values of the incoming data.

The SIC metric has been used to implement a fair shedding policy, that aims at an even allocation of system resources among the running queries, regardless of their size or characteristics. This allows all queries to achieve a similar quality-of-service, reducing the differences in terms of SIC values of results. This is particularly useful in a shared processing infrastructure, so that all users receive a fair share of the processing resources.

A set of experiments has been devised to test the characteristics of the SIC metric and its applicability in the context of semantic load-shedding. It has been shown that there is a good correlation between the error that occurs in the results and their associated SIC value. The fair shedding policy was compared

6.5 Summary

This chapter presented the experiments performed to evaluate the advantages of employing the SIC quality metric in a stream processing system. All experiments were performed using the DISSP prototype described in Chapter 4. The first set looked at the correlation between SIC values and the correctness of results. Even though the SIC quality metric was not designed as a direct measure of the accuracy of results, experiments showed that for many classes of queries there is a good correlation, confirming that a high value of the metric indicates a good confidence in the results. The second set of experiments focused on the evaluation of the *fair shedding policy*, designed to exploit the SIC metadata to evenly allocate resources among queries, comparing its performance with a random shedder. Results showed that employing the fair shedding policy leads to a more even distribution of SIC values for the delivered output tuples. Finally, a set of experiments was devised to investigate the correlation between cost and quality of the results. These experiments showed that it is possible to strike a trade-off between the achieved SIC value of results and a reduced rental cost for the processing infrastructure.

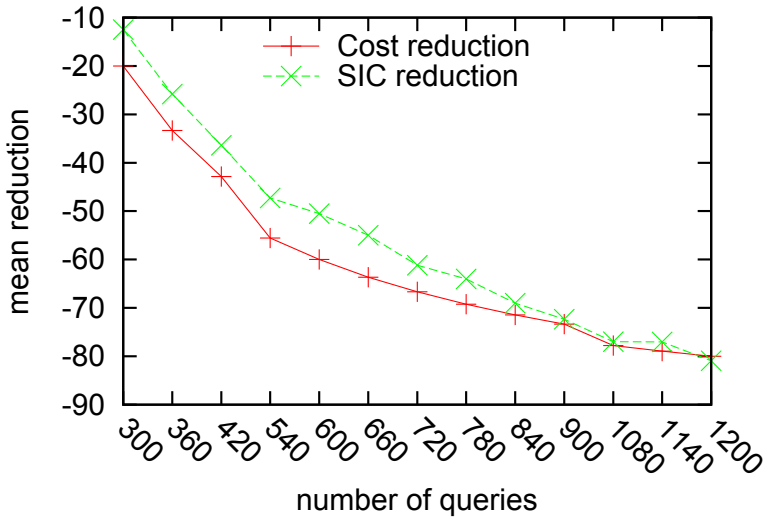

Figure 6.11: Trade-off between the reduction of cost per query and reduction in QoS expressed as mean SIC values.

service achieved deploying x queries.

The deployment of new queries is considered to be advantageous when:

$$\Delta(QoS) > \Delta(Cost) \tag{6.5}$$

The difference between the reduction in cost and quality-of-service is inversely proportional to the number of queries. The more queries are added, the lower the cost advantage, until a point is reached where increasing the number of queries is no longer advantageous. A user can keep adding new queries until this point is reached, until the difference goes under a certain threshold, or until a minimum SIC values is reached.

Experimental Results

For the trade-off evaluation, the data from Figure 6.10 is used. The cost of the infrastructure remains stable, so the cost per query is calculated as the inverse of the number of queries: $1/N_{queries}$.

Figure 6.11 uses the mean SIC value as the quality-of-service metric. The base case is at 180 queries and at every step 60 more queries are deployed. The graph shows how at 300 queries there is almost an 8% difference between the reduction in quality-of-service and the cost reduction. This difference grows smaller as more queries are deployed until a "tipping point" is reached at around 900 queries. From this point on, adding more queries is no more advantageous.

percentile reduction in SIC value is lower than the relative increase in number of queries. At a certain point this changes and the system performance becomes sublinear. The next section will show how it is possible to exploit this behaviour to strike a trade-off between the quality of results in terms of SIC value and the cost per query.

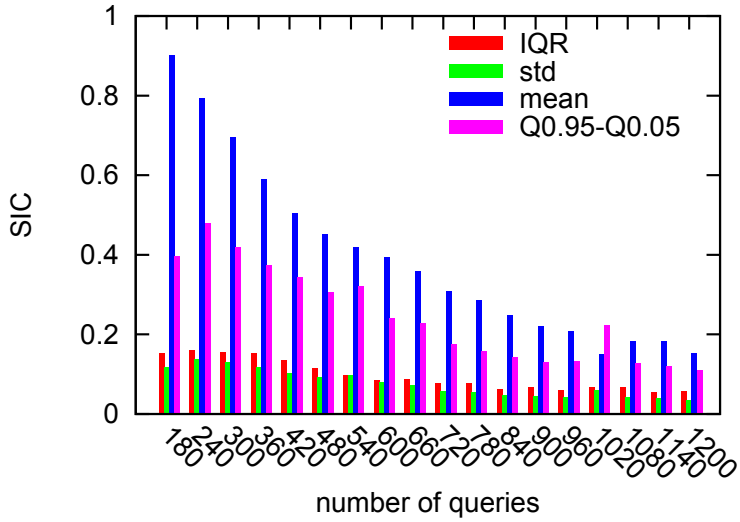

Figure 6.10: Fairness for increasing number of queries.

6.4 Cost/QoS Trade-off

When the user is willing to accept approximate results, it is possible to use the SIC metric to strike a trade-off between quality-of-service and costs. If we maintain the amount of processing resources stable, increasing the number of deployed queries leads to a reduction of the cost per query. When the percentage of cost reduction is greater than the reduction in quality-of-service (i.e. SIC values), there is a cost advantage adding more queries.

The difference in cost, $\Delta(cost)$, is calculated as:

$$\Delta(cost) = \frac{C_{base} - C_x}{C_{base}} \tag{6.3}$$

where C_{base} is the base cost for the starting deployment and C_x is the cost of deploying x queries.

The difference in quality-of-service, $\Delta(QoS)$, is calculated as:

$$\Delta(QoS) = \frac{QoS_{base} - QoS_x}{QoS_{base}} \tag{6.4}$$

where QoS_{base} is the quality-of-service value for the starting deployment and QoS_x is the quality-of-

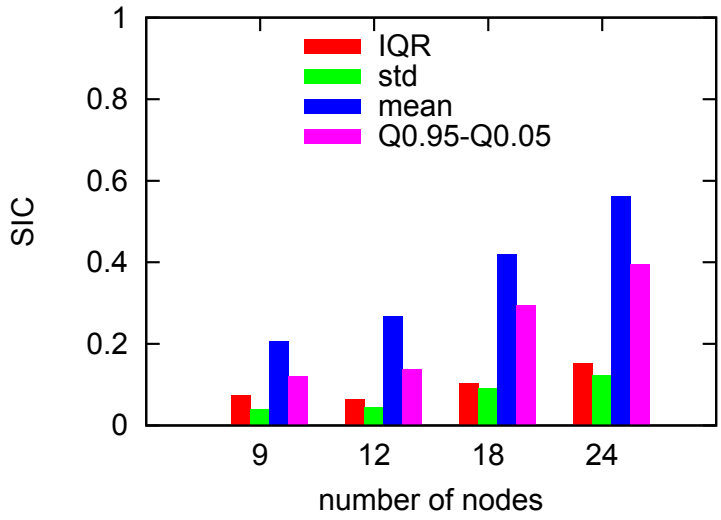

Figure 6.9: Fairness for increasing number of nodes.

the need for load-shedding. The higher amount of processing resources available leads to an increase in average SIC values of the output tuples. All dispersion measures show a reduction, which means that a better performance also is achieved in terms of the variability of the output SIC values. The value of the dispersion measures grows together with the value of the mean. If the value of the mean doubles, in fact, the value of the dispersion metrics tends to double as well, because the variability range is larger. With a mean of 0.2, we can expect the SIC values to be distributed in the interval [0-0.4], while with a mean of 0.4, we can expect the range to be [0-0.8]. Since the range is doubled also the dispersion measures should be doubled. What should remain constant is the ratio between the dispersion measure and the mean. For instance, the ratio between the standard deviation and the mean takes the name of *coefficient of variation* [Bla09]. All these ratios present a stable behaviour, confirming the good scalability of the fair shedder when varying the number of processing nodes.

Increasing the Number of Queries

This set of experiments observes the performance in terms of SIC value when varying the number of queries (i.e. increasing the load) while maintaining a constant amount of processing resources. The total number of nodes used in these experiments is 25, with the characteristics described in Table 6.1. The nodes are loaded with an increasing number of queries, with an evenly mixed workload of COV, AVG-all and TOP-5 queries (see Table 6.2).

Figure 6.10 shows the degradation in SIC values when deploying a number of queries that varies from 180 to 1200. The results show that the value of all queries is reduced proportionally to the number of queries. From the graph, it is possible to observe that the performance of the system is not strictly linear. The slope of the descending mean SIC values changes sign, meaning that at the beginning the

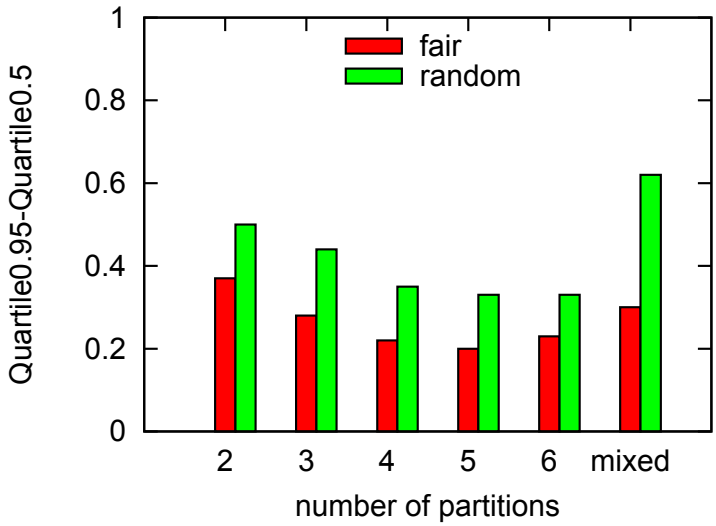

Figure 6.8: Comparison of fair and random shedders (Q0.5-Q0.95).

Fair Shedder Scalability

The following set of experiments evaluate the scalability of the *fair shedder* in terms of the number of nodes and queries. The aim is to test the variation of the SIC values of results when the amount of processing resources changes. In the first set of experiments, the amount of load remains constant, while increasing the processing resources. Adding more nodes leads to an increase in the resulting SIC values, as the amount of load-shedding required is reduced. In the second set of experiments, the variation is in terms of processing load, while maintaining a constant amount of processing resources. In this case, increasing the number of deployed queries leads to a reduction of SIC values. Both sets of experiments are deployed on the Emulab testbed, as described in Table 6.1. The processing nodes are loaded with an evenly mixed workload of COV, AVG and TOP-5 queries, as described in Table 6.2.

Increasing the Number of Nodes

This set of experiments measures the scalability of the fair shedding policy in terms of number of nodes. The aim is to measure the performance of the fair shedder when varying the amount of processing resources. Increasing the number of nodes, keeping the amount of queries stable, means progressively reducing the overload on each processing node. The fair shedder should achieve a proportionally better performance in terms of mean SIC value. Ideally, reducing the number of nodes of 50% should result in the same reduction in the SIC values of the computed results.

Figure 6.9 shows the results obtained after running the experiment on a number of nodes varying from 9 to 24. Increasing the number of processing nodes reduces the average load on each node and thus

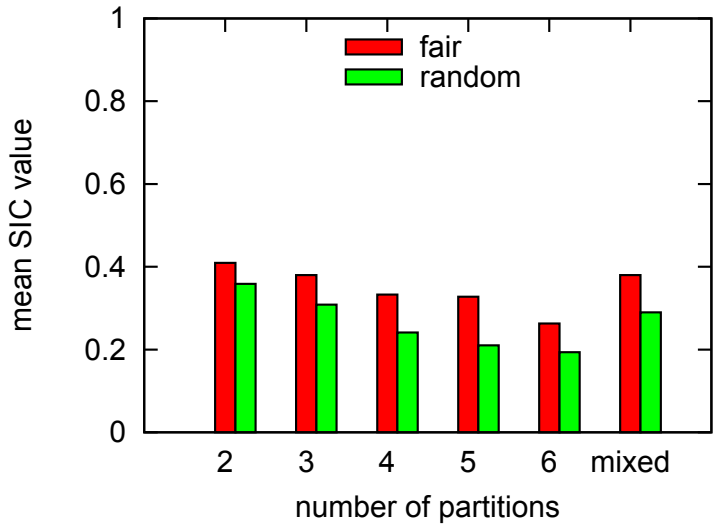

Figure 6.5: Comparison of fair and random shedders (MEAN).

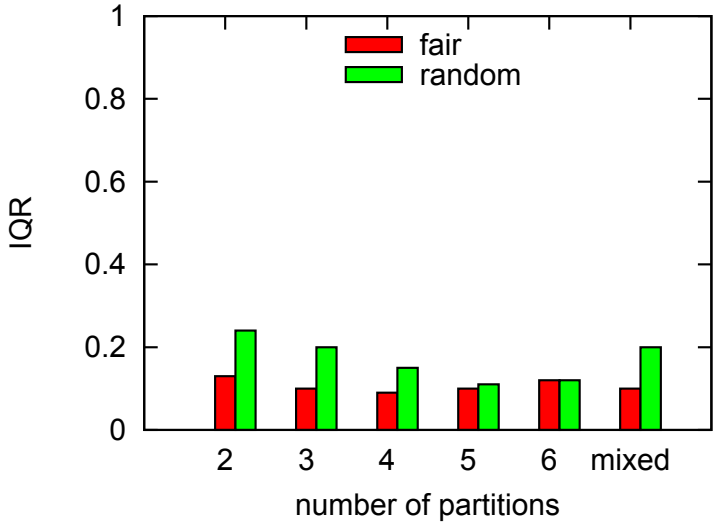

Figure 6.6: Comparison of fair and random shedders (IQR).

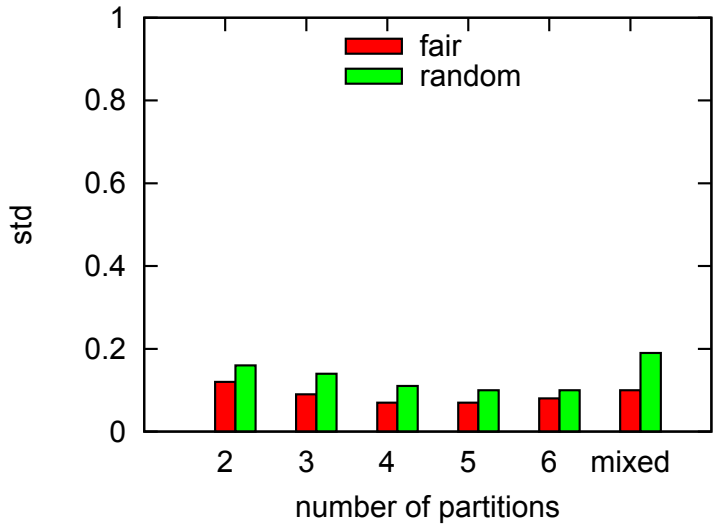

Figure 6.7: Comparison of fair and random shedders (Standard Deviation).

its variance. Variance is defined as the sum of the squared distances of each term in the sample from the mean, divided by the total number of elements in the sample. It shows how much variation or "dispersion" exists from the mean. A low standard deviation indicates that the data points tend to be close to the mean, whereas a high standard deviation indicates that the data points are spread out over a large range of values.

Interquartile range: The interquartile range (IQR) [UC96] is a measure of variability, based on dividing a data set into quartiles. Quartiles divide a rank-ordered data set into four equal parts. The values that divide each part are called the first, second and third quartiles. They are denoted by Q1, Q2, and Q3, respectively. Q1 is the middle value in the first half of the rank-ordered data set; Q2 is the median value in the set; Q3 is the middle value in the second half of the rank-ordered data set. The interquartile range is equal to Q3 minus Q1.

Q0.95-Q0.05: This is a measure of variability used in a similar way to the interquartile range. The ordered data is divided into 100 equal parts, called percentiles, and Q0.95-Q0.05 captures the spread of the middle 90% of the ordered data values. It shows the spread of the majority of the data values and captures a larger set of values than the IQR.

Experimental Results

The first graph, in Figure 6.5, shows the comparison between the random and the fair shedding policies in terms of average quality of the delivered results. In all the experiments, the fair shedder outperforms the random one, achieving on average results with a higher SIC value than the random shedder. The last three graphs, in Figure 6.6, 6.7 and 6.8, show the comparison between the random and the fair shedding policies in terms of variability. For all the dispersion measures used, the fair shedder delivers a lower value compared to the random shedder. This means that the fair shedder chooses a better set of tuples to be dropped, resulting in a higher mean SIC value and a lower dispersion of SIC values.

All experiments show the impact of breaking the queries into more partitions for all the analysed measures. This is due to the higher cost of inter-node communication. A larger number of subqueries provides a better load distribution among all the processing nodes, but it also increases the total load imposed on the system. The *mean* SIC value is higher in the case of two partitions and decreases as the number of partitions increases. The *dispersion* metrics, instead, have lower values for two partitions. Their value increases when increasing the number of subqueries.

Each run of the experiment compares the performance of the random and fair shedding policies, varying the number of query partitions (i.e. subqueries) for each query, while trying to maintain a similar total number of queries. This means that, when the the number of partitions per query increases, the total number of queries decreases (as shown in Table 6.4). Using a larger number of subqueries means that the the load of each individual query is distributed over a larger number of nodes, increasing the overhead due to the inter-node network communication. The goal is to explore the effect of increasing the number of partitions for the same number of queries, while comparing the two load-shedding policies.

Table 6.4 shows a breakdown of the load characteristics for each experimental run. The first column shows the number of subqueries that each query has been partitioned into, the second column shows the total number of deployed queries, the third shows the number of query types used (always 3), and the final column shows the total number of subqueries deployed. The last row contains the data for the *mixed* run, in which each query has been divided into a random number of partitions between 2 and 6.

Statistical Measures

The following statistical measures are used to compare the two load-shedding policies. The first, mean, is used to evaluate the average performance of the system among all queries, while the other three capture the dispersion of results. Before discussing the experimental results, we provide definitions for each of them:

Mean: The arithmetic mean is defined as the value obtained by summing all elements of the sample and dividing this value by the total number of elements in the sample. It is used to provide an indication of the central tendency of the data set.

Standard deviation: The standard deviation [Rek69] of a data set is defined as the square root of

Partitions	Queries	Total Partitions
2	334	2004
3	220	1980
4	170	2040
5	134	2010
6	112	2016
2-6	190	∼2280

Table 6.4: Workload breakdown for experiments comparing the random and fair load-shedding.

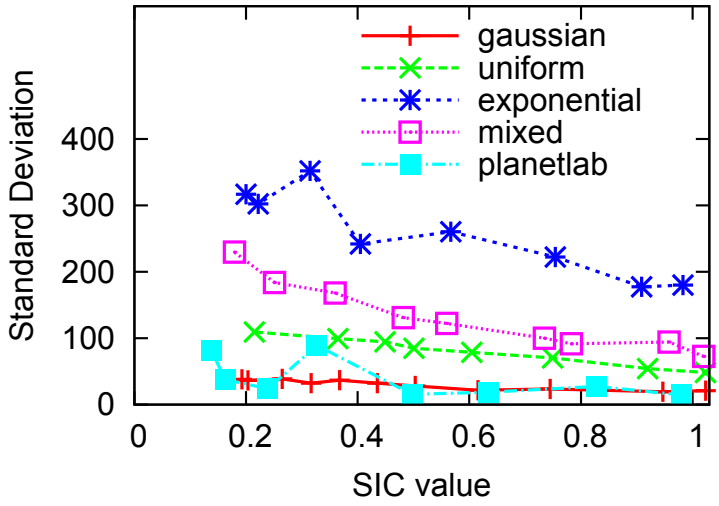

Figure 6.4: Correlation of the SIC values with the query output performance for *Covariance* queries.

the SIC metric is designed as a generic quality metric, it represents a good indicator of the quality of the computed results.

6.3 Load-shedding Policies

This section presents experiments that evaluate the use of the SIC quality metric in the context of load-shedding. Using the SIC metric, it is possible to implement *semantic shedding policies* [Ma+08] that discard tuples based on their information content. These experiments evaluate a *fair shedding* policy (see Section 5.3) that is designed to provide an equal allocation of system resources, with the goal of equalising the quality-of-service (i.e. same SIC value) for all running queries under constant overload. We compare the *fair shedding* to a *random shedding* policy.

Fairness Comparison

This section compares the random and the fair load-shedding policies. The fair 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) result SIC values are numerically close. The random shedder, instead, picks the tuples to be discarded at random. The comparison shows that the fair shedder always outperforms the random shedder, achieving a higher average quality of the results (mean) and also a lower "spread", measured using some dispersion metrics (IQR, Q.95-Q.05 and STD). The experimental setup for this set of experiments consists of 25 Emulab nodes, as described in Table 6.1. The query workload is a mix of 3 different queries: Average, Covariance and Top-5, as described in Table 6.2.

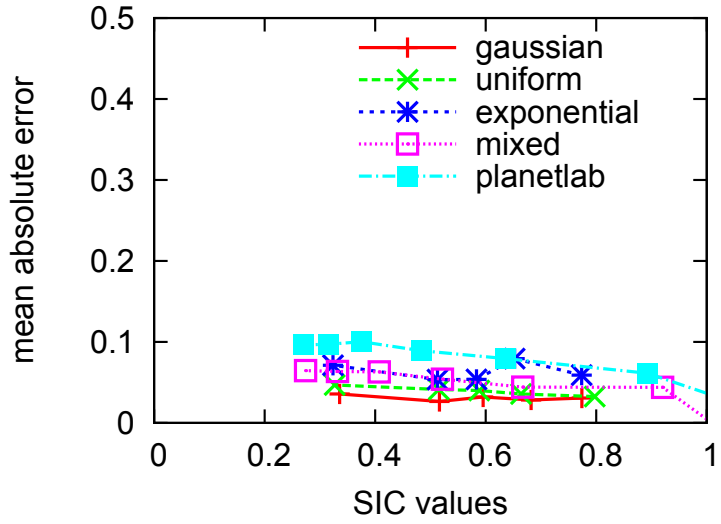

Figure 6.1: Correlation of SIC values with the query output performance for *Average* queries.

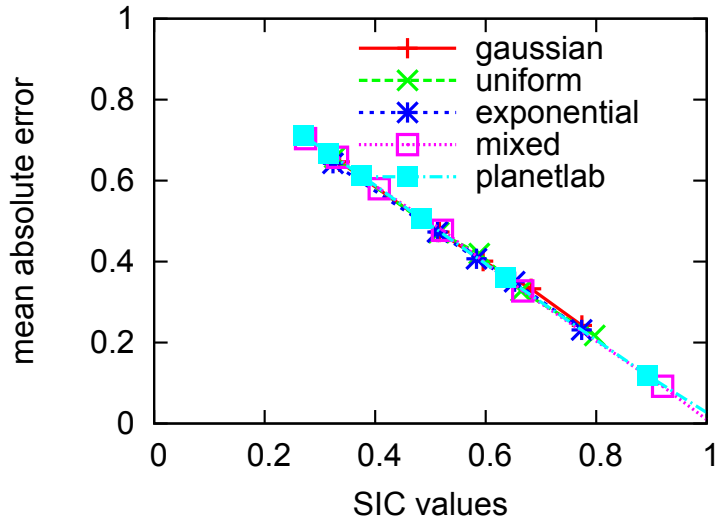

Figure 6.2: Correlation of SIC values with the query output performance for *Count* queries.

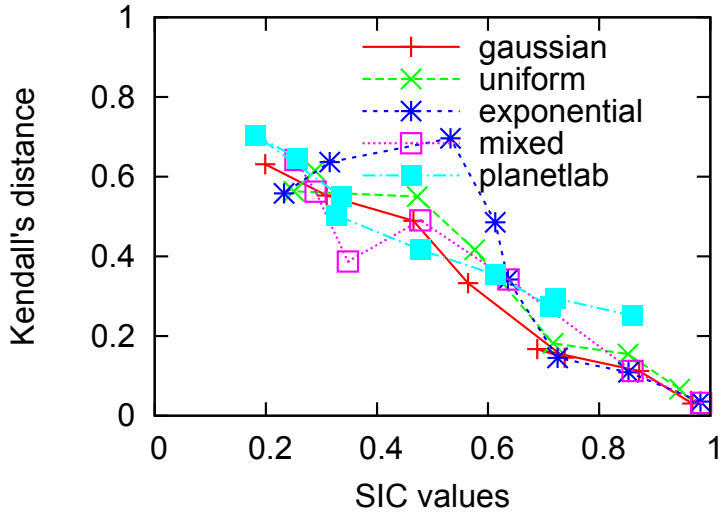

Figure 6.3: Correlation of the SIC values with the query output performance for *Top-5* queries.

where C_p is the number of concordant pairs, D_p is the number of discordant pairs and n is the total number of pairs. This value is normalised to lie within the [0,1] interval.

In the case of the COV query, we compare the standard deviation of the real covariances with the one under overload. The real covariance come from perfect processing and this value is matched with the covariance obtained for degraded processing. Hence, we can evaluate the correlation between the SIC values and the quality of the COV query results.

Experimental Results

The following graphs present the results from the experiments running queries belonging to the *aggregate* and *complex* classes. A graph is shown for each query type (AVG, COUNT, TOP-5 and COV). Each graph contains the measurements obtained from the different runs of the experiment, one for each input data set.

Aggregate queries. For the AVG queries (see Figure 6.1), the graph shows a small error even under heavy overload. This is due to the particular nature of the average operation. Since the load-shedder discards tuples at random, the distribution of the data is not significantly affected. However, we observe that as the amount of load-shedding diminishes and the SIC values increase, the degraded values are closer to the perfect ones.

The COUNT query (see Figure 6.2) shows a different behaviour. The error grows linearly with the amount of discarded data. This is an extreme case, in which the occurrence of load-shedding has always a direct effect on the final results. Each tuple that is discarded, in fact, reduces the total output value (i.e. the final count) and thus increases the error.

Complex queries. For the TOP-5 query (see Figure 6.3), the error expressed as the Kendal distance decreases almost linearly with the amount of load-shedding performed, showing a good correlation between the SIC metric and the correctness of results for this type of queries.

For the COV query (see Figure 6.4), the standard deviation of results shows a decreasing trend as the amount of load-shedding diminishes and the SIC value increases. This means that when the amount of overload is reduced, the spread in the value for the results is also reduced. All input distributions, real and synthetic, show a low error despite the large amount of load-shedding.

Even for this other set of queries, the experimental results show a good correlation between the achieved SIC values and the correctness of the results. The severity of the overload condition is, in general, directly proportional to the error observed in the output results. This means that even though

number of query partitions of 150. According to the Zipf's law, each cluster is assigned twice as many subqueries as the previous one. Starting with the deployment of 10 partitions onto the first cluster, the second would receive 20, the third 40 and the fourth 80.

6.2 SIC Values and Correctness

First, we evaluate the correlation between the SIC values of the output tuples and the correctness of the computed results, across a representative range of *aggregate* and *top-k* classes of queries as described in Table 6.2. The experiments use as input synthetic workloads that contains inputs arranged according to some well-known statistical distributions as well as real load measurements collected on PlanetLab, as described in Table 6.3.

For this set of experiments the local testbed is used, as described in Table 6.1. A single DISSP processing node is overloaded by instantiating an increasing number of queries of one class. As we increase the number of queries, the node is forced to discard an increasing number of input tuples. The node uses a load-shedder that drops tuples randomly from each query. The experiment is repeated for every query class and across the five different sets of source data.

Correlation Metrics

We compare the results of a query with degraded processing (i.e. with result SIC values of less than 1) against the result of a query with perfect processing. Each query, degraded and perfect, runs for 5 minutes and the error in the results is measured every second as a function of the achieved SIC value. For each query class, the experiment is repeated for all the different input data sets.

For the AVG and COUNT aggregate queries, we use the *Mean Absolute Error* (MAE) to quantify the relative distance of the *degraded* from the *perfect* result value across all measurements for the duration of the experiment:

$$\text{MAE} = \frac{1}{n} \sum \left| \frac{degraded - perfect}{perfect} \right| \tag{6.1}$$

For the TOP-5 query we calculate the error using the Kendall distance metric [FKS03] that counts the differences (i.e. permutations and elements in only one list) of pairs of distinct elements between the two lists. The Kendall's distance (τ) is defined as:

$$\tau = \frac{C_p - D_p}{\frac{1}{2}n(n-1)} \tag{6.2}$$

Aggregate Queries	
AVG	Calculates the average value over 1 sec.
COUNT	Counts the number of tuples with values >= 50.0 over 1 sec.
Complex Queries	
TOP-5	Shows the 5 PlanetLab nodes with the highest amount of available CPU and at least 100 KB of free memory over 1 sec.
COV	Shows the covariance of the CPU consumption between two PlanetLab nodes.

Table 6.2: Query workload for experiments.

Synthetic source data	
Gaussian	Gaussian data distribution of mean 50.
Uniform	Uniform data distribution of mean 50.
Exponential	Exponential data distribution of mean 50.
Mixed	Random mix from the gaussian, uniform and exponential input sets.
Real-world source data	
PlanetLab	CPU and memory measurements collected on PlanetLab in April 2010 by the CoTop project.

Table 6.3: Input source data for experiments.

cific synthetic workload. The real-world data streams are traces from CPU load and memory usage measurements from all PlanetLab nodes [PLB12] collected in April 2010, as recorded by the CoTop project [CoD10]. The values of the synthetic data follow either a *gaussian*, *uniform* or *exponential*, as described in Table 6.3. Furthermore, a *mixed* synthetic workload is used that mixes all three distributions randomly. These synthetic workloads provide controlled experimental input sets that allow for an easier understanding of the results. In all experiments, the duration of the source time window (STW) is set to 10 seconds for all sources. This value stays well within the variation of processing delays of all the chosen queries. The shedding interval is set to 250 milliseconds, a value that provides a good trade-off between throughput and management overhead.

In the experiments using more than one processing node, comparison and scalability, the deployment of queries tries to emulate the scenario of a federated resource pool, in which the allocation of resources is not homogeneous. Each query is partitioned into a number of subqueries and these fragments are deployed on the available node according to the Zipf's distribution [AH02]. Consider, for instance, a scenario in which the number of nodes is 20, divided into 4 clusters of 5 nodes each, with a total

6.1 Experimental Setup

This section describes the experimental setup used in this chapter. We use two testbed: a local one and one on Emulab, as described in Table 6.1. The local testbed consists of 3 machines with 1.8 Ghz CPUs and 4 GB of memory running Ubuntu Linux 2.6.27-17-server. They are connected over a 1 Gbps network. One machine is used as an oracle, with a global system view, one for the input sources and query submission and one as a processing node. The Emulab testbed consists of a varying number of pc3000-type machines connected over a 1 Gbps LAN network. Each machine has a 3 Ghz CPU, 2 GB of memory and runs the FBSD410+RHL90-STD Emulab-configured Linux image. One machine is used as an oracle, three as input sources and three for the submission of queries.

The workload chosen for the experiments belong to two query classes: aggregate (i.e. AVG and COUNT) and complex (i.e. TOP-5, AVG-all and COV) queries. They are summarised in Table 6.2. The queries use a diverse set of operators, namely: average, top-k, group-by, filter, join, covariance, time-window, remote-sender, remote-receiver and output. The first query class consists of two aggregate queries, chosen to investigate the behaviour of the SIC metric under the different operator semantics. The second class consists of more complex queries, used to explore the properties of the SIC metric in workloads employing a broader variety of operators and outside the aggregate domain. These query types are used in a variety of data processing applications, such as sensor networks and social media analysis.

The input data generated by the sources contains either real-world load measurements or a spe-

Local Testbed	
Physical configuration	3 machines with 1.8 Ghz CPUs and 4 GB of memory running Ubuntu Linux 2.6.27-17-server and connected a over 1 Gbps network.
System layout	1 machine: oracle node, 1 machine: source data generation and query submission, 1 machine: DISSP processing node.
Data sources	400 tuples/sec in 5 batches/sec of 80 tuples/batch.
Emulab Testbed	
Physical layout	pc3000-type machines connected over a 1 Gbps LAN network. Each machine has a 3 Ghz CPU, 2 GB of memory and runs the FBSD410+RHL90-STD Emulab-configured Linux image.
System layout	1 machine: 1 oracle node, 3 machines: source data generation, 3 machines: query submission, 18 machines: 18 DISSP processing nodes.
Data sources	150 tuples/sec in 3 batches/sec of 50 tuples/batch.

Table 6.1: Testbed configurations for experiments.

Chapter 6

Evaluation

This chapter presents the experiments devised to evaluate the advantages of employing the SIC quality metric in a stream processing system. All experiments were performed using the DISSP prototype implementation described in Chapter 4. The research questions that this chapter addresses aim at a better understanding of the characteristics of the SIC metric and its applicability in the context of a constantly overloaded stream processing system. The SIC metric was designed to be operator agnostic, so that it could be employed in a general purpose system supporting any type of query, but was intended to be used as an indicator about the quality of the computed results. Does the SIC value reflect the correctness of the delivered results? What is the correlation between the error of the output and the reduction in SIC values? The SIC metric is thought to be helpful in the implementation of semantic shedding policies. A fair shedding policy was designed and the experiments compare its performance to a random shedder. Does this policy achieve a better quality for the results? Is it more fair in the allocation of system resources among queries? We were also interested in evaluating the performance of this fair shedding policy in terms of scalability, both when varying the number of nodes and the number of queries deployed. What is the behaviour of this fair shedder, when the amount of processing resources changes? Does the performance scale linearly with the number of processing nodes? What happens when the number of queries is increased on the same processing infrastructure? A user may be willing to trade some correctness of result with a reduction of costs, deploying a larger number of queries on the same processing infrastructure, if the reduction in cost is proportionally greater than the reduction in SIC value. Is it possible to strike a trade-off between the achieved SIC value and the cost of renting the processing infrastructure? Loading the system with an increasing number of queries reduces the cost per query. How does this reduction in cost correlate with the reduction in SIC values? All these questions are answered in this chapter, by looking at a range of sample queries using both synthetic and real-world input data.

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.

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.

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 *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

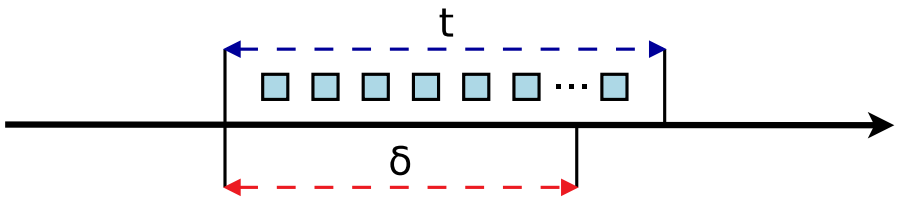

Figure 5.2: Graphical representation of the time intervals δ and t used to calculate the *average tuple cost* C_T needed to predict the *future processing capacity* N_{tuples} of a node.

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

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.

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

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.

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

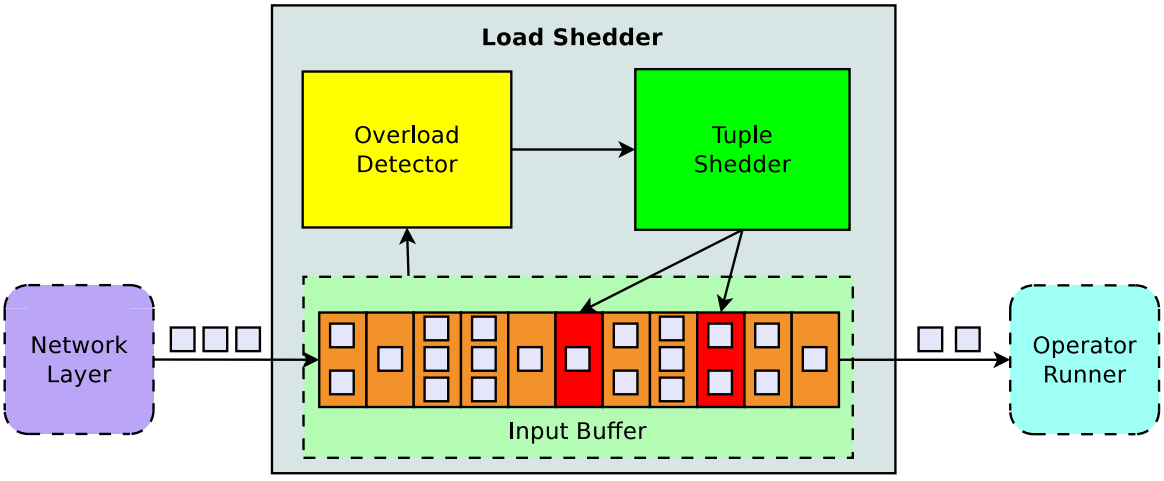

Figure 5.1: Abstract representation of a load-shedder, showing its internal conceptual model.

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.

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.

according to the user needs. It explains how the theoretical entities presented in the description of the data model have been actually implemented in the basic components of a stream processing system.

It then described the *system-wide components*, their role and their interaction. In every DISSP deployment queries are deployed one or more *processing nodes*, processing tuples provided by a set of *input providers* acting as gateways that convert external inputs to the appropriate system format. New queries are introduced into the system by a *submitter*, for each a *coordinator* is spawned, that is in charge of its deployment and management. An *oracle* overlooks the processing of all queries, gathering information about their performance and providing a global view of the system.

After looking at the system as a whole, the focus shifted to the internal components of a processing nodes. Every node exchanges command messages and tuples with the other nodes and the oracle through the *network layer*. The *operator runner* is in charge of routing tuples to the assigned subqueries and to manage their processing through the graph of operators. A *load-shedder* is in charge of overcoming overload by selecting some tuples to be discarded, while the *statistic manager* keeps track of some important metrics about the performance of the node.

The chapter ends following the deployment of a query, using this workflow to describe some details about the algorithms implemented within the system. It describes the steps involved from the submission of a query to when it starts processing. It explains how tuples and operators are compiled at run-time from an XML description, and how the original query graph can be partitioned into subqueries to be deployed on multiple processing nodes.

The next chapter describe the load-shedding process in more details, presenting a *fair shedding* algorithm, that exploits the SIC metric to evenly allocate resources among queries so that they all achieve the same performance in terms of quality of the computed results.

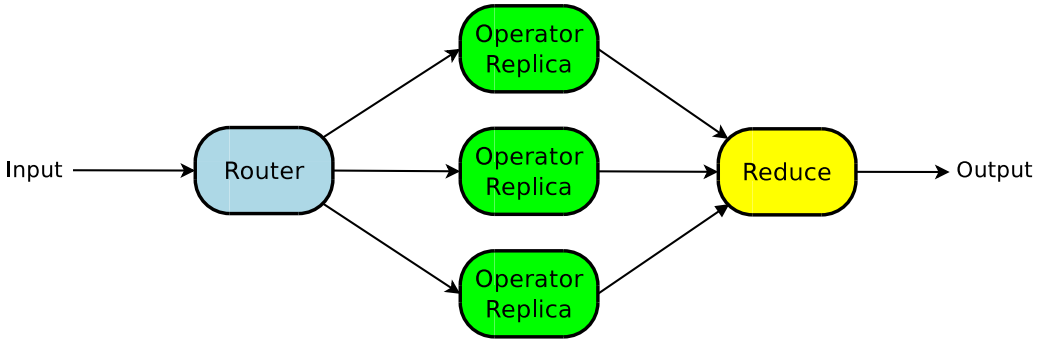

Figure 4.5: Map-Reduce decomposition of an operator.

due to a high rate of incoming tuples or to the semantic complexity of an operator. In this cases the partitioning of the query it is not limited to the grouping of operators into subqueries, but goes further rewriting the query graph. The costly operator, or subquery, is multiplied so that it can run in parallel onto several processing nodes. The original input rate is divided by a *router* operator that sends the chunks in turn to the nodes hosting the replicated subqueries. This approach mimics the *map-reduce* approach, with these parallel subqueries representing the map phase. The output of these subqueries has then to be sent to a collector operator that finalises the processing completing the semantic of the original subquery.

Every aggregate operator can be decomposed using a copy of it in each subquery and again a copy for the reduce part. For instance an *average* operator would be multiplied into several *map* copies each calculating the average for a portion of the input. A final average operator is then used in the *reduce* phase, generating a global average from the partial results calculated. Other operators, like *filter*, do not need a further reduce phase, and only make a union of the partial results to generate the final output.

Figure 4.5 shows the original and modified query graph of a query rewritten using the map-reduce partitioning strategy. The original query, with one heavy operator, is transformed so that three copies of this operators run on different processing nodes. A round-robin router operator evenly partition the incoming tuples among them. A final copy of the operator, an aggregate in this case, collects the output of the three copies and performs the final reduce phase.

4.5 Summary

This chapter presented the design of the DISSP prototype, a stream processing engine designed to implement and test the quality-centric data model described in Chapter 3. First, it presented the goals that drove the design of the system, such as the ability to perform efficiently under overload, the embedded calculation of the SIC metric and the possibility of tweaking the load-shedding policy

The DISSP system already implements a number of general purpose operator templates, the following list contains some examples. *Window Operators* provide transformations over window sizes, holding the input of an operator until a certain condition is reached. *I/O Operators* are the gateways to the system, in input they provide the conversion from external to system tuple representation, in output they deliver the query results to the user. *Network Operators* allow the inter node communication, so that a query can be partitioned and distributed onto several nodes. Finally, *Data Manipulation Operator* are used to perform the processing on the data, and are equivalent to the *relation-to-relation* operators in CQL.

Partitioning of Queries

Once a query has been instantiated from its XML representation, it is broken down into chunks called *subqueries*. These are computationally less intensive than the complete query and can be deployed onto a set of *processing nodes*. The partitioning process is arbitrary and can follow many different strategies. This section presents two policies: one that takes into account the processing *load* of the newly created partitions, and another that multiplexes certain particularly heavy partitions over several nodes and executes them in parallel, following a *map-reduce* approach.

Load-Aware partitioning. The partitioning of a query graph can follow the principle of grouping operators so that they have a similar processing load. This *load-aware* strategy tries to obtain subqueries with similar cost in order to evenly spread the processing load and it is similar to the one exposed in [Xin+06]. Since the partitioning happens before the query is deployed, the system estimates the cost of an operator based on its expected *input rate* and *load coefficient*. The expected input rate can be calculated for an operator analyzing its direct predecessors in the query graph and their window operators. The estimate can be more precise in case of *aggregate operators*, like average, since they produce a single output every time they are triggered. For *filters*, the estimate is more difficult and the system assumes a 50% drop rate. The load coefficient of an operator has to be determined offline and indicates the computational complexity of the operator. A synthetic load of fixed size is fed to the operator on a testbed machine and the CPU load is measured. Multiplying the input rate and the load coefficient it is possible to guess the future load of an operator. The initial partitioning can then try to divide the query graph evenly, even though this should only be thought as a starting point to be paired with the run-time migration of operators or subqueries.

Map-Reduce partitioning The cost of a query is not only dictated by the number of operators it employs. Even queries with a very simple query graph can be very computationally expensive

Listing 4.1: XML description of a Tuple

```
<schema name="simpleSchemaONE">
    <field type="long"   name="ts"  />
    <field type="double" name="idx" />
    <field type="double" name="tmp" />
</schema>
```

As described in Section 4.4, Tuples are implemented using a template file. One parent class called "Tuple" is provided, containing the basic logic common to all tuples, and every other tuple class derive from this. In the XML query description, the user specifies a schema and a name for the tuples that will be used by the query and the system creates a new tuple class, with the name and fields required. A generic tuple `.template` file is filled, substituting some placeholders with the provided data, producing the `.java` source file of the new tuple class.

Listing 4.1 shows the XML description of a Tuple object with 3 fields: one `long` for the timestamp named *ts*, and two `double`, one with a numerical identifier *idx* and the other carrying a temperature reading *tmp*. These values will be inserted in the tuple .template in correspondence with the "`$FIELD`" placeholder, transforming it into a complete .java source file. The compiled object will contain 3 public fields with the type and name specified in the XML listing.

Listing 4.2 shows the XML description of an Average operator. In the first line the operator is characterized as being an instance of class "Average", described in the correspondent .template file, and it is given the name of "MyAvgCpu'. Then there is the declaration of a *next* operator, this means that this is not a terminal operator and thus its output should be delivered to a single local operator named "MyOutput". Next are 3 parameter definitions, in the form $\langle name, value \rangle$. The system will look for the "$NAME" placeholder and will replace it with the string given by "value". Once the substitution has taken place, the .template file becomes a complete .java source file and is then compiled by the *CharSequenceCompiler*.

Listing 4.2: XML description of an Average operator

```
<operator name="MyAvgCpu" type="Average">
    <next name="MyOutput"/>
    <parameter name="tuple"    value="simpleSchemaONE" />
    <parameter name="field"    value="cpu"/>
    <parameter name="groupby"  value="idx"/>
<operator>
```

contained in the XML file describing the query producing a new .java file, which is then compiled to bytecode by a run-time compiler.

XML query representation. Queries are submitted to the system using and XML representation. An XML query file contains the complete description of the query. It specifies what kind of tuples will be processed, providing a description of the *tuple schemas* so that operators are aware of the number, type and name of the fields contained in the tuples they process. It also contains the list of operators implementing the query semantic. Every operator is represented by an XML block, containing its description. Every operator needs to be specialised before it can be used within a query. A generic implementation is provided in a *template file*, which acts as a blueprint for the operator, which is then finalised with the data included in the XML block, allowing the correct instantiation of the operator. The information needed to complete an operator templates includes its *name, parameters* and an indication about its *following operator*. Every operator either declares itself as a *terminal* operator, by declaring no downstream operators, or as an *intermediate* operator by declaring that its output should be fed in input to another operator. This allows the system to reconstruct the complete query graph. The information contained in the XML representation is used to generate the complete .java files describing the customised tuples and operator that will be used in the query. This can then be compiled and instantiated so that the query can begin processing.

Templates. Semi-complete .java files, representing the skeleton of a class. They are used for the efficient instantiation of customised *tuples* and *operators*. All tuples share some common characteristics, but are differentiated by the number of fields, their names and types. The same is true for all operators belonging to the same type. For instance Average operators are all equals when it comes to the processing semantic, but they differ about the name and type of the field to be averaged. So a generic blueprint for these object comes from the generalisation of a specific instance, where specific names and types are replaced by some place text tokens called *place holders*. Using the information contained in their XML description it is possible to substitute these place holders with the correct details, transforming a template into a complete .java ready for compilation. Place holders are all capital keywords preceded by a dollar sign, in the form "`$PLACEHOLDER`". They are meant to be completed using the information provided in the XML file describing the query and the compiled into actual POJO objects with the desired characteristics. For this transformation the system employs a *CharSequenceCompiler*, which takes in input a string containing the content of a completed .template file and produces an instance of the customised object. This allows the system to operate always on compiled bytecode, granting the maximum performance of execution together with the flexibility of working with customised versions of tuples and operators tailored to the specific query requirements.

1. First, the query graph is submitted in XML format. This file contains the query specification, with a list of tuples and operators to be used. It also contains information about the links among operators.

2. The *submitter* compiles and instantiates the XML query specification, compiling the tuples and operators objects. Then, it starts a *coordinator*, the entity in charge of deploying and monitoring the status of the query.

3. The *coordinator* creates N subqueries from the original query graph employing a user defined partitioning policy. Then it contacts the *oracle* to obtain a set of N nodes for deployment.

4. The *oracle* returns a list of the N most suitable nodes ready for deployment. This choice is dictated by system-wide policy, like choosing the least loaded nodes or nodes with a low latency among them.

5. Once the *coordinator* receives this information, it proceed by deploying the individual subqueries onto the provided *processing nodes*.

6. Each *processing node* instantiate the assigned subquery and connects the query, establishing the needed remote connections among operators.

7. Finally, the *input provider* operators connect to the external sources and start feeding tuples to the query. The query is then deployed and starts processing.

Submission of Queries

The life of a query begins with the submission of its query plan by a user. This is done through the *submitter* component. This interprets the XML query specification and compiles a set of custom operator instances. Operators and tuples are compiled at run-time for better performance. The next paragraphs introduce the concept of *compiled* operators and tuples and how they are implemented in DISSP.

Compiled Operators and Tuples

In DISSP tuples and operators are implemented as *compiled POJO objects*. This choice was made taking performance into consideration, since other options, based for instance on *reflection*, were considered too slow and cause of a potential performance bottleneck. To instantiate a custom tuple or operator object, based on the query requirements, a text *template* is completed with the information

even deployment or to a skewed distribution of the subqueries. This number is sent to the oracle and can be used to evaluate the goodness of the chosen deployment strategy.

Number of connected subqueries. A value used to check the correct deployment of a query. Every subquery contains at least one incoming network operator (Remote Receiver) and one outgoing network operator (Remote Sender). When the deployment is successful the number of established connection is equal to the number of these operators, a discrepancy indicates that one or more queries are disconnected either temporary or permanently.

4.4 Lifetime of a Query

This section describes the workflow of a query, from when it is submitted to when it starts processing. It illustrates how different aspects of the system have been implemented by following the process of instantiating, deploying and connecting a query. First, a query plan is submitted in XML format. Based on this, the system creates a set of customised operator instances. Once the query has been instantiated, it is broken down into subqueries employing a flexible partitioning policy. Each subquery is then deployed on a processing node, chosen from a set of suitable candidates provided by the oracle. Finally the different subqueries connect to each other, tuples start flowing from the input providers and the processing can begin.

Figure 4.4 shows the steps involved in the deployment of a query.

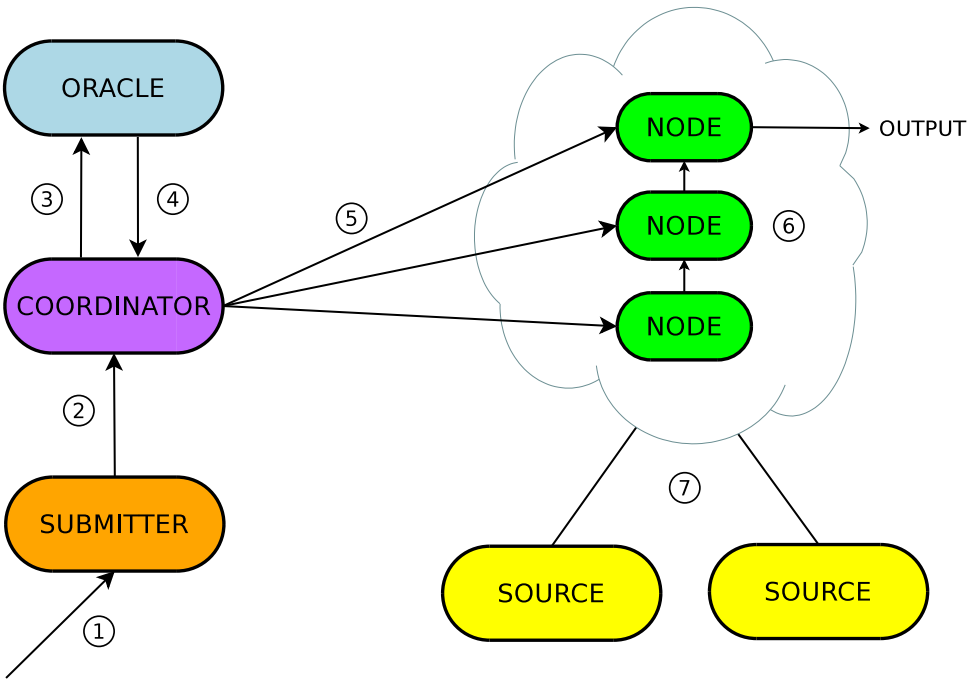

Figure 4.4: Steps involved in the deployment of a query.

Statistics Manager

The *statistics manager* is the component responsible for the calculation of the global performance metrics of a processing node. The metrics gathered by this component are used to improve the overall performance of the system. The load-shedder can exploit the knowledge about the the local utility of a query to implement a fair shedding policy, trying to provide the same quality-of-service to all the queries. The oracle, using the information about latency and throughput of nodes, can select the set of nodes for a new deployment trying to avoid those already heavily loaded.

It runs periodically, by default once a minute, and calculates a number of statistics, some of which are logged locally and some propagated either to the coordinator or to the oracle. The following is a list of the most significant metrics and their function:

Average SIC. The average SIC value achieved by queries hosted at this node. Every subquery is aware of its global SIC value, as this value is propagated from the coordinator to all the participants nodes. This metric is sent to the oracle and can be used during the deployment of a new query. In the scope of a fair deployment strategy, a node already hosting pieces of queries achieving a low global SIC value should not be chosen as a host for a new subquery. Adding further load to the node in fact may lead to an overload condition, thus further depleting the performance of the currently running queries.

Average tuple drop. The average number of tuples dropped by the node in a chosen time interval. This value helps evaluating the amount of overload experienced by a node. This value is sent to the oracle and it can be used by a deploying strategy to place a new subquery onto nodes that are not overloaded or to avoid placing it on a node experiencing already a high tuple loss.

Average throughput. The average number of tuples processed by the node in a chosen time interval. This value can be used to evaluate the spare processing capacity of a node. By recording the value of this metric when hitting an overload condition it is possible to have an indication about the number of tuples per second that this node is able to process on average without any loss. This value is sent to the oracle and can help a deployment strategy to choose the least loaded nodes out of those currently not experiencing overload.

Average latency. The average latency in milliseconds for all the hosted subqueries. A measure of how much time a tuple spends in this node, from when it is received to when it is delivered in output to the next node. This value is sent to the oracle and can be used to detect performance bottlenecks and to evaluate the goodness of the chosen deployment strategy. A good strategy would result in similar latencies at all nodes.

Number of running subqueries. A number used to check if the deployment strategy leads to an

The system keeps track of its *output rate*, the amount of tuples processed within one shedding interval. It uses this to calculate the average time needed to process one tuple, called *tuple cost*. Since the shedding interval is fixed, it is possible to calculate the amount of tuples that the system will be able to process before the new shedding iteration, by dividing the *shedding interval* by the *tuple cost*. This provides the amount of tuples *to keep*, subtracting this number from the *total number of tuples waiting*, the system obtains the *amount of tuples to be dropped*. These numbers depend on many factors, for instance the number of tuples received by each subquery hosted can be different in different interval, or the query semantic might produce a variable load depending on the values contained in the incoming tuples (i.e. in the presence of a Filter). Therefore the system uses a *sliding window* to average a value over a few past iteration, reducing the variability of these variables.

Execution cycle. The periodic cycle followed by the loads-shedder is as following:

1. Pause the thread pool
2. Calculate the amount of tuples to be shed
3. Choose what tuples to shed
4. Shed the chosen tuples
5. Resume the thread pool
6. Update the load metrics

First, the thread pool has to be stopped, in order to count the number of tuple it contains in its pending queue at this moment. The processing of tuples can not happen during the execution of the load-shedder, so that it can choose what tuples to drop from a static set. Then, it estimates the amount of tuples to be dropped, using the logic exposed in the previous paragraph. Once the number of tuples to keep is calculated the load-shedder can make a decision about what tuples should be dropped out of the ones available. This choice depends on the semantic of the Load Shedder and takes the name of *shedding policy*. A more detailed description of different shedding policies will be given in Chapter 5.

Then the actual tuples are dropped by removing the corresponding work unit from the thread pool queue. In the DISSP prototype the granularity of dropping is at the level of *batches*, since this is the input/output unit between operators and every work unit is associated with a batch. After performing the dropping of tuples, the thread pool can be restarted and thus the processing of tuples. Before ending its cycle, the load-shedder calculates the new updates the current *tuple cost* and other internal metrics. Once it has finished executing, it calculates the time it took for processing, called *shedding time*, it subtracts it from its fixed interval of execution, called *shedding interval*, obtaining the amount of milliseconds to sleep before its next execution.

Work unit. A work unit is a container class implementing Runnable that contains the blueprint to deserialise, route and process a batch of tuples. Once a work unit is removed from the pending jobs queue and selected for processing, it gets executed by a thread from the pool. It starts by deserialising and instantiating the tuples contained in the message received from the net. Then it checks to what query it should be delivered. In case the recipients are multiple, it creates a copy of itself for each other query interested in its payload. Then it calls the `.process()` method on the first operator of the query graph. This method takes the incoming tuples ans executes the operator logic, producing one or more batches in output. The work unit passes these tuples to the next operator and continues processing until it reaches the end of the local subquery graph.

A work unit will continue processing as long as possible, instead of creating a new work unit for each operator, allowing the system to achieve a higher throughput by reducing overhead. Not introducing a new work unit for each operator also guarantees that once a batch of tuples starts processing it will not be touched by the load-shedder, while the one work unit per operator approach sometimes resulted in a batch being processed by a few operators just to be discarded by a later execution of the load-shedder. If the subquery graph contains an operator with more than one follower, like in a fan-out query, the current work unit will continue processing on one branch, while creating a new work unit for each other following operator.

Load-Shedder

The *load-shedder* is the component delegated to the *overload management* of a processing node. When the amount of input tuples rises over a certain threshold, the resources available at a node become insufficient for their processing. More tuples are given in input to a node than the ones it can process, thus leading to the accumulation of jobs in the pending queue. This leads to a growing increase in latency for the tuples and to the never or the cost of processing ending growth of the internal queues, until the available memory finishes, leading to a system crash.

Periodic evaluation. The *CheckOverload* thread periodically runs and evaluate the current load situation of the node. In the DISSP prototype a *shedding interval* of 250ms has been chosen, which allows a timely response to overcome an overload condition while keeping a reasonable performance overhead. Every time this thread executes, it calculates the number of tuples the node will be able to process before the next load check. If this number is greater than the number of tuples currently in the queue waiting to be processed, the system is not overloaded and there will be no dropping of tuples. On the other hand, if the number of awaiting tuples is larger than the ones the node will manage to process, a certain amount needs to be discarded.

connecting to the oracle, the web interface produces a report about the status of all processing nodes, together with a summary of the performance achieved by all queries, sorted by SIC value. The web interface of the oracle is useful to monitor the overall performance of the system and to evaluate the effectiveness of the shedding policy.

Operator Runner

Once a batch of tuples has been received it gets passed to the *operator runner* to be processed. This is the component in charge of handling the operator processing. It employs a fixed amount of threads, each executing a chain of operators at the time. Bounding the number of processing threads has the advantage that tuples are processed in the order of arrival and provides more flexibility to the load shedder, that can freeze the queue of pending jobs and analyse it to make the dropping decisions.

The original message, still in its network format, is encapsulated into a *work unit*, a Runnable class representing a future job to be processed. This is submitted to a ThreadPoolExecutor and placed into a pending queue, until one of the threads in the pool becomes available and executes it. The work unit contains the logic to deserialise, route and process the tuples contained in the message it was assigned. The last operator of the subquery graph usually is a RemoteSender, that takes the result of the subquery computation and send it to the next processing node through the network layer. Once a work unit has terminated its execution it terminates and frees its execution thread so that a new work unit can be processed.

Thread pool. At the core of the *operator runner* there is a *thread pool*, ready to executes work units as they are submitted. It is a subclass of ThreadPoolExecutor, that augments its parent class with the possibility of stopping and resuming execution, needed by the load-shedder to inspect the pending jobs queue. The number of threads is set to the number of available CPU cores. The pending jobs queue is a list of work unit. As soon as one of the threads in the pool is available it removes the oldest work unit from the queue and executes it.

When the system is overloaded the number of items added to the queue is larger than the number of items removed, thus making the size of the queue grow, and thus the latency of processing, eventually leading to an out-of-memory exception. For this reason, the Thread Pool is periodically interrupted by the load-shedder, that inspects the pending batches and decides if there is the need of dropping a certain portion of them in order to overcome overload. The choice about which ones to discard is part of the shedding policy of the load-shedder.

which states that the conversion from network format of tuples should happen as late as possible. Tuple objects are instantiated only when their are scheduled for processing, even their destination, the queries they belong to, is not known until then.

The reason for this choice is that instantiating tuples and finding their destination is a costly process and it could also be useless to perform it early, since at a later stage the whole batch could be dropped by the *load-shedder*. Delaying this operation as much as possible ensures that no resources are wasted to instantiate and route tuples that might never be processed. Tests have shown that an early deserialisation also posed a great processing burden on the RequestHandler thread, which was not able to handle all messages in a timely fashion under heavy load, leading to a growing message queue and eventually raising an out-of-memory exception.

Command messages. A message can also contain a command, triggering the corresponding remote procedure call. This can be used for communication purposes, reporting information about the status of a node, or of a query to the oracle or the query coordinator. Every node, for instance, periodically reports its throughput, average tuple latency and load information to the oracle. Every node then also reports the achieved SIC value for each query to the respective coordinator, which in turns calculate an aggregated value to be sent to the oracle.

Command messages can also be used to obtain information from another node or from the oracle. For instance, when a coordinator is instantiating a query it sends a message to the oracle requesting a list of nodes onto which to deploy the query. During the initial connection stage, when the subqueries running at different nodes need to connect to each other, it is usual that a node has to wait a certain amount of time for the other node to be ready to accept a connection. A message is used in this situation to probe the availability of the another node and to wait until it becomes ready.

Another use for command messages is to propagate certain values to a set of nodes. Every node, for instance, needs to be aware of the final SIC value achieved by all the queries that are running on it. Using the global SIC value achieved by every query and comparing the local value of the tuples it processes, the load-shedder can implement an intelligent dropping policy. So every coordinator periodically spreads the global SIC value of its query to all the nodes hosting its subqueries.

Web interface. The RequestHandler is also the gateway to the node *web interface*. When receiving a message it checks if it is an HTTP request and if so it replies with a web page containing information about the current status of the node, using the information obtained from the StatisticsManager. This includes the number of subqueries hosted together with their performance, as well as some global metrics describing the global status of the node, like its average throughput and latency. When

Network Layer

The network layer is the component responsible for all the incoming and outgoing communications of a processing node. It is composed of two threads: the *NioConnector*, in charge of handling all network communications, and the *RequestHandler*, which interprets the content of the received messages and acts upon it. The choice of employing a many-to-one design, where many remote connections are handled by a single pair of receiving threads, is motivated by the better scalability that this approach offers. In this way the number of active threads within a processing node is constant, avoiding the context switch overhead that a one-to-one threading approach would have.

The *NioConnector* thread acts as a receiver for remote messages, as well as a sender for the outgoing messages. All communication is done through TCP connections, this choice was made in order to preserve the integrity of a network message. Using UDP would have increased the chances of a long message being voided by the loss of one of its UDP chunks. The handling of sockets is based on the Java NIO library and is completely asynchronous. This ensures a low communication overhead and allows one thread to deal with all the I/O requests. This many-to-one design, where one thread is responsible for all sockets, was preferred to the one-to-one approach, with one thread for each open socket, because of its simplicity and efficiency, only employing one CPU core to handle network communication. Tests have shown that even under heavy load and the presence of a large number of connections, the NioConnector never becomes a performance bottleneck, since its semantic is only limited to the reading and writing of network data. As soon as an incoming message has been fully read, it is placed in a queue of pending messages, waiting to be interpreted by the RequestHandler.

The *RequestHandler* thread is responsible for the processing of incoming messages. It waits for the NioConnector to place them into its queue and process them as soon as they become available. All inter node communication within the system is encapsulated into *messages*. The chosen format for messages is UTF-8 text, with binary chunks expressed in Base64 encoding. This choice was made for simplicity and because it allows for easier debugging, even though a completely binary representation could have been used as a further optimisation. The initial task performed by the RequestHandler is to interpret the beginning of the message to categorise it and process it accordingly. A message can be of 3 kinds. It can be a message containing *tuples* to be delivered to some subquery to be processed, it can contain a *command message* triggering the execution of a remote procedure call, or it can simply be the request to access the *web interface*.

Tuples handling. Once a message has been determined as containing a tuple payload, it is directly passed to the *Operator Runner* component, without further processing. The content of the message stays in network format, thus being passed as a string. This follows the principle of *lazy deserialisation*,

multiple queries.

All inter node communication passes through the *network layer*, the components responsible for the handling of *command messages* and *tuples*. When a command message is received, it triggers an action that is performed directly at the network layer level. If a *batch of tuples* is received instead, it is passed directly the *operator runner* component. This layer also provides a *web interface* that allows its remote monitoring.

The *operator runner* is in charge of delivering the incoming tuples to the appropriate subqueries, and to execute the operators in their graphs. It employs a pool of concurrent threads to achieve scalability, so that multiple operators can run in parallel. Once a subquery has completed its processing on a set of tuples, it sends them to the network layer, that will send them to the following node to continue processing.

If the incoming input is too large and the node resources are not sufficient to processes it completely without running out of resources, the *load-shedder* component is in charge of selecting a portion of it to be discarded in order to overcome the *overload* condition. The selection of the tuples to discard is determined by a *shedding policy*. Section 5.3 deals with a strategy trying to *fairly* shed tuples with the objective of maintaining a similar quality-of-service(i.e. SIC value) for all queries

Finally, the last important component of each processing node is the *statistic manager*, that is in charge of collecting a number of statistics about the performance of the different queries and of the node as a whole.

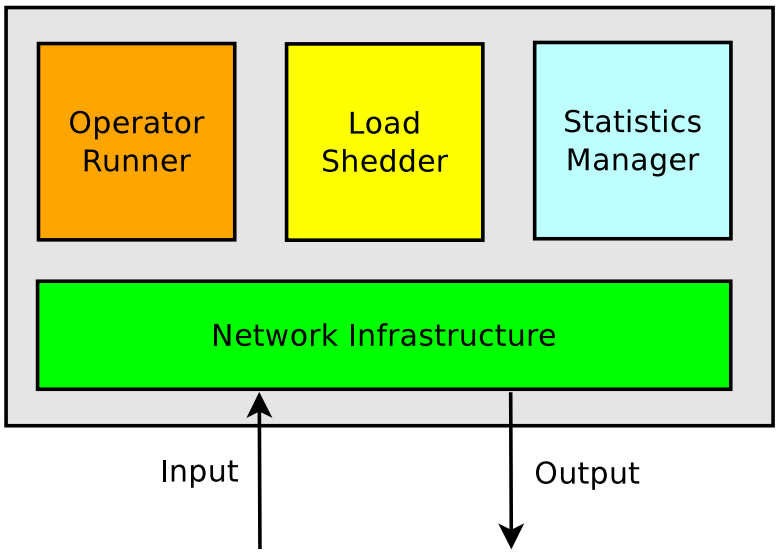

Figure 4.3: A high level view of a Processing Node, showing its internal logical components.

for processing. Every source can be the entry point to many data sources, by employing multiple *InputDevice* operators. Each of them connects to an external source, for instance the sink of a sensor network or to the Twitter firehose.

Oracle. The *oracle* is the system-wide component that overlooks all the processing happening within the system. It has two main functions: *monitoring* and *managing*. It provides an overview about the status of all queries, so that is possible to know their achieved quality-of-service in real-time. It also reports the status of all processing nodes, providing information like their throughput, number of queries running and their load status. The oracle is also in charge of providing a list of suitable nodes ready for new queries. In fact, whenever a new query is submitted to the system, its *coordinator* asks the oracle to provide a list of N nodes, which are best suited to host the new job. The *oracle* provides these nodes following a *deployment strategy*, ranking all processing nodes according to some criteria. For instance, it could choose to provide a list of the *least-loaded* nodes, to reduce the occurrence of overload conditions, or simply provide a set of node based on a *round-robin* policy.

In the DISSP prototype the *oracle* is a centralised component and only one instance of it exists for each system deployment. It lives on a dedicated machine and receives messages from *coordinators*. These report the quality-of-service and other information about the query they manage, and also ask for a list of nodes suitable for deployment when a new query is instantiated. The *oracle* also provides a web interface, so that it is possible to connect to it using a browser and access a dashboard providing real-time updated information about all that is happening in the system.

In a truly distributed system this choice of a centralised monitoring entity would seriously hinder its scalability. The same functionalities could be implemented using an *epidemic* approach [VGS05], in which for instance all nodes exchange gossip messages and participate in multiple overlays. Therefore a *coordinator* could find the list of the least loaded nodes by participating in the overlay minimising the load metric [VS05], and a user could monitor the status of a query by participating in the overlay of its processing nodes. Even though such a solution would be much more scalable and truly distributed, a centralised approach has been chose for its simplicity, as the gossip-based approach would have been too complex and difficult to implement correctly.

Node Architecture

This section will describe the internal structure of a DISSP *processing node*. A system deployment comprises several processing nodes, that can be hosted at many different sites, onto which queries can be deployed. Each node only hosts a partition of a query graph, called *subquery*, while the complete query can span onto several nodes. Every processing node can host several subqueries belonging to

into concrete instances. These tuples are sent to the graph of operators in charge of their processing, usually a subset of query deployed onto many nodes. After they have been processed, tuples are sent to the next node to be further processed by the next subquery, or are sent in output if there is no more processing to be done. Every *processing node* periodically reports to the query *coordinator* about the performance achieved in terms of quality-of-service, and to the *oracle* about its throughput and load condition. All *processing nodes* are also in charge of performing *load-shed* whenever an overload condition arises, making local decisions about what input tuples should be dropped in order to reduce load. A load-shedding policy designed to achieve *fairness* among all queries running in the system will be the topic of Chapter 5.

Submitter. Queries are deployed onto the system through the invocation of a *submitter*. This receives in input an XML file containing the query specifications, including the *tuple schemas* to be used and a *list of operators*, specifying their type, name, parameters and connections. Based on this query description, the *submitter* then compiles the individual components of the query using a runtime compiler, obtaining a set of customised object instantiations representing the concrete query in memory. The newly instantiated query is then transformed to its network representation so that it can be transmitted to a *coordinator*, the component responsible for deployment and management of the query.

Coordinator. Every time a new query is instantiated a new *coordinator* is created to manage it. This is the entity in charge of deploying the query on the different *processing nodes* and all nodes report to it about their performance. While a submitter can spawn many queries and then terminate, every query is assigned a *coordinator* that is active for the whole query lifetime. When the query has to be initially deployed, the coordinator contacts the *oracle* to receive a list of nodes suitable to host it. It then partitions the query graphs in a number of processing units, and assigns one of them to each *processing node*. After the query has been deployed and started, it stays alive and acts as a mediator between the *processing nodes* and the *oracle*, gathering statistics about the query and transferring them to the *oracle*.

Input Providers. An *input provider* is the component that allows input tuples to enter the system, acting as a collector of external data. The DISSP system is not concerned with the harvesting of input tuples. The data it processes is collected or generated elsewhere, for instance by a *sensor network* or through a *social media environment*. When the data is available for processing it is passed to the stream processing system so that queries can be computed over it. The input provider component acts as a *gateway*, where data is converted to the system representation and made available to queries

System-wide Components

This section presents the system-wide components composing our prototype stream processing system. The most important component of all is the *processing node*, a dedicated machine used only for the processing of queries. One or more processing nodes compose the resource pool onto which all queries are deployed. Since a query is usually divided into sub-queries that are hosted on different processing nodes, a *coordinator* is deployed for each query. This is usually hosted on a separate machine and is in charge of monitoring the status of the query, detecting failing nodes and gathering statistics about the query performance in terms of achieved quality-of-service. A number of *sources* provide input to the processing nodes. A system-wide *oracle* is used to obtain a global view of the system, it is aware of all queries and their performance, allowing to effectively monitor the behaviour of the system as a whole, and is used during query deployment by the coordinator, providing a list of processing nodes suitable for the deployment of a new query. Finally, there is also a *submitter*, which is the initial user interface used to submit one or more queries for execution.

Figure 4.2 provides a high-level overview of the DISSP system. Many *processing nodes* are deployed on a cloud environment, while some *sources* provide the input data to be processed. Queries are deployed through a *submitter*, each having a *coordinator* in charge of its management. An *oracle* overlooks the processing of all queries, gathering information about their status and performance, also providing the the set of nodes onto which nodes new queries should be deployed.

Processing Node. The *processing node* is the component in charge of the execution of operators. It receives tuples through the network, which are converted them from their serialised representation

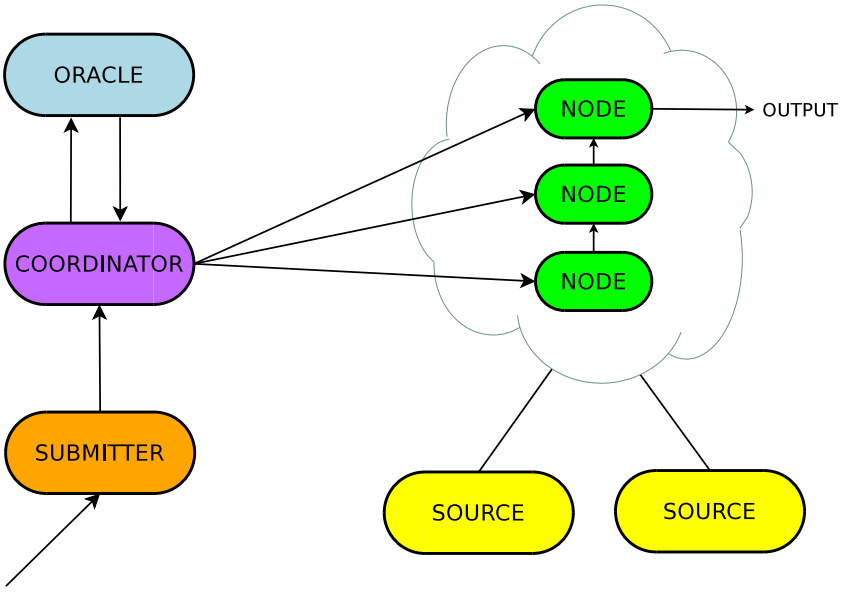

Figure 4.2: A system-wide overview of the components acting within the DISSP prototype.

delays, time windows, etc. To address this challenge, we consider a source information tuple set that includes all source tuples generated at sources during a predefined time period, referred to *Source Time Window (STW)*.

The length of a STW is chosen so that is always longer than the end-to-end query processing delay. In this way the STW contains all tuples that will go into the generation of future results. In reality, a STW might include source tuples that would generate multiple different result tuples, or belonging to different source information tuple sets. To this end, we apply the same STW at the result stream as well: the SIC value of a query result is the aggregate SIC value of all result tuples generated during the same STW. In order to capture the continuous processing of data streaming, we treat the STW as a sliding window with a much smaller slide than the window itself.

The SIC value of a derived tuple t, processed by operator o, i.e. $t \in \mathcal{T}^o_{out}$, is:

$$t_{SIC} = \frac{\sum_{t \in \mathcal{T}^o_{in}} t_{SIC}}{|\mathcal{T}^o_{out}|}. \tag{4.1}$$

By using the STW, we no longer require the precise definition of the $\mathcal{T}^o_{in}$ and $\mathcal{T}^o_{out}$ sets. Rather, we use two new sets for each operator o: $\mathcal{T}^{o\prime}_{in}$ is formed by the window operator before o; and, $\mathcal{T}^{o\prime}_{out}$ depends on the semantics of o. Note, that, $\mathcal{T}^{o\prime}_{in}$ and $\mathcal{T}^{o\prime}_{out}$ might not contain the correct set of tuples to produce a result tuple. However, we ensure that the calculation of the result tuples is correct according to the data model definition of SIC, by applying the same STW at the result stream.

4.3 DISSP Architecture

This section describes the overall architecture of the DISSP system. First, it presents the *high-level components* of a system deployment. DISSP is a distributed system that is designed to run on a large number of nodes. Every system-wide component taking part in the processing of a query will be described taking into account its role within the system.

Then, the *node architecture* is presented, looking at the internal workings of a DISSP node. The different components of a processing node are examined, providing details about some implementation choices and the reasoning behind them. In particular, the presentation focuses on the network layer, that handles inter-node communications, the operator runner, that executes the query semantic, the load-shedder, used to overcome overload and the statistics manager, that collects information about the status of the system.

Queries. A query logically groups operators that cooperate in the same processing task. In our implementation a query is organised according to the *boxes-and-arrows* model. The user submits a query specification containing the list of operators and the connections among them. Within a query operators are organised in a directed acyclic graph.

A query always starts with one or more *input operators.* These are in charge of transforming the incoming data streams into the DISSP system format. They act as a gateway, allowing external input generators, like a sensor network, to plug into the system an make their data available for processing. Then there are a number of *data manipulation operators*, that take these data items and process them according to the query semantic. Once a final result has been obtain it is delivered to the user through an *output operator*. The different classes of operators will be further described in Section 4.2.

As described in Section 3.3, the graph of a query can assume many shapes. Fan-in queries only include a single output operator, while fan-out allow for more than one final result and thus may have multiple terminal operators.

Subqueries. Queries are usually too computationally expensive to be run completely on a single processing node. To avoid overload the query graph is usually split into a number of partitions that take the name of *subqueries*. The partitioning of a query can follow many strategies like trying to group operators in groups of equal computational cost, or distribute the same subquery to be processed in parallel. The different query partition schemes will be described in Section 4.4.

These are then deployed onto several processing nodes. For each query running in the system there is a *coordinator*, that is in charge of the partitioning, deployment and management of the query. It uses a system of *command messages* to obtain a set of suitable nodes for deployment and to install the subqueries onto the processing nodes. Once the query has been deployed, data can start flowing through the input operators and the processing begins.

Source Time Window. Section 3.4 introduced the concept of *source information tuple set*: the set of input tuples from which a final result is generated. The final SIC value of a tuple, in absence of failure, is the sum of the SIC values of all tuples contained in the corresponding source information tuple set. According to the model definitions stated in Chapter 3, every source contributes a total unnormalised SIC value of 1 for all result tuples.

To implement this part of the data model it would be necessary to know in advance which tuples will contribute to the creation of a future result, but because this result has not yet been created it is difficult to identify the source information tuple set. However, in practice, it is very difficult to accurately define this set due to time delays occurring during query processing, for instance processing

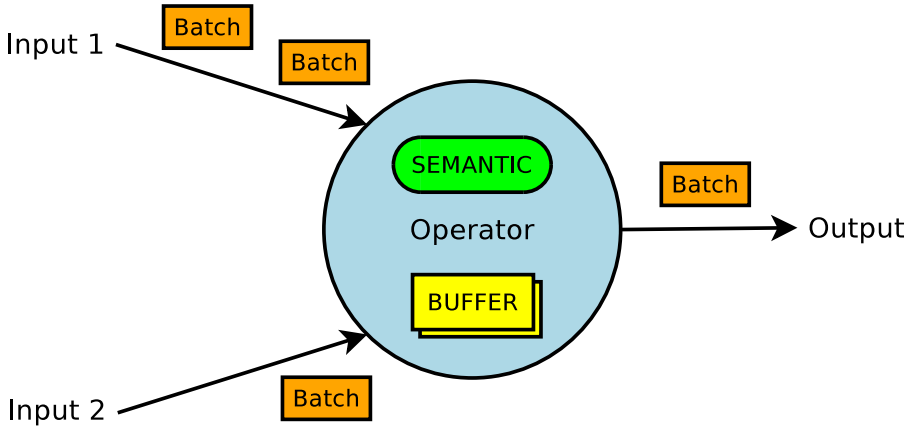

Figure 4.1: An operator with its internal structure.

for operators. A stream becomes then an abstract entity, as a possibly infinite set of batches, while batches serve as the actual units of information within the system. In our system batches are simple objects that contain a list of tuples, a SIC value and some other metadata. They also contain the logic for converting to and from their network representation.

Operators. Operators are the basic unit of computation within a stream processing system. They implement a *function* that transforms one or more input batches into one output batch. A set of operators linked together in a directed acyclic graph takes the name of *query*.

Operators can be classified as *blocking* or *non-blocking* based on their behaviour when dealing with input data. Many operators can be configured to work in either mode. *Blocking operators* need at least one input batch on every input channel before they can produce an output, while *non-blocking operators* execute every time a new batch of tuples is delivered on either input. The system implements the logic described in Section 3.1.

Figure 4.1 shows the internal structure of an operator, with its two main components: the *buffer pool* and the *semantic module*. Every operator is equipped with a pool of internal buffers, one for each input channel. These are used to stage tuples before they are processed. Every time an operator triggers for execution, it first moves one or more batches of tuples into the corresponding buffer within the pool. Every buffer might still contain some tuples from the previous execution cycle, in this case the new batch is merged with the left over tuples.

After the input data has been staged, the operator executes the logic contained in the *semantic module*. This is what really characterises an operator, where its processing logic is outlined. From the implementation point of view, an operator is an abstract class with an unimplemented `execute()` method. Every concrete operator class has to provide the code for this method, specifying a processing *semantic*. Once done processing, this method returns a batch of tuples which is delivered in output.

4.2 Implementing the Model

This section introduces the basic building blocks of our system. It explains how the theoretical entities presented in the description of the data model in Chapter 3 have been actually implemented. It starts describing the basic containers for data, such as *tuples* and *batches*. Then it covers the *operators*, providing a taxonomy and describing some details about the implementation of the most common ones. It then describes *queries*, showing how they serve as a logical container for operators and can be partitioned in smaller entities for distribution. Finally, it introduces the *Source Time Window*, that implements the abstract concept of Source Information Tuple Set that is used to assign SIC values to input tuples.

Tuples. Tuples are the simplest unit of information processed by a stream processing system. As described in Section 3.1, they are modeled after a *schema*, stating the number, the name and the type of the values they carry. These represent the *payload* of a tuple, the data processed by operators in a query. Every tuple also contains a *timestamp*, a temporal indication about their time of creation. In our system this is expressed in POSIX time, the number of seconds that have elapsed since midnight Coordinated Universal Time (UTC), January 1, 1970. A timestamp is usually set externally for *base tuples*, or set by the system to the system when creating *derived tuples*. Tuples are also associate with an individual SIC value. This is the metric introduced in the previous chapter, expressing the quality of the data carried in a tuple. In our implementation this value is actually carried by the container batch and not attached to every individual tuple. This reduces the overhead of transporting a `double` value (32 bits) for each tuple, exploiting the consideration that all tuples within a batch have the same SIC value.

Batches. Batches are logical groups of tuples with the same SIC value. Instead of associating an individual metadata value to each tuple, the system uses batches as containers with a single SIC value, which is considered to be assigned to all the tuples contained in that batch. Batches are the input an output units of operators. The output of an operator is usually composed of several tuples, which are encapsulated within a batch. The operator calculates a SIC value to be assigned to all tuples in the batch based on the SIC values of the input batches and its processing semantic. The newly produced batch is delivered in input to another operator or is returned as a result to the user.

Batches are an implementation of CQL concept of *relation* described in Section 2.2. They represent a finite snapshot of a stream, which allows operators to process a possibly infinite continuous stream of tuples in consecutive discrete steps. By using batches the system does not need to use *stream-to-relation* and *relation-to-stream* operators, as batches provide a unified input/output units

the quality-of-service achieved by the queries running on it. Every stream is augmented with a SIC value and operators automatically perform the calculation of the new values for their output. Using SIC values the system is able to operate under overload, to reason about the performance of queries and to make intelligent load-shedding decisions.

Efficiency under overload. The system provides an *efficient processing* of tuples and operators because, even though it is designed to cope with resource overload, its occurrence should be avoided as much as possible. Hitting an overload condition means having to discard some of the input and this situation is always detrimental to the user. Especially because the system is designed to operate in resource constraint environments it should try to maximise its throughput so that the limited resources available can be fully exploited. Overload is considered to be the normal running condition of the system and not a transient condition to be overcome. xt Under overload the efficiency of the system is inversely proportional to the quantity of load-shed necessary, so that every increase in system performancebf translates to a decrease in the amount of input to be discarded, and thus to an increase of the SIC values of the delivered tuples.

Meta-data management. The system employs the SIC metric to reason about its performance under overload. The SIC value of the tuples delivered in output to the user is an indicator of the amount of load-shedding experienced by that query. It provides a mechanism to the system to track the occurrence of failure and to estimate its impact on the quality of the processed data. It also provides a constant feedback to the user about the quality-of-serviced achieved by the running queries. Every tuple is assigned a SIC value indicating the amount of information that went into its creation. The mechanism for SIC metric calculations is embedded in all operators that, after processing, assign a new value to their output tuples. Every time a tuple is lost, either due to failure or to load-shedding, the SIC value of that query is decreased.

Flexible load-shedding policy. The use of SIC values also allows the system to employ flexible policies when making load-shedding decisions. When facing the decision about what tuples should be dropped, their SIC values provide valuable information to the shedder. A perfect value indicates that a tuple was produced in absence of failure, while a lower value gives a measure of the amount of lost information occurred during its creation. Using this information the shedder can decide to privilege some queries more than others, reasoning about the impact that the selection would have on their performance. It uses the local decisions at every processing node to implement a global shedding policy. Chapter 5 will deal with the implementation and testing of a *fair policy*, that tries to minimise the difference in SIC values experienced by all queries running within the system.

Chapter 4

DISSP Design

This chapter introduces the DISSP stream processing system, a prototype developed to implement and evaluate the quality-centric data model described in the previous chapter. It describes some of the novel features of its design and their implementation. It shows how the system employs the Source Information Content (SIC) metric to *perform efficiently under overload* and to *provide feedback* to the user about the achieved quality-of-service of queries.

The system has been designed from the ground up to be distributed, allowing the processing of queries to span onto many computing nodes. It was particularly targeted at environments with a *constrained amount of resources* that can not easily be scaled. Examples of such deployment settings include, for instance, *federated resource pools* with different authorities administering different processing sites; or *cloud deployments* where the resources are rented and it might be more cost effective to voluntarily reduce the amount of processing resources, trading an acceptable reduction in the correctness of results for a substantial reduction of the operational costs.

Overload is assumed to be the normal condition of operation of the system and not a transient, rare event. The SIC values of tuples are used as a hint to their importance, so that the system can perform an *intelligent selection* of the tuples to shed. This allows the creation of *flexible shedding policies*, as it will be illustrated in the next chapter. This chapter describes the implementation choices made to provide a reliable and efficient processing of queries in such an hostile environment. It also presents and overview of the different system-wide components and their interaction to deploy and run queries.

4.1 Design Approach

The design of the DISSP prototype follows the ideas behind the quality-centric data model described in the previous chapter. DISSP places the SIC quality metric at its core and uses it to reason about

the ones produces by the counter operator and 1/2 the one produced by the Ratio operator. If we sum together these value though we obtain a total SIC value of 1 for the query, which indicates that no failure occurred during the processing.

3.6 Summary

This chapter presented a data model for stream processing systems that takes the quality of the processing into account. First, there was the definition of its basic entities, such as tuples, streams and queries, as intended in the scope of this work. The need for handling and quantifying failure led to the introduction of a quality metric called Source Information Content (SIC). The definition of this metric followed the reasoning outlined in some assumptions and considerations to make it applicable in a generic stream processing system supporting operators with arbitrary semantics. Then, there was an introduction to the different classes of queries, namely fan-in and fan-out. This led to the definition of the SIC metric with a set of formulas for its calculation. A few real-world queries were then used to show how this can be applied and how it can be used by the system to keep track of the amount of failure occurring during their processing. The next chapter describes the design of a stream processing prototype developed to implement the quality-aware query model and exploit the SIC quality metric.

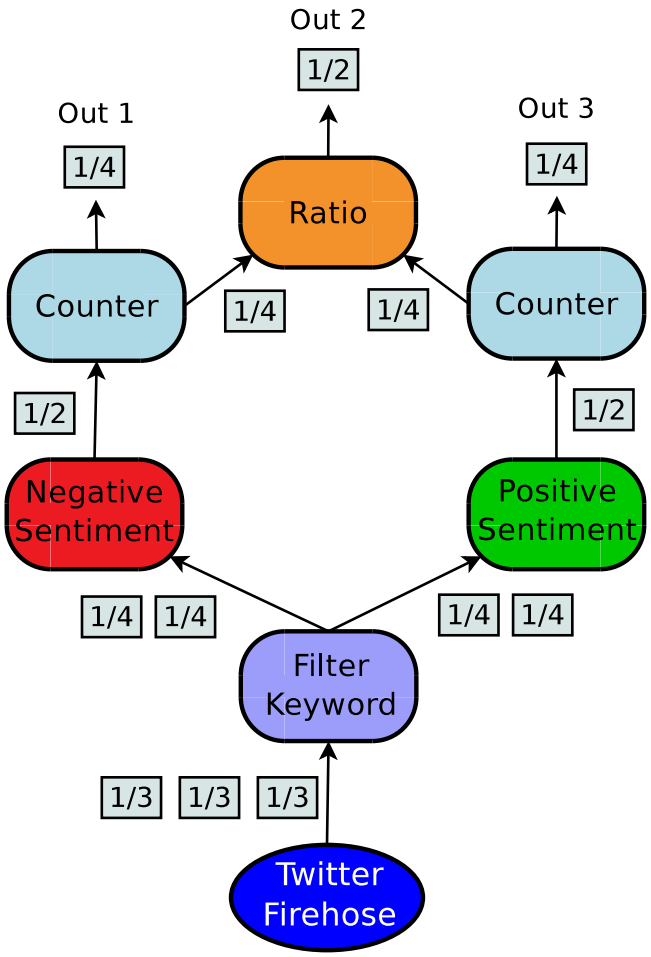

Figure 3.9: An example of fan-out query processing Twitter data. It counts the occurrence of positive and negative mentions of a certain keyword, giving also their ratio. The numbers shown on tuples are the individual SIC values.

the total SIC value has to be distributed over all the output tuples. This is also in accordance to Assumption 6 (Information Conservation), as the amount of information in input is equal to the amount in output. Each output stream of the Keyword Filter operator then contains an identical set of 2 tuples, each having an individual SIC value of 1/4. The positive and the negative filter then each outputs a single tuple, with a SIC value of 1/2.

These go in input to a Counter operator on each side. These operators produce 2 output streams, one that is terminal and is delivered as a result and another that feeds into a Ratio operator. Again the output of the Counter operators is multiplexed and thus the individual SIC values of the produced tuples is scaled accordingly. In our example each Counter produces 2 identical batches of 1 tuple having a SIC value of 1/4. The Ratio operator outputs the third result of the query, producing a single tuple with SIC value of 1/2.

This query computes 3 different results: the number of messages with a positive mention of a keyword, the number of with a negative one and their ratio. The result tuples have different SIC values, 1/4

having an individual SIC value of 1/300. Each stream is sent to a different processing node where a NLP operator assigns a coefficient to each message. These operators output the same number of tuples they receive in input, thus producing 100 tuples each with an individual SIC value of 1/300. Finally all the tuples are received by a Top-10 operator which outputs the 10 tuples having the maximum coefficient. Each output tuple has a SIC value of 1/10, bearing a total final SIC value of 1.

Multiple results queries. Figure 3.9 shows a query calculating the occurrence of positive and negative mentions of a certain keyword and also calculates their ratio. The original stream comes from the Twitter firehose and contains an unsorted stream of messages. In our example, during a time-window, it delivers 3 messages with individual SIC values of 1/3. Then a Filter operator select only those messages containing a certain keyword, in our example it outputs 2 tuples. At this point the output of the Filter operator is *multiplexed* over two different operators, one that filters only messages containing a positive reference to the keyword and one filtering for negative references.

Assumption 9 (Queries are DAGs) tells us that when such a multiplication of the output occurs,

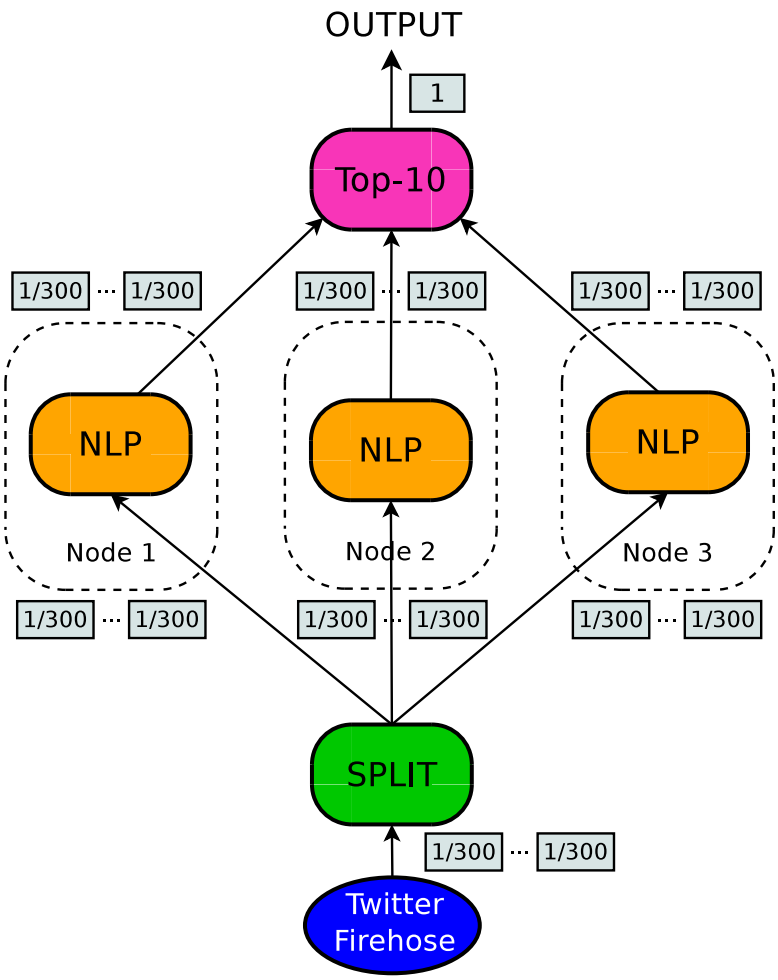

Figure 3.8: An example of fan-out query processing Twitter data implementing the map-reduce paradigm. Source Information Content values are shown for each tuple.

tuple set contain the same amount of information. Figure 3.7 show that each source assigns a total value of 1 to the tuples belonging to the same source information tuple set window, and that the individual SIC values are different according to the number of tuples contained in it. In this example the first source produces only 2 tuples and thus the individual SIC value is 1/2, the second source produces 3 tuples with an individual value of 1/3 and the third 3, each with a value of 1/3.

The *Average* operator takes in input a batch of tuples, of size 2, 3 and 4 respectively, and outputs a single tuple. Assumption 6 (Conservation of Information) states that the amount of information going into an operator is equal to the amount in output, therefore the newly generated tuples are all assigned a SIC value of 1. This is the sum of the individual SIC values of the input tuples.

The final *Max* operator takes in input 3 tuples, each with SIC value of 1, and outputs a single tuple with SIC value of 3. Even though the processing semantics of the Max and Average operators are different, the calculation of SIC values remains the same, as stated in Assumption 3 (A Generic Metric). This final tuple contains the total amount of information carried by the all the initial input tuples.

Let us consider what would happen to the final SIC value in case of the loss of a tuple, for instance a source tuple produced by Source 3. In this case the amount of information of the tuple generated by the average operator on the right would be 3/4 instead of 1. The loss of information, in the amount of 1/4 would then be propagated to the final tuple, which would have a SIC value of 9/4 instead of 3. Since the calculation of the SIC values is additive, the amount of information missing, compared to the maximum theoretical value, would be equal to the sum of the SIC values of the individual tuples that were lost during the processing.

Fan-out

Split computation queries. In this query 300 messages are processed during a time-window, each is analysed by a Natural Language Processing (NLP) operator that calculates a coefficient based on the text of the message text, finally a max operator outputs the message with the highest coefficient. We assume that the NLP operator is very computationally intensive and would overload a single node at the current rate. Therefore the computation is split onto 3 different nodes, each processing 1/3 of the messages, which can then process all the messages without the need to discard a portion of them. In this case the *Split* operator only makes a partition of the original stream, without data duplication, meaning that the tuples it outputs still have the same SIC value as the ones in input. This follows from Assumption 9 (Queries are DAGs). In this query the Split operators receives in input 300 tuples each with a SIC value of 1/300 and outputs 100 tuples on each output stream, again

explored in Section 5.3.

3.5 SIC Use Cases

This section returns on the sample queries presented in Section 3.3 and uses them as running examples to illustrate the calculation of SIC values.

Fan-in

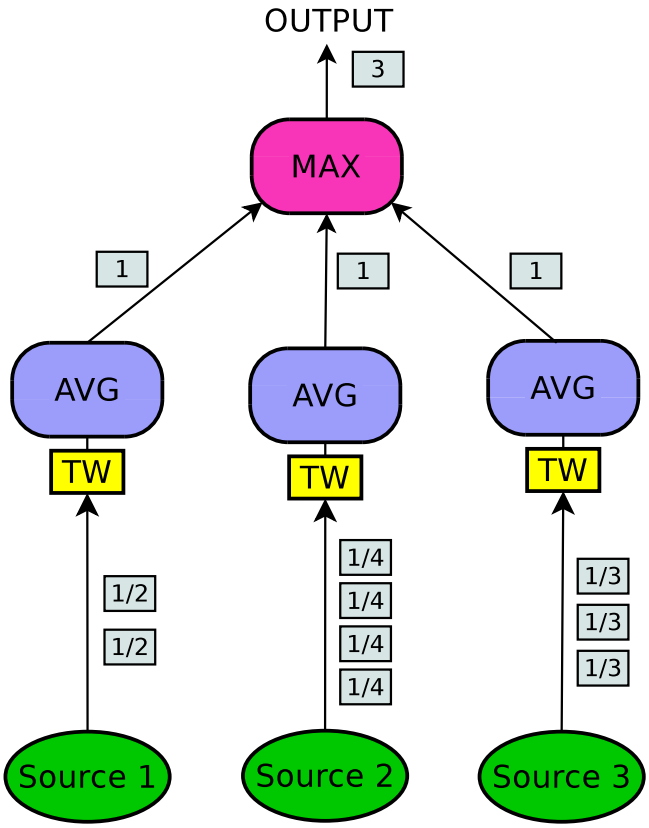

Figure 3.7: An example of fan-in query, input values for each source are first averaged over a certain time window, then the maximum is selected. Source Information Content values are shown for each tuple.

In Figure 3.7 the final SIC value obtained by the depicted query is 3, which is equal to the number of input sources. This follows from Assumption 8 (Total Information Content). To simplify the exposition, this value is absolute, not scaled to the $[0, 1]$ interval for comparison. It is also a perfect value, meaning that there was no loss of information during its creation. In case of failure its value would be reduced by the amount of information contained in the lost tuples.

According to Assumption 4 (Source Equality) all sources equally contribute to the creation of the final result. Assumption 5 (Tuple Equality) then states that all tuples from a source in a *source information*

the Source Tuple Information Set has been used for the computation. A small value indicated a large information loss, while a value close to 1 indicates an almost perfect result.

Model Benefits

Query Performance. Even though SIC values give a good estimate about the quality of the processing, it should be noted that this value is not directly linked to the absolute error of the query results. In general it is not possible to compute error bars on the delivered results based on this metric. SIC values are an indication to the user about the amount of failure that occurred during the generation of a query result. It is then up to the user to decide if the delivered answer should be considered valid of should be instead discarded.

Nevertheless SIC values are a good hint about the quality of the results delivered by a query. The knowledge of the query semantics might allow the user to correlate SIC values and actual absolute error of the results. The simpler the query semantic, the easier it is to establish such a correlation.

Completeness of Results. Using SIC values it is possible to compare the results of the same query at different times to evaluate the quality of the delivered results in terms of information completeness. A result tuple t_1^R delivered a time t_1, compared to another result tuple t_2^R delivered at time t_2, contains more information iff $t_1^{SIC} > t_2^{SIC}$.

A user can monitor the status of a computation by observing the changes in SIC values delivered by a query at different times. A sharp decrease in SIC values means that a lot more information is been discarded and there might be the need to act upon it. Reasons for a sudden reduction could be:
1) the failure of one or more processing nodes, requiring the migration of some operators and maintenance of the failed infrastructure,
2) a sharp increase in input rates, requiring the increase of processing resources to maintain the delivered SIC value with acceptable boundaries.

Fair resource allocation. Using SIC values it is possible to compare the performance achieved by different queries running concurrently on a shared processing platform. A fair system would try to equalise the quality-of-service of running queries, allocating resources in such a way so that all queries delivered results with similar SIC values.

When the system experiences *overload* and needs to discard a certain amount of tuples, it can use SIC values to determine which tuples to drop and which tuples to spare aiming at the equalisation of result SIC values among all queries. This process takes the name of *fair shedding* and will be further

by the source during the interval. Thus, the SIC value of an individual source tuple t^s is inversely proportional to the number of tuples in $|\mathcal{T}_s^S|$. Furthermore it is necessary to normalise to the number of sources $|\mathbb{S}|$ connected to a query by dividing the initial SIC values to the number of sources. In this way all final SIC values lie in the interval $[0, 1]$, making possible to compare the performance of different queries.

SIC propagation from source to result tuples

So far, we considered queries as black-boxes and discussed only the relation of SIC values between source and result tuples. In order for the formal description to be complete, it is important to understand how intermediate SIC values are calculated at the individual operators and how these propagate within the system.

Source tuples from $\mathcal{T}^S$ are processed by the query operators in set $\mathcal{O}$, these produce new derived tuples that flow to the next downstream operator, until they reach a terminal operator that outputs result tuples. More formally, for a given intermediate derived tuple t_o^R and for any operator $o \in \mathcal{O}$, we define $\mathcal{T}_{in}^o$ to be the set of all input tuples to the operator o required for the generation of t_o^R. Similarly, $\mathcal{T}_{out}^o$ denotes the set of derived tuples produced by operator o after the processing on $\mathcal{T}_{in}^o$.

The SIC value of a derived tuple t, processed by operator o, i.e. $t \in \mathcal{T}_{out}^o$, is:

$$t_{SIC} = \frac{\sum_{t \in \mathcal{T}_{in}^o} t_{SIC}}{|\mathcal{T}_{out}^o|}. \tag{3.3}$$

This means that the set of derived tuple produced by an operator contains all the information contained by the set of input tuples that the operator processed. This information is considered to be equally distributed over all the output tuples and so the total value is divided by the number of output tuples.

The way SIC values are calculated is somehow recursive, as it is the same if we considered a single operator or a complete query seen as a complex black box operator. The sum of the information in input is always propagated to the output tuples and equally divided among them.

SIC relation to query performance

The SIC values of the result tuple lie in the $[0, 1]$ interval :

(a) A value equal to 1 means that the complete set of $\mathcal{T}^S$ tuples is used and the result is perfect.

(b) A 0 value means that all source tuples from $\mathcal{T}^S$ have been discarded or lost during the processing.

(c) A value in $(0, 1)$ indicates a degraded result, meaning that only a subset of the tuples contained in

lost during its creation, meaning that all tuples in its source information tuple set or their derivatives are correctly processed. If one of the tuples from the source information tuple set or a derivative is lost, either because of failure or deliberate shedding, the information contained in the result tuple is not perfect and its SIC value is decreased accordingly.

Definition 3.8 (Source Information Content) *The sum of the source information content (SIC) of all result tuple for query q, denoted as t_R^{SIC}, measures the contribution of source tuples, belonging to its Source Information Tuple Set denoted as $\mathcal{T}^S$, processed for its generation. Its values are calculated by:*

$$\sum_{t_R \in \mathbb{T}_q^R} t_R^{SIC} = \sum_{t_{src} \in \mathcal{T}^S} t_{src}^{SIC}, \tag{3.1}$$

where t_{src} denotes a source tuple in $\mathcal{T}^S$ used for the calculation of the result tuple. In particular, t_{src}^{SIC} shows the contribution of a source tuple coming from source $src \in S$ to the perfect result and is quantified by:

$$t_{src}^{SIC} = \frac{1}{|\mathcal{T}_{src}^{\mathcal{S}}||\mathbb{S}|} \tag{3.2}$$

where $|\mathcal{T}_{src}^{\mathcal{S}}|$ is the total number of tuples in the Source Information Tuple Set of source $\mathcal{S}$, and $|\mathbb{S}|$ is the total number of sources for the query.

Equation (3.1) states that the total SIC value of all result tuples for a certain query, is equal to the sum of the SIC values of all tuples in its Source Information Tuple Set. It aggregates all the SIC values of all source tuples that contributed to its creation. Since a query can have multiple terminal operators, each potentially producing more than one result tuple at the time, we consider the *sum* of all the result tuples produced by all terminal operators. If all tuples where correctly processed, the resulting SIC value is 1, a lower value indicates that some information loss happen during the query execution.

Equation (3.2) describe how source tuples are assigned their individual SIC value. First, a value of 1 is assigned to the totality of the tuples produced by a source in its Source Tuple Information Set. This means that all sources are considered to equally contribute to the final result, regardless of the amount of tuples they produce during the same interval. Since every tuple in this set is considered to contain the same amount of information, this value is then divided by the number of tuples produced

already compromised tuples before other which where produced with perfect information.

When dealing with a system that allows the the concurrent execution of multiple queries, SIC values help discarding tuples so that all queries in the system are affected in the same way. This process takes the name of *fair shedding* and will be further analysed in Section 5.3.

Data Model Formalisation

The purpose of the SIC metric is to capture the amount of information that goes into the creation of an output tuple. In absence of failure the *perfect value* of the SIC metric is given by the number of sources, or it is 1 when scaled to be comparable across queries. A reduction from this value is determined by the occurrence of failure during the processing.

Since it is not possible to know in advance which tuples will contribute to the creation of an output batch of tuples, the final SIC value is not calculated on the single tuples, but instead over a window of tuples. The amount of information captured is calculated over a time interval, so that the final resulting SIC value delivered is then the sum of the individual SIC values of all tuples produced within the corresponding time window. Every tuples within the interval is assigned the same value, as for Assumption 5 (Tuple Equality), and all sources assign the same total amount of SIC value during the time window, as for Assumption 6 (Source Equality). Therefore a source producing 100 tuples within the time interval assigns a value of 1/100 to each one of them. Another source producing only 20 tuple during the same interval assigns individual values of 1/20, and the result tuples obtained after processing would have an aggregated SIC value of 2 in absence of failure. The totality of the tuples produced by the sources within a time interval takes the name of Source Information Tuple Set.

Definition 3.7 (Source Information Tuple Set) *The source information tuple set $\mathcal{T}^S$ of a result tuple t^R is the set of source tuples given by function $f^{-1} : \mathbb{T}^R \rightarrow \mathbb{T}^S$ when applied over t^R, i.e. $f^{-1}(t^R) = \mathcal{T}^S$. $\mathbb{T}^S = \{\mathcal{T}^S_s | s \in \mathcal{S}\}$, where $\mathcal{T}^S_s$ denotes the set of source tuples in $\mathcal{T}^S$ generated from source s.*

In our query model every result tuple t^R is associated to the set of source tuples $\mathcal{T}^S$ that contributed to its generation. We consider a query as a black-box modeled after a one-to-one *query function* f that maps source to result tuples and ignores any derived tuples generated by the operators. The *source information tuple set* of a result tuple, is the set of all source tuples that were processed for its creation.

This concept is central to the definition of our quality centric metric called *Source Information Content* (SIC). The main idea behind it is that a result tuple is considered to be perfect when no information is

scope of this project, derived following the reasoning of all the model assumptions made in Section 3.2. In Section 3.5 there will be a further analysis of these sample queries, which will be used as running examples of real-world applications of the SIC metric.

3.4 Source Information Content

This section provides the definition of a quality-metric called *Source Information Content (SIC)*, which has been developed in the scope of this project. The complete set of formulas to be for its calculation is derived, following the reasoning expressed in the assumptions presented in Section 3.2. A discussion about what benefits this novel quality metric can bring when employed by a stream processing system closes this section.

Introduction

Typically a stream processing system does not perform any self-inspection to determine if all the available data is being correctly processed, assuming failure as a transient, rare condition. We know this is not a realistic assumption and that failure should be monitored and accounted for. In our model, we propose to augment streams with a new metric called *Source Information Content* (SIC) that gives a hint about the amount of information contained in a tuple and thus to its importance for the current query results. This is added to streams in the form metadata, whose value is calculated by the query operators and is inversely proportional to the occurrence of failure.

A tuple should convey information about the amount of failure experienced during its creation. There should be a way to indicate if the data carried by a tuple was created using all the available information, and is then 100% accurate, or if some information was lost in the process, thus reducing the dependability of that data. *Source Information Content* tries to do achieve this goal, by measuring the amount of lost information in the creation of a tuple.

The system can use this value to evaluate the importance of a tuple towards the creation of the final query result. When the system is overloaded for instance, it has to decide which tuples should be dropped in order to recover. SIC values in this case can be used to assess the individual value of tuples and guide the system to a selection that helps improving the quality of the results.

In a single query scenario the system is able to reason about the amount of information carried by tuples, by dropping the ones with the lowest values it minimises the impact on the query results. Even in the absence of failure, it is possible that some tuples aggregate more information than others and should be treated with more care. In case of failure then, the system would prefer to drop some

analysing a stream of data, a user might be interested in obtaining several different results. Instead of submitting N queries though, it can submit a single fan-out query with multiple ending points. This is conceptually simpler than having to design and submit several almost identical queries and directly exploits the processing redundancy by reusing a number of streams and operators.

EXAMPLE: Figure 3.9 shows a query calculating the occurrence of positive and negative mentions of a certain keyword and also calculates their ratio. The original stream comes from the Twitter firehose and contains an unsorted stream of messages. First, a filter eliminates all the tweets not containing the keyword of interest. Then its output stream is multiplexed over two different natural language processing (NLP) operators that only forward tweets if they contain either a positive mention or a negative mention of the keyword. The resulting streams are each sent to a counter operator, thus producing two output streams counting the number of positive and negative references to the keyword. The resulting streams from the counter operators are also sent to a final operator which calculates the ratio between them, thus providing an indication about the general feeling about the keyword. In this query a single input source is processed to produce 3 different results.

The next section will describe the *Source Information Content* (SIC) quality metric developed in the

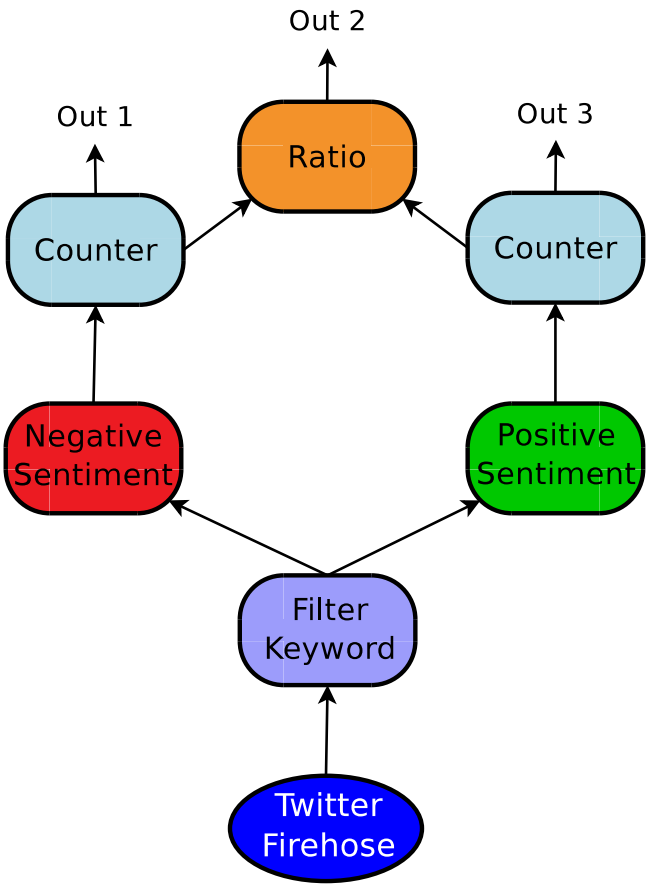

Figure 3.6: An example of fan-out query processing Twitter data. It counts the occurrence of positive and negative mentions of a certain keyword, calculating also their ratio.

processing node, their computation is split among a certain number of copies, running at different sites. These copies are all composed by the same set of operators, but process only a subset of the original data. Splitting the computation allows the system to spread the processing cost of a set of operators over several nodes and thus overcome the overload condition at the original hosting node.

EXAMPLE: Figure 3.9 shows a query processing Twitter data in real-time. A stream of Twitter messages can be analysed to calculate the "rudest" status updates. Every message contains a lot of information in addition to the message text itself, like the location where it was sent from and some information about the author.

A single stream of Twitter messages is split and scattered over 3 different processing nodes, each hosting a Natural Language Processing (NLP) operator that calculates some coefficient for each message, for instance its "rudeness". The output of these NLP operators is finally collected by a single Top-10 operator, preceded by a 1 minute time-window. The query outputs then the 10 "rudest" Twitter messages posted every minute.

Multiple results queries. These queries produce several outputs from the same set of input data. These can be seen as multiple single output queries, with a partial overlap in their computation. When

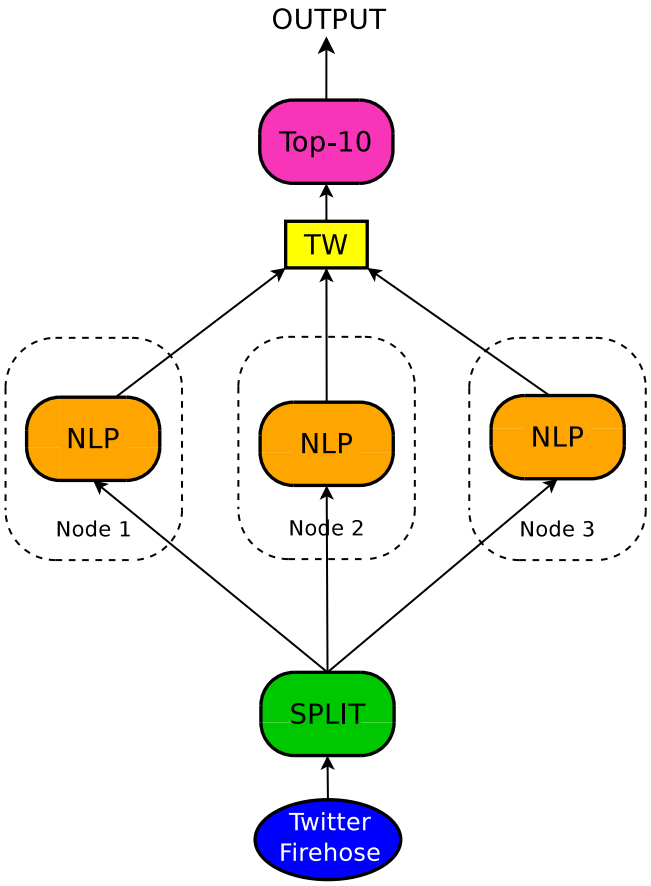

Figure 3.5: An example of fan-out query processing Twitter data implementing the map-reduce paradigm.

series of processing steps and a single point of output.

EXAMPLE: Consider the query depicted in Figure 3.4, calculating the max average value produced by the sources. Three sources produce input tuples at different rates. All the tuples from each source are first collected by a time window and then averaged. Finally the maximum value among the one produced by the average operator is selected and output. This is a typical fan-in query where input from many sources is collected and processed to produce a single output.

Fan-out queries

The second class of queries that we consider takes the name of *fan-out*. This is the family of queries where one or more operators send their output to *more than one* downstream operators. Differently than *fan-in* queries, where the graph resembles a tree, the *fan-out* query graph is free to take any configuration, as long as as it does not contain a cycle. Fan-out queries can be divided into 2 possibly overlapping main classes: *a)* queries with one result but split computation, and *b)* queries calculating more than one result. I am going to present an example for each case.

Split computation queries. The semantic of these queries is similar to the map-reduce processing paradigm. Since one or more operators are computationally too heavy to be hosted on a single

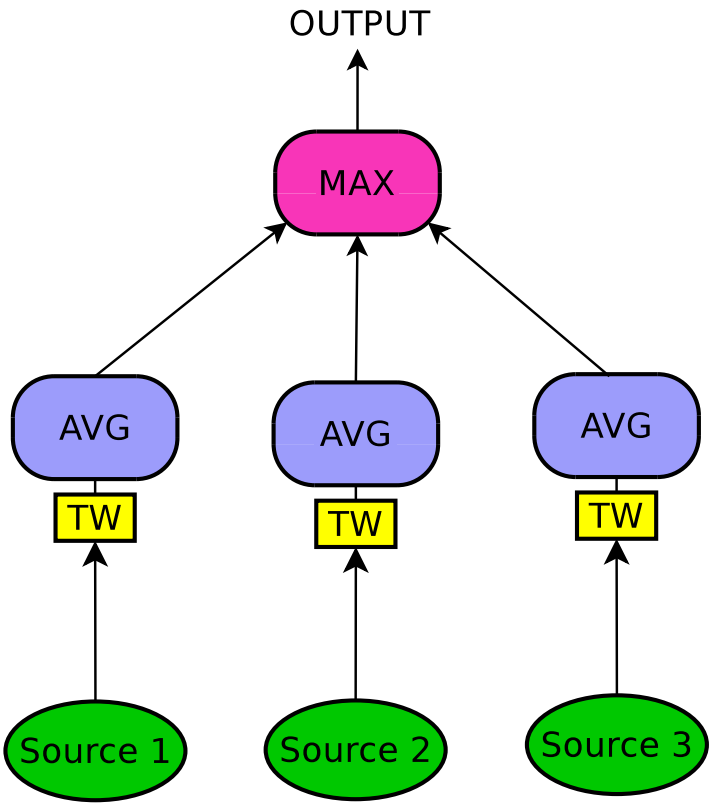

Figure 3.4: An example of fan-in query, input values for each source are first averaged over a certain time window, then the maximum is selected.

query submitted by the second user as single computation yielding 9 different results, it would evenly shed tuple among the two queries, leading to a fair allocation of the available computing resources. Fair shedding will be explained in more detail in Section 5.3.

The second reason for distributing the output of an operator is when the downstream computation is too intensive to be executed on a single machine. In this case the output is split among several processing node, each hosting the same set of operators, processing a portion of the output tuples. The computed results are then collected to produce a single result. This is the stream equivalent of the *map-reduce* paradigm.

When computing the values of the quality metric, the system needs to take this into account. In particular, an operator distributes the total value of information in input among all its output tuples. A tuple is assigned a value that is inversely proportional to the number of the output streams of the present operator. In the first case, when multiple results are calculated, each terminal operator will produce tuples with a value equal to the number of sources, divided by the number of results. In this way the cumulative information content value for the query is the same as if there is only one terminal operator. In the case of a map-reduce partitioning of the query, the individual information content value of tuples is lowered during branching, but is restored during the reduce phase. The details about this calculations are covered in Section 3.4.

3.3 Types of Queries

This section provides an overview on the different query types. The categorisation of queries is based on the shape of their graph. When every operator have at most one successor, the query graph resembles a tree, having input sources as leaves and one output stream as root. These queries belong to the *fan-in* type, as the number of edges in the graph decreases moving from sources to output. A query containing operators with more than one successor instead belongs to the *fan-out* type. In this case the query graph can be more complex, with the only restriction of being acyclic. These queries can have multiple input sources as well as multiple resulting streams. The following sections will provide more details on these types of queries.

Fan-in queries

The first class of queries we are going to analyse takes the name of *fan-in*. In these queries operators have *at most* one successor and the input data from one or more sources is elaborated to produce a *single result*. Typically in this category we find queries computing a single aggregation, usually dealing with sensor data. These queries present a graph that resembles a tree, with many input sources, a

final value of a result batch is equal to the number of sources.

Let us consider a simple query calculating an average pollution value every minute from a set of 10 sensors located at different locations in a city. This query produces a result batch of a single tuple, which aggregates all the information gathered by the 10 sensors over a one minute window. Let us assume that no failure happens and that all tuples are correctly processed. Since all sources are considered as equals, the total amount of information carried by the tuples produced by each source over a minute is 1. When the final result is computed then, the total information it carries is equal to 10, the number of sources.

When dealing with a system supporting the concurrent execution of multiple queries it becomes important to be able to compare the information values of the different queries. This is needed for instance in an overload condition, when trying to achieve *fair shedding*. This is the process of selectively drop tuples to reduce the load of the system, so that the final quality values of all queries is equalised. In this scope the system must be able to compare the different information contents of tuples. A simple way to achieve this is to divide the total information value of a query result to its number of sources. Doing so all values are scaled down in the interval $[0, 1]$ and become comparable.

9. QUERIES ARE DAGs

An operator can have more than one downstream operator, metric calculation should take this into account.

A query can be represented as a Directed Acyclic Graph (DAG), where nodes represent operators and arrows the streams flowing through them. This means that, as long as there are no cycles, every operator is free to have multiple downstream operators which receive its output as input. There are two main reasons for distributing the output of an operators, 1) the query can compute multiple results, or 2) the downstream computation is split over multiple nodes because it would be too computationally intensive on a single one.

The first case deals with queries computing multiple results. These are called *fan-out* queries and will be further analysed in Section 3.3. They compute multiple results based on some common input information, but are logically seen as a single query. The reason for treating a query with multiple results as a single query is to allow the system to be fair when making shedding decisions under overload. Let us consider a system with 2 running queries, submitted by 2 users. The first user submitted a query with a single terminal operator, while the query submitted by the other user have 9. If the system considered each result computed as a single query, the second user would be allocated 90% of the available resources, leaving only 10% to the first. If the system instead considered the

of tuples and drop the ones with the lowest values first. This is done to reduce the amount of missing information in the final query result.

Let us consider a query whose graph representation resembles an unbalanced tree as in Figure 3.3. In this query there are 3 sources, all producing tuples at the same rate. The input tuples produced by source 1 and 2 are first combined together by operator A, then the resulting tuples are processed by operator B together with the tuple produces by source number 3. We can assumed that operator B resides on a different machine than operator A and that, because of overlaoad, it needs to discard one tuple out of the four currently present in its input buffers. When comparing the information values of the single tuples it finds that the tuples received on the left input have an information content which is double compared to the tuples received on its right input. This is because the tuples on the left contain the information produced by 2 sources, while the others are carrying the information of a single source. Since the goal of the system is to deliver results containing the maximum amount of source information, it decides that the one tuple to drop is one of the two received on its right input.

8. TOTAL INFORMATION CONTENT

The total amount of information contained in a result in absence of failure is equal to the number of sources, 1 if scaled to be comparable with other queries.

In absence of failure, every result batch produced by a query contains an amount of information which is equal to the sum of the individual information values carried by the source tuples that concurred to its creation. Considering every source as equal, we can assign a value of 1 to the complete set of source tuples for each source that contributed to the calculation of a result batch. If we do so, the

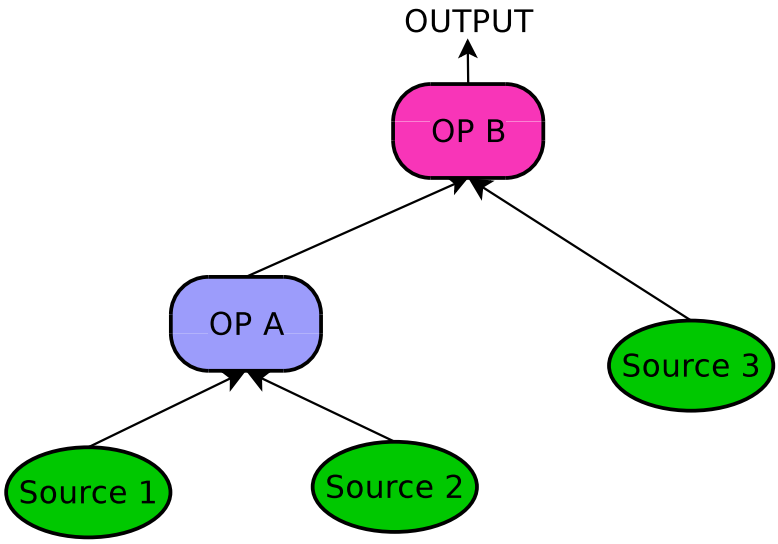

Figure 3.3: An unbalanced query tree, showing the different information contents of the tuples flowing through it.

Consider then a *filter* operator, which discards a certain number of tuple not satisfying a set of requirements. As an example, we can imagine an operator filtering all temperature readings with a value below 30 degrees Celsius. In input it receives 10 tuples, out of which only 5 carry a reading above 30 degrees. The number of output tuples in this case is only 50% of the one in input, yet the total information carried by these is unchanged. What changes is the individual information value associated with the single tuples, which in this case in doubled compared to when they entered the operator.

In all the previous examples the number of tuples produces by an operator is smaller or equal to the number in input. This is not true for all operators though. If we consider the *join* operator for instance, the number of tuples in output can also be greater than the input one. Let us consider a join operator that has on one input containing temperature readings for a certain room and on the other humidity values. The tuples on the first input have the schema ⟨RoomID, Tmp⟩, while for the second the schema is ⟨RoomID, Humidity⟩. The operator joins the two streams on the field RoomID, producing a stream of output tuple having schema ⟨RoomID, Tmp, Humidity⟩. This amount of tuples produced by this operator is in the range $[0, NxM]$, where N is the number of temperature tuples and M the number of humidity ones. This means that is possible for such an operator to produce more tuples in output than the number it has in input. In the perspective of our model the information is still conserved, even though the individual quality values of tuples might decrease.

[**Note**] I included a special case below, I think we never discussed it, see if you agree with this.

Finally we should consider the case when an operator produces no output at all. This can often happen with filtering operators, when all the input tuples do not satisfy the filtering criteria and therefore are dropped. Since there is no output tuple to carry the input information content, this situation would be indistinguishable from the case where all the input tuple are lost, for instance because the system had decided to shed them in order to overcome an overload condition. In this case the system still needs to produce an *empty batch*, containing no tuples but with a total quality metadata value equal to the sum of the values received in input. In this way the total amount of information is preserved and the final calculation is not affected.

7. INFORMATION IS VALUE

The importance of a tuple is directly proportional to the amount of information needed to generate it.

When we consider the importance that a tuple has within a query it is important to take into account the amount of information that went into its creation. The more information is conveyed within a tuple, the more important it becomes for the computation of the query result. If the system is in need of making a decision about what tuples to drop, it has to consider the individual information content

result. Since the system employs an operator independent model it cannot know which tuples to drop in order to reduce the error in the results, so it treats all source tuples as equals and makes a random decision.

This is a simplification to keep the model abstract enough to be usable in a general purpose system. Of course there are situation, like in detection queries, where some source tuples are definitely more important than others. Having the knowledge of the query semantic could enable the system to be more selective and make distinctions among tuples when making shedding decisions, but it would bind it to a specific set of operators and queries.

A tuple acquires more value when it is obtained from the processing of many other tuples. The tuple equality principle only applies to source tuples, not derived. These are obtained as output from operators and their value is proportional to the amount of information (i.e. number of source tuples) they aggregate. The difference in terms of information content among derived tuples is exploited to implement intelligent overload management strategies, as described in Chapter 5.

6. CONSERVATION OF INFORMATION

The amount of information going into an operator is equal to the amount in output, regardless of the number of tuples produced.

An operator processes one or more input batches and produces a single output batch. Every tuple that enters the operator has a certain value for the quality metadata, which is the same for all tuples in a batch and might be different for different batches. The operator transform these tuples into a new set according to its processing semantic. Even though the amount of tuples in output might be different than the amount in input, the *amount of information* does not change. In our model we assume that the total value for the quality metric in input to an operator is not altered by the processing and is transmitted to the output tuples.

Let us consider a simple *map* operator, which transforms a Fahrenheit temperature reading into its Celsius equivalent. This operator receives N tuples in input, applies a simple formula and produces N tuples in output. The information carried by the input tuples is transformed, but its total amount does not change.

Another class of operators deals with *aggregation*, we can use a simple average as an representative. This operators receive N tuples in input and produce a single output tuple. Consider an operator calculating the average temperature in a room every minute. It receives all the input tuples produced by the sensors during the specified time window an produces a single average value. Even though the output tuple produced is only one, it carries all the information of the ones in input, only transformed into a single aggregate value.

distance, as it requires a higher transmission power.

Consider a sensor network monitoring moist levels on a rural area, composed of sensors randomly scattered over a certain surface. The processing query produces an average moist value for the area every hour, aggregating readings from all sensors which are produced at a rate of one every 10 minutes. Due to the uneven distribution of sensors some require more power to transmit their readings and in order to save battery decide to reduce the transmission rate to half the original. When aggregating the moist values then, the amount of tuples for each sensor would be different. The processing query would first group the readings by sensor id, average a single result for each sensor over the specified time window, and finally compute a global average. We could see this as an average over a single reading from every sensor, each conveying information about a certain location, but with a different resolution. Nevertheless the information produced by all units has the same value for the final calculation of the global value.

In the previous example all tuples generated by a sensor in a one hour time-window convey the same information, regardless of the fact that some sources produced more tuples and others less. All sources equally contributed to the production of the final result.

In our model we decided to treat all sources as equal, but this is not true in all cases. There might be scenarios in which a particular source should be considered more important than others, for instance because it is placed in a strategic location or because it is equipped with a more sophisticated instrumentation. A possible extension to our model that would account for this differences is to include a *weight* parameter. This would allow the user to indicate to the system which tuples should be regarded are more valuable. The system then would assign a different importance to these tuples and try to avoid dropping them, with a probability proportional to their weight.

5. TUPLES EQUALITY

All tuples from a source are equally important for the generation of the final result.

All tuples produced by a source that concur to the creation of the same result are considered to contain the same amount of information. Since the model abstracts from the semantic of the query and is designed to be used for all queries, it treats all source tuples as equals.

Let us consider a simple query, composed by one source and one average operator producing a single result every minute. The source produces 1000 tuples per second, the system is overloaded and is not able to process all of them. If the distribution of values carried by the source tuples is uniform it does not make any difference which tuples are dropped. If instead the distribution is highly skewed, presenting a small number of outliers, the dropping of one of them could sensibly change the aggregated

close to 0.5. In this scenario the loss of half of the input values is almost unnoticeable from a practical point of view.

A filter operator with the same input set, eliminating from the input stream all tuples with values below 0.5, would on average produce an output of 50 tuples, since the input values are uniformly distributed in the interval. When shedding half of the input values though, the operator result changes considerably. On average it would now produce 25 output tuples, which is half of the number of tuples it would produce with the complete set of input tuples. With this simple example we can observe how the processing semantic of the operator can be very different from one another and how the impact of input loss can be almost unnoticeable for some operators while can be disruptive for others.

Therefore we decided that the quality metric employed by our system should be operator independent and valid for any kind of processing semantics, efficiently capturing the processing degradation under failure. Even though the knowledge of the internal functioning of operators would allow the system to have a more precise reasoning about the impact of failure on the quality of the results, this is impractical for a general purpose system. Instead of trying to quantify the approximation of the result in terms of precision (i.e. providing error bars) we try to quantify the amount of information that was lost during the computation of a result compared to the total amount that would have been processed in absence of failure.

Other systems adopted this operator independent approach. The *network imprecision* metric accounts for the state of all participating nodes in large-scale monitoring systems [Jai+08a]. It estimates the percentage of nodes available when calculating an aggregate value over a set of data sources. In contrast, our metric operates at the granularity of individual tuples, not sources, to reason about the impact of information loss on the results. Another example is the *harvest* quality metric proposed independently by Murty and Welsh [MW06] and Fox and Brewer [FB99]. In the former case, the authors argue that harvest should capture the fraction of data sources used in Internet-scale sensor systems. In the latter case, the metric captures the fraction of data included in the response of Internet applications.

4. SOURCE EQUALITY

All sources are equally important for the generation of the final result.

In our model sources are all considered equally important, regardless of their tuple production rate. If we consider a sensor network deployment it is common to observe different rates of input readings. This is due to different settings or different battery constraint of the sensors. A unit with a depleted battery would usually reduce the rate at which it propagates information to save energy. Another reason could be the position of sensor, the energy cost of propagating a reading in fact grows with

large set of sensors the lack of input from a certain number of them does not usually mean that the computation should be considered void as a whole. Instead the results simply become less accurate due to the missing input data. If we consider queries aggregating readings of sensors scattered over a certain area, like an average of pollutants in a city area, it is possible that a failing sensor is located close to others that will record a similar reading. Thus the missing information might not produce a great variation in the final result, that is approximate but still meaningful.

If we consider queries trying to detect a certain event instead, the failing of a single sensor can become crucial. The failing unit could be the one that would have received the reading of interest, and because of the failure this reading would never be generated. In this scenario failure reduces the user confidence in the results and may not tolerable. Even if an approximation of the results may not be acceptable, it is important to notify the user that some failure occurred during the collection of the input data, leaving the final decision about the confidence in the results to the user.

An analogous reasoning can be applied to the social media analysis scenario. In this case the enormous amount of available input data can determine an overload condition of the processing infrastructure. Differently than the sensor data example, where the failure usually occurs at the input level, here the failure would more probably happen at the processing stage. Queries dealing with aggregation usually process copious amounts of data and, in case of the loss of a small percentage of it, they can still produce results close to the ones that would be obtained in absence of failure. In general when dealing with aggregations, the larger is the set of input data the more tolerable becomes failure.

3. A GENERIC METRIC

A quality metric should be operator independent and abstract from the query semantics.

Stream processing systems are usually very versatile and can compute a large variety of queries. Each query is composed by operators whose semantics can be very different. If we consider the streaming equivalent of some traditional operators present in a relational database, like *filter* and *average*, it is notable how the the dropping of a tuple from the input of these operators can have dramatic differences in terms of the produced output. Due to their different processing semantics, missing one tuple can have almost no influence or can completely subvert the output of an operator. Furthermore it is common for stream processing systems to be extensible, allowing users to implement their own custom operators, whose processing semantics are unknown.

Let us consider the different impact that the loss of a tuple can have on the result of different operators. An average operator, having in input a set of 100 tuples carrying integer values uniformly distributed in the range $[0, 1]$, produces a single output tuple with a value close to 0.5. If we imagine to drop 50% of the input because of overload, the operator would still produce a single tuple with again a value

In a paper released by Google [PWB07] it is shown how the occurrence of failure in a large hard drive population is much higher than what declared by vendors. They analysed a large population of disks and showed how the Annualized Failure Rate (AFR) ranged from 1.7% for first year drives to over 8.6% for three-year old drives. Jeff Dean of Google also presented some statistics [Dea09] about the real world occurrence of failure in their data centres. In a typical rack of 1800 machines it is expected that more than 1000 machines will fail in the first year alone, and there is a 50% chance that a cluster will overheat, taking down most of the servers in less than 5 minutes and taking 1 to 2 days to recover.

Overload can be considered as a kind of failure, because the system is not able to process all the incoming data and needs to discard some of it. In this case an approximate result is produced even though all the processing units are functioning correctly. Input data rates can be highly unpredictable, with variations that can often be of orders of magnitude [Tat+03]. This leads to an objective difficulty in the provisioning of a system. If we considered a cloud deployment scenario, where resources are rented, overload can be not only tolerable but also desirable. It might be cheaper to underprovision the system on purpose, in order to keep the costs down, when the approximation of results is not an issue. In case of a federated resource pool, in which several parties share their local cluster to gain access to the complete processing infrastructure, the possibilities for overload are even higher. Each local site, in fact, is under the control of a different authority and so is its maintenance, making recovery times unpredictable. Due to its nature of shared platform, the processing resources available at each clusters are not uniform, so the query fragments distributed across several processing sites will experience a skewed availability of resources. The same query fragment can run without failure at one site, while experiencing heavy overload at another.

2. APPROXIMATE PROCESSING

Users can accept approximate processing, but need to have a way of evaluating the quality of the computed results.

Failure should be considered a normal condition of operation for a large enough data stream processing system. We propose to augment data streams with a metric capturing the amount of failure occurred during the processing so that this failure can be quantified instead of hidden. In many applications an approximate result is acceptable for users, but the it is important that when failure occurs its amount is calculated and reported. It is then up to the user to decide if the quality of the delivered results is good enough for these to be accepted or should be discarded instead.

In sensor networks it is common to have a lot of failure at the source level. Sensors can suddenly stop working because of hardware failure or can become temporarily unreachable. In this cases a query processing their readings delivers results that are incomplete. Nevertheless the quality of the query results can be good enough to be meaningful for the final user. Especially when dealing with a

A query is a graph where vertices correspond to operators and arcs indicate the direction of tuples flowing from one or more sources to one or more terminal operators. The set of query operators is given by $\mathcal{O}$ and cumulatively computes the query function f. Operators might be distributed over a set of nodes N, when the query function is too complex to be computed on a single machine. The discussion about our implementation of queries is given in Section 4.2.

3.2 Model Assumptions

This section states the assumptions that led to the definition of a quality metric called Source Information Content (SIC), which is presented formally in Section 3.4. The reasoning started with the consideration that failure is a common condition and should not be ignored but accounted for. Instead of trying to eradicate it, the system should accept it, monitor and report its impact. Since queries can be composed of an arbitrary set of operators, often employing very diverse processing semantic, a quality metric must be generic enough to be usable across the whole spectrum of possible queries and not tied to a specific domain. When considering sources, it is possible that some produce inputs that are more valuable than others, but it is often difficult to decide beforehand which ones they will be. This almost impossibility of determining which sources and which tuples are more valuable than others led to the idea of considering all of them as equals. When an operator receives some tuples in input, it transforms them but does not change the amount of information they carry, thus leading to the consideration that there is a principle of conservation of information that the quality metric should follow. Another consideration is that the amount of information embedded in a tuple is proportional to the amount of tuples needed for its generation. Since we assumed that every source produces the same amount of information per interval time as the others, the total amount of information processed by a query in absence of failure is then a multiple of the number of sources. Finally we considered what shapes a query graph can take, which in its most generic form is a directed acyclic graph, the computation of the metric then should also be flexible enough to usable in all these cases. The rest of the section describes these ideas in more details, laying out the reasoning behind these considerations that lead the design choices behind the Source Information Content (SIC) quality metric.

1. FAILURE IS EVERYWHERE

Failure is always present in large-scale query processing, overload is a kind of failure.

In large-scale distributed systems failure is not to be considered as a rare and transient condition. In fact evidence indicates that in such a system a certain percentage of nodes will be failing at all times. Even though the Mean Time Between Failure (MTBF) for a single component might be quite high, once the number of components grows the incidence of failure becomes more and more relevant.

can divide operators based on their behaviour when dealing with input streams into two categories: *blocking* and *non-blocking*.

Blocking operators need to have at least one input batch ready on each of their input streams. For instance, if there are two inputs to an operator, one containing two input batches while the other one is currently empty, the operator blocks until an input batches arrives on the second input stream. When this happens the first batch of each input is removed and processed, producing one derived batch. Assuming no other batch has arrived on the second input, the operator then blocks again, as only the first input contains data to be processed.

Non-Blocking operators do not need input on all channel to be triggered. Instead they produce a derived batch as soon as some input batch is present on one of their input channels. This means that such an operator will never block waiting for input, and will never have pending input data. The discussion about our implementation of queries is given in Section 4.2.

Query

The processing needed to transform the information contained into some input tuples into a useful output for the users is usually not performed by a single operator, but by a group of them arranged into a logical processing graph that takes the name of *query*.

Definition 3.6 (Query) *A query logically defines a series of processing steps over set of input streams* $S_{in} = \{S_{in}^1, \ldots, S_{in}^n\}$ *to produce a desired set of output streams* $S_{out} = \{S_{out}^1, \ldots, S_{out}^n\}$ *by the mean of a finite number of operators.*

A query describes the processing to transform a set of input streams into a set of output streams. In the boxes-and-arrows model, queries are depicted as a directed acyclic graph (DAG) where arcs represent streams and vertices represent operators. One or more sources produce streams of tuples in various time-variant rates, which are fed in input to the system. Then one or more operators process these tuples, either in sequence or in parallel.

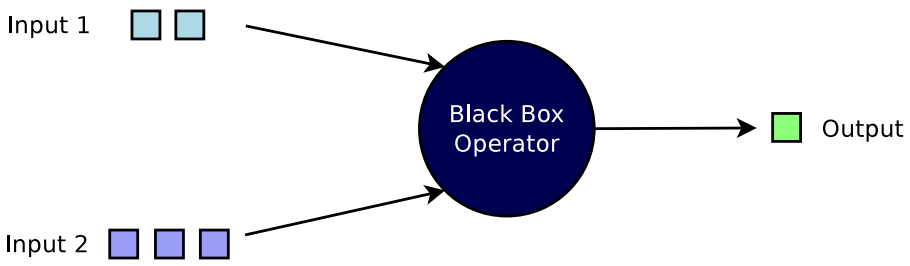

Figure 3.2: A black-box operator with two inputs and one output.

A stream is only used logically to describe a possibly unbounded bag of similar tuples flowing between two operators. As for CQL the system needs a finite amount of tuples to operate, which in our system is represented by a *batch*.

Batch

Streams are a continuous abstract entities, while operators need a finite set of tuples to operate. The system thus logically partitions streams in *batches*: finite snapshots of a stream, containing tuples that have the same *quality metric* value, to be used as input and output units for operators.

Definition 3.4 (Batch) *A batch is a finite set of tuples $B = \{t_1, \ldots, t_n\}$, all having the same quality metric metadata value.*

A *batch* is a logical group of tuples all having the same quality metric value. In our model an operator does not work on a single tuple but on batches, it processes one or more input batches and produces one output batch, which might be composed of a single tuple. Using batches allows for a more compact representation of the quality metric metadata, as it does not have to be included with each tuple. It also speeds up the calculation of the new quality metric value when an operator outputs a new set of tuples.

Batches represent a snapshot of a stream, a finite amount of tuples that can be processed by an operator. In our system they represent the equivalent of *relations* used by CQL. The discussion about our implementation of batches is given in Section 4.2.

Operator

A stream processing system transforms a set of input tuples into a set of output tuples representing the answer to a given query. This data transformation used to produce a result is carried out by a number of *operators*.

Definition 3.5 (Operator) *An operator is a function f_{op} over a set of input streams $S_{in} = \{S_1, \ldots, S_n\}$, that generates a new output stream S_{out}=$f_{op}(S_{in})$.*

An *operator* is the basic processing unit within a stream processing system. It represents a function over a set of input streams, transforming one or more input batches into one output batch.

Operators are seen as *black boxes* in our system, meaning that their internal semantic is not taken into account when calculating the quality metric value for the newly generated derived tuples. We

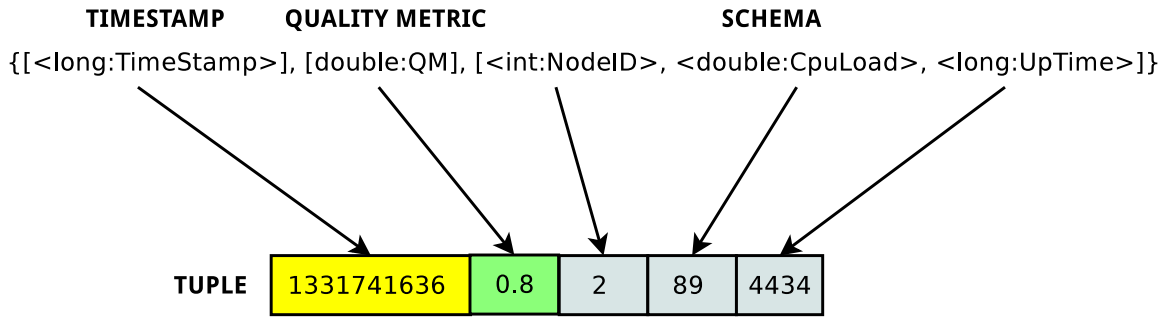

Figure 3.1: A simple tuple with the relative schema.

The third element composing a tuple is the *payload* ($\mathcal{V}$). This contains the actual information carried by the tuple. It is formed by one or more values of any primitive data type. The type and the order of the values forming a tuple payload is defined by a schema.

Logically we can divide tuples in three categories. A *source* tuple (t_{src}) is a tuple generated from a source representing a single input to the system. A *derived* tuple ($t_{op} \in \mathcal{T}^{o}_{out}$) is a tuple generated by an intermediate operator. Finally there are *result* tuples (t_{res}), which are derived tuples produced by a terminal operator. These contains the final results of the processing which are delivered to the user together with the complete quality metric value.

EXAMPLE: Figure 3.2 shows a simple tuple with the relative schema. It shows a *timestamp* expressed in POSIX time, a *quality metric* value and a a *payload* with three fields carrying information about the CPU load and uptime of a machine monitored by the system. The discussion about our implementation of tuples is given in Section 4.2.

Stream

A *stream* is an abstract entity that describes the totality of tuples flowing between two operators. These are possibly infinite and all belonging to the same schema.

Definition 3.3 (Stream) *A stream is a possibly unbounded time-ordered set of tuples, all belonging to the same schema.*

A *stream* is a logical abstraction, representing the totality of the tuples flowing between two operators. It is time ordered, meaning that a tuple t_2 received after a tuple t_1 will always have a timestamp greater or equal than the former, such as $\tau_2 \geq \tau_1$.

Similarly to tuples, streams can be divided into three categories. *Base* streams (S_{src}) are generated from sources and are the input to the system. *Derived* streams (S^{o}_{out})are produced as an output by the operators of a running query. *Result* streams (S_{res}) are derived streams produced by a terminal operator, these contains the results of the processing delivered to the user.

Model Notation	
t	tuple
τ	tuple timestamp
QM	tuple quality metric value
$\mathcal{V}$	tuple payload
S	abstract stream of tuples
B	batch of tuples
O	operator
f_{op}	operator function
Q	query
f_Q	query function
$\mathcal{S}$	set of all sources attached to a query
$\mathcal{T}^S$	source information tuple set

Table 3.1: Notation used in the model definitions.

Tuple

A schema describes the prototype for a unit of information processed by a stream processing system. A instance of a schema containing real information to be processed is called a *tuple.*

Definition 3.2 (Tuple) *A tuple is an element $t = \langle \tau, QM, \mathcal{V} \rangle$, where $\tau \in T$ is the timestamp of the element, QM is a quality metric metadata value and $\mathcal{V}$ is a set of values defined by a schema S.*

A *tuple* is the basic information vector within a stream processing system, representing a single unit of data. In our model a tuple is composed by three main elements: a *timestamp*, a *quality metric* and a *payload*.

With timestamp (τ) we mean an temporal indication of when the tuple was produced. This usually is equivalent to the time at which the tuple enters the system, in this case the sources are only concerned with the production of the raw data and the timestamp is assigned by the system as the UTC time at the moment of input. It is also possible for the timestamp to be set externally, which is particularly useful in case of synthetic workloads. In this case the timestamp is determined by an external entity and the system merely accepts this values without checking their correctness.

In our model a tuple is always augmented with a *quality* metadata value (QM), which is an indication of the amount of information carried by the tuple. This value is calculated autonomously by the system and varies according to the amount of failure (i.e. tuple loss) which occurred during the processing of the tuple. This value has two main functions: it reports back to the user the achieved quality-of-service for the current processing and it is used internally by the system to make intelligent load-shedding decisions under overload.

employing such a metric.

The chapter closes with some running examples of real queries taking advantage of the SIC metric to enhance their processing, so that failure can be detected and accounted for automatically. It show how the SIC metric can be applied to different query types and how its behaviour directly follows from the previously stated assumptions.

3.1 Model Definitions

This section presents the definition of the basic components of a stream processing system as they are intended in the scope of this work. Many of the concepts are similar or equivalent to their counterparts already described when presenting the CQL and Boxes-and-Arrows query models. Nevertheless, a precise definition of many concepts used throughout this document is necessary for the correct understanding of this work.

Schema

In order for the system to be able to interpret the content of a unit of information it is necessary to describe the values it contains, providing a name and a data type for each data item. This definition is contained in a *schema*.

Definition 3.1 (Schema) *A schema S is a data structure of elements in the form: $\langle Type, Name \rangle$.*

A *schema* defines the structure of the payload contained in a tuple. It is formed by a series of $\langle Type, Name \rangle$ pairs, each specifying the abstract type and the name of an element. It is equivalent to a schema used in a relational database, where it is used to describe the columns forming a table.

Example: Consider a tuple with the following schema:

Integer	NodeID
Double	CpuLoad
Long	UpTime

It contains information about load and uptime for a specific processing node. Each field is named so that it can be referred to when processing the tuple and is defined using an abstract type which will be translated to a system type once the tuple is instantiated.

Chapter 3

Quality-Centric Data Model

Failure in a stream processing system has been usually considered as a transient, rare condition. The common approach to handle it has been to recover from it as soon as possible, without quantifying its impact on the current processing. In many large-scale deployments instead, the occurrence of failure is common and not always avoidable. In this cases it is better to instrument the system so that it can self-inspect and quantify the impact of failure on the computed results and let the user decide if the quality of the output is acceptable or not.

In order to do so, the system should be able to quantify the amount of information lost during the processing because of failure. A quality metric should be used to enhance streams and used to detect and estimate the impact of failure on the current computation. This metric should be generic enough so that it can be used in any stream processing system, supporting the semantic of traditional as well as custom operators, and be applicable with a broad variety of query types.

This chapter presents the theoretical background about the quality-centric data model we have developed to allow a stream processing system to calculate the impact of failure on its running queries. First, a set of definition are provided, about the basic components of a stream processing system as they are intended in the scope of this work. These fundamental concepts are used to describe the working of a stream processing system and to derive a suitable quality metric.

In order to define such a metric, it is important to understand the goals and the assumptions made during its design. These are provided with an explanation about the reasoning behind them and about what characteristics such a metric should have. Next, there is an introduction to the two main families of queries, fan-in and fan-out, providing some sample queries to illustrate them.

The Source Information Content (SIC) metric is then introduced, a quality metric derived from the previous assumptions and having the required characteristics. A set of mathematical formulas are given for its calculation within the system, describing its propagation with the system and the benefits of

processing systems. The continuous query language (CQL), that extends the relational model for the processing of unbounded streams, and the boxes-and-arrows representation that employs a visual paradigm to compose queries by creating a network of operators. The next section introduced the system models for stream processing, presenting examples of systems that pioneered the centralised as well as the distributed approach. When dealing with a distributed environment the correct allocation of resources across processing sites is essential. In this regard, there has been a description of efficient resource allocation techniques for both data-center and wide-area deployments. A major focus of this work is on techniques for the management of overload, in particular load-shedding. An overview of the most relevant strategies to efficiently discard input data was provided, emphasising the different issues related to load-shedding. Finally, there has been a description of techniques used to avoid and recover from failure, with a discussion on the possible consistency and replication models. The next chapter will describe the data model developed in the scope of this work, deriving a quality metric that can used to augment streams and allow the system to detect and measure the effect of overload.

availability, since the operator continues to process even in presence of failure on its input streams. Output tuples though, are then marked as *tentative*, in order to signal the incorrectness of these results. Once the system has heal from the failure, it tries to reconcile the state of operators which processed tentative tuples, by re-running their computation with the stable data tuples. This also means that every source or operator has to buffer all their outgoing tuples, at least for a certain amount of time, to replay them in case of failure. The goal, in Borealis, is to achieve *eventual consistency*, or the guarantee that eventually all clients will receive the complete correct results.

Another approach to the management of replicas has been proposed in [MW06]. The authors take a different stand point on failure, not considering it as an infrequent event, but as a constant condition within the system. This is due to the increased size of the infrastructure, which is considered to be deployed at Internet-scale. In this conditions maintaining consistency among replica becomes even harder. In fact, the authors claim it is better to drop this constraint all together and to employ instead *free-running* operators. Operator, thus, are allowed to independently update their internal state as incoming tuples arrive. In case of recovery from failure, for instance, an operator restarts from scratch, without any knowledge on its previous state. This, on the other hand, means that an operator's state can diverge from its replicas, due to missing tuples. In general, though, the state of an operator is based only on the tuples in its *causality window*. Thus, assuming a window size of w seconds and no additional failure, the system will regain consistency without the need for explicit synchronization. However, whenever the replicas are out of sync and produce different results, the system needs to choose which set of results represents the "best" answer. It can do so in different ways, for instance by choosing tuples generated by the replica with the biggest uptime, with a majority vote among replicas, or by relying on some *quality metrics* describing the quality of the data. This process takes the name of *best-guess reconciliation*.

2.6 Summary

This chapter presented the relevant background work useful to understand the work presented in this thesis. It started with a description of some stream processing applications, in particular those that have to deal with large data sets and can tolerate a certain degree of approximation in their results. Environmental monitoring is crucial for the timely prediction of possibly disruptive weather phenomena and for the long term understanding of world-wide climate change. Social media analysis deals with the processing of user generated content in online social networks, a valuable tool for researchers that investigate the spread of information in social cascades and for online social marketers. Then there has been a presentation of the most important query languages used to express queries in stream

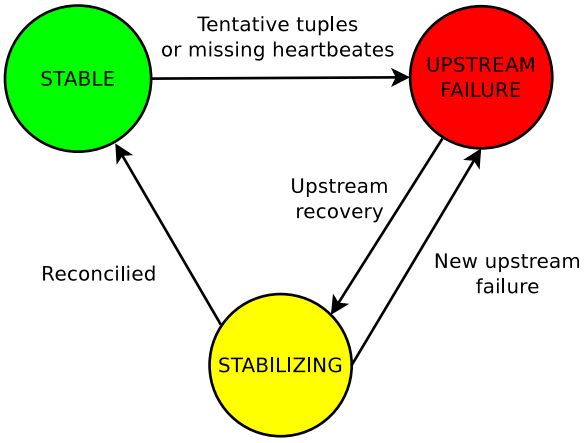

Figure 2.5: State machine describing Borealis fault tolerance mechanism.

called *harvest* and *freshness*, to provide the user with a hint about the quality of the results. Harvest represents the coverage of a query result in terms of sources and its value is reduced by failures during the query processing. An harvest of 100% indicates that the result has been computed using data coming from all sources. This metric is not directly related to correctness, aiming at providing an indication about the confidence the user should have about the delivered results. A high value for harvest indicates that the provided result represent a large number of the input sources. Freshness represents the age of the input data that generated the result. A high network latency or a high system load affect negatively the freshness value of the delivered tuples. The lower bound for freshness is the network diameter.

Replication Model

In order to be more resilient to failure, stream processing systems often employ *operator replication*. The system identifies the most important operators in a query and run them concurrently. In this way, if a node hosting a part of the query fails, one of its replicas can be employed instead, without any damage to the processing. Replication is not only useful to increase the dependability of the system, but can also improve its performance by running the replicas in competition (i.e. to reduce latency) [HcZ08; HCZ07].

Early works on replication in a DSPS have been focusing on masking software failure [Hwa+05], by running multiple copies of the same operators at different nodes. Others [SHB04] strictly favoured consistency over availability, by requiring at least one fully connected copy of the query tree to be correctly working in order for the computation to proceed.

In Borealis, the approach has been to provide a configurable trade-off between availability and consistency [Bal+05]. This is done by letting the user specify a time threshold, within this time window all tuples are processed regardless of whether failures occur or not on input streams. This increase

Consistency Model

When failure occurs in a stream processing system, a choice arises between stopping the delivery of results while recovering or delivering incorrect results. In the first case the *availability* of the system is reduced, because no results are delivered during the recovery phase, in the latter it is the *consistency* of results to be compromised. The system can also opt for a mixed approach [Bal+04], delivering first imperfect results, marking them as "unstable", while trying to correct them as soon as the failure has been overcome. This model takes the name of *eventual consistency*, because eventually all the delivered results will be correct. Another approach, called *relaxed consistency*, is to avoid correcting results and instead trying to augment them with a metric describing the amount of failure occurred [MW06]. This has been the chosen approach in this thesis, since we considered failure a constant operating condition of the system and the cost to achieve eventual consistency too high to be acceptable.

Borealis [Bal+05] adopts a strict consistency model which aims at eventual consistency. This means that in case of tuple loss or misordering, the system always tries to obtain the correct final result. It achieve this by marking as *tentative* a stream on which an error has occurred. Later it tries to restore the correct result by employing a revision process which allow the final result to be correct. This, of course, assumes a scenario where fault is more an exception than the rule, as the cost for this revision process can become really high. There is no mechanism to quantify the error to the user. As the error is thought to be transient and recoverable the only feedback given is that the stream content is invalid and cannot be trusted until recovery. We developed a quality-centric data model that allows the system to quantify the impact of the error and to provide the user with better feedback on the actual performance of the system. The revision mechanism used in Borealis also allow to perform "time travel". It means that it's possible to recalculate tuples of the past and even in the future, by providing a prediction formula. Rolling back to a past configuration is a heavy-duty process and needs a large amount of storage which is not always feasible due to the nature of stream processing. Figure 2.5 shows a state machine depicting the fault tolerance mechanism employed by Borealis. When failure occurs upstream an operator is alerted by the receival of tentative tuples or by the lack of heartbeat messages. It then marks his tuples as tentative, while waiting for stabilization. In the end, the failure is overcome, the state is reconciled and the system becomes stable once again.

Another more relax approach to consistency has been proposed by Murty and Welsh [MW06]. In this model, operators are considered to be *free-running*, updating their internal state independently as incoming tuples arrive. This means that two replicas of the same operator might produce different outputs, because of their difference in internal state due to failure. Instead of trying to achieve eventual consistency, they propose to deliver imperfect results, but augmenting them with two quality metrics

Approximate Operators

Overload can also occurs when the total state of all running operators exceeds the available memory. In this case, the system has to reduce the memory usage by reducing the state kept by operators. In general the system will try to minimize its memory footprint as much as possible employing a number of techniques. For instance it can exploit constraints on streams to reduce state, either by letting the user specify them or by inferring them at run-time. It can also *share state* among operators when they materialize nearly identical relations. It can also schedule operators intelligently in order to minimize the length of queues in memory [Bab+03]. While these techniques do not lower the quality of the processing, sometimes the memory usage might still exceed the limits of the systems. In this situation, the internal state of operators can be approximate. In the case of aggregation and join operators, the state can be reduced by employing sampling techniques, like using *histograms* [Tha+02]. These are approximations to data sets, achieved by partitioning the data into subsets, which are in turn summarised by an aggregate functions. Every column in the histogram is a compact representation of one of these subsets. Another summarising technique that can be employed makes use of wavelet synopses [Cha+01]. For set difference, set intersection and duplicate elimination operators Bloom filters can be employed [Blo70]. These are compact set representation that allow a fast way of testing whether an element is a member of the set. All these techniques trade memory use against precision, but can greatly help the system in overcoming or avoiding memory-limited overload conditions.

2.5 Failure

In a large-scale distributed stream processing system, failure should not be considered a rare, transient condition. The previous section dealt with failure due to overload, when the system resource are insufficient to carry out perfect processing. Failure, tough, can also occur for many other reasons. When dealing with a large number of processing units, it is not uncommon for a node to crash, thus leading to the partial disconnection of a query. It could also happen that a processing location becomes temporarily unreachable because of a network partition, especially in a widely geographically distributed processing infrastructure.

A stream processing system should address these issues in order to achieve a good level of dependability. The term dependability [Ucl+01] in this context refers to the ability of a system to withstand failure and to efficiently recover from it. It should choose an appropriate *consistency model*, a contract with the user stating how failure will be handled, and employ a *replication model* to be more resilient to the occurrence of failure.

maximise the output rate at the query end-points. The goal is to insert a number of load-shedding operators and to choose their drop selectivities, the percentage of tuples to be discarded, so that the total throughput is maximised. The solution of the load-shedding problem is broken down into four steps:consists of four steps: (i) advanced planning, (ii) load monitoring, (iii) plan selection, and (iv) plan implementation. In the first step, a series of load-shedding plan is computed, one for each predicted overload condition. Then, during the lifetime of the query, the load is constantly monitored at each node. When an overload condition is detected, one of the previously computed shedding plans is selected and implemented at the various nodes, installing the corresponding set of load-shedding operators. The idea is to prepare the system against all possible overload conditions before hand, and to modify the query at run-time depending on the overload scenario. This approach has been implemented in two ways: *centralised* and *distributed*.

Centralised Approach. In the centralised approach, a central server called *coordinator* is used to calculate all the load-shedding plans. It uses information obtained from all processing nodes about the operator their are hosting, their input rate and selectivities. Based on this information it generates a number of load-shedding plans to address various overload scenarios. Once the plans are generated they are pushed to the processing nodes together with their plan ids. The coordinator starts then monitoring the load of each node. If an overload condition is detected, the coordinator selects the most suitable plan to address it and triggers its application at the processing nodes, signaling which plan id they should implement among those previously received.

Distributed Approach. In the distributed approach the four load-shedding steps are performed cooperatively by all the processing nodes, without the help of a centralised coordinator. All nodes communicate with their neighbours and propagate information about their input rates and processing capabilities. Each node identifies a feasible input load for itself and all its downstream neighbours. This information about is used to compute a *Feasibility Input Table (FIT)*. Once a node has calculated its FIT table it propagates it to its parent nodes. Upon receival of such information, the parent node aggregates the FITs of all children and eliminates those that are infeasible for itself. The propagation continues until the input nodes receive the FITs of all their downstream nodes. At this point , when an overload conditions arises, every node is able to select a load-shedding plan among those contained in its FIT, reducing the load for itself and all its downstream nodes.

Load-shedding Selection

Another important aspect of load-shedding is the correct selection of tuples to be discarded. The simplest approach is to discard the required amount of tuples at random, avoiding the selection step in the load-shedding algorithm. On the other hand choosing a specific set of tuples according to some criteria allows the implementation of semantic shedding strategies that can improve the quality of the computed results. This kind of approach has been used, for instance, in the context of aggregate queries and in the context of streams presenting irregular frequency patterns.

Mozafari et al. [MZ10] propose an algorithm for optimal load-shedding for aggregates when queries have different processing costs, different importance and, importantly, the users provided their own arbitrary error functions. load-shedding is treated as an optimisation problem, where users state their needs in term of quality-of-service requirements (i.e. the maximum error tolerated) and the system implements a shedding policy that meet those guarantees. If this minimum quality of the results can not be achieved for a query with the available resources, all inputs for that query are discarded and the freed resources are given to the other remaining queries. Using this approach the user has to choose among approximate and intermittent results. The most demanding queries in terms of processing capacity are tend to deliver intermittent results, since all their data is often dropped, while small lightweight queries tend to deliver approximate results, due to a partial drop of their input data.

Chang et al. [CK09] propose an adaptive load-shedding technique based on the frequency of input tuples. In many data stream processing applications, such as web analysis and network monitoring systems, where the purpose is to mine frequent patterns, a frequent tuple in the data stream can be considered more significant compared to an infrequent one. Based on this observation,frequent tuples are more likely to be selected for processing while other infrequent tuples are more likely to be discarded. The proposed technique can also support flexible trade-offs between the frequencies of the selected tuples and the latency defined as the gap between their generation time and processing time. One of the proposed algorithms, in fact, buffers tuples for a certain amount of time, so that the load-shedding decision can be made based on the frequency of tuples over multiple time slots.

Distributed load-shedding

In a distributed stream processing system every node acts as an input provider its downstream nodes. The reduction of load at a node thus also reduces the amount of load on all other nodes hosting the following operators in the query graph. This makes the problem of identifying the correct load-shedding plan more challenging than its centralised counterpart.

In Borealis [Tat+08] the load-shedding problem is solved using linear optimisation. The goal is to

a fraction of the total capacity C that network $N(I)$ pose on the system. An overload condition is detected when $Load(N(I)) > H \times C$, where H is constant called *headroom factor* to avoid trashing. The total load calculation is a summary of the individual load of all network inputs. Each input has an associated *load coefficient*, that represents the number of CPU cycles required to push a single tuple across the entire local operator graph. Observing the input rates it is possible to calculate if the current load will lead to an overload condition and trigger the load-shedding mechanism when needed.

Load-shedding Location

Once an overload condition has been detected, the system has to choose the location where it is best to drop the input data. In general, it is better to shed tuples before they get processed, so that no CPU cycles are wasted to process data that is then going to be discarded. For some classes of applications though, it might be wiser to perform the dropping at different locations. In the case of aggregate operators then, a load-shedding operator, called WinDrop, has been proposed in order to optimise the dropping for this particular class of queries.

Babcock et al. [BDM04] proposed random drop operators carefully inserted along a query plan such that the aggregate results degrade gracefully. If there is no sharing of operators among queries, the optimal solution is to introduce a load-shedding operator before the first operator in the query path for each query . Introducing a load shedder as early in the query path as possible reduces the effective input rate for all downstream operators and conforms to the general query optimization principle of pushing selection conditions down. In the case of stream sharing among queries, the authors provide an algorithm that chooses the location and the sampling rate of the load-shedding operators in a an optimal way.

Tatbul et al. [TZ06] showed that an arbitrary tuple-based load-shedding can cause inconsistency when windowed aggregation is used. They proposed a new operator called *WinDrop* that drops windows completely or keeps them in whole. The idea behind this approach is that placing a tuple-based load-shedding operator before an aggregate does not reduce the load for the downstream operators, as the aggregate still produces tuples at the same rate. Placing the load-shedding operator after the aggregate instead, is not effective in reducing the system load, as all tuples are still processed and a valuable aggregate value is computed just to be discarded. Following this reasoning the WinDrop operator is designed to operate on the window of tuples to be delivered to the aggregate operator and discards or forward this set of tuples as a whole, according to some parameters.

cost among processing nodes, and then adapts to the changes in the data and workload within the system. Ying et al. [Yin+09] propose to also account for the state of operators when performing migration decisions. When the resources are administered by multiple authorities, the degree of trust and collaboration among them must be taken into account. The algorithm from Balazinska et al. [BBS04] achieve this by the mean of of prenegotiated load-exchange contracts between the nodes.

2.4 Overload Reduction

When the rate of input data exceeds the processing capabilities of a node, the system becomes overloaded and needs to take action. The rate of input data can vary rapidly, sometimes growing of orders of magnitude, and the resource provisioning of the system can quickly become inadequate. Scaling the processing resources is not always feasible or cost effective, and in order to overcome an overload condition it is possible to either to employ *load-shedding*, the deliberate discarding of a portion of the input data, or to employ *approximate operators* that produce a semantically similar output, but with a lower the CPU and memory footprint.

Load-Shedding

The arbitrary discarding of a certain amount of input data takes the name of *load-shedding* [Tat+03]. The need to load-shed comes from the inability of the system to process all the incoming tuples in a timely fashion. If the system is not able to sustain the incoming rate of data with its processing throughput, the internal buffers holding the tuples waiting to be processed grow in size, leading to an increased *latency* of the result tuples and eventually to an out-of-memory failure. The most important choices that the system faces when overloaded are about the amount of tuples that is necessary to discard, where to drop in order to maximise the impact of the shedding and what tuples to discard.

Overload Detection

In order to detect an overload condition the system has to constantly monitor its input rate and the size of its internal buffers. The periodic self-inspection has to be performed often enough so that the system can promptly react to a sudden variation of input rates, but trying to avoid an excessive overhead from the monitoring procedure.

Aurora [Tat+03] evaluates load based on a calculation that takes into account the processing cost of operators and their selectivity. Given a query network N, a set of input streams I with certain data arrival rates, and a processing capacity C for the system that runs N. Let $N(I))$ denote the load as

clusters, these needs to be mapped to some available physical computing sites. A placement algorithm assigns operators to processing nodes while satisfying a set of constraints and attempts to optimize some objective function. Finding the optimal assignment among the total possible assignments is an NP-complete problem and thus computationally intractable [SMW05]. Despite this, many strategies has been devised for efficient operator placement and depending on the assumptions and requirements of the system, different approaches can be more suitable than others. A taxonomy of various placement strategies for operators can be found in [LLS08].

Closely tied to operator placement is the problem of load balancing. While the former is more concerned with the placement at deployment time, load balancing has to deal with the moving of operator at run time. During the execution of the query the amount of data carried by the stream can greatly vary and this may result in computational overload at some nodes. To recover from this overload condition, the system can decide to migrate some operators to better location, attempting to balance the load among the available resources. Many operator placement algorithms recognize the need of placement reconfiguration at run time and make load balancing a key component of their strategy.

When the processing nodes of a DSPS are located within the same data center, the topology management can be assigned to a centralised placement controller. In these environments the controller can easily be aware of if the current state for the entire network, including workload information and resource availability. Having access to this information allows it to reason about placement choices, with results that are potentially optimal for small deployments. When the number of available resources becomes very large though, these strategy are not indicated because of their low degree of scalability. Abadi et al. [Aba+05] propose an example of such a solution, which assumes a fairly constant workload and an heavy cost for operator migration. Xing et al. [XZH05] instead, consider the workload to be highly unpredictable, thus requiring some load balancing at runtime. They also acknowledge the importance of initial deployment, and a high migration cost for operators, thus developing an algorithm resilient to load variations [Xin+06]. All these algorithms have been implemented within the Borealis DSPS.

When processing nodes are instead distributed over wide-are networks, a central coordinator becomes ineffective and a decentralized approach to the problem have to be employed. These algorithms make decisions based on local workload and resource information. Pietzuch et al. [Pie+06], for instance, employ a distributed algorithm based on multiple spring relaxation, trying to globally minimize a metric called network usage, based on both bandwidth and latency. Another approach has been taken by Amini et al. [Ami+06], in this case the algorithm maximises weighted throughput of processing elements, while ensuring stable operation in the face of highly bursty workloads. Zhou et al. [Zho+06] propose an algorithm in which the initial deployment is determined by minimising the communication

between them.

Distributed

The natural step in the evolution of DSPSs is distribution. Centralized systems are inherently subject to scalability issues, being able to distributed the processing over a number of nodes is the logical approach to overcome this issue. Distribution also allow for better dependability as operators can be replicated at different locations increasing availability. Fault-tolerance can also be better achieved by employing distribution, as different replicas of operators can be deployed, achieving resilience to failure. The overall performance of the system is also increased by this approach, as the computation can be partitioned and run in parallel at different computing nodes. Distribution can be realised at different scales. Processing nodes can be located within the same data center, taking advantage of the high-bandwidth and low-latency typical of these environment, or spanned over wide-area networks to push scalability even further.

Distribution presents several advantages, but it does come at a cost. Many issues arise when the system is distributed: the operational complexity the system increases and resources need to be allocated efficiently. In a DSPS distribution is achieved by partitioning the query plan and running clusters of operators at different sites. Placing these operators correctly at deployment time and moving them at run time to rebalance the system is not an easy task. Running replicas, also, can greatly help to increase the dependability of the system, but their correct management also represent a key issue. Finally, being able to fully exploit the system resources is also major challenge.

Building on the experience of Aurora, a centralised DSPS, other systems have been developed to explore the possibilities offered by distribution. Aurora* [Che+03] was an evolution of the Aurora system that aimed at providing interconnection among multiple Aurora instance running on a cluster environment. A similar exampled was Medusa [Sbz+03], that tried to expand the boundaries of distribution outside a single cluster, onto different processing sites administered by different authorities. Borealis [Aba+05] has been the realization of these experiences into a complete distributed stream processing systems. Borealis has been used to develop many research questions, like resource allocation, load-shedding and replica management. In the following sections, many of these topics will be covered in more details.

Operator Placement and Load Balancing

The optimal distribution of streaming operators over a number of network machines is key to maximise the performance of a distributed stream processing system. Once the query plan has been divided into

processing it would introduce an unacceptable delay, due to the latency cost of storing and retrieving data on disk.

Figure 2.4 shows the two different architecture. The left image depicts a traditional DBMS. Queries are issued over previously stored data. Since the data is already in the system, it is possible to create indexes over it, in order to reduce the query execution time. Results are generated from data received by the system *up to the time* when the query is issued. The picture on the right, instead, depicts a DSPS. Queries are issued over a constantly changing stream of data. Once the data enters the system, it is matched against the registered queries to generate the results. Results are generated from data received by the system starting from the time when the query is issued.

Centralised

The first generation of stream processing systems was centralised, with one processing node handling all the computation. Examples of such systems are STREAM and Aurora.

STREAM [Ara+04; Mot+02; Bab+03] is a centralised DSPS which tries to address the issue of long-running continuous queries. It introduced a relational language called Continuous Query Language (CQL) [ABW06], which allows the expression of SQL-like queries over continuous data streams (see Section 2.2). Once a query is registered in the system, a correspondent *query plan* is compiled from it. Query plans are composed of *operators*, which perform the actual processing, *queues* which buffer tuples as they move between operators, and *synopses*, which store operator state. All queries are expressed in CQL and then converted into an actual query plan which is composed by these basic elements. Aurora [Car+02] instead, introduced a different representation model for queries, called *boxes-and-arrows*. In this model, operators are seen as boxes, with an input, an output and a specific processing semantic. Operators are linked together by arrows representing the stream of tuples flowing

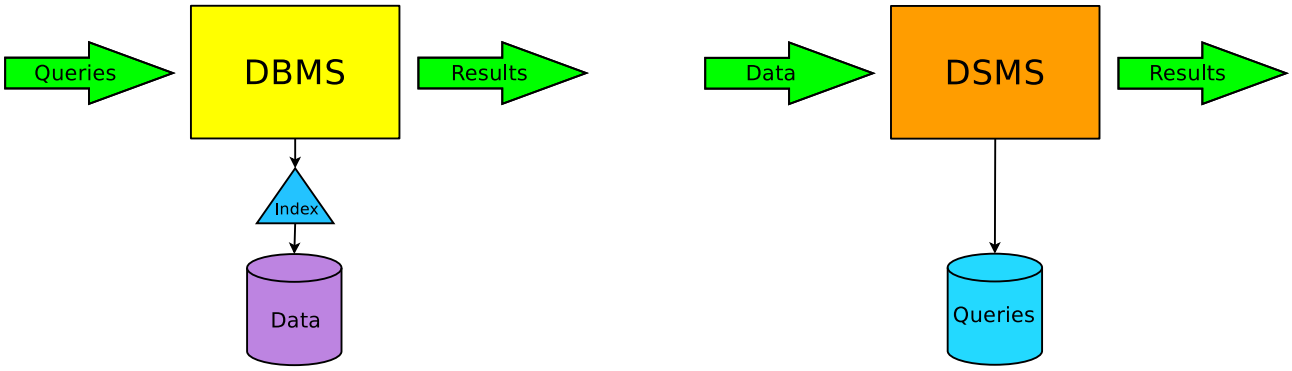

Figure 2.4: Comparison between a traditional Data Base Management System and a Data Stream Management System. On the left queries are issued over stored data, while on the right data flows through continuous queries.

Join is another classical stateful operator. In this case, there are two input streams and two windows. Join pairs tuples from the two input windows that satisfy some specified predicate. This operator very much resembles its relational counterpart. A special kind of stateful operators are synchronization operators. In this class we find *Lock, Unlock* and *WaitFor*. These are used to temporarily buffer tuples until a certain condition is reached.

Load-shedding operators are used to overcome overload conditions. The load on an operator is reduced by discarding a portion of its input. In this category we find, for instance, RandomDrop and WindowDrop. *RandomDrop* operates by randomly discarding a certain percentage of a stream. Every time the operator is scheduled, it drops a number of tuples on its input window at random, according to specified dropping parameters. *WindowDrop*, instead, allows for a more sophisticated specification of the windowing parameters. Once a window has been computed, it probabilistically choose if it should be kept or dropped as a whole, according to a dropping parameter.

Example:

In the boxes-and-arrows model, implementing our temperature monitoring query is simple. Figure 2.3 shows the graphical representation of this query in this model. In this case, the query plan is composed only by a *Filter* operator. It receives the stream of readings in input and outputs all tuples satisfying the predicate T $\geq$ 30.

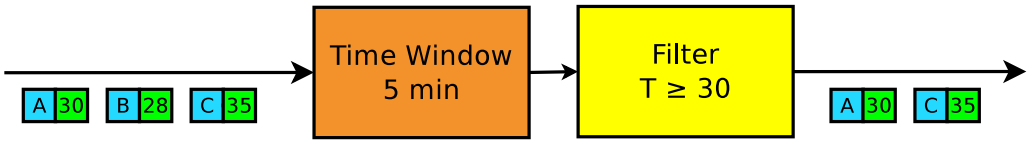

Figure 2.3: Temperature monitoring query example, expressed according to the boxes-and-arrows representation.

2.3 System Model

Data Stream Management Systems (DSMS) have been developed to process large volumes of stream data, generated in real-time with new values constantly being introduced in the system. Using a traditional approach of first storing the data and then issuing the queries over it is not appropriate in this case. The first reason is the amount of available data [Tur+10; ScZ05], which is usually very high and potentially too large to be stored entirely in the system. Another reason not to store all the incoming data has to deal with its nature, as often only a subset of the data streams are relevant to the application. Moreover, stream data is usually transient, with new values constantly updating old ones and rendering them irrelevant. Data stream processing is also tied to strict latency constraints. Processing data streams with a low latency in this case is very important and storing the data before

WHERE TMP > 30

Boxes-and-Arrows Model

A different paradigm to express queries over streams takes the name of *boxes-and-arrows* [Aba+03]. It has been introduced by Aurora [Sbz+03] and also employed by Borealis [Aba+05]. In this model a query is represented graphically as a data-flow. Operators are depicted as boxes with streams as arrows connecting them. Tuples flow in a loop-free, directed graph of processing operations. Queries in this model can be expressed by composing the query plan using a graphical interface or by employing a query language called SQuAl (Stream Query Algebra). Aurora employs its own set of primitive operators, even though many of these have equivalents in the traditional relational query model. In this model, we can divide operators in two main categories: stateless and stateful [Tea06].

Stateless operators do not need to store any information about previous tuples in order to execute. *Map* is an example of such operators. It transforms some attributes of a tuple applying a predicate. If we consider a tuple carrying a temperature reading, for instance, a Map operator can be employed to convert the temperature field from Fahrenheit degrees to Celsius.
In the same category we also find *Filter*. This can be seen as the equivalent of a *Select* in the relational model. It applies predicates to each input tuple and route it to the output stream of the first predicate that evaluates to *true*. A Filter operator can have multiple output streams, depending on the number of specified predicates.
Union is a simple operator that produce a single output stream from many input streams with the same schema. The operator itself doesn't produce any modification on the tuples, it simply merges streams. It is used, for example, to provide a single stream as an input to other operators, like an Aggregate or a Filter.

Stateful operators, instead, maintain some internal state, determined by tuples previously processed. In the boxes-and-arrows model, there is no explicit concept of *window* operator, unlike in CQL. Instead, some operators are designed to support windowing on their input streams. The semantics for expressing the organizations of these windows are rich, and it is possible to express a wide range of windows, such as count-based, time-based and partitioned. A sliding parameter can also be specified to express the updating policy of the window.
Aggregate is a typical example of such an operator. It computes an aggregate function, like a sum or an average, over a window of tuples. A wide range of aggregate functions is provided, and others can be specified by the user. It accepts a single input stream, and it is usually preceded by a Union operator.

$$IStream(R) = \bigcup_{\tau>=0}((R(\tau) - R(\tau-1)) \times \{\tau\})$$

2. **DStream** (for "delete stream") outputs only tuples that have disappeared since the last execution. It compares the current output of the preceding relation-to-relation operator and outputs all the tuples that were present in the previous input, but not in the current one. Informally it generates the stream of tuples that have been deleted from the relation. Formally the output of *DStream* applied to a relation R contains a stream element $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau - 1) - R(\tau)$.

$$DStream(R) = \bigcup_{\tau>=0}((R(\tau-1) - R(\tau)) \times \{\tau\})$$

3. **RStream** (for "relation stream") informally outputs all tuples produced by the relation-to-relation operator. Formally the output of *RStream* applied to a relation R contains a stream element $\langle s, \tau \rangle$ whenever tuple s is in R at time τ.

$$RStream(R) = \bigcup_{\tau>=0}((R(\tau) \times \{\tau\})$$

EXAMPLE: Figure 2.2 illustrates a simple CQL query implementing out temperature monitoring example. It signals all rooms where a temperature above 30 degrees celsius is detected in the last 5 minutes. The input stream is called ROOMTMPSTR and carries tuples formed by a room identifier (ROOMID) and a temperature reading (ROOMTMP). A time-based window id applied to it, to obtain a snapshot that contains only fresh measurements generated in the last five minutes. Once this relation has been created a conditional SELECT is applied to it, filtering out all tuples with a temperature value lower than 30 degrees. Finally, a new stream, called HOTROOMSSTR, is created by an ISTREAM relation-to-stream operator, containing tuples coming from over heated rooms, which are then fed to the air conditioning control subsystem.

SELECT ISTREAM(*)
FROM ROOMTMPSTR[RANGE 5 MIN]

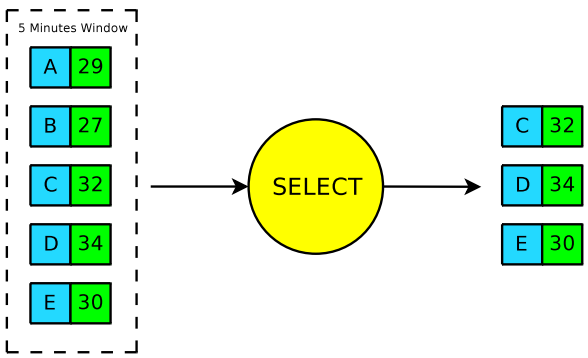

Figure 2.2: This query outputs all tuples having a temperature reading ≥ 30 in the last five minutes.

partitioned window is a special kind of the tuple-based. It allows the user to specify a set of attributes as a parameter and splits the original stream based on these arguments, similarly to the SQL GroupBy, and computes a tuple-based window of size N independently creating a number of substreams.

Sliding Windows produce a snapshot of a stream based on two parameters: the *window size* and the *sliding factor*. Once the window is triggered it outputs all the tuples it contains, but it does not discard them all. The sliding factor determines a criteria to hold some tuples in the window so that they can be used in the next iteration as well. A tuple-based window can be set to trigger every 5 tuples, but to slide of 1, so every time it reaches 5 tuples it outputs all of them, but retains the 4 newest tuples for the next iteration. A time-based window can be set to trigger every five minute with a sliding factor of 1 minute, so at every iteration it outputs all the tuples arrived in the last 5 minutes, but retains all tuples arrived in the last 4 minutes.

Relation-To-Relation. operators are derived from the traditional SQL relational model, They perform the bulk of the data manipulation and are equivalent to many canonical SQL operators. In this category we find, for instance, Select and other familiar operators. These are the main operators in the system, as every stream is converted in a relation by a window operator, then processed by a relation-to-relation operator and finally output by relation-to-stream operator.

Example: Consider the following query for our temperature monitoring example:

```
Select *
From RoomTmpStr[Range 5 Min]
Where Tmp > 30
```

This query is composed by a stream-to-relation time-window operator, followed by a relation-to-relation operator performing a projection. The output is a relation that contains, at any instant, the set of all temperature readings received by the system in the last five minutes, with temperatures greater than 30 degrees Celsius.

Relation-To-Stream. operators convert the result of relation-to-stream operators back into a stream. Three kinds of such operators exists in CQL: *IStream, DStream* and *RStream.*

1. **IStream** (for "insert stream") outputs only the new tuples that have been generated since the last execution. It compares the current output of the preceding relation-to-relation operator and outputs all the tuples that are present in the current input, but not in the previous one. Informally it inserts new tuples into the stream. Formally the output of *IStream* applied to a relation R contains a stream element $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau) - R(\tau - 1)$. Assuming $R(-1) = \emptyset$, we have:

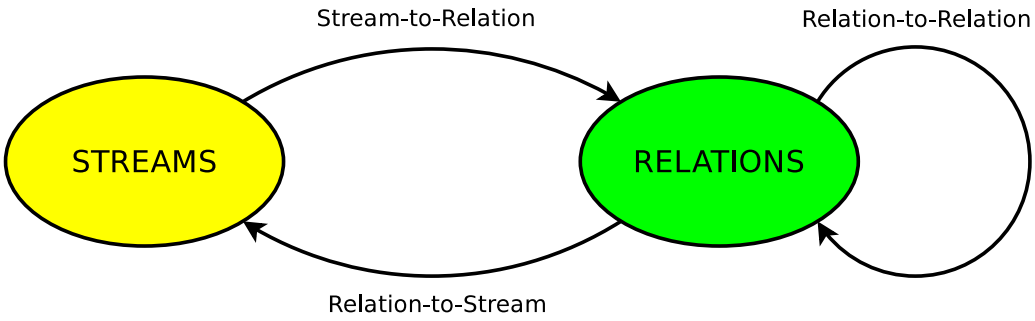

Figure 2.1: Interaction among the three classes of operators in CQL.

located, while RoomTmp contains the current detected temperature.

Definition 2.2 (Relation) *A relation R is a mapping from each time instant T to a finite, but unbounded bag (multi-set) of tuples belonging to the schema of R.*

A relation R defines and unordered multi-set of tuples at any instant in time $\tau \in T$, denoted as $R(\tau)$. The difference between this definition and the one used in databases is that in the standard relational model a relation is simply a set of tuples, with no notion of time.

Example: In our example, a relation is created form the base stream of temperature readings through a time-window operator. It is a snapshot of of all readings received within the last minute. The concept of windows over streams will be better explained in th next paragraph.

Operator Classes

CQL employs three classes of operators to process stream data. First is the *Stream-to-Relation*, which allows the creation of a snapshot of the stream. Once this finite bag of tuples has been obtained it employs *Relation-to-Stream* operators, which are equivalent to the ones employed in a traditional DBMS. Finally there are *Relation-to-Stream* operators, to recreate streams out of the newly computed relations. Figure 2.1 shows the interaction of these classes of operators.

Stream-To-Relation. operators are used to isolate a subset of a stream, or snapshot, so that one or more relation-to-relation operators can act on it. All the operators in this class are based on the concept of *sliding window* over a stream. This contains, at any point in time, an historical snapshot of a finite portion of the stream.

Three kinds of window operators exists in CQL: time-based, tuple-based and partitioned. A *time-based* window contains all the tuples in the stream with timestamp within the specified boundaries. For example it could hold all the tuples arrived in the last minute. In a *tuple-based* window instead, the number of tuples is specified and fixed, so it only contains the last N tuples of the stream. A

The next sections present some more details about the the two main paradigm used to express queries in a stream processing system: the *Continuous Query Language* and the *Boxes-and-Arrows* representation.

CQL: Continuous Query Language

CQL (Continuous Query Language) [ABW06] is a declarative language for expressing continuous queries over streams of data. It has been introduced by STREAM (Stanford Stream Data Manager) [Ara+04] and has become one of the de facto standards in stream processing. It is an SQL-like language designed to work on streams as well as with relations and to allow and easy conversion among them. It extends SQL, introducing semantics to enable queries to be issued over streams of data. A detailed description of the more formal aspects of the language can be found in [KS05].

To better illustrate the main concepts in CQL, let's consider a *temperature monitoring* query. The purpose of such query is to detect if the temperature in any room of a house rises over a certain threshold. If a room becomes too hot the system detects it and reacts by switching the local air conditioner on. In this scenario all rooms in the house are equipped with a thermal sensor. These are connected to a stream processing system, which is able to command the air conditioner of the room. The thermal sensors sample the temperature in the room every minute, sending these readings, together with their room id and timestamp, to the stream processing system. This executes a simple query, filtering all tuples with a temperature reading above a certain threshold (i.e. $T >= 30$. When the system detects that a room is too hot, it switches on the air-conditioning system in the that room.

Data Types

The two core data types manipulated by CQL are *streams* and *relations*.

Definition 2.1 (Stream) *A stream S is a (possibly infinite) bag (multiset) of elements $\langle s, \tau \rangle$, where s is a tuple belonging to the schema of S and T is the timestamp of the element.*

Timestamps are not part of the schema of a stream and there could be zero, one or multiple elements with a certain timestamp in a stream. The number of elements with the same timestamp in a stream is finite but unbounded.In CQL the data part (not timestamped) of a stream element takes the name of *tuple*. There are two classes of streams: *base streams*, which are the input of the system, and *derived streams*, which are intermediate streams produced by operators.

Example: In our temperature monitoring example there would be only one base stream with schema: RoomTmpStr(RoomID, RoomTmp). Attribute RoomID identifies the room where the sensor is

observation that a social cascade tends to propagate in a geographically limited area, since social connections often reflect real life relationships. It easy to see how a video spoken in a particular national language would tend to propagate within the national borders of the country in which that language is native. This study analyses a corpus of 334 millions messages shared on Twitter, extracting about 3 millions single messages with a video links. It was found that about 40% of steps in social cascades involve users that are, on average, less than 1,000 km away from each others.

2.2 Query Languages

In a DSPS data is constantly pushed into the system in the form of streams. These are real-time, continuous, ordered sequences of data items. Streams have different characteristics than traditional relations in a DBMS, and the way of expressing queries in a DSPS has to take these differences into account. Streams can be unbounded, but queries are usually issued over a recent snapshot of the data stream, which takes the name of *window*. A query language for data streams must be capable of operate over windows of data, should be extensible, in order to support custom user operators, and simple enough to enable the easy expression of complex queries.

Three query paradigm have been proposed to operate over streaming data [GO03]. The first and most adopted is the *relational model*. This comes directly from the DBMS world and is usually implemented as and SQL-like language, it provides the syntax to express windows over data streams, similarly to the extensions introduced by SQL-99. A query describes the results, in a declarative way, giving the system the flexibility of selecting the optimal evaluation procedure to produce the desired answer. Examples of languages employing this model are CQL [ABW06], StreamSQL [Jai+08b] and SPACE [Ged+08].

Another method for expressing queries is through *procedural languages*. In this paradigm the user constructs queries in a graphical way, having maximum control over the exact series of steps by which the query answer is obtained. In this model operators are depicted as boxes, and arrows represents the data streams flowing through them, thus taking the name of *Boxes-and-Arrows* model. One drawback of this paradigm is the difficulty of expressing complex query plans, as the diagram describing the query can be become very intricate. Aurora [Car+02] and Borealis [Aba+05] are examples of systems employing this query model.

The third model for expressing queries is the *object-oriented* paradigm. In this approach the query elements are modeled in an hierarchical way, taking inspiration from the object-oriented programming world. In Cougar [BGS01], for instance, sources are modeled as ADTs (Abstract Data Types), exposing an interface consisting of the sensor's signal processing methods. A discussion on query languages and practical issues on building a stream processing system can be found in [Tur+10].

More formally a user *Bob* is reached by a social cascade when he receives a certain content *c* and:

1. User *Alice* already posted content *c* before user *Bob* and
2. There is a social connection between user *Alice* and user Bob

Social cascading in Twitter. The diffusion of links through Twitter can be used to better understand how a social cascading tree looks like. Twitter is a micro-blogging website where users can share a short message of maximum 140 characters with other users called followers. In this particular social network it is also possible to further characterise cascades into to main groups: L-cascades and RT-cascades [Gal+10].

L-cascades occur when a certain content is shared by direct followers. More formally we can say that an L-cascade is the graph of all users who tweeted about a certain content *c*. A cascade link is formed when 1) User *Alice* and user *Bob* shared a content *c*, 2) User *Alice* posted *c* before user *Bob*, 3) User *Bob* is a follower of user *Alice*.

RT-cascades instead do not only take into account only direct connections, but also the possibility of crediting a certain content to a user even without being a direct follower. If user *Bob* wants to give credit to user *Alice* for a certain content, he prepends his new tweet with the conventional *RT @Alice* followed by the original tweet. In this way *Bob* gives a direct credit to *Alice* for the content of the tweet even without being a direct follower. This phenomenon became known as retweetting [RTW12]. More formally we can say that an RT-cascade *R(c)* is the graph all the users who have retweeted content *c* or have been credit for it. A cascade link is formed when 1) User *Bob* tweeted about content *c*, 2) User *Alice* tweeted about content *c* before user *Bob*, 3) User *Bob* credited user *Alice* as the original source of the content.

Impact of social cascades on CDNs. It is difficult to predict where and when a certain content will become popular. Many content providers rely on Content Delivery Networks (CDNs) to distribute their content across several geographically distributed locations in order to enhance the availability of the content by their users. The choice of *what* content to replicate, *where* and for *how long* is crucial to the reduction of the costs associate with the use of a CDN. Being able to predict the popularity trend of a certain content is needed for the correct provisioning of resources. This is especially important since it has been found that the top 10% of the videos in a video-on-demand system account for approximately 60% of accesses, while the rest of the videos (the 90% in the tail) account for 40% [SYC09].

An attempt to characterise the spread of social cascades focuses on improving the caching of contents exploiting the geographic information contained in the cascades [Sce+11]. This approach leverage the

the establishment of interactive closed loops between the forecast analysis and the instruments. This sensing paradigm, based on numerous input devices and the possibility of directly process their data into a forecasting model, is changing the way meso-scale weather events are detected and will help to greatly mitigate their impact. Stream processing has the potential of enhancing the CASA/LEAD infrastructure, extending its real-time monitoring capabilities.

Real-time Social Media Analysis

Social networks have experienced an large growth in the past years [Mis+08], changing the Internet landscape by shifting the balance of the available information towards user-generated content. Numerous sites allow users to interact and share content using social links. Users of these networks often establish hundreds to even thousands of social links with other users, constantly reshaping the social graph [Vis+09]. Users share photos [FLK12; FBK12], videos [YTB12], status updates [TWT12] and locations [FSQ12], providing a constant stream of valuable information for researchers and companies. Social media analysis has become an essential tool for online marketing and many tools have been developed to help the discovery and reach of possible customers [WDF12; BWH12]. The possibility of target specific segments of the online population with customised advertisements allows companies to increase their return on investment and users to avoid unwanted communications. Furthermore, analysing Twitter streams it is possible to predict in real-time the evolution of numerous phenomena, like the spreading of a disease [MPH12] or the fluctuation of stock quotes [BMZ10]. Other applications analyse keywords in status updates to determine the public sentiment towards a certain topic, brand or public figure [SCM12; TWZ12; TWF12]. The diffusion of a user generated content through a social network takes the name of *social cascade*. The next section describes this phenomenon and explains why its prediction can be useful for the correct allocation of resource by content distribution networks.

Social Cascades. One interesting application of this social media real-time analysis is the possibility of predicting social cascades. A *social cascade* is the phenomenon generated by the repetitive sharing of a certain content over Online Social Networks (OSNs). One person discovers an interesting piece of information and shares a link to it with a few friends, who share it themselves and so on. When a content is considered interesting by a community it starts being shared over and over again, possibly reaching a large number of hits in a short period of time [Cha+08]. Due to the characteristics of social networks, this process mimics the spread of an epidemics [SYC09]. Initial studies about this phenomenon date back to the 1950s [RR03], with the theory of Diffusion of Innovation. It is only now though, with the wealth of information shared through OSNs that is possible to study social cascading in much more detail.

times is most probably infeasible and it is important for the computing backbone of such projects to be able to withstand a certain degree of failure within the system [MW06].

Distributed Collaborative Adaptive Sensing. Meso-scale weather events, such as tornadoes and hurricanes, cause every year a large number of deaths and a great deal of damage to infrastructures. Being able to understand and predict when these hazardous weather conditions will occur is going to greatly mitigate their consequences.

In this regard, the National Science Foundation has recently established the center for *Collaborative Adaptive Sensing of the Atmosphere (CASA)* [Mcl+05]. This project aims at enhancing the current infrastructure of long-range weather observing radars, with a large number of small solid-state radars able to increase the sampling resolution throughout the entire troposphere. Although current long-range radar technology allows the coverage of large areas with a relative small number of devices, these are not able to correctly measure the lowest part of the atmosphere in areas faraway from the radar. This is due to Earth curvature and terrain-induced blockage. This new sensing paradigm, based on a greater number of smaller devices, goes under the name of Distributed Collaborative Adaptive Sensing (DCAS). Distributed refers to the use of numerous small and inexpensive radars, spread near enough to fully measure the area even at lower altitudes where the traditional approach fails. Collaborative refers to the coordination of multiple devices covering overlapping areas to increase the resolution and the precision of the measurements compared to a single radar. Adaptive refers to the ability of the infrastructure to dynamically adjust its configuration based on the current weather conditions and user needs.

Another project, closely linked to CASA is the *Linked Environments for Atmospheric Discovery (LEAD)* [Li+08]. This gives scientists the tools with which they can automatically spawn weather forecast models in response to real-time weather events for a desired region of interest. It is a middleware that facilitates the adaptive utilization of distributed resources, sensors and workflows. While CASA is primarily concerned with the reliable collection of the data, LEAD is the backbone processing infrastructure. It allows the automation of time consuming and complicated tasks associated with meteorological science through the creation of workflows. The workflow tool links data management, assimilation, forecasting, and verification applications into a single experiment. Weather information is available to users in real-time, whom are not being restricted to pre-generated data, greatly expanding their experimental opportunities.

The integration of these two systems offers an invaluable infrastructure to atmospheric scientists [Pla+06]. First of all, it allows meteorologists to directly interact with data from the instruments as well as control the instruments themselves. Also, unlike traditional forecasting approaches, it allows

number of large-scale sensing projects are being developed and we can expect even more to come soon into place [ERS12; NEO12; Pla+06; SWX12].

One area in which large-scale sensing has flourished is *environmental monitoring*. The growing concern about climate change has brought a lot of attention to environmental studies and many large-scale sensing projects have been launched to better understand the behaviour of many natural processes. Examples of such efforts are for instance the Earthscope [ERS12] and the Neon [NEO12] projects.

Earthscope is a multi-disciplinary project across earth sciences to study the geophysical structure and evolution of the American continent. It uses a large number of sensors geographically distributed over the whole continent to answer some of the outstanding questions in Earth Sciences by looking deeper, at increased resolution, and integrating diverse measurements and observations. Thousands of geophysical instruments measure the motion of Earth's surface, record seismic waves and recover rock samples from depths at which earthquakes originate. All the collected data is freely available and a large community of scientists is conducting several multidisciplinary researches based on it. Examples of sensing projects within Earthscope include the constant monitoring of the San Andreas fault and the Plate Bound Observatory, which collects information about the tectonic movements across the active boundary zones

NEON stands for National Ecological Observation Network. This project aims at creating a new national observatory network to collect ecological and climatic observations across the United States. It is an example of continental-scale research platform for discovering and understanding the impacts of climate change, land-use change, and invasive species on ecology. It is the first observatory designed to detect and enable forecasting of ecological change at continental scales over multiple decades. Obtaining this kind of data over a long-term period is crucial to improving ecological forecast models and will greatly help understanding the effects of human interference on climate change. These are just two examples of large-scale monitoring projects, and many others already exists or will be started in the near future [TBN12; SKS12; NEO12; USV12].

Environmental sensing is a natural application for stream processing systems, as a constant flow of measurements is generated by a large number of sensors. Stream processing offers an intuitive and flexible paradigm to harness the complexity of mining such a high volume of data. It also give the possibility of obtaining real-time picture of the measured phenomena, enabling quick reactions to possibly disruptive events. The core components of these projects are sensing stations geographically widely distributed and often subject to harsh conditions. In this scenario, the failure of these sensing devices is not going to be uncommon and often their replacement is going to be problematic. Long running continuous queries need to be able to deal with some missing data, adapting to the new conditions and reporting the achieved quality-of-service. To collect and process all the data at all

2.1 Stream-based applications

Traditionally stream processing has been applied to application domains that require very low tolerance of failure. These applications are able to produce meaningful results only when the processing is carried out without any loss of data or approximation. Financial algorithmic trading [Str08] is an example of such an application. Investment banks, trading on the stock market, need to process a great deal of financial events in real-time. Complex algorithms are used to capture the current situation of the market and give hints on profitable trades exploiting temporary arbitrage conditions. For these applications the focus is on efficiency: being able to make a trade even a split second before a competitor is the key to achieve maximum profits. In this scope the necessity of processing all the available information is evident, as a wrong choice based on approximate data could mean the loss of large amounts of money.

On the other hand, there are some classes of applications which are able to operate correctly even not all the data can be correctly processed. In many cases a certain degree of failure in the system is tolerable and does not hinder the application from producing meaningful results.

Two broad classes of applications will be described that can benefit from this approximate result approach. The first comprises research projects aiming at the construction of a *global sensing infrastructure* [Kan+07; AHS06; Gib+03]. These are concerned mostly with sensor data, trying to develop a worldwide infrastructure in which different classes of sensors can be connected and accessed through a common interface, allowing users to process data streams generated by different sensor networks.

A second category of applications that can benefit from this approach is the *real-time processing of social media events*. These analyse the constant stream of user generated content trying to extract useful information from it. It is possible to process user generated data streams, coming for instance from Twitter, to understand how the public reacts to a certain product or event [TWS12; TWA12], identifying trends as they rise [BMZ10; GBH09]. It also possible to exploit the locality information disclosed by the users to identify differences in lifestyle and preferences, as does FourSquare [FS112; FS212].

Environmental Monitoring

The first domain of applications I will take into consideration is large-scale scientific sensing, in particular I will describe projects dealing with large-scale environmental monitoring. In recent years, the availability of cheap micro-sensing technology has unleashed the capability of sensing at an unprecedented scale. We can expect in the future to see everything of material significance being *sensor-tagged* and report its state or location in real-time [Gib+03; GM04; Bal+07]. At the present time, a great

Chapter 2

Background

In recent years there has been a growing demand for applications capable of processing high-volume data streams in real-time [ScZ05]. These stream based applications include, among others, financial algorithmic trading [Str08], environmental monitoring [SWX12] and the real-time processing of social media events [PZYD11]. This research work focuses on the issues arising when the amount of input data exceeds the processing capabilities of the system. In this scenario it is necessary to employ techniques apt to overcome the overload condition and to keep delivering results in a timely fashion. In this circumstance the system should gracefully degrade the correctness of the computed results, following the principle that an approximate result can be better than no result at all.

Many classes of applications do not require the delivery of perfect results at all times. Examples are environmental monitoring and social media analysis. These application domains are the most suitable to be operated under overload, as they are able to tolerate a certain degree of approximation in the results. They deal with large sets of input data, that may require an extremely large amount of processing resources, possibly not in the availability of the user. In many cases it is possible to trade some correctness of the results, with a reduction of cost for the processing infrastructure.

Overload is a particular kind of failure, as an excessive amount of data can render a stream processing system unusable. The management of overload can be achieved mainly with two techniques: load-shedding and approximate operators. Load-shedding is the process of deliberately discarding a certain percentage of the input data. The correct execution of this procedure, choosing the right set of tuples to de dropped, is crucial and can have a great impact on the quality of the computed result. In parallel it is also possible to reduce the complexity of operators, employing approximate versions that present a similar semantic but pose a significantly lower computational cost.

of stream processing. It presents different approaches to overcoming overload, focusing on the different load-shedding strategies available. It also look at techniques used to deal with failure, as consistency and replication.

Chapter 3 introduces a quality-centric data model, where streams are augmented with the SIC meta-data metric. This is a quality metric designed to capture the amount of failure that occurs in the generation of a tuple. It provides a hint about the amount of information captured by a tuple and thus about its contribution to the query results.

Chapter 4 presents the DISSP stream processing system, a research prototype that implements the proposed quality-centric data model. It explains how the abstract concepts introduced by the model has been implemented in a concrete system. It describes the design of the prototype and shows how it the SIC metric is calculated and used.

Chapter 5 focuses on the overload management process by analysing the load-shedding process. It introduces the concept of fair resource allocation among queries and a semantic shedding policy that implements it. It describes the algorithm used to discard tuples in an overloaded nodes so that all queries achieve a similar SIC value for their results.

Chapter 6 presents a set of experiments designed to evaluate some properties of the SIC quality metric. It investigates its correlation with the accuracy of results. It compares a random shedding policy with the proposed fair shedding algorithm. Finally it looks at the trade-off between cost and SIC value degradation, offering the possibility to voluntarily accept imperfect results in order to reduce the operational costs.

Chapter 7 concludes the document and present a discussion on the research contribution of this work.

DISSP prototype design. The theoretical concepts introduced by the data model have been applied in the DISSP stream processing prototype. This system was designed to make use of the SIC quality metric at the core and is able to operate under heavy overload while tracking the quality-of-service achieved by every query. It also implements a semantic shedding policy that aims at providing a fair amount of resources to all queries. Overload should be considered a common operating condition on such systems and not an exception. Overload can be considered a kind of failure, since the system is not able to fully carry out the required computation. A system operating under constant overload is then subject to continuous failure. Instead of considering this failure as a rare, transient condition, the system should accept it and degrade its processing quality, while reporting to the user the achieved quality-of-service. In some cases it is better to reduce the quality of the delivered results in order to maintain the cost within the user's budget. Some classes of applications can still produce meaningful results even if some data is lost during the computation, since, in many cases, an imperfect result is better than no result at all.

Experimental evaluation. The first set of experiments investigated the correlation between the degradation of SIC values and the correctness of results. Even though the SIC quality metric was not designed as an accuracy metric, it was found that for many aggregate queries there is a good correlation between SIC value and correctness.

Having introduced the possibility of implementing semantic shedding policies, a second set of experiments was designed to test the advantages of employing the SIC quality metric in the load-shedding process. In particular the experiments focused in the comparison between a random shedder and the fair shedding algorithm introduced in Chapter 5. It was found that employing the fair shedding policy results in a better resource allocation among queries for many classes of queries.

Finally, a set of experiments tested the correlation between SIC degradation and reduction of cost. In this scenario the user voluntarily accepts a reduction of quality in the results, i.e. lower SIC values, in order to reduce the amount of required resources and thus the cost in a cloud deployment. The results showed that it is possible to strike a trade-off between a reduced SIC value of results and the costs of renting the processing infrastructure.

1.5 Document Outline

The rest of this document is organised as follows. Chapter 2 presents the background material relevant to this research. It describes some applications of stream processing, focusing on those that better tolerate approximate processing. It introduces different streaming data models and the basic concepts

1.4 Contributions

The choice about what tuples should be shed and what should be kept is crucial to the quality of the results. The system has to decide which tuples are more important for the processing and privilege these over others other tuples of lower quality. Not all tuples contain the same amount of information and thus they have a different value to the query. A tuple containing a value aggregated over a large number of sensors is probably more valuable than a a tuple containing a single reading. The system should be able to reason about the information content of tuples in order to identify the most valuable set of tuples to be spared from shedding. To do so it is necessary to introduce a *quality metric* that captures the quantity of information that went into the creation of a tuple and thus its value. Employing this metric the system is then able to implement a *semantic shedding policy*, an algorithm that allows to reason about the quality of tuples and to choose the most valuable to the computation.

SIC quality metadata. In order to allow the system to self-inspect and reason about the achieved quality-of-service, this work proposes the introduction of a quality metric called *Source Information Content (SIC)*. The purpose of this metric is to provide a hint about the amount of information contained in a tuple and thus to its importance for the current query results. It is added to streams in the form metadata, whose value is calculated by the query operators and is inversely proportional to the occurrence of failure. Employing the SIC metric the system is able to reason about the individual value of tuples and can implement intelligent resource allocation policies based on it. Chapter 3 introduces a quality-centric data model for stream processing system based on the SIC metric. It presents the assumptions behind the design of the SIC quality metric and explains how this is calculated by the system.

Semantic shedder. When overload is considered the normal operational mode of the system and not a rare, transient condition, it becomes crucial to choose intelligently what input tuples to discard in the load-shedding phase. If the rate at which tuples are received in constantly higher than the rate at which they can be processed, a node is in the condition of continuously having to discard a certain portion of its input tuples. Augmenting streams with the SIC quality metric provides a valuable indicator about the amount of information captured during the creation of a tuple and thus about its importance to the query processing. When performing load-shedding, the system can leverage this knowledge to implement a semantic shedding policy, an algorithm to choose the set of tuples to discard not at random, but according to specific criteria. Chapter 5 presents the design and implementation of a fair shedding policy, which tries to allocate the system resources proportionally to the requirements of queries, so that the achieved quality-of-service is similar for all queries.

is might not be feasible or it could be preferable not to do so. Instead of trying to recover, it might be beneficial to accept the overload condition if the cost of running the processing is reduced while maintaining an acceptable quality of the results.

In a resource constraint deployment, with a large number of users and queries, the correct allocation of resources becomes critical. The problem is how to allocate resources fairly so that all queries achieve a similar quality-of-service, regardless of their nature. The system should be able to reason also about the relative performance of queries, so that, when it has to perform load-shedding, it can discard tuples in a fair fashion. Having a metric that captures the achieved quality-of-service for all queries, allows the system to implement a shedding policy that allocates resources fairly among queries. In this way the quality of the delivered results is equalised and queries receive the correct amount of processing resources according to their needs.

Is is possible to track the loss of information and report its entity to the user? Is there a correlation between the quality of the processing and the correctness of the delivered results? When the system have to choose what tuples to discard in an overload condition, is possible to design a fair resource allocation so that all queries experience the same quality-of-service? In a cloud deployment, where resources are rented on a per need basis, what is the trade-off between the cost of processing and the quality of the delivered results? These are the research questions that this thesis tries to address.

1.3 Quality-Aware Stream Processing

A stream processing system that operates under constant overload has to design with the ability to self-inspect and evaluate the quality-of-service it delivers to its users. When the amount of input data exceeds the processing capabilities of a node, the system has to discard a certain amount of tuples in input. By refusing to accept new tuples for a certain amount of time, the load on the system can be reduced and the the overload condition overcome. This approach, however, does not take into account the nature of the discarded tuples and results in a random shedding of input data.

When overload is not an exceptional condition, but is instead considered the normal operating condition, the system needs to be able to reason about the information carried by the individual tuples of each query. This allows the system to reason about what tuples it receives in input so that it can implement a semantic shedding policy. By augmenting tuples with a metadata value expressing their importance to the query calculation, it is possible to select tuples based on their information content and avoid resorting to a random shedding approach. This thesis proposes the introduction of a quality metric to quantify the amount of information captured by a tuple, tracking the degradation of information quality due to overload and failure.

in the availability of the user or too costly to purchase. The cost of purchasing and administering the needed processing infrastructure can be too high and outside the financial possibilities of a user. In order to obtain a the needed processing capacity without having to own the entire infrastructure, it is possible to opt either for a *federated resource pool* or for a *cloud deployment*.

In a federated resource pool, many parties contribute their resources to create a larger shared infrastructure that can be used by all according to their needs. In this scenario each institution is only responsible for the costs of purchasing and administering their processing units, while being able to use the complete set of resources when available. These kinds of deployments, where many parties pool together their resources are subject to a phenomenon similar to the *tragedy of the commons* [Har68], since every party tends to consume more resources than what it contributed, the amount of available resources is always scarce. Another problem with such a deployment is that, when resources are scarce, it is only possible to increase the provisioning of the local domain, while the other processing sites are under the control of third parties.

Another way to acquire a large amount of processing resources is to rent them on a per need basis. It is possible in fact to opt for a cloud deployment, renting the required processing nodes from a cloud provider only for the time needed. In this way the cost of purchasing and maintaining the infrastructure is outsourced and the user pays only the resources it needs for a limited amount of time. This allows the acquisition of a large processing capacity for a reasonable cost and only for the time of the queries. The amount of resources needed, however, can be very large and the rental cost may be out of the financial reach of a user. Even if the user opted for a cloud deployment, renting all the resources needed, perfect processing might not be feasible due to financial constraints.

The loss of information that determines the quality reduction of results is due to either the loss of tuples or their intentional discarding. Processing nodes can fail and some might become temporarily unreachable because of a network failure, thus causing a loss of a portion of the processed tuples. In this scenario the stream processing system should be able to track the loss of information and to quantify its impact on the quality of the results. The user should be provided with a quality metric that reports in real-time the achieved quality-of-service, so that it can deliberate over the goodness of the provided output.

Another common reason that leads to the reduction of the processed information is the deliberated shedding of a portion tuples. This happens when the rate at which tuples are delivered in input to a processing node is greater than the rate at which it is able to process them. The throughput of the node reaches a plateau and can not increase any longer. Therefore, the excess tuples delivered in input need to be discarded, this process takes the name of *load-shedding*. In order to overcome this overload condition it is possible to increase the amount of processing resources, but, as explained earlier, this

time. One area that has recently flourished in this field is environmental monitoring. Understanding how and why the climate is changing so rapidly is one of the top priority of many scientists and will be crucial to plan future policies. Sensor networks can also be employed to monitor in real-time weather phenomena that are potentially catastrophic, like hurricanes and tornadoes. The creation of large-scale weather sensing infrastructures will increase the possibility of identifying these phenomena before they become disruptive, thus allowing for an early response that will lead to a mitigation of their impact.

Social media. Social media represent a great source of user generated data, which offers new interesting mining possibilities. These data streams are populated with status updates, check-ins, photos, videos, etc., an ever increasing flux of everyday experiences that the users decide to share. The amount of available data is theoretically enormous and opens the doors to a new class of streaming applications. Dealing with the totality of the generated information might be over costly, while the analysis of a sample, capturing a certain percentage of the original stream, might be sufficient. Accepting the discarding of a portion of the input data allows a substantial reduction of the processing resources needed to run the queries and thus a reduction in the operational costs. What is important is that this reduction in quality of the generated results is quantified and reported to the user. This allows to establish a trade-off between the processing costs and the quality-of-services achieved by the query results.

In these application domains a result may be acceptable even if imperfect, as long as there is an indication about the entity of the quality degradation. The judgement about the quality of the delivered results should be left to the end user, while the system should provide constant feedback about the achieved quality-of-service.

1.2 Problem Statement

Sometimes the over provisioning of a stream processing system is difficult to achieve, like in the case of a federated resource pool, or might be too costly, like in a cloud deployment. When the amount of resources needed for perfect processing is not available, it is often possible to deliver meaningful results even processing only a subset of the input data. In many cases an imperfect result is better than no result at all.

The constant increase in data availability allows the computation of a richer variety of stream processing queries, but also renders the provisioning of the system resources difficult and costly. An extremely large amount of input data requires an amount of processing resources that might not be

system should reliably collect, filter and process stream data from potentially thousands of data sources on behalf of many users. Such systems should support a large number of queries, possibly running for extended periods of time. Data should be processed in a decentralised fashion, distributing the computation over several data centres.

In this context the notion of *sensor* is not only limited to a device measuring some physical quantities, but refers to a more extended range of devices. *Physical sensors* are devices taking measurements of some physical quantity at interval time, for instance a temperature sensor deployed in a sensor network, on the other hand a *software sensor* takes measurements about non-physical events, like social media events.

Figure 1.1 shows an ISSP at work. Sources, either physical or software, are located at different locations around the world (i.e. weather towers). Data streams are depicted as arrows. Several data centres, also located at different locations, are used to process these streams, in a decentralised fashion. Many users can concurrently issue queries, which the system tries to satisfy at the best of its possibilities, according to the available resources.

We can expect the rise of globally distributed ISSP systems in the future, in which several parties contribute processing resources to be shared by all users. These federated resource pools are going to join several clusters distributed across the globe, each administered by a local authority. Users will be able to deploy queries processing data coming from all over the world in real-time. Some systems started developing in this direction, like MaxStream [Bot+09], a federated stream processing infrastructure designed to support real-time business intelligence applications, and Cosm [CSM12], a system aiming at the construction a world-wide network of intelligent devices to build the Internet-of-Things.

1.1 Applications

These shared platforms can support the concurrent execution of a large number of queries, submitted by multiple users. The amount of available input data renders the provisioning of the processing resources difficult as it might be too costly to purchase or rent the necessary infrastructure. The variety of queries that a stream processing system can support is very broad and allows the support of a large number of applications. Among these, two interesting application domains that can be used as examples, deal with queries analysing *sensor data* and *social media* streams.

Sensor data. The advances in cheap micro-sensing technology is increasing the availability and resolution of sensor data. In the future the number of sensing devices will increase to a point where everything of material significance will be "sensor tagged" and will report its state and position in real-

Chapter 1

Introduction

Data stream processing systems (DSPSs) compute real-time queries over constantly changing streams of data [ScZ05]. A stream is a possibly infinite sequence of tuples, or timestamped entities [ABW06]. Differently than traditional databases, where queries are issued over stored data, in DSPSs queries are set first and results are generated as new data enters the system. This allows the generation of real-time updated results based on the constantly changing available data streams. Real-time in this context refers to the ability of delivering results *as they become available*. Stream queries are *continuous* [BW01], they process tuples as they are pushed into the system, delivering results as they are computed.

Internet-scale stream processing (ISSP) systems represent the next step in the evolution of stream processing [Pie08]. Similarly to how search engines make static web data useful to users, an ISSP

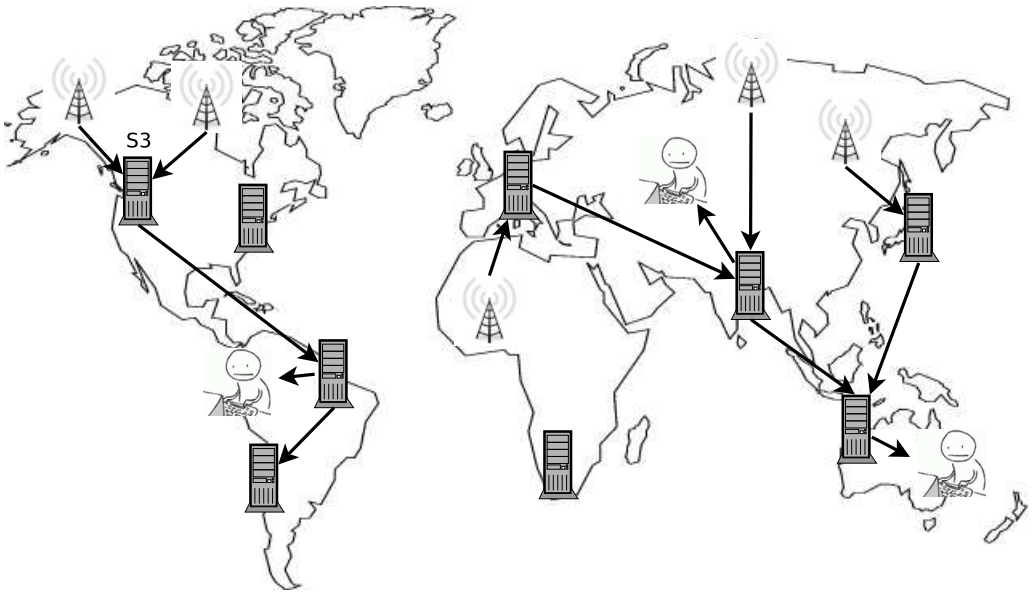

Figure 1.1: Graphical representation of an Internet-scale stream processing system at work.

Abstract

Data stream processing systems (DSPSs) compute real-time queries over constantly changing streams of data. A stream is a possibly infinite sequence of tuples, or timestamped data items. In contrast to traditional databases, in which queries are issued over stored data, results in DSPSs are generated continuously as new data enters the system.

The constant increase in data volume renders the provisioning of a DSPS difficult and costly. It may require computing resources that may not be available or too costly to purchase. Even if a user opts for a cloud deployment, thus renting all resources on demand, perfect processing may still not be possible due to the financial cost.

In the future we can expect the development of processing infrastructures in which different parties cooperate to the creation of large-scale federated resource pools. The correct allocation of resources on these federated pools is going to be difficult, as the management of the single clusters will be be under the control of the local authority, often leading to unbalanced resource allocations and the local occurrence of overload. For these reasons, *overload* should be considered a common operating condition for such DSPS and not an exception.

Overload can be considered a type of failure because the system is not able to fully carry out the required computation. A system operating under constant overload is thus subject to continuous failure. In this situation, the system needs to discard some of its input data, an operation called *load shedding*.

Many streaming applications are able to produce valuable results even after some failure has occurred during the processing. Examples of such applications are meso-scale weather prediction, tornadoes and hurricanes forecasting, and real-time social media monitoring. An approximate result may still be useful to the user, as long as it is delivered with a low latency and it contains some information about its quality. In many cases, an imperfect result is better than no result at all.

We propose a new stream processing model under overload. The system constantly estimates the impact of overload on the computation and reports to the user the achieved quality-of-service. We introduce a quality metric called *Source Information Content (SIC)*. This can be used by the user as an indicator for the achieved quality-of-service and by the system to implement intelligent shedding policies and to better allocate the system resources among users. When an overloaded system performs *load-shedding*, the choice of how much and what to discard is crucial for the correct functioning of the system. The SIC quality metric helps the system make more informed decisions when to shed data. It allows the implementation of a *fair shedding* policy, giving an equal quality-of-service to all users, without penalising certain kinds of queries.

We develop these ideas as part of a research prototype called DISSP, the Dependable Internet-Scale Stream Processing engine. With it we explore the issues related to overload management and fair resource allocation. We show that augmenting streams with the SIC metric allows the system to make better load-shedding decisions, leading to more accurate results for many queries. It also allows the user to reason about the amount of processing resources that are needed to run a given query, striking a balance between the quality of the delivered results and the cost of operating the system.