Imperial College of Science, Technology and Medicine
Department of Computing

# Quality-Aware Overload Management For Stream Processing

Marco Fiscato

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of the University of London and the Diploma of Imperial College

October 2012

# Declaration of Originality

I certify, as the author of this thesis, that I was the person primarily involved in the study designs, implementation, analysis and manuscript preparation. I declare that the work presented in the thesis is to the best of my knowledge and belief, original (except as acknowledged in the text) and that the work has not been previously submitted for a degree or diploma at any institution.

Marco Fiscato

# Contents

**Abstract** 5

**1 Introduction** 7

1.1 Applications . . . . . 8

1.2 Problem Statement . . . . . 9

1.3 Quality-Aware Stream Processing . . . . . 11

1.4 Contributions . . . . . 12

1.5 Thesis Outline . . . . . 13

**2 Background** 15

2.1 Stream-based applications . . . . . 16

2.2 Query Languages . . . . . 21

2.3 System Model . . . . . 27

2.4 Overload Reduction . . . . . 31

2.5 Failure . . . . . 35

2.6 Summary . . . . . 38

**3 Quality-Centric Data Model** 39

3.1 Model Definitions . . . . . 40

3.2 Model Assumptions . . . . . 45

3.3 Types of Queries . . . . . 54

3.4 Source Information Content . . . . . 57

3.5 Use Cases for Source Information Content . . . . . 61

3.6 Summary . . . 65

**4 DISSP Design 67**

4.1 Design Approach . . . 68

4.2 Implementing the Model . . . 69

4.3 DISSP Architecture . . . 72

4.4 Life Cycle of a Query . . . 82

4.5 Summary . . . 87

**5 Quality-Aware Load-Shedding 89**

5.1 Fairness in Load-Shedding . . . 90

5.2 Abstract Shedder Model . . . 91

5.3 Quality-Aware Fair Shedding . . . 93

5.4 Implementation of the DISSP Load-Shedder . . . 97

5.5 Summary . . . 98

**6 Evaluation 101**

6.1 Experimental Set-up . . . 102

6.2 SIC Values and Correctness . . . 104

6.3 Load-shedding Policies . . . 107

6.4 Trade-off between Quality of Processing and Resource Cost . . . 113

6.5 Summary . . . 114

**7 Conclusion 115**

# List of Figures

1.1 Graphical representation of an Internet-scale stream processing system. . . . . . . . . 8

2.1 Interaction among the three classes of operators in CQL. . . . . . . . . . . . . . . . . . . 23

2.2 This query outputs all tuples with a temperature reading $\geq 30$ in the last five minutes. 25

2.3 Temperature monitoring query example, expressed in the boxes-and-arrows model. . . 27

2.4 Comparison between a traditional Data Base Management System (DBMS) and a Data Stream Processing System (DSPS). On the left, queries are issued over stored data, while, on the right, data flows through continuous queries. . . . . . . . . . . . . . . . . . 28

2.5 State machine describing Borealis fault tolerance mechanism in Borealis. . . . . . . . 36

3.1 A simple tuple with the relational schema. . . . . . . . . . . . . . . . . . . . . . . . . . . 42

3.2 A black-box operator with two input and one output stream. . . . . . . . . . . . . . . 44

3.3 An unbalanced query tree, showing the different information content of tuples. . . . . 51

3.4 An example of a fan-in query. Input values from each source are first averaged over a certain time window, and then the maximum is selected. . . . . . . . . . . . . . . . . . 54

3.5 An example of fan-out query. It processes Twitter data by counting the top-rated messages on a certain topic. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

3.6 An example of a fan-out query for processing Twitter data. It counts the occurrence of positive and negative mentions of a certain keyword and calculates their ratio. . . . . 56

3.7 An example of fan-in query. Input values for each source are first averaged over a certain time window, and then the maximum value is selected. Source Information Content values are shown for each tuple. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

3.8 An example of a fan-out query processing Twitter data. The Source Information Content values are shown for each tuple. . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

3.9 An example of fan-out query processing Twitter data. It counts the occurrences of positive and negative mentions of a given keyword. The numbers shown on tuples are their individual SIC values. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

4.1 A generic operator with its internal structure shown. . . . . . . . . . . . . . . . . . . . 70

4.2 A system level overview of the components in DISSP prototype. . . . . . . . . . . . . 73

4.3 A high level view of a processing node, with its internal components. . . . . . . . . . . 76

4.4 Steps involved in the deployment of a query. . . . . . . . . . . . . . . . . . . . . . . . 83

4.5 Horizontal partitioning of an operator. . . . . . . . . . . . . . . . . . . . . . . . . . . 87

5.1 Abstract representation of a load-shedder, showing its internal conceptual model. . . . 91

5.2 Graphical representation of the time intervals $\delta$ and $t$ used to calculate the *average tuple cost* $C_T$ needed to predict the *future processing capacity* $N_{keep}$ of a node. . . . . . . . . 95

6.1 Correlation of SIC values with the query output quality for *average* queries. . . . . . . 105

6.2 Correlation of SIC values with the query output quality for *count* queries. . . . . . . . 106

6.3 Correlation of the SIC values with the query output quality for *top-5* queries. . . . . . 106

6.4 Correlation of the SIC values with the query output quality for *covariance* queries. . . 107

6.5 Comparison of fair and random load shedders (MEAN). . . . . . . . . . . . . . . . . . 109

6.6 Comparison of fair and random load shedders (IQR). . . . . . . . . . . . . . . . . . . . 110

6.7 Comparison of fair and random load shedders (Standard Deviation). . . . . . . . . . . 110

6.8 Comparison of fair and random load shedders (Q0.5-Q0.95). . . . . . . . . . . . . . . . 110

6.9 Fairness for an increasing number of nodes. . . . . . . . . . . . . . . . . . . . . . . . . 111

6.10 Fairness for an increasing number of queries. . . . . . . . . . . . . . . . . . . . . . . . 112

6.11 Trade-off between the reduction of resource cost per query and the reduction in quality of processing, as expressed by mean SIC values. . . . . . . . . . . . . . . . . . . . . . . 114

# List of Tables

3.1 Notation used in the model definitions. . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

6.1 Testbed configurations for experiments. . . . . . . . . . . . . . . . . . . . . . . . . . . . 102

6.2 Query workload for experiments. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103

6.3 Input source data for experiments. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103

6.4 Workload breakdown for experiments comparing the random and fair load-shedding partitions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

## Abstract

Data stream processing systems (DSPSs) compute real-time queries over continuously changing streams of data. A stream is a potentially infinite sequence of tuples, or timestamped data items. The constant increase in data volume renders the provisioning of a DSPS challenging, requiring computing resources that may not be available or too costly to purchase. Even if a user opts for a cloud deployment model, thus renting all resources on demand, acquiring sufficient resources 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 create federated resource pools. These kinds of deployments, in which many parties pool together their resources, are subject to a phenomenon similar to the *tragedy of the commons.* Since every party tends to consume more resources than what they contributed, the amount of available resources is always scarce.

For these reasons, *overload* should be considered a common operating condition for such DSPS and not an exception. In such situations, the system needs to discard some of its input data, which is an operation called *load-shedding.* 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.

Many streaming applications are able to produce useful results even after some data has been discarded 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 contains some information about its quality.

We propose a new model for federated stream processing under overload. The system constantly estimates the impact of overload on the computation and reports to the user the achieved processing quality. We introduce a quality metric called *Source Information Content (SIC).* This can be used by the user as an indicator for the achieved processing quality and by the system to implement intelligent shedding policies and to better allocate the system resources among users. The SIC quality metric allows the implementation of a *fair shedding* policy, giving an equal processing quality to all users without penalising individual queries.

We experimentally show that augmenting streams with the SIC metric allows the system to make better load-shedding decisions, leading to more accurate results for many types of 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 resource cost.

# Chapter 1

# Introduction

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

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 system reliably collects, filters and processes 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 centers.

In this context, the notion of *sensor* is not only limited to a device measuring some physical quantity but also refers to a more extended range of devices. *Physical sensors* are devices taking measurements of some physical quantity at interval time, e.g. a temperature sensor deployed in a sensor network. On the other hand, *software sensors* take measurements of non-physical events, such as social media events.

Figure 1.1 shows the deployment of an ISSP system. Sources, either physical or software sensors, are located at different locations around the world (i.e. illustrated as 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 ISSP system tries to satisfy using the available processing resources.

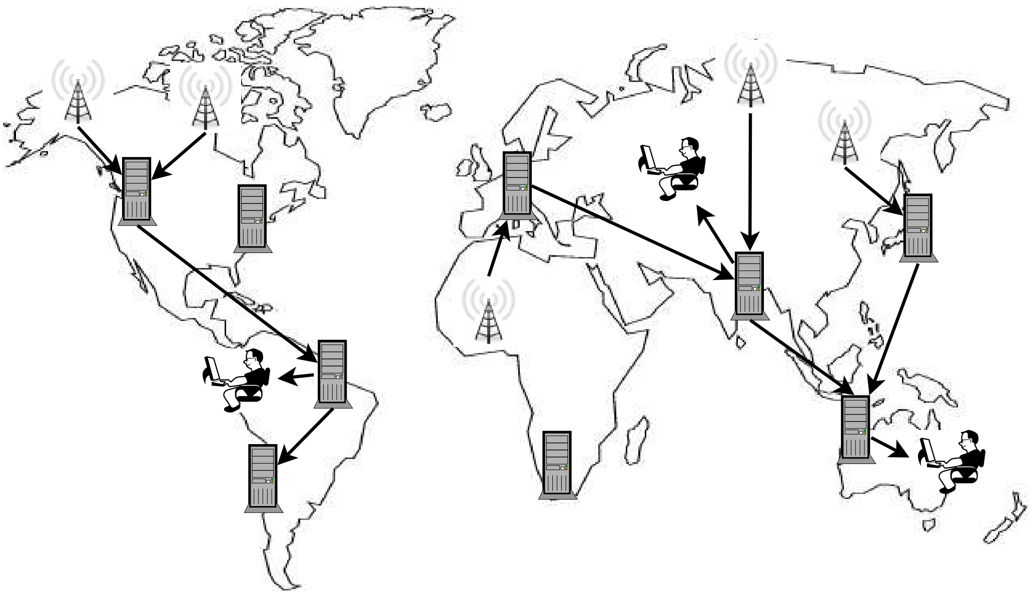

Figure 1.1: Graphical representation of an Internet-scale stream processing system.

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. Such federated resource pools join clusters distributed across the globe, each administered by a local authority. Users deploy queries processing data from sources located all over the world in real-time. MaxStream [Bot+09] is a federated stream processing infrastructure designed to support real-time business intelligence applications, and Cosm [CSM12] is a system aiming to construct a world-wide network of intelligent devices for the Internet-of-Things. The amount of available input data renders the provisioning of processing resources challenging because it may be too costly to purchase or rent the necessary infrastructure. A world-wide federation of resources allows the creation of large infrastructures, in which users take advantage of each others processing capabilities when needed.

## 1.1 Applications

A federated stream processing system can support the concurrent execution of a large number of queries, submitted by multiple users. The variety of queries that a stream processing system must support is broad and can be the basis for a large number of applications. Among these, two interesting application domains, which will be used as examples, deal with queries analysing *sensor data* and *social media* streams.

**Sensor data.** Advances in cheap micro-sensing technology led to an increase in the availability and resolution of sensor data. In the future, the number of sensing devices will reach a point at which everything of material significance is "sensor tagged", reporting its state and position in real-time.

One area that has recently flourished in this field is environmental monitoring. Understanding how and why the climate is changing is a scientific priority and will be crucial to plan future policies. Sensor networks can also be employed to monitor weather phenomena that are potentially catastrophic, such as hurricanes and tornadoes. The creation of large-scale weather sensing infrastructures allows to identify these phenomena before they become disruptive, thus enabling an early response to mitigate their impact.

**Social media.** Social media represent a large source of user generated data, which offers new interesting possibilities for data mining. Data streams are populated with status updates, location check-ins, photos, videos, etc. This results in an ever increasing influx of everyday experiences that users decide to share. The amount of available data is vast and opens the doors to a new class of streaming applications. Dealing with the totality of the generated data may be too costly, and the analysis of samples, capturing a certain percentage of the original stream, may be sufficient. Discarding a fraction of the input data allows a substantial reduction of the processing resources needed to execute queries, thus reducing operational costs. What is important is that this reduction in quality of the generated results is quantified and reported to the user. This leads to a trade-off between the processing cost and the quality of processing achieved by the system.

In this application domain, a result may be acceptable even if imperfect, as long as there is an indication of the degree of quality degradation. The judgement about the quality of the delivered results should be left to the end user, while the system should provide continuous feedback about the achieved quality of processing.

## 1.2 Problem Statement

In certain domains, the over-provisioning of a stream processing system is not possible, such as in the case of a federated resource pool, or it may be too costly, such as in a cloud deployment. When the amount of resources needed for perfect stream processing is not available, it may be possible to deliver meaningful results by 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 more difficult and costly. A large amount of input data requires processing resources that may not be available to the user or too costly to purchase. In order to obtain the necessary 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, which can be used by all according to their needs. In such a scenario, each institution is only responsible for the costs of purchasing and administering their own processing resources, while being able to use the global set of resources. These kinds of deployments, in which many parties pool together their resources, are subject to a phenomenon similar to the *tragedy of the commons* [Har68]. If every party tends to consume more resources than what they contributed, the amount of available resources becomes scarce.

Another way to acquire a large amount of processing resources is to rent them on demand [Arm+10]. In a cloud deployment model, the required processing nodes are rented from a cloud provider for the time needed. In this way, the cost of purchasing and maintaining the infrastructure is outsourced, and the user pays only for the incurred resource usage. This allows the acquisition of a large processing capacity for a reasonable cost. The amount of resources needed, however, can be large and the rental cost may become too expensive. Therefore, even if a user opts for a cloud deployment model, perfect processing may not be feasible due to these 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 may become temporarily unavailable due to a network failure, thus causing a loss of a portion of the processed tuples. In these cases, the stream processing system should track the loss of information and quantify its impact on the quality of the results. Therefore, the user should be provided with a quality metric that reports the achieved quality of processing so that they can assess the "goodness" of the provided output.

A common reason that leads to the reduction of processing quality is the deliberated shedding of a fraction of tuples. This happens when the rate at which tuples are received by a processing node is greater than the rate at which it is able to process them. The throughput of the node saturates and cannot increase any more. The excess tuples need to be discarded, which is referred to as *load-shedding* [Tat+03]. In order to overcome an overload condition, it is possible to increase the amount of processing resources but, as explained above, this may not be feasible. Instead, it may be acceptable to sustain the overload condition if an acceptable quality of the results can be maintained.

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

This thesis attempts to address a set of research questions. Is it possible to track the loss of information and report it to the user? In a DSPS is there a correlation between the quality of the processing and the correctness of the delivered results? When the DSPS has to choose which tuples to discard in an overload condition, is it possible to design a fair resource allocation so that all queries experience the same quality of processing? In a cloud deployment, in which resources are rented according to demand, what is the trade-off between the cost of processing and the quality of the delivered results?

## 1.3 Quality-Aware Stream Processing

A federated stream processing system that operates under constant overload has to be designed with the ability to evaluate its achieved quality of processing. When the amount of input data exceeds the processing capabilities of a node, the system has to discard a given amount of input tuples. By discarding new tuples for a certain amount of time, the load on the system can be reduced and the overload condition overcome. Such an approach, however, does not take into account the nature of the discarded tuples.

When overload is not an exceptional condition but is considered normal, the system needs to be able to reason about the information carried by the individual tuples of each query. This allows the system to implement a *semantic* shedding policy, in which the tuples to be discarded are chosen according to a given set of rules. By augmenting tuples with a metadata value expressing their importance to the query computation, it is possible to select tuples based on their information content and avoid random data loss. This thesis proposes a new quality metric to quantify the amount of information captured by a data tuple, thus tracking the degradation of processing quality due to overload.

The choice of what tuples should be discarded and what should be kept affects the quality of the results. A federated stream processing system has to decide which tuples are more important for the processing and prioritise them over other tuples of lower quality. Not all tuples contain the same amount of information and thus have a different value to the query. For example, a tuple containing a value aggregated over a large number of sensors is likely to be more valuable than a tuple containing a single sensor reading. The system should be able to reason about the information content of tuples in order to identify the most valuable tuples to be spared from shedding. To do so, we introduce a *quality metric* that captures the quantity of information that went into the creation of a tuple and thus its value. Using this metric, the system is able to implement a *semantic shedding policy*, which is based on the quality of tuples and retains the most valuable tuples for the computation.

## 1.4 Contributions

**SIC quality metadata metric.** In order to allow the system to reason about the achieved quality of processing, this thesis proposes a quality metric called *Source Information Content (SIC)*. It provides an estimate of the amount of information contained in a tuple and thus its importance for the query results. It is added to streams in the form metadata and its value is calculated by the query operators. The system uses the SIC metric to implement intelligent resource allocation policies. Chapter 3 introduces the quality-centric data model for federated stream processing system based on the SIC metric. It presents the assumptions behind the SIC metric and explains how it is calculated.

**Semantic quality-aware load shedding.** When an overload condition is considered normal, it becomes important to choose intelligently which input tuples to discard during load-shedding. If the rate at which tuples are received is higher than the rate at which they can be processed, a node has to discard a certain fraction of its input tuples. Augmenting streams with the SIC quality metric provides an indicator of the amount of information captured during the creation of a tuple, thus quantifying its importance to query processing. When performing load-shedding, the system can leverage this knowledge to realise a semantic load-shedding policy (i.e. an algorithm that chooses the set of tuples to discard according to specific criteria). Chapter 5 presents the design and implementation of a fair load-shedding policy, which tries to allocate system resources proportionally to the requirements of queries so that the achieved quality of processing is similar for all queries.

**Prototype system design.** The concepts introduced by the quality-aware data model have been realised in the DISSP stream processing prototype system. This system is designed to make use of the SIC quality metric to operate under overload while tracking the quality of processing achieved by each query. The system implements a semantic load-shedding policy that provides a fair amount of resources to all queries. The prototype is described in Chapter 4, which shows how the basic concepts of the data model are realised. It presents the system-wide components interacting at a high level as well as the low level architecture of each processing node.

**Experimental evaluation.** We report results on the correlation between the degradation of SIC values and the correctness of query results. Even though the SIC quality metric is not designed as an accuracy metric, for many aggregate queries, there is a good correlation between SIC values and correctness. Having introduced semantic load-shedding policies, we investigate the advantages of employing the SIC quality metric for load-shedding. In particular, we compare a random shedder and the fair shedding algorithm introduced in Chapter 5. We find that the fair shedding policy leads to better resource allocation among queries for many classes of queries.

Finally, we explore the correlation between SIC degradation and cost. In this scenario, the user voluntarily accepts a reduction of result quality in order to reduce the amount of required resources and thus the cost of a cloud deployment. Our results show that there is a trade-off between a reduced SIC value of results and the costs of renting cloud resources.

## 1.5 Thesis Outline

The rest of this thesis is organised as follows. Chapter 2 presents the background material for this research work. It describes some applications of stream processing, focusing on those that can tolerate imperfect processing. It introduces different streaming data models and the basic concepts of stream processing. It presents different approaches for overcoming overload, focusing on load-shedding strategies.

Chapter 3 introduces our quality-centric data model, in which streams are augmented with the SIC metadata metric. It states the assumptions used to derive the quality metric and presents an overview of the different categories of queries. The chapter ends with the formalisation of the model and some examples of its application.

Chapter 4 presents the design of the DISSP stream processing system, a research prototype that implements the proposed quality-centric data model. It explains how the abstract concepts from the the model can be implemented as a part of a real-world system. We describe the design of the prototype and show how the SIC metric is calculated and used.

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

Chapter 6 presents a set of experiments designed to evaluate properties of the SIC quality metric. It investigates its correlation with the accuracy of results and compares a random shedding policy with our proposed fair shedding algorithm. Finally, it looks at the trade-off between cost and SIC value degradation.

Chapter 7 concludes the thesis and includes a discussion of future research.

# 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 that overcome an overload condition and keep delivering results in a timely fashion. The system should gracefully degrade the correctness of the computed results, following the principle that an approximate result is better than no result at all.

Many classes of applications do not require 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 beyond what is available to the user. In many cases, it is possible to trade the correctness of results against a reduction of the cost for the processing infrastructure.

Overload is a particular kind of failure because 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 load-shedding, (i.e. choosing the right set of tuples to be discarded, can have a great impact on the quality of the computed result). In addition, 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.

## 2.1 Stream-based applications

Traditionally, stream processing has been used in application domains with low tolerance of failure. Such applications can only produce meaningful results when the processing is carried out without any loss of data or approximation. Financial algorithmic trading [Str08] is a prime example. Investment banks, trading on the stock market, need to process a large volume of financial events in real-time. Complex algorithms are used to capture the current market situation and to 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 make profit. In this scope, the necessity of processing all the available data is evident because a wrong choice based on approximate data could lead to financial loss.

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

We will describe two broad classes of applications that can use an approximate result approach. The first class are *global sensing infrastructures* [Kan+07; AHS06; Gib+03], which are concerned mostly with sensor data, developing a worldwide infrastructure in which different classes of sensors can be connected and accessed through a common interface. They allow users to process data streams generated by different sensor networks.

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

### Environmental Monitoring

The first domain of applications that we will take into consideration are related to large-scale scientific sensing, in particular we will describe projects dealing with large-scale environmental monitoring. In recent years, the availability of cheap micro-sensing technology has unleashed 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 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. Many large-scale sensing projects have been launched to understand better 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 an increased resolution, and integrating diverse measurements and observations. Thousands of geophysical instruments measure the motion of the 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 studies 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 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 a continental-scale research platform for discovering and understanding the impact 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 time period is crucial to improving ecological forecast models and will greatly help understand the effects of human interference on climate. 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 because a continuous flow of measurements are generated by a large number of sensors. Stream processing offers an intuitive and flexible paradigm to harness such a high volume of data. It also gives the possibility of obtaining a real-time picture of the measured phenomena, enabling quick reaction to possibly disruptive events. The core components of these projects are sensing stations, which are geographically distributed and frequently subject to harsh conditions. In such scenarios, the failure of sensing devices is common and often their replacement is problematic. Long running continuous queries need to be able to handle some missing data, adapting to the new conditions and reporting the achieved quality of processing. To collect and process all data at all times remains 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 a large number of deaths and a great deal of damage to infrastructures every year. Being able to understand and predict when these hazardous weather conditions will occur can 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]. Its goal is to enhance the current infrastructure of long-range weather observing radars with a large number of small solid-state radars in order to increase the sampling resolution throughout the entire troposphere. Although current long-range radar technology allows the coverage of large areas with a relatively small number of devices, these are not able to measure correctly the lowest part of the atmosphere in areas far away from the radar. This is due to the Earth's curvature and terrain-induced blockage. This new sensing paradigm, based on a large number of smaller devices, is referred to as *Distributed Collaborative Adaptive Sensing (DCAS)*. Distributed refers to the use of numerous small and inexpensive radars, spread densely enough to measure the area fully, even at lower altitudes at which the traditional approach fails. Collaborative refers to the coordination of multiple devices that cover 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. Similar to CASA is the *Linked Environments for Atmospheric Discovery (LEAD)* [Li+08]. It gives scientists the tools with which they can automatically spawn weather forecast models in response to real-time weather events in 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 system. Weather information is available to users in real-time, not being restricted to pre-generated data, which greatly expands analysis capabilities.

The integration of these two systems offers a useful infrastructure to atmospheric scientists [Pla+06]. First of all, it allows meteorologists to interact directly with data from the instruments as well as control the instruments themselves. Unlike traditional forecasting approaches, it allows 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 feeding their data into a forecasting model, is changing the way meso-scale weather events are detected and will help 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 a large growth over 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 based on social links. Users of these networks often establish hundreds or 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 continuous 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 discovery and to reach new potential customers [WDF12; BWH12]. The possibility to target specific segments of the online population with customised advertisements allows companies to increase their return on investment and users to avoid unwanted communication. Furthermore, by analysing Twitter streams, it is possible to predict in real-time the evolution of numerous phenomena, such as the spreading of diseases [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 user-generated content through a social network is referred to as a *social cascade*. The next section describes social cascades and explains why their prediction can be useful for the correct allocation of resources in content distribution networks.

**Social Cascades.** One interesting application of social media real-time analysis is the possibility of predicting social cascades. A *social cascade* is the phenomenon generated by the repeated sharing of a certain content over an Online Social Network (OSN). 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 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 epidemic [SYC09]. Initial studies about this phenomenon date back to the 1950s, with the theory of Diffusion of Innovation [Rog03]. It is only now though, with the wealth of information shared through OSNs, that is possible to study social cascading in much more detail.

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 users *Alice* and Bob.

**Social Cascading in Twitter.** The diffusion of links through Twitter can be used to understand better what a social cascading tree looks like. Twitter is a micro-blogging website on which 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 characterise cascades further into two 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) Users *Alice* and *Bob* shared content *c*, (2) User *Alice* posted *c* before user *Bob*, and (3) User *Bob* is a follower of user *Alice*.

Instead, RT-cascades do not only take into account direct connections but also the possibility of crediting certain content to a user even without being a direct follower. If user *Bob* wants to give credit to user *Alice* for certain content, he prepends his new tweet with the conventional *RT @Alice* tag followed by the original tweet. In this way, *Bob* gives direct credit to *Alice* for the content of the tweet even without being a direct follower. This phenomenon is 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 received 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*, and (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 certain content will become popular. Many content providers rely on Content Delivery Networks (CDNs) to distribute their content across multiple geographic locations in order to enhance the availability of their content to their users. The choice of *what* content to replicate, *where* and for *how long* is crucial to the reduction of the costs associated with the use of a CDN. Being able to predict the popularity of certain content is needed for the correct provisioning of resources. For example, 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 (i.e. the 90% are the tail) account for 40% [SYC09].

An attempt to characterise the spread of social cascades focuses on improving the caching of content, exploiting the geographic information from the cascades [Sce+11]. This approach leverages the observation that a social cascade tends to propagate in a geographically limited area, because social connections typically reflect real life relationships. It is also likely that a video spoken in a particular language would tend to propagate within the national borders of a country, in which that language is native. Based on an analysis of 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 hops in social cascades involve users that are, on average, less than 1,000 km away from each other.

## 2.2 Query Languages

In a DSPS, data is continuously 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 is referred to as *window*. A query language for data streams must be capable of operating over windows of data, should be extensible in order to support custom user operators, and simple enough to enable the easy specification of complex queries.

Three query paradigms have been proposed for streaming data [GO03]. The first and most adopted is the *relational streaming model*. It comes from the DBMS world and is usually implemented as an SQL-like language. It provides the syntax to express windows over data streams, similarly to the extensions introduced in SQL-99. A query describes the results in a declarative way, giving the system the flexibility of selecting an optimal evaluation procedure for producing 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 streaming 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 represent the data streams flowing through them, thus taking the name of *boxes-and-arrows* model. One drawback of this model is the difficulty of expressing complex query plans because the diagram describing a query can become 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 streaming paradigm*. In this approach, the query elements are modeled in a hierarchical way, taking inspiration from the object-oriented programming world. In Cougar [BGS01], for example, sources are modeled as ADTs (Abstract Data Types), exposing an interface consisting of a sensor's signal processing methods. A discussion of query languages and practical issues on building a stream processing system can be found in [Tur+10].

The next sections present more details on the two main models for expressing queries in a stream processing system: the *continuous query language* and the *boxes-and-arrows* representation.

### CQL: Continuous Query Language

The Continuous Query Language [ABW06] is a declarative language for expressing continuous queries over streams of data. It has been introduced as part of 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 an 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 us consider a *temperature monitoring* query. The purpose of such a query is to detect if the temperature in any room of a building rises above a certain threshold. If a room becomes too hot, the system detects it and reacts by switching on the local air conditioning unit. In this scenario, all rooms are equipped with thermal sensors. These are connected to a stream processing system, which is able to control the air conditioning unit. 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 (e.g. $T >= 30$). When the system detects that a room is too hot, it switches on the air conditioning unit in 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 potentially 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 can be zero, one or more elements with a given timestamp in a stream. The number of elements with the same timestamp in a stream is finite but unbounded. In CQL the data part of a stream element is referred to as a *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 scenario there would be only one base stream with a schema: ROOMTMPSTR(ROOMID, ROOMTMP). Attribute ROOMID identifies the room where the sensor is 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 (multiset) of tuples belonging to the schema of $R$.*

A relation $R$ defines an 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.

<u>Example</u>: In our scenario, a relation is created from the base stream of temperature readings through a time-window operator. It is a snapshot of all readings received within the last minute. The concept of windows over streams will be explained in the next paragraph.

### Operator Classes

CQL employs three classes of operators to process stream data. First is the *Stream-to-Relation* operator, which creates a snapshot of a stream. After a 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 create streams from newly computed relations. Figure 2.1 shows the interaction between 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 a *sliding window* over a stream. This contains, at any point in time, a historical snapshot of a finite portion of the stream.

Three kinds of window operators exist in CQL: time-based, tuple-based and partitioned. A *time-based* window contains all the tuples in the stream that have timestamps within the specified boundaries. For example, it could hold all the tuples that have arrived in the past minute. In a *tuple-based* window, instead, the number of tuples is specified and fixed so that it only contains the last $N$ tuples of the stream. A *partitioned* window is a special kind of the tuple-based window. It allows the user to specify a set of attributes as a parameter, splits the original stream based on these arguments, similarly to the SQL GroupBy statement, 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 that it contains but it does not discard them all. The sliding factor determines a strategy to hold some tuples in the window so

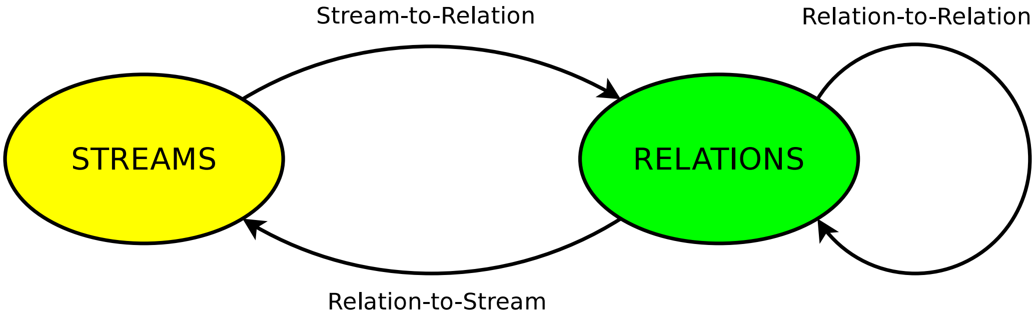

Figure 2.1: Interaction among the three classes of operators in CQL.

that they can be used in the next iteration. A tuple-based window, for example, can be set to trigger every 5 tuples with a slide of 1. Every time that it reaches 5 tuples, it outputs all of them but retains the 4 most recent tuples for the next iteration. A time-based window can be set to trigger every five minutes with a sliding factor of 1 minute. At every iteration, it outputs all the tuples that have arrived in the past 5 minutes, retaining all tuples from the past 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 canonical SQL operators. In this category, we find, for example, SELECT, PROJECT and other familiar operators. A stream is converted into a relation by a window operator, then processed by a relation-to-relation operator and finally output by a relation-to-stream operator.

EXAMPLE: Consider the following query for our temperature monitoring scenario:

SELECT *
FROM ROOMTMPSTR[RANGE 5 MIN]
WHERE TMP > 30

This query is composed of 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 time, the set of all temperature readings received by the system in the past 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:

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

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 $R(\tau - 1) - R(\tau)$.

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

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 $\tau$.

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

EXAMPLE: Figure 2.2 illustrates a simple CQL query implementing our temperature monitoring scenario. It signals all rooms, in which a temperature above 30 degrees Celsius is detected in the past 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 is applied to it to obtain a snapshot that contains only fresh measurements generated in the past five minutes. Once this relation has been created, a conditional Select is used to filter 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 from overheated rooms, which are then sent to control the air conditioning subsystem.

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

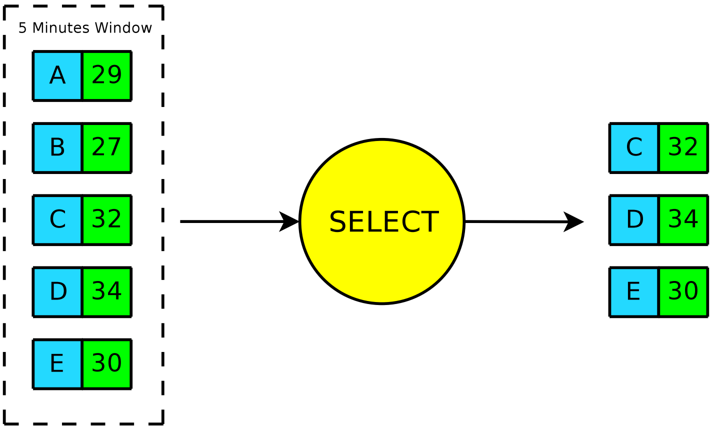

Figure 2.2: This query outputs all tuples with a temperature reading $\geq 30$ in the last five minutes.

### Boxes-and-Arrows Model

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

**Stateless operators.** Steteless operators do not need to store any information about previous tuples in order to execute. *Map* is an example of such an operator. It transforms some attributes of a tuple by applying a predicate. If we consider a tuple carrying a temperature reading, for example, a Map operator can be used to convert the temperature field from degrees Fahrenheit to Celsius.

In the same category, there is also a *Filter* operator. It can be seen as the equivalent of a *Select* in the relational model. It applies predicates to each input tuple and includes them in an output stream if the predicate evaluates to *true*. A Filter operator can have multiple output streams, depending on the number of specified predicates.

*Union* is a simple operator that produces a single output stream from many input streams with the same schema. The operator itself does not modify the tuples but it simply merges streams. For example, it is used to provide a single stream as an input to other operators, such as an Aggregate or a Filter.

**Stateful operators.** Stateful operators maintain internal state, which is determined by the tuples processed previously. In the boxes-and-arrows model, there is no explicit concept of a *window* operator, unlike in CQL. Instead, some operators are designed to support windowing on their input streams. The semantics for expressing these windows are rich, and it is possible to state a wide range of windows, such as count-based, time-based and partitioned. A sliding parameter can be used to specify the updating policy of the window.

*Aggregate* is a typical example of such an operator. It computes an aggregate function, such as 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 is usually preceded by a Union operator.

*Join* is another classical stateful operator with two input streams and two windows. Join pairs tuples from the two input windows that satisfy a given predicate, resembling its relational counterpart.

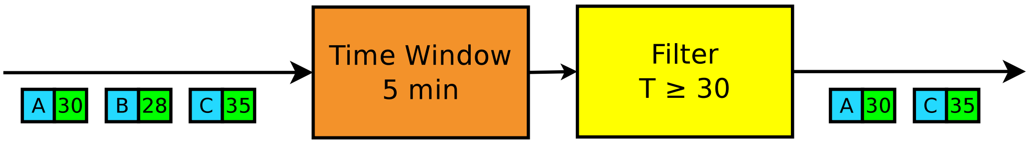

Figure 2.3: Temperature monitoring query example, expressed in the boxes-and-arrows model.

A special kind of stateful operators are synchronisation operators. This class includes *Lock, Unlock* and *WaitFor*. These are used to buffer tuples temporarily until a certain condition is reached.

**Load-shedding operators.** Load-shedding operators are used to overcome overload conditions. They reduce the load of an operator by discarding a portion of its input. This category includes RandomDrop and WindowDrop. *RandomDrop* randomly discards a certain fraction of a stream. Every time the operator is scheduled, it discards a number of tuples on its input window at random according to specified dropping parameters. *WindowDrop* allows for a more sophisticated specification of the windowing parameters. Once a window has been computed, it choose probabilistically if it should be kept or discarded 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 case, the query plan is composed only of a *Filter* operator. It receives the stream of readings as input and outputs all tuples satisfying the predicate T $\geq$ 30.

## 2.3 System Model

Data Stream Processing Systems (DSPS) have been developed to process large volumes of stream data, generated in real-time with new values constantly being introduced by the system. Using a traditional approach of first storing the data and then executing the queries is not appropriate. The first reason is the amount of available data [Tur+10; ScZ05], which is typically high and potentially too large to be stored entirely by the system. Another reason not to store all the incoming data is related to its nature: often only a subset of the data streams are relevant to the application. Moreover, stream data is transient, with new values constantly updating old ones and rendering them irrelevant. Finally, data stream processing also has strict latency constraints and processing data streams with low latency may be important. Storing the data before processing introduces an unacceptable delay due to the latency cost of storing and retrieving data on disk.

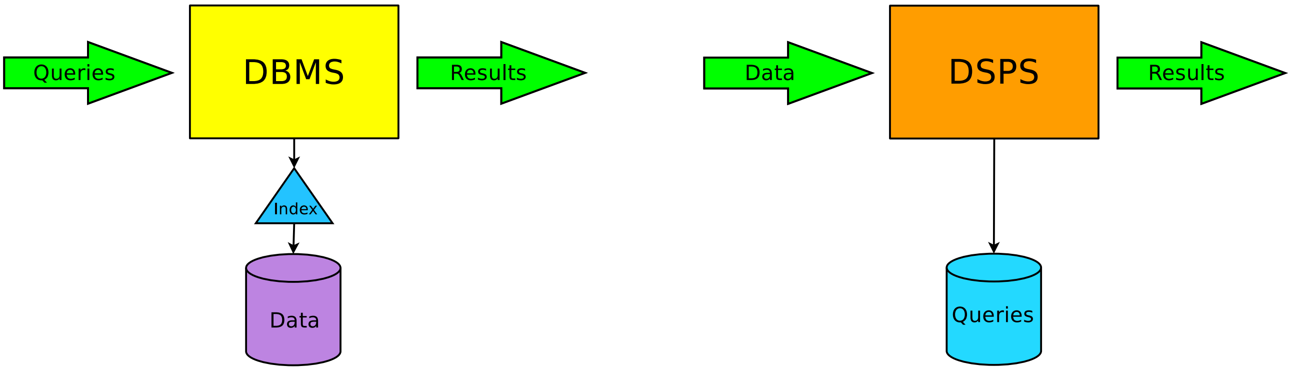

Figure 2.4: Comparison between a traditional Data Base Management System (DBMS) and a Data Stream Processing System (DSPS). On the left, queries are issued over stored data, while, on the right, data flows through continuous queries.

Figure 2.4 shows the two different architectures of a traditional DBMS (left image) and a DSPS (right image). In the 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 was issued. The picture on the right, instead, depicts a DSPS. Queries are issued over a continuously changing stream of data. Once the data enters the system, it is matched against the registered queries to produce the results. Results are generated from the data received by the system starting *from the time* when the query was first registered.

### Centralised Architectures

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 uses the relational Continuous Query Language (CQL) [ABW06] and supports SQL-like queries over continuous data streams (see Section 2.2). Once a query is registered in the system, a corresponding *query plan* is derived 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 consists of these basic elements. In contrast, Aurora [Car+02], introduced the boxes-and-arrows model for queries. Operators are regarded as boxes, with an input, an output and a specific processing semantic. They are linked together by arrows representing the stream of tuples flowing between them.

### Distributed Architectures

The natural step in the evolution of DSPSs is distribution. Centralised systems are inherently subject to scalability issues. Distributing the processing over a number of nodes is the logical approach to overcome this issue. Distribution also allows for better dependability because operators can be replicated at different locations, thus increasing availability. Fault-tolerance can also be achieved better by employing distribution because different replicas of operators can be deployed, which achieves resilience to failure. The overall performance of the system is also increased by this approach because 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 centre, taking advantage of the high bandwidth and low latency typical of these environments, or spread out over wide-area networks to push scalability even further.

Distribution presents several advantages but it comes at a cost. Many issues arise when the system is distributed: the operational complexity of 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 are not easy tasks. Running replicas can help increase the dependability of the system but their correct management represents a key issue.

Building on the experience of Aurora, a centralised DSPS, other systems have been developed to explore the possibilities offered by distribution. Aurora* [Che+03] is an evolution of the Aurora system that provides interconnection among multiple Aurora instances running in a cluster environment. A similar example is Medusa [Sbz+03], which expands the boundaries of distribution outside a single cluster onto different processing sites administered by different authorities. Borealis [Aba+05] realises these experiences into a complete distributed stream processing system. Borealis has been used to explore many research questions related to resource allocation, load-shedding and replica management. In the following sections, we cover these topics in more detail.

### Operator Placement and Load Balancing

The optimal distribution of streaming operators over a number of machines is key to maximise the performance of a distributed stream processing system. Once a query plan has been divided into clusters, these need to be mapped to available physical computing sites. A placement algorithm assigns operators to processing nodes while satisfying a set of constraints and attempts to optimise some objective function. Finding the optimal assignment among the total possible assignments is an NP-hard problem and thus computationally intractable [SMW05]. Despite this, many strategies have been devised for efficient operator placement and, depending on the assumptions and requirements of the system, different approaches are 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 movement of operators at run time. During the execution of a query, the amount of data carried by a stream can greatly vary. This may result in computational overload at some nodes. To recover from this overload condition, the system can decide to migrate certain operators to a better location, attempting to balance the load among the available resources. Many operator placement algorithms recognise the need for 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 centre, the topology management can be performed by a centralised placement controller. In such environments, the controller can be aware of the current state of 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 large though, such an approach suffers from low scalability. Abadi et al. [Aba+05] describe an approach which assumes a fairly constant workload and a heavy cost for operator migration. Xing et al. [XZH05] consider the workload to be highly unpredictable, thus requiring load balancing at runtime. They also acknowledge the importance of initial placement and assume 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 distributed over wide-area networks, a central coordinator becomes ineffective and a decentralised approach to the problem is more appropriate. Such algorithms make decisions based on local workload and resource information. Pietzuch et al. [Pie+06] propose a distributed algorithm based on spring relaxation, to minimise a global metric called network usage based on both bandwidth and latency. Another approach has been taken by Amini et al. [Ami+06]. In this case, an algorithm maximises the 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 through minimisation of the communication cost among processing nodes. It then adapts to the changes in the data and workload within the system. Ying et al. [Yin+09] propose to 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 means of pre-negotiated load-exchange contracts between 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 by orders of magnitude, and the resource provisioning of the system thus quickly becomes inadequate. Scaling the processing resources is not always feasible or cost effective. In order to overcome an overload condition, it is possible to employ either *load-shedding* (i.e. the deliberate discarding of a fraction of the input data), or *approximate operators*, which produce a semantically similar output but with a lower CPU or memory footprint. Next we describe each of these two approaches in turn.

### Load-Shedding

The discarding of a certain amount of input data is referred to as *load-shedding* [Tat+03]. The need to shed load comes from the inability of the system to process all the incoming data 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 awaiting to be processed grow in size, leading to an increased latency of the result tuples and eventually to an exhaustion of buffer memory. The most important choices that a system faces when overloaded are about (a) the amount of tuples to discard, (b) where to discard tuples in order to maximise the impact of the shedding and (c) what tuples to discard.

### Overload Detection

In order to detect an overload condition, the system has to continuously monitor its input rate and the size of its internal buffers. This periodic self-inspection has to be performed frequently enough so that the system can promptly react to a sudden variation of input rates, while avoiding an excessive overhead from the monitoring process.

Aurora [Tat+03] evaluates load based on a calculation that takes into account the processing cost of operators and their selectivity. It assumes a query network $N$, a set of input streams $I$ with certain data arrival rates and a processing capacity $C$ for the system that executes $N$. Let $N(I)$ denote the load as a fraction of the total capacity $C$ that network $N$ exerts on the system. An overload condition is detected when $Load(N(I)) > H \times C$, where $H$ is constant called the *headroom factor*. The total load calculation is a summation of the individual load of all network inputs. Each input has an associated *load coefficient*, which represents the number of CPU cycles required to transfer a single tuple across the entire local operator graph. By observing the input rates, it is possible to calculate if the current load leads to an overload condition and trigger the load-shedding mechanism when needed.

### Load-Shedding Location

After an overload condition has been detected, the system has to choose the location, at which it is best to discard the input data. In general, it is better to shed tuples before they are processed so that no CPU cycles are wasted processing data that will be discarded. For some classes of applications though, it may be better to perform the load-shedding at different locations, as explained below.

Babcock et al. [BDM04] propose 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 load-shedding 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 shared streams among queries, the authors provide an algorithm that chooses the location and the sampling rate of the load-shedding operators in an optimal way.

Tatbul et al. [TZ06] show that arbitrary tuple-based load-shedding can cause inconsistencies when windowed aggregation is used. They propose a new operator called *WinDrop* that discards entire windows. 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 because 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, either because all tuples are still processed and aggregate values are computed just to be discarded. Following this reasoning, the WinDrop operator is designed to operate on the window of tuples to be sent to the aggregate operator and discards or forwards this set of tuples as a whole.

### 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 a selection step in the load-shedding algorithm. In contrast, 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. Such approaches have been used, for example, in the context of aggregate queries and streams presenting irregular frequency patterns.

Mozafari et al. [MZ10] propose an algorithm for optimal load-shedding for aggregate queries when queries have different processing costs, different priorities and, importantly, the users provide their own error functions. Load-shedding is treated as an optimisation problem, in which users state their needs in term of quality-of-service requirements, e.g. the maximum error tolerated. The system then implements a shedding policy that meets those requirements. If the minimum quality of the results cannot be achieved for a query with the available resources, all inputs for that query are discarded, and the freed resources are allocated 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 tend to deliver intermittent results because all their data is discarded frequently; small lightweight queries tend to deliver approximate results due to a partial loss 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, where the aim 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 infrequent tuples are more likely to be discarded. The proposed technique can also support a flexible trade-off between the frequencies of the selected tuples and the latency defined as the delay between the generation 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 for its downstream nodes. The reduction of load at a node thus also reduces the amount of load on all other nodes hosting the subsequent operators in the query graph. This makes the problem of identifying the correct load-shedding plan more challenging than in a centralised environment.

In the Borealis system [Tat+08], the load-shedding problem is solved using linear optimisation. The goal is to maximise the output rate at the query end-points. The system inserts a number of load-shedding operators and chooses 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: (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 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, adding the corresponding set of load-shedding operators. The idea is to prepare the system against all possible overload conditions beforehand and to modify the query at run-time depending on the expected overload scenario. This general approach has been implemented in two ways: in a *centralised* and in a *distributed* fashion.

**Centralised Coordination.** In the centralised approach, a central server called the *coordinator* is used to calculate all the load-shedding plans. It uses information obtained from all processing nodes about the operators they are hosting, their input rates and their 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 passed to the processing nodes. The coordinator then starts monitoring the processing load at each node. If an overload condition is detected, the coordinator selects the most suitable plan to address it and triggers its execution at the processing node.

**Distributed Coordination.** 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 is used to compute a *Feasibility Input Table (FIT)*, a table containing the input rate combinations that are feasible (i.e. not causing overload) for that node. Once a node has calculated its FIT table, it is propagated to its parent nodes. 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.

### 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 can try to minimise its memory footprint through a number of techniques. For example, 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 materialise nearly identical relations. It can also schedule operators intelligently in order to minimise the length of queues in memory [Bab+03]. While these techniques do not lower the quality of the processing, the memory usage may still exceed the limits of the systems. In such situations, the internal state of operators can be approximated. In the case of aggregation and join operators, the state can be reduced by employing sampling techniques, such as using *histograms* [Tha+02]. These are approximations to data sets, achieved by partitioning the data into subsets, which are in turn summarised by 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 representations that provide a fast way of testing whether an element is a member of a set. All these techniques trade memory usage against precision and can greatly help the system overcome or avoid memory-induced 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 described failure due to overload (i.e. when the system resources are insufficient to carry out perfect processing). Failure, however, can also occur for 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 the query graph. It is also possible that a processing node becomes temporarily unreachable because of a network partition, especially in a widely geographically distributed processing infrastructure.

A stream processing system should address such issues in order to achieve a high 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.

### 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 produced during the recovery phase; in the latter case it is the *consistency* of results that is compromised. The system can also opt for a hybrid 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 is called *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 quantifying the amount of failure incurred [MW06]. This approach has been adopted in this thesis because we consider failure a constant operating condition of the system and the cost to achieve eventual consistency would become too high to be acceptable.

Borealis [Bal+05] adopts a strict consistency model that aims at eventual consistency. This means that in the case of tuple loss or misordering, the system tries to obtain the correct final result. It

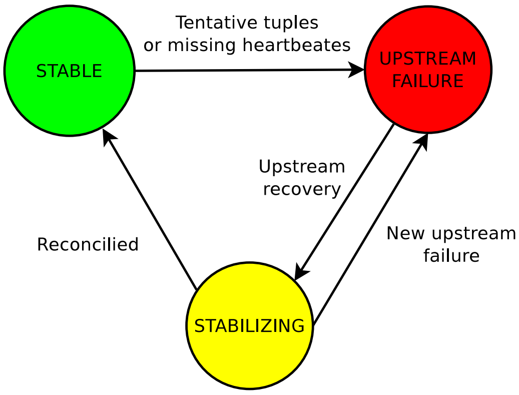

Figure 2.5: State machine describing Borealis fault tolerance mechanism in Borealis.

achieves this by marking a stream on which an error has occurred as *tentative*. Later the system attempts to restore the correct result by employing a revision process, which ensures that the final result is correct. Of course, this assumes a scenario in which a fault is more an exception than the rule, as the cost for the revision process can become high. There is no mechanism to quantify the input of the fault to the user. As the fault is thought to be transient and recoverable the only feedback given to the user is that the stream content is invalid and cannot be trusted until recovery. Instead, we propose a quality-centric data model that allows the system to quantify the impact of the error and to provide the user with feedback on the actual processing quality of the system.

The revision mechanism used in Borealis also supports "time travel". It is possible to recalculate tuples for the past and even in the future (by providing a prediction formula). Rolling back processing to the past is a heavy-weight process and requires a large amount of storage, which is not always feasible in stream processing.

Figure 2.5 shows the state machine for the fault tolerance mechanism employed by Borealis. When failure occurs upstream in the query graph, an operator is alerted by the arrival of tentative tuples or by the lack of heartbeat messages. It then marks its tuples as tentative while waiting the restoring of a stable state for the processing. When failure has been overcome, the state is reconciled and the system becomes stable once again.

In the relax consistency model proposed by Murty and Welsh [MW06], 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 may produce different outputs because of their different 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 called *harvest* and *freshness*. These metrics provide the user with information about the quality of the results. Harvest represents the coverage of a query result in terms of data sources. Its value is reduced by failures during query processing. A harvest of 100% indicates that the result was computed using data from all sources. This metric is not directly related to correctness but rather provides an indication of the confidence the user should have in the delivered results. A high value for harvest means that the provided result incorporates 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 the freshness value of the delivered tuples negatively. The lower bound for freshness is the network latency.

### Replication Models

In order to be more resilient to failure, stream processing systems frequently employs *operator replication*. The system identifies the most important operators in a query and runs multiple copies of them concurrently. In this way, if a node hosting a part of the query fails, one of its replicas can be used instead, without any impact on the processing. Replication is not only useful to increase the dependability of the system but it can also improve its performance by running the replicas in competition in order to reduce latency [HcZ08; HCZ07].

Early work on replication in a DSPS have been focused on masking software failures [Hwa+05] by running multiple copies of the same operators at different nodes. Others [SHB04] proposed to strictly favour consistency over availability by requiring at least one fully connected copy of the query graph to be working correctly 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 which all tuples are processed regardless of whether failures occurred or not on the input streams. This increases availability because the operator continues to process even in the presence of failures on its input streams. Output tuples, are then marked as *tentative* in order to signal the incorrectness of the results. After the system has recovered from the failure, it reconciles the state of operators that 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 of Borealis is to achieve *eventual consistency* and provide all clients with complete correct results.

Another approach to the management of replicas has been proposed in [MW06]. The authors take a different standpoint on failure by not considering it as an infrequent event but as a constant condition of the system. In this case, maintaining consistency among replicas becomes even harder. The authors claim that it is better to ignore this requirement and to employ *free-running* operators instead. Operator are permitted to update their internal states independently as incoming tuples arrive. In case of recovery from failure, for example, an operator restarts, without any knowledge of its previous state. However, this 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. Whenever the replicas are out of sync and produce different results, however, the system needs to choose, which set of results represents the "best" answer. It can do so in several ways, for example, by choosing tuples generated by the replica with the largest uptime, with a majority vote among the replicas, or by relying on some *quality metric* describing the quality of the data. This process is referred to as *best-guess reconciliation.*

## 2.6 Summary

This chapter presented the relevant background to understand the work presented in this thesis. It started with a description of a range of stream processing applications, in particular those that have to deal with large amount of data 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, which is a valuable tool for researchers to investigate the spread of information through social cascades. We also presented the most important query models used to express queries in stream processing systems: the continuous query language (CQL), which extends the relational model for the processing of unbounded streams, and the boxes-and-arrows model, which employs a visual paradigm to compose queries by creating a network of operators. After that, we introduced a system model for stream processing, presenting examples of systems that pioneered centralised as well as the distributed processing. When dealing with a distributed environment, the correct allocation of resources across processing sites is crucial. In this regard, we described efficient resource allocation techniques for both data centre 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 discard input data efficiently was provided emphasising the different issues related to load-shedding. Finally, there was a description of techniques used to avoid and recover from failures, with a discussion on the possible consistency and replication models.

The next chapter will describe the data model developed in the scope of this thesis which includes a quality metric that can be used to augment streams and allow the system to detect and measure the effect of overload.

# Chapter 3

# Quality-Centric Data Model

Overload in a stream processing system has been usually considered as a transient and rare condition. The common approach to handle overload has been to recover from it as quickly as possible, without quantifying its impact on the current processing. In many large-scale deployments, however, the occurrence of overload is common and not always avoidable. In such cases, it is better to adapt the system so that it can self-inspect and quantify the impact of overload 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 quantify the amount of information that was lost during the processing because of failure. A quality metric can be used to enhance streams and detect and estimate the impact of failure on the current computation. Such a metric should be generic enough so that it can be used in any stream processing system, supporting the semantics of traditional as well as custom operators, and be applicable to a broad variety of query types.

This chapter presents the quality-centric data model that we have developed to allow a stream processing system to calculate the impact of failure on its running queries. First, we provide a set of definitions about the basic components of a stream processing system, as they are assumed in the scope of this work. These fundamental concepts are used then to describe the workings of a stream processing system and to define a suitable quality metric.

In order to define such a metric, it is important to understand the goals and the assumptions. We provide an explanation about the reasoning behind such a metric and about the characteristics that it should have. Next, we introduce the two main families of queries, fan-in and fan-out, providing sample queries to illustrate them.

We then introduce the Source Information Content (SIC) metric, a quality metric defined based on the previous assumptions and with the required characteristics. A set of equations are given for its calculation, describing its propagation within the system and the benefits of its use.

The chapter closes with 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. They show how the SIC metric can be applied to different query types and how its behaviour directly derives 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 used in 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 in Section 2.2. Nevertheless, a precise definition of many concepts used throughout this thesis is necessary for the correct understanding of this work. Table 3.1 summarises the notation used below.

### 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 that it contains, providing a name and a data type for each item. Such a 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 item. It is equivalent to a schema used in a relational database, where it is used to describe the columns forming a table.

<u>EXAMPLE</u>: Consider a tuple with the following schema:

| | |
|---|---|
| *Integer* | NODEID |
| *Double* | CPULOAD |
| *Long* | UPTIME |

It contains information about load and uptime for a specific sensor. Each field has a name so that it can be referred to when processing the tuple and it includes an abstract type, which will be translated to a system type once the tuple is instantiated.

| Model Notation | |
|---|---|
| $t$ | tuple |
| $\tau$ | 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 data item processed by a stream processing system. An instance of a schema containing actual 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 in 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 ($\tau$) we mean an temporal indication of when the tuple was produced. This typically is the time at which the tuple entered the system. In this case, the sources are only concerned with the production of the raw data and timestamps are 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 useful in case of synthetic workloads. In this case, the timestamp is determined by an external entity.

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

The third element composing a tuple is the *payload* ($\mathcal{V}$). It 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

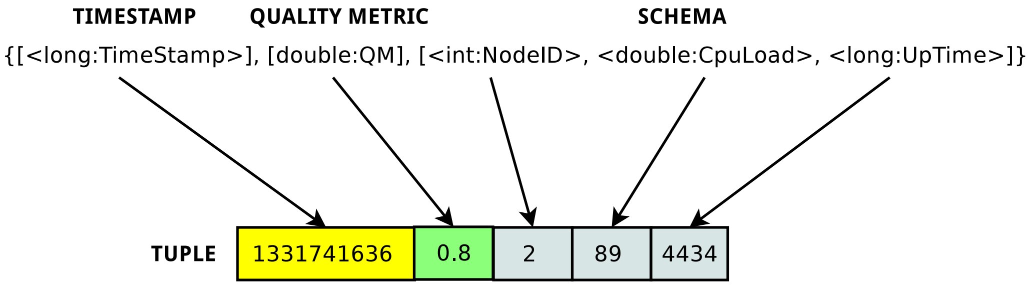
TIMESTAMP
QUALITY METRIC
SCHEMA
{[<long:TimeStamp>], [double:QM], [<int:NodeID>, <double:CpuLoad>, <long:UpTime>]}
TUPLE
1331741636
0.8
2
89
4434

Figure 3.1: A simple tuple with the relational schema.

the values forming a tuple payload is defined by the schema.

Logically, we can divide tuples in three categories: a *source* tuple ($t_{src}$) is a tuple generated from a data 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; and, finally, there exist *result* tuples ($t_{res}$), which are derived tuples produced by a terminal operator. They contains the final results of the processing, which are delivered to the user together with the quality metric value.

EXAMPLE: Figure 3.2 shows a simple tuple with its relational schema. It shows a *timestamp* expressed as POSIX time, a *quality metric* value and a *payload* with three fields carrying information about the CPU load and uptime of a machine monitored by the system. We discuss the implementation of tuples in Section 4.2.

### Stream

A *stream* is an abstract entity that describes all tuples flowing between two operators. The number of tuples is potentially infinite and all tuples share a common schema.

**Definition 3.3 (Stream)** *A stream is a potentially unbounded time-ordered sequence 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 $t_1$ always has a timestamp greater than or equal to $t_2$ (i.e. $\tau_2 \geq \tau_1$).

Analogous 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; and *result* streams ($S_{res}$) are derived streams produced by a terminal operator, which contain the results of the processing delivered to the user.

A stream is used only logically to describe a potentially unbounded sequence of tuples flowing between two operators. The system, however, needs a finite amount of tuples to operate, which in our system is represented by a *batch*.

### Batch

Streams are continuous abstract entities, while operators need a finite set of tuples to operate. The system thus partitions streams into *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 with 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 may be composed of a single tuple. Using batches allows for a more compact representation of the quality metric metadata because 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 (i.e. a finite amount of tuples that can be processed by an operator). In our system, they are the equivalent to *relations* used in 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 in 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 can divide operators based on their behaviour when handling input streams into two categories: *blocking* or *non-blocking* operators.

Blocking operators need to have at least one input batch available on each of their input streams. For example, 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 batch 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 blocks again, as only the first input contains data to be processed.

Non-Blocking operators do not need input on all channels to be triggered. Instead, they produce a derived batch as soon as at least one input batch is present on one of their input channels. Such an operator never blocks waiting for input and never has pending input data.

### Query

The processing required to transform the information contained in input tuples into a useful output for the user is typically not performed by a single operator but by a group of them, arranged into a logical processing graph. This graph is referred to as a *query*.

**Definition 3.6 (Query)** *A query logically defines a series of processing steps over a 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 means 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), in which arcs represent streams and vertices represent operators. One or more sources produce streams of tuples at time-varying rates, which are the input to the system. One or more operators process these tuples, either in sequence or in parallel.

A query is a graph, in which 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 $f_Q$. Operators may be distributed over a set of nodes $N$ if 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.

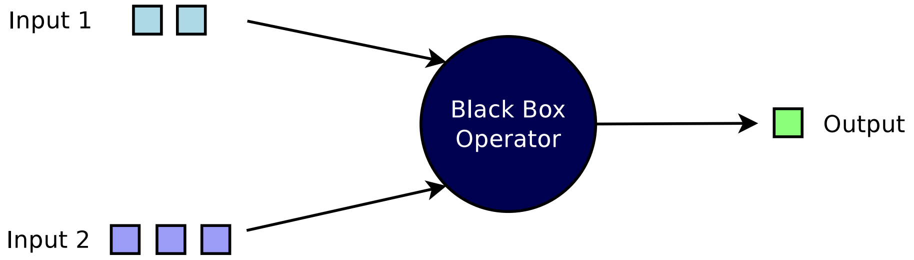

Figure 3.2: A black-box operator with two input and one output stream.

## 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. It is based on the consideration that overload is a common condition and should not be ignored but accounted for. Instead of trying to mask it, the system should monitor it and report its impact. Since queries can be composed of an arbitrary set of operators, frequently employing diverse processing semantics, a quality metric must be sufficiently generic to be usable across the whole spectrum of possible queries and not tied to a specific domain.

When considering data sources, it is possible that some of the produced inputs are more valuable than others but it is often difficult to decide this beforehand. The challenge of determining which sources and tuples are more valuable than others leads to the idea of considering all tuples as equally important. When an operator receives some tuples as input, it transforms them but does not change the amount of information that they carry. Conceptually, this is similar to the conservation of energy law in physics. Another consideration is that the amount of information in a tuple is proportional to the amount of tuples required for its generation. Since we assume that every source produces the same amount of information per time interval, the total amount of information processed by a query in the absence of failure is a multiple of the number of sources. Finally, we consider the shape of a query graph, which, in its most generic form, is a directed acyclic graph. The computation of the quality metric should be flexible enough to be usable for all graphs.

The rest of the section describes these assumptions in more detail, laying out the reasoning that lead to the Source Information Content (SIC) quality metric.

### 1. FAILURE IS EVERYWHERE

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

In large-scale distributed systems, failure cannot be considered a rare and transient condition. In fact, evidence suggests that, in such systems, a certain percentage of nodes will be failing at all times. Even though the Mean Time Between Failure (MTBF) for a single component may be quite high, once the number of components increases, failures become more and more relevant.

In a paper released by Google [PWB07], it is shown how the occurrence of failure in a large hard disk population is higher than what is declared by vendors. They analyse a large population of disks and showed how the Annualized Failure Rate (AFR) ranges from 1.7% for first-year drives to over 8.6% for three-year old drives. Jeff Dean of Google also presents statistics [Dea09] about the real world occurrence of failure in their data centres, showing that a typical cluster of 2400 machines it is expected to experience around 1000 individual machines failures in the first year alone. Overload can be considered as a type of failure because the system cannot process all the incoming data and must discard some of it. In this case, an approximate result is produced even though all the processing units function correctly. Input data rates can be highly unpredictable, with variations that can often be of orders of magnitude [Tat+03]. This makes it challenging to provision a system. If we consider a cloud deployment scenario, in which resources are rented, overload can not only be tolerated but may be deliberate. It may 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 clusters to gain access to a unified processing infrastructure, the possibilities for overload are even higher. Each local site, in fact, is under the control of a different authority, making recovery times unpredictable. Due to its nature as a shared platform, the processing resources available at each cluster are not uniform. The query fragments distributed across several processing sites thus experience a skewed availability of resources. For example, the same query fragment may run without failure at one site, while experiencing heavy overload at another.

### 2. APPROXIMATE PROCESSING

***Users can accept approximate processing but they 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 data stream processing system. We propose to augment data streams with a metric that captures the amount of failure during the processing. This quantifies the impact of failure instead of hiding it. In many applications, an approximate result is acceptable for users. It is up to the user to decide if the quality of the delivered results is good enough for these to be accepted or if they should be discarded.

In sensor networks, it is common to have a lot of failure at the data source level. Sensors can suddenly stop working because of hardware failure or can become temporarily unreachable. In such cases, a query that processes sensor readings delivers results that are incomplete. Nevertheless, the quality of the query results can be sufficient to be meaningful to the user. Especially when dealing with a large set of sensors, the lack of input from a certain number of them does not mean that the computation should be discarded altogether. Instead, the results become less accurate due to the missing input data. If we consider queries aggregating readings for sensors scattered over a geographic area, such as an average of measured pollutants in a city, it is possible that a failed sensor is located close to others that record an equivalent reading. Thus the missing information may not produce a significant variation in the final result, which is approximate yet still meaningful.

If we consider queries that detect a certain event instead, the failure of a single sensor can become critical. The failed sensor could be the one that would have generated a reading of interest but, due to the failure, this reading is not available. In this scenario, failure reduces a user's confidence in the results and may not be 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 argument can be made for the social media analysis scenario. In this case, the enormous amount of available input data can cause an overload condition of the processing infrastructure. Unlike the sensor data scenario, in which the failure usually occurs at the input level, the failure here is more likely to happen at the processing stage. Aggregation queries typically process vast amounts of data and, in case of the loss of a small percentage of data, they can still produce results close to what would have been obtained in the absence of failure. In general, when dealing with aggregation, the larger the set of input data, the more tolerable failure becomes.

### 3. A GENERIC QUALITY METRIC

***A quality metric should be operator independent, abstracting from the query semantics.***

Stream processing systems are versatile and can compute a variety of queries. Each query is composed of operators whose semantics can be different. If we consider the streaming equivalent of traditional operators present in a relational DBMS, such as *filter* and *average*, it is notable how the discarding of a tuple from the input of these operators can lead to significantly different quality reductions in the produced output. Due to their different processing semantics, missing a single tuple can have almost no influence or can completely alter 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 a priori.

Let us consider the different impact that the loss of a tuple can have on operator results. An average operator, that inputs a set of 100 tuples with integer values uniformly distributed in the range $[0, 1]$, produces a single output tuple with a value close to 0.5. If we assume that 50% of the input tuples are discarded due to overload, the operator still produces a single tuple with a value close to 0.5. In this scenario, the loss of half of the input tuples has virtually no impact.

A filter operator with the same input tuples outputs only tuples with values below 0.5. Since the input is uniformly distributed, it produces on average an output of 50 tuples. When shedding half of the input values though, the operator result changes considerably. On average, it now produces 25 output tuples, which is half of the number of tuples that it would produce with the complete set of input tuples. With this simple example, we can observe how the processing semantics of the operator can be different: the impact of input loss can be almost unnoticeable for some operators while it can be disruptive for others.

Therefore, the quality metric 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, 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 the absence of failure.

Other proposal for an operator independent approach exists. The *network imprecision* metric accounts for the state of all participating nodes in a large-scale monitoring system [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 by Murty and Welsh [MW06] and Fox and Brewer [FB99]. The authors argue that harvest should capture the fraction of data sources available in an Internet-scale sensor system.

### 4. DATA SOURCE EQUALITY

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

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 sensor settings or different battery constraints of the sensors. A unit with a depleted battery may reduce the rate at which it propagates information to save energy. Another reason could be the position of a sensor. The energy cost of propagating a reading grows with distance because it requires a higher transmission power.

Consider a sensor network monitoring humidity levels in a rural area, composed of sensors randomly scattered over a certain region. The processing query produces an average humidity value for the area every hour, aggregating readings from all sensors, which are produced once every 10 minutes. Due to the uneven distribution of sensors, some may require more power to transmit their readings and, in order to save battery power, may decide to reduce their transmission rate to half the original rate. When aggregating the humidity values, 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 can 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 sensors has the same value for the final calculation of the global humidity 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 produce more tuples than others. All sources equally contribute to the final result.

In our model, we treat all sources as equal but this is not valid in all cases. There may be scenarios in which a particular source should be considered more important than others. For example, it may be placed in a strategic location or it may be equipped with a more sophisticated sensor.

### 5. TUPLE EQUALITY

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

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

Let us consider a simple query, composed of one data source and one average operator, producing a single result every minute. The source produces 1000 tuples per second. The system is overloaded and unable to process all of the tuples. If the distribution of values carried by the source tuples is uniform, it does not make a difference which tuples are discarded. If instead the distribution is highly skewed, including a small number of outliers, discarding one of them could significantly change the aggregated result. Since the system employs an operator independent model, it cannot know which tuples to drop in order to reduce the error of the results.

This is a simplification to keep the model abstract enough to be usable in a general purpose system. Of course, there are situations, such as when using detection queries, in which some source tuples are more important than others. Having the knowledge of the query semantics could enable the system to be more selective and make distinctions between tuples when making shedding decisions but it would limit the system to support a set of operators and queries.

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

### 6. CONSERVATION OF INFORMATION

***The amount of information going into an operator is equal to the amount of its 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 an operator has a certain value for its quality metadata. It is the same for all tuples in a batch and may be different for different batches. The operator transforms these tuples into a new set according to its processing semantics. Even though the amount of tuples in the output may be different for the amount in its input, the *amount of information* does not change. We assume that the total value for the quality metric in the input to an operator is not changed 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 as input, applies a simple equation and produces $N$ tuples as 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 representative example. Such an operator receives $N$ tuples as input and produces 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 and outputs a single average value. Even though only a single output tuple is produced, it carries all the information for the input tuples, transformed into only a single aggregate value.

Consider a *filter* operator, which discards a certain number of tuples that do not satisfy a set of predicates. As an example, we can imagine an operator filtering all temperature readings with a value below 30 degrees Celsius. As 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 input, yet the total information carried remains unchanged. What changes is the individual information value associated with the single tuples, which is doubled compared to when the tuples entered the operator.

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

Finally, we consider the case, in which an operator produces no output at all. This can happen with filtering operators, when all the input tuples do not satisfy the filtering predicate and therefore are discarded. Since there is no output tuple to carry the input information content, this case would be indistinguishable from the case where all the input tuples have been lost, for example, because the system decided to discard them 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 the input. In this way, the total amount of information is preserved.

### 7. INFORMATION IS VALUE

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

When we consider the importance of an individual tuple within a query, it is important to take into account the amount of information that went into its creation. The more information is part of a tuple, the more important it becomes for the computation of the final query result. If the system needs to make a decision about which tuples to discard, it has to consider the individual information content of tuples and discard 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 shown in

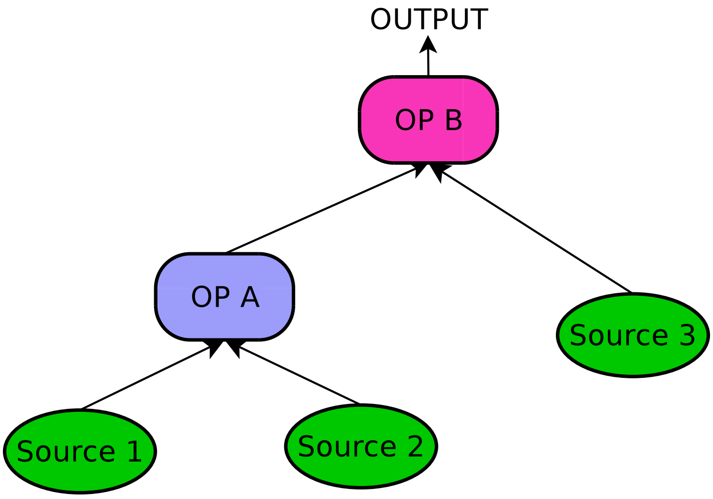

Figure 3.3: An unbalanced query tree, showing the different information content of tuples.

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. The resulting tuples are processed by operator B, together with the tuple produces by source number 3. We can assume that the operator B resides on a different machine than operator A and that, due to overload, it needs to discard one tuple out of the four present in its input buffers. When comparing the information values of the tuples, the tuples received for the left input stream have an information content that is twice that of the tuples received on its right input stream. This is because the tuples on the left contain the information for 2 sources, while the others carry the information for a single source. Since the goal of the system is to deliver results with the maximum amount of source information, the system decides that the tuple to discard should be one of those received on its right input.

### 8. TOTAL INFORMATION CONTENT

***The total amount of information contained in a result in the absence of failure is equal to the number of sources***

In the absence of failure, every result batch produced by a query contains an amount of information that is equal to the sum of the individual information values carried by the source tuples. 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 final value of a result batch is equal to the number of sources.

Let us consider a simple query that calculates 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 has happened and that all tuples are correctly processed. Since all sources are considered equal, the total amount of information carried by the tuples produced by each source over a minute is 1. When the final result is computed, the total information is equal to 10 (i.e. the number of sources).

When dealing with a system that supports the concurrent execution of multiple queries, it becomes important to compare the information values of the different queries. This is needed in an overload condition, when trying to achieve *fair shedding*. In fair shedding, the system selectively discards tuples to reduce the load of the system so that the final quality values of all queries is equalised. Therefore, the system must compare the different information content values of tuples. A simple way to achieve this is to normalise the information content value by dividing the total value of a query result by its number of sources. This scales all values in the interval $[0, 1]$, making them comparable.

### 9. QUERIES ARE DIRECTED ACYCLIC GRAPHS

***An operator may have more than one downstream operator.***

A query can be represented as a directed acyclic graph (DAG), in which nodes represent operators and arrows the streams flowing through them. This means that, as long as there are no cycles, every operator can have multiple downstream operators that receive its output as input. There are two reasons for distributing the output of an operator: (1) the query has to compute multiple results; and (2) the downstream computation has to be split over multiple machines because it would be too computationally intensive for a single one.

The first case deals with queries computing multiple results. These are called *fan-out* queries and will be further discussed in Section 3.3. They compute multiple results based on common input information but are logically 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 two running queries, submitted by two users. The first user submits a query with a single terminal operator, while the query submitted by the other user may have 9. If the system considered each computed result 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 considers the query submitted by the second user as single computation with 9 different results, it can evenly discard tuples among the two queries, leading to a fair allocation of the available computing resources. Fair shedding will be described in more detail in Section 5.3.

The second reason for the distribution of the output of an operator is when the downstream computation is too intensive to be executed by a single machine. In this case, the output is split among several processing nodes, each hosting the same set of operators that process a fraction of the output tuples. The computed results are then collected to produce a single final result.

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 from its input to all its output tuples. A tuple is assigned a value that is inversely proportional to the number of the output streams of the 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 value for the query is the same as in cases with only a single terminal operator. The details of this are covered in Section 3.4.

## 3.3 Types of Queries

This section introduces a categorisation of different query types, which is based on the shape of their graphs. When each operator has at most one successor, the query graph resembles a tree, with input sources as leaves and one output stream as root. Such queries belong to the *fan-in* type: the number of edges in the graph decreases from sources to output. A query containing operators with more than one successor instead belongs to a *fan-out* type. In this case, the query graph can be more complex, with the restriction of being acyclic. Fan-out queries can have multiple input sources as well as multiple result streams. The following sections provides more detail on these two types of queries.

### Fan-in Queries

In *fan-in* queries, operators have *at most* one successor operator, and the input data from one or more sources is used to produce a *single result.* In this category, we find queries computing a single aggregation result. Such queries have a graph that resembles a tree with many input sources, a series of processing steps and a single output stream.

Example: Consider the query depicted in Figure 3.4 that calculates the maximum average value produced by the data sources. Three sources produce input tuples at different rates. All tuples from a source are first collected by a time window and then averaged. Finally, the maximum value among the averages is selected and output. This is a typical fan-in query, in which input tuples from several sources are collected and processed to produce a single output.

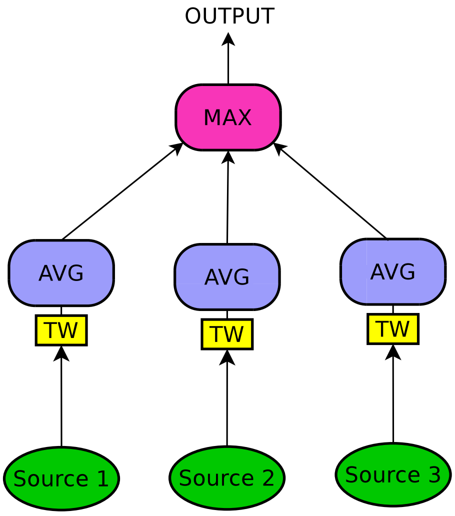

Figure 3.4: An example of a fan-in query. Input values from each source are first averaged over a certain time window, and then the maximum is selected.

### Fan-out Queries

The second class of queries that we consider are *fan-out* queries. There one or more operators send their output to *more than one* downstream operators. Unlike fan-in queries, whose graph is a tree, a fan-out query graph can have any configuration, as long as as it does not contain cycles.

Fan-out queries can be divided into two possibly overlapping classes: (a) queries with one result but split computation; and (b) queries with more than one result. Next, we present examples for each case.

**Split Computation Queries.** Since operators may be computationally too demanding to be hosted on a single processing node, their computation may have to be split among a number of copies running at different sites. These copies are composed of 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 multiple nodes and thus overcome an overload condition at a single 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 "top rated" status updates. In addition to the message text itself, every message contains information about the location where it was sent and the author. A single stream of Twitter messages is split and scattered over three different processing nodes, each hosting a Natural Language Processing (NLP) operator that calculates some coefficient for each message, for

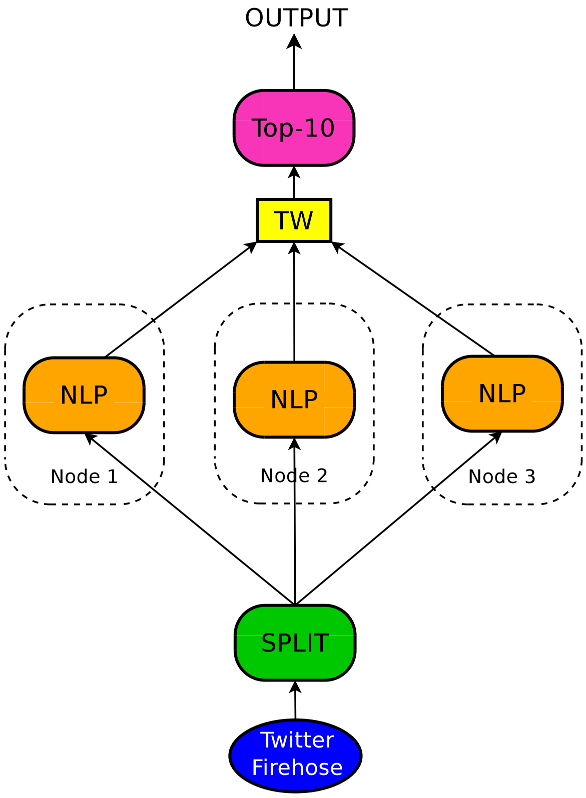

Figure 3.5: An example of fan-out query. It processes Twitter data by counting the top-rated messages on a certain topic.

example, its "rating". The output of these NLP operators is collected by a single Top-10 operator preceded by a one minute time-window. The query then outputs then the ten "top-rated" Twitter messages posted every minute.

**Multiple Result Queries.** Some 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 analysing a stream of data, a user may be interested in obtaining several different results. Instead of submitting $N$ queries, they can submit a single fan-out query with multiple end points. Conceptually, this is simpler than having to design and submit several almost identical queries. It directly exploits the processing redundancy by reusing a number of streams and operators.

Example: Figure 3.9 shows a query that calculates the occurrence of positive and negative statements about particular keywords on Twitter. The original stream contains an unsorted stream of messages. First, a filter eliminates all the messages that do not contain a keyword of interest. After that, the output stream is multiplexed over two different NLP operators, which forward tweets if they contain either a positive or a negative mention of a keyword. The resulting streams are sent to a counter operator, which produces two output streams counting the number of positive and negative references to a given keyword. The resulting streams from the counter operators are sent to a final operator that calculates the ratio between them, resulting in an indication of the general feeling about a keyword. In this query, a single input source is processed to produce three different results.

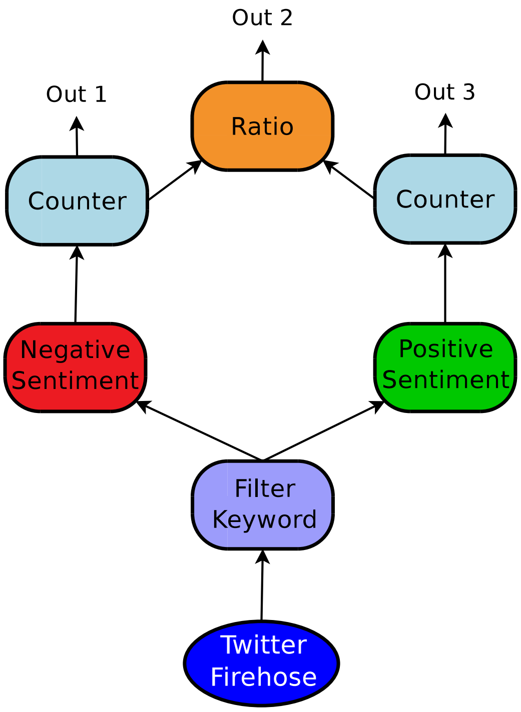

Figure 3.6: An example of a fan-out query for processing Twitter data. It counts the occurrence of positive and negative mentions of a certain keyword and calculates their ratio.

## 3.4 Source Information Content

This section provides the definition of a new quality metric called *Source Information Content (SIC).* A complete set of equations is derived for its calculation, following the reasoning expressed in the assumptions presented in Section 3.2. The section finishes with a discussion of the benefits of the SIC metric when employed by a stream processing system.

### Introduction

Typically a stream processing system does not have global knowledge if all the available data was processed correctly. Therefore, we propose that the impact of failure should be monitored and accounted for. In our model, we augment streams with a new metric called *Source Information Content* (SIC), which quantifies the amount of information in a tuple, thus recording its importance for the current query results. The SIC metric is added to streams in the form of metadata, whose value is calculated by the query operators and is inversely proportional to the level 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 includes all the available information, making it 100% accurate, or if some information was lost in the process, thus reducing its quality. *Source Information Content* tries to achieve this goal by measuring the amount of lost information.

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

When only a single query is deployed, the system can minimise the impact of overload by selecting the tuples to be discarded among those with the lowest SIC values. When dealing with a system that allows the concurrent execution of multiple queries, SIC values help discard tuples in a way that affects all queries in the system equally. We refer to this as *fair shedding.*

### Data Model Formalisation

The purpose of the SIC metric is to capture the amount of information that contributed to the creation of an output tuple. In the absence of failure, the *perfect value* of the SIC metric is equal to the number of sources. A reduction from this value is caused by failure during 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 based on individual tuples but instead over a window of tuples. The amount of information is defined over a time interval so that the final result SIC value is the sum of the individual SIC values of all tuples produced within the corresponding time window. Every tuple within the interval is assigned the same value, as for Assumption 5 (Tuple Equality). All sources assign the same total SIC value in a time window, as for Assumption 6 (Source Equality).

For example, a source producing 100 tuples in a given time interval assigns a SIC value of 1/100 to each tuple. Another source producing only 20 tuples during the same time interval assigns individual SIC values of 1/20. The result tuples obtained after processing would have an aggregated SIC value of 2 without failure. We refer to the complete set of tuples produced by the sources in a given time interval as the 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 that contributed to the creation of $t_R$ (i.e. the set of tuples that were processed by the query function $f_Q : \mathcal{T}^S \rightarrow t_R$ to generate $t_R$).*

In our query model, every result tuple $t_R$ is associated with the set of source tuples $\mathcal{T}^S$ that contributed to its creation. We consider a query as a black-box with a *query function* $f_Q$ that maps source tuples to result tuples. The *source information tuple set* of a result tuple is the set of all source tuples.

This concept is central to the definition of the *Source Information Content* (SIC) quality centric metric. The intuition behind it is that a result tuple is considered to be perfect when no information from the sources was lost, meaning that all tuples in its source information tuple set (and any derived tuples) were processed correctly. If one of the tuples is lost, either due to failure or deliberate load-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 source information content (SIC) of all result tuples for query Q, denoted as $SIC_Q$, measures the contribution of all source tuples, belonging to the source information tuple set $\mathcal{T}^S$, in the result tuple set $\mathcal{T}^R$. It is calculated as:*

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

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

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

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

Equation (3.1) states that the total SIC value of all result tuples for a given query, is equal to the sum of the SIC values of all tuples in the Source Information Tuple Set. 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 are correctly processed, the resulting SIC value is 1. A lower value indicates that some information loss happened during the query execution.

Equation (3.2) describes how source tuples are assigned their individual SIC values. First, a value of 1 is assigned to the totality of tuples produced by a source in its Source Tuple Information Set. This means that all sources are considered to contribute equally to the final result, regardless of the amount of tuples that they produce during an interval. Since every tuple in this set is considered to contain the same amount of information, this value is divided by the number of tuples produced by the source during a time interval. Thus, the SIC value of an individual source tuple $t_{src}$ is inversely proportional to the number of tuples in $|\mathcal{T}^S_{src}|$. Furthermore, it is necessary to normalise by the number of sources $|\mathbb{S}|$ connected to a query, dividing the initial SIC values by the number of sources. In this way, all resulting SIC values are in the interval $[0, 1]$, which makes it possible to compare the quality of different queries.

### SIC Propagation from Source to Result Tuples

Up to now, we considered queries as black-boxes and discussed only the relation of SIC values between the 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 individual operators and how these propagate within the system.

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

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

$$t^{SIC}_{out} = \frac{1}{|\mathcal{T}^o_{out}|} \cdot \sum_{t_{in} \in \mathcal{T}^o_{in}} t^{SIC}_{in} \tag{3.3}$$

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

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

### Relation of SIC to Query Processing Quality

The SIC values of result tuples lie in the $[0, 1]$ interval:
(a) A value equal to 1 indicates that the complete set of $\mathcal{T}^S$ tuples was used during processing and that the result is perfect.
(b) A value of 0 indicates that all source tuples from $\mathcal{T}^S$ were discarded or lost during processing.
(c) A value between $(0, 1)$ indicates a degraded result, meaning that only a subset of the tuples contained in the Source Tuple Information Set was used for computation. A small value means a large information loss, while a value close to 1 indicates an almost perfect result.

### Model Benefits

**Query Performance.** Even though SIC values provide a good estimate of the quality of the processing, it should be noted that their value is not linked directly 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 or should be discarded.

Nevertheless SIC values are a good indication about the quality of the results. The knowledge of the query semantics may allow a user to correlate SIC values with an absolute result error. 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 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 if and only if $t_1^{SIC} > t_2^{SIC}$.

A user can monitor the status of a computation by observing the changes in SIC values of a query at different times. A decrease in SIC values means that more information was discarded, and there may be a need to act upon it. Reasons for a sudden reduction may be:
(1) the failure of one or more processing nodes, requiring the migration of some operators and recovery of the failed infrastructure; or

(2) a sharp increase in input rates, requiring the increase of processing resources to maintain SIC values within acceptable bounds.

**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 processing of running queries, allocating resources in such a way as to provide results with similar SIC values for all queries.

When the system experiences *overload* and needs to discard a certain number of tuples, it can use the SIC values to determine which tuples to drop and which tuples to preserve. This process is achieved through *fair shedding* and will be further explored in Section 5.3.

## 3.5 Use Cases for Source Information Content

This section returns to the sample queries for Section 3.3 to illustrate the calculation of SIC values.

### Fan-in Queries

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, we show absolute values (i.e. not scaled to the $[0, 1]$ interval for comparison). It is also a perfect value (i.e. there was no loss of information during processing). In the case of failure, its value would be reduced by the amount of information contained in the lost tuples.

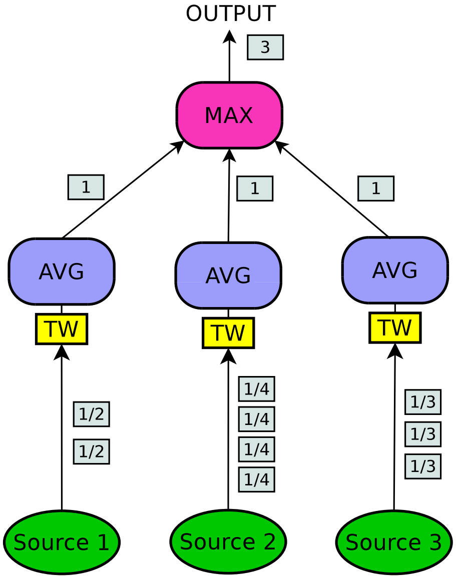

Figure 3.7: An example of fan-in query. Input values for each source are first averaged over a certain time window, and then the maximum value is selected. Source Information Content values are shown for each tuple.

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

Figure 3.7 shows 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 4 tuples with an individual value of 1/4; and the third 3 tuples, 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) requires that the amount of information going into an operator is equal to the amount in the output. Therefore, the newly generated tuples are all assigned a SIC value of 1, which is the sum of the individual SIC values of the input tuples.

The final *max* operator inputs 3 tuples, each with a SIC value of 1, and outputs a single tuple with a 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 required by Assumption 3 (A Generic Metric). The final tuple contains the total amount of information carried by 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 example, a source tuple produced by Source 2. In this case, the amount of information in the tuple generated by the central average operator would be 3/4 instead of 1. The loss of information (i.e. to the amount of 1/4) would then be propagated to the final tuple, which would have a SIC value of 11/4 instead of 3. Since the calculation of the SIC values is additive, the amount of information missing, compared to the maximum value, is equal to the sum of the SIC values of the individual tuples that were lost during processing.

### Fan-out Queries

**Split Computation Queries.** Figure 3.8 shows a query that processes 300 messages during a time window. Each message is analysed by a Natural Language Processing (NLP) operator that calculates a coefficient based on its text. Finally, a max operator outputs the message with the highest coefficient.

We assume that the NLP operator is computationally intensive and would overload a single node at the current rate. Therefore the computation is split across 3 different nodes. Each node processes 1/3 of the messages, and all nodes manage to process all the messages without the need to discard data. The *split* operator creates a partition of the original stream without data duplication, which means that the tuples that it outputs still have the same SIC value as the ones in the input.

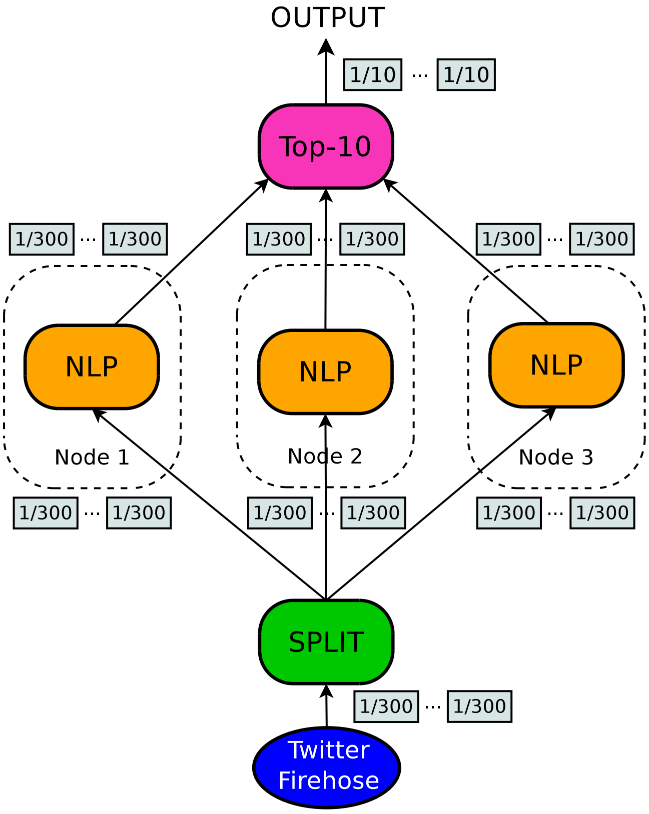

Figure 3.8: An example of a fan-out query processing Twitter data. The Source Information Content values are shown for each tuple.

This follows from Assumption 9 (Queries are DAGs). In this query, the split operators receive as input 300 tuples. Each with a SIC value of 1/300. The operator outputs 100 tuples on each output stream, again with 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 that they receive as input, thus producing 100 tuples each with an individual SIC value of 1/300. Finally, all the tuples are processed by a Top-10 operator, which outputs the 10 tuples with the highest coefficients. Each output tuple has a SIC value of 1/10, for a total final SIC value of 1.

**Multiple Results Queries.** Figure 3.9 shows a query calculating the occurrences of positive and negative mentions of a given keyword. 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 original stream comes from the Twitter platform and contains an unsorted stream of messages. In our example, during a time window, there are 3 messages with individual SIC values of 1/3. A filter operator selects only those messages that contain a given keyword, in our example outputting 2 tuples. At this point, the output of the filter operator is multiplexed over two different operators, one that filters only messages with positive references to the keyword and one filtering negative references.

Assumption 9 (Queries are DAGs) requires that, when such a duplication of the output occurs, the total SIC value must be distributed over all output tuples. This is also in accordance with Assumption 6 (Information Conservation) because the amount of information in the input is equal to the amount in the output. Each output stream of the *keyword filter* operator contains an identical set of 2 tuples, each with an individual SIC value of 1/4. The positive and negative filters each output a single tuple with a SIC value of 1/2.

These output tuples propagate to *counter* operators. 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 are scaled accordingly. In our example, each counter produces 2 identical batches of 1 tuple with a SIC value of 1/4. The ratio operator outputs the third result of the query, producing a single tuple with a SIC value of 1/2. The result tuples have different SIC values: 1/4 for the ones produced by the counter operators and 1/2 the ones produced by the ratio operator. If we sum these together, we obtain a total SIC value of 1 for the query, which indicates that no failure occurred during processing.

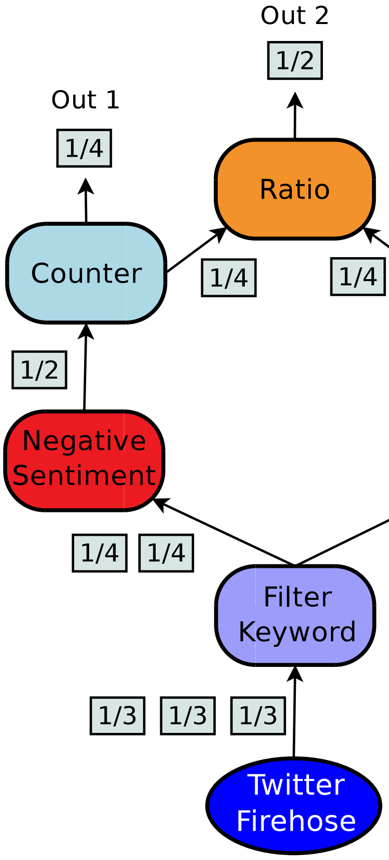

Figure 3.9: An example of fan-out query processing Twitter data. It counts the occurrences of positive and negative mentions of a given keyword. The numbers shown on tuples are their individual SIC values.

## 3.6 Summary

This chapter presented a data model for stream processing systems that takes the quality of the processing into account. First, we defined basic entities, such as tuples, streams and queries. The need for handling and quantifying failure led to the introduction of a quality metric called the Source Information Content (SIC). The definition of this metric followed the reasoning outlined in a set of assumptions and considerations, making it applicable to a generic stream processing system supporting any kind of operators. After that, there was an introduction of the different classes of queries, namely fan-in and fan-out queries. This led to the definition of the SIC metric with a set of equations for its calculation. A few real-world queries were then used to show how the metric can be used to keep track of the amount of failure that occurred during processing. The next chapter describes the design of a stream processing prototype system that implements the quality-aware query model, thus exploiting the SIC quality metric in practic.

# Chapter 4

# DISSP Design

This chapter introduces the design of the DISSP system, a prototype stream processing system developed to implement and evaluate the quality-centric data model described in the previous chapter. The chapter describes some of the novel features of the design and its 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 processing of queries.

The system is designed from the ground up to be distributed, allowing the processing of queries to span multiple compute 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 *federated resource pools* with different authorities administering different processing sites; or *cloud deployments* in which the resources are rented on demand and it may be more cost effective to reduce the amount of processing resources voluntarily, trading an acceptable reduction in the correctness of results for a substantially lower operational costs.

*Overload* is assumed to be the normal condition of operation for the DISSP system and not a transient, rare event. The SIC values of tuples are used to quantify their importance so that the system can perform an *intelligent selection* of which tuples to shed. This allows the creation of *flexible shedding policies*, as will be shown in the next chapter. This chapter describes the design choices made to provide a reliable and efficient processing of queries in such a hostile environment. It also gives an overview of the different system-level components and their interaction when deploying and running queries.

## 4.1 Design Approach

The design of the DISSP prototype system follows the ideas behind the quality-centric data model from the previous chapter. The DISSP system places the SIC quality metric at its core and uses it to reason about the quality of processing achieved by queries. Every stream is augmented with a SIC value and operators automatically calculate the new values for their output. Using SIC values, the system is able to operate under overload, reasoning about the performance of queries and making intelligent load-shedding decisions.

**Efficiency under overload.** The system achieves *efficient processing* of tuples and operators. Even though it is designed to handle resource overload, its occurrence is mitigated as much as possible. In an overload condition, having to discard some of the input data is always undesirable to the user. Especially since the system is designed to operate in resource constraint environments, it should try to maximise its throughput so that the limited available resources can be fully exploited. Under an overload condition, the efficiency of the system is inversely proportional to the quantity of load-shedding so that every increase in system performance leads to a decrease in the amount of input to be discarded, and thus to an increase of the SIC values of the computed tuples.

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

**Flexible load-shedding policy.** The use of the SIC values also allows the DISSP system to employ flexible policies when making load-shedding decisions. When deciding which tuples should be discarded, their SIC values provide valuable information to the load-shedding component. A perfect value indicates that a tuple was produced in the absence of failure, while a lower value gives a measure of the amount of lost information. Using this information, a load-shedder can decide to prioritise some queries over others, reasoning about the impact that tuple loss would have on their performance. It uses the local decisions at every processing node to implement a global shedding policy. Chapter 5 focuses on the implementation and testing of a *fair policy*, which tries to minimise the difference in SIC values experienced by all queries running in the system.

## 4.2 Implementing the Model

This section introduces the basic building blocks of our system. It explains how the theoretical concepts presented in the description of the data model in Chapter 3 can be actually implemented. It starts describing the basic concepts related to data, namely *tuples* and *batches*. After that, it covers *operators*, providing a taxonomy and a description of the most common ones. It then describes *queries*, showing how they serve as logical containers for operators and can be partitioned into smaller entities for distribution. Finally, we introduce the *source time window* that implements the abstract concept of the *source information tuple set*, which associates SIC values with input tuples.

**Tuples.** Tuples are the simplest unit of information processed by a stream processing system. As described in Section 3.1, they comply with a *schema*, stating the number, the name and the type of their values. Values represent the *payload* of a tuple (i.e. the data processed by operators in a query). Every tuple also contains a *timestamp*, an indication of their time of creation. In our system, this is expressed as POSIX time (i.e. the number of seconds that have elapsed since midnight Universal Time Coordinated (UTC) on January 1, 1970). A timestamp is typically set externally for *base tuples*, or set by the system when creating *derived tuples*.

Tuples are also associated with an individual SIC value, which expresses the quality of the data carried in the tuple. In our implementation, this value is associated with the container batch and not attached to each individual tuple. This reduces the overhead of transporting a `double` value (32 bits) with each tuple, exploiting our assumption 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 with each tuple, the system uses batches as containers with a single SIC value, which is considered valid for all the tuples contained in that batch.

Batches are the input and output units of operators. The output of an operator may be composed of several tuples, which are encapsulated within a batch. The operator calculates a SIC value for all tuples in the batch based on the SIC values of the input batches and its processing semantics. The newly produced batch is delivered as input to another operator or is returned as a result to the user.

Batches are an implementation of the CQL concept of a *relation*, as described in Section 2.2. They represent a finite snapshot of a stream, which allows operators to process a potentially infinite continuous stream of tuples in consecutive, discrete steps. By using batches, the DISSP system does not need to use *stream-to-relation* or *relation-to-stream* operators because batches provide a unified input/output unit for operators. A stream becomes an abstract entity as a potentially infinite set of batches, while batches serve as the actual units of information in the system. In our system, batches are simple objects that contain a list of tuples, a SIC value and some additional metadata.

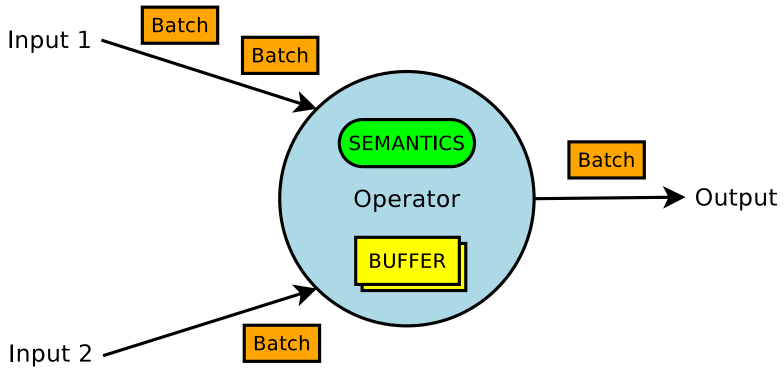

Figure 4.1: A generic operator with its internal structure shown.

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

Operators can be classified as *blocking* or *non-blocking* based on their behaviour when handling 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.

Figure 4.1 shows the internal structure of an operator with its two main components: the *buffer pool* and the *semantics 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. A buffer may still contain tuples from the previous execution cycle. In this case, the new batch is merged with the leftover tuples.

After the input data was staged, the operator executes the logic contained in the *semantics module*. This is what characterises an operator by defining its processing logic. From an implementation point of view, an operator is an abstract class with an unimplemented `execute()` function. A concrete operator class has to provide the code for this function. Once an operator finished processing, this function returns a batch of tuples that are delivered as its output.

**Queries.** A query logically groups operators that cooperate in the same processing task. In our design, a query is organised according to the *boxes-and-arrows* model. A 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*, which transform the incoming data streams into the DISSP system format. They thus act as a gateway, allowing external input generators, such

as a sensor network, to connect to the system and make their data available for processing.
In addition, there are a number of *data manipulation operators* that take data items and process them according to the query semantics. After a final result was obtained, it is delivered to the user through an *output operator*.

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

**Subqueries.** Queries may become too computationally expensive to execute completely on a single processing node. To avoid overload, the query graph can be split into a number of partitions, which are referred to as *subqueries*. The partitioning of a query can follow different strategies, e.g. grouping operators according to equal computational costs or distributing the same subquery for parallel processing. The different query partitioning schemes are described in Section 4.4.2.

Subqueries are deployed onto several processing nodes. For each query 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 on the processing nodes. After the query was deployed, data can start flowing through the input operators and the processing begins.

**Source Time Window.** Section 3.4 introduced the concept of the *source information tuple set*: the set of input tuples from which a final result is generated. In the absence of failure, the final SIC value of a tuple 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 to all result tuples.

To implement this part of the data model in practice, it would be necessary to know in advance which tuples will contribute to the creation of a future result. Since the result has not yet been created, however, it is infeasible to identify the source information tuple set and thus to define it accurately due to the time delays during query processing caused by processing delays, variable-sized time windows, etc. To address this challenge, we consider an approximation: a source information tuple set that includes all source tuples generated by sources during a predefined time period, referred to as the *source time window (STW)*.

The length of a STW is chosen so that it is always longer than the least possible end-to-end query processing delay. In this way, the STW contains all tuples that can possibly contribute to the generation of future results. In practice, a STW may include source tuples that generate multiple different result tuples, or belong to different source information tuple sets. Therefore, we apply the same concept of the STW to the result streams as well: the result SIC value of a query is the aggregate SIC value of all result tuples generated during the same STW. In order to capture the continuous nature of processing in data streaming, we treat the STW as a sliding window with a small slide compared to its size.

By using the STW, we no longer require the precise definition of the source information tuple set $\mathcal{T}^S$ and result tuple set $\mathcal{T}^R$. Rather, we can use two new sets: $\mathcal{T}^S_{STW}$, which is the set of input tuples belonging to a source time window, and $\mathcal{T}^R_{STW}$, which is the set of output tuples belonging to the corresponding window on the result streams.

Using these new tuple sets, the definition of the result SIC value of a query becomes:

$$\mathrm{SIC}_Q = \sum_{t_R \in \mathcal{T}^R_{STW}} t_R^{SIC} = \sum_{t_{src} \in \mathcal{T}^S_{STW}} t_{src}^{SIC}, \tag{4.1}$$

This definition states that, in the absence of failure, the sum of all SIC values of tuples in a source time window is equal to the sum of all SIC values of the tuples present in the corresponding window on the result streams.

## 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 be deployed on a large number of nodes. Every system-wide component that takes part in the processing of a query will be described, emphasising its role within the system.

After that, we present the *node architecture*, focusing on the internal operations of a DISSP node. The different components of a processing node are examined, providing details of some of the implementation choices and the reasoning behind them. In particular, we describe the network layer that handles inter-node communication, the operator runner that realises the query semantics, the load-shedder used to manage overload and the statistics manager that collects information on the system status.

### 4.3.1 System-level Components

This section presents the system-level components in the DISSP prototype stream processing system. The most important component of all is the *processing node*, a dedicated machine used for the processing of queries. One or more processing nodes constitute the resource pool onto which all queries are deployed. Since a query is usually divided into subqueries that are hosted on different processing nodes, a *coordinator* is deployed for each query. It is hosted on a separate machine and is in charge of monitoring the status of the query (i.e. detecting failed nodes and gathering statistics about the query performance in terms of achieved quality of processing).

A number of data *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 monitor the behaviour of the system as a whole, and is used during query deployment by the coordinator to provide a list of processing nodes suitable for the deployment of a new query. Finally, there is also a *submitter*, which provides the user interface to submit one or more queries for execution.

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

**Processing Node.** The *processing node* is the component in charge of the execution of operators. It receives tuples through a network connection. Tuples are converted from their serialised network representation into concrete instances. Tuples are then sent to the graph of operators in charge of their processing. After processing by one node, tuples are sent to the next node to be further processed by the next subquery or are sent to the 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 processing and to the *oracle* about its throughput and load condition. All *processing nodes* are also in charge of performing *load-shedding*, whenever an overload condition arises. They then make local decisions about which input tuples should be discarded in order to reduce load. A load-shedding policy designed to achieve *fairness* among all queries running in the system is the topic of Chapter 5.

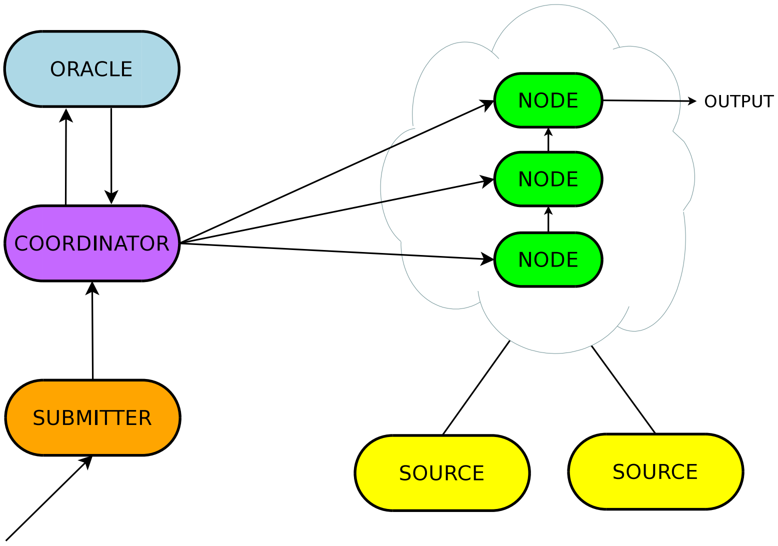

Figure 4.2: A system level overview of the components in DISSP prototype.

**Submitter.** Queries are deployed in the system through the invocation of a *submitter*. It receives as input an XML file with the query specifications, including the *tuple schemas* to be used and the *list of operators*, specifying their type, name, parameters and connections. Based on this query description, the *submitter* 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 into a network representation so that it can be transmitted to a *coordinator*, which is responsible for deployment and management of the query.

**Coordinator.** Each 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, a query is assigned a *coordinator* that is active for the whole query lifetime. When a query is to be deployed, the coordinator contacts the *oracle* to receive a list of nodes suitable to host it. It then partitions the query graph across a number of processing units and assigns each to a *processing node*. After the query was deployed and started, the coordinator 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. They are generated elsewhere, for example, 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. The input provider component acts as a *gateway*, at which data is converted to the system representation and made available to queries for processing. Every source can be the entry point for many data sources by employing multiple *InputDevice* operators. Each of them connects to an external source, for example, the sink of a sensor network or to the Twitter firehose.

**Oracle.** The *oracle* is a system level component that supervises all the processing happening in the system. It has two main functions: *monitoring* and *management*. It provides an overview of the status of all queries so that it is possible to know their achieved quality of processing in real-time. It also reports the status of all processing nodes, providing information such as their throughput, number of queries running and their load status. The oracle is also responsible for providing a list of suitable nodes for deployment of new queries. When a new query is submitted to the system, its *coordinator* asks the oracle to provide a list of $N$ nodes, which are most suited to host the new operators. The *oracle* selects these nodes following a *deployment strategy*, ranking all processing nodes according to some criteria. For example, it may choose to provide a list of the *least-loaded* nodes in order 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 in a system deployment. It is hosted on a dedicated machine and receives messages from *coordinators*. These messages report the quality of processing and other information about the query that they manage, and also request a list of nodes suitable for deployment when a new query is instantiated. In addition, the *oracle* provides a web interface so that it is possible to connect to it using a web browser and access a dashboard with real-time updated information about the system.

In a distributed system, the design choice of a centralised monitoring entity could seriously hinder scalability. Therefore, the same functionality could be implemented using an *epidemic* approach [VGS05], in which all nodes exchange gossip messages. A *coordinator* could discover the list of the least-loaded nodes by participating in gossip communications regarding the load metric [VS05]. A user could monitor the status of a query by participating in gossip communications with its processing nodes. Even though such a solution would be more scalable, a centralised approach is simpler to realise.

### 4.3.2 Node Architecture

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

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

The *operator runner* delivers the incoming tuples to the appropriate subqueries and executes the operators. It employs a pool of concurrent threads so that multiple operators can execute in parallel. Once a subquery has completed its processing on a set of tuples, it sends them to the network layer that forwards them to the following node to continue processing.

If the incoming input is too large and the node resources are not sufficient to process it completely without exhausting resources, the *load-shedder* component selects 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 shed tuples *fairly* with the objective of maintaining an equal quality of processing (i.e. SIC value) for all queries.

The final important component of a processing node is the *statistic manager*, which is in charge of collecting statistics about the performance of queries and of the node as a whole.

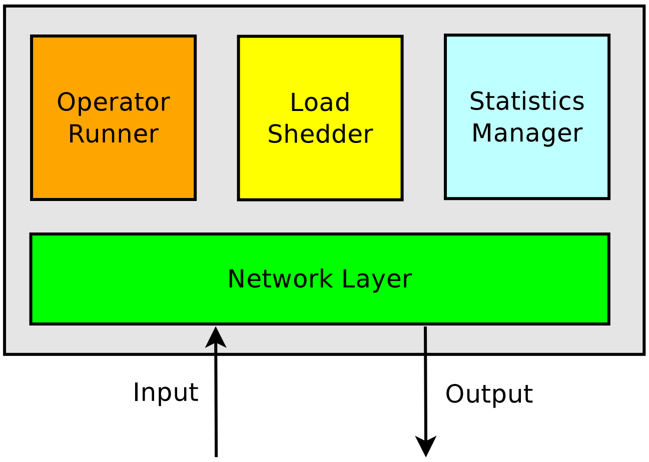

Figure 4.3: A high level view of a processing node, with its internal components.

### Network Layer

The network layer is the component that is responsible for all the incoming and outgoing communication of a processing node. It is composed of two threads: the *NioConnector* thread handles all network communication, and the *RequestHandler* thread interprets the content of the received messages and acts upon it. We employ a many-to-one design, in which many remote connections are handled by a single pair of receiving threads because it offers better scalability. In this way, the number of active threads in a processing node is constant, thus 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, which preserve the integrity of a network message. 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 handle all the I/O requests. This many-to-one design, in which one thread is responsible for all sockets, is preferable to a one-to-one approach, with one thread for each open socket, due to its simplicity and efficiency. It only employs one CPU core to handle network communication. Empirical evidence shows that, even under heavy load and the presence of a large number of connections, the NioConnector thread never becomes a performance bottleneck because it only reads or writes network data. As soon as an incoming message was fully read, it is placed in a queue of pending messages, waiting to be interpreted by the RequestHandler thread.

The *RequestHandler* thread processes incoming messages. It waits for the NioConnector thread to place them into its queue and handles them as soon as they become available. All inter-node communication in the system takes place through *messages*. The chosen format for messages is UTF-8 text, with binary chunks expressed in Base64 encoding. This simple format allows for easy debugging, even though a binary representation would be more efficient. The initial task performed by the RequestHandler thread is to interpret the beginning of the message to categorise it and process it accordingly. A message can be of three kinds: it can be a message containing *tuples* to be delivered to some subquery for processing; it can contain a *command message* that triggers the execution of a remote procedure call; or it can be the request to access the *web interface*.

**Tuple Handling.** If a message contains a tuple payload, it is directly passed to the *operator runner* component without further processing. The content of the message remains in the network format (i.e. it is passed as a string). This follows the principle of *lazy deserialisation*, which states that the conversion from the network format of tuples should happen as late as possible. Tuple objects are instantiated only when they are scheduled for processing. Even their destination (i.e. the queries they belong to) is not known until then.

The reason for this design choice is that instantiating tuples and determining their destination is a costly process. In particular, it may be unnecessary to perform it early, when at a later stage the whole batch may be discarded by the *load-shedder*. Delaying this operation as much as possible ensures that no resources are wasted to instantiate and route tuples that are never processed. Our empirical evidence suggests that an early deserialisation also poses a significant processing burden on the RequestHandler thread, which is unable to handle all messages in a timely fashion under heavy load.

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

Command messages can also be used to obtain information from another node or the oracle. For example, when a coordinator instantiates 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 on different nodes need to connect to each other, it is typical for a node to wait a certain amount of time for the other node to be ready to accept the 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 example, needs to be aware of the final SIC value achieved by all queries that are running on it. By using the global SIC value achieved by every query and comparing it to the local values of the tuples that it processes, the load-shedder can implement an intelligent load-shedding policy. Therefore, every coordinator periodically disseminates the global SIC value of its query to all the nodes hosting subqueries.

**Web Interface.** The RequestHandler thread is also the gateway to the node's *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 *statistics manager*. This includes the number of hosted subqueries together with their performance as well as some global metrics describing the status of the node, such as its average throughput and latency. When 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

After a batch of tuples was received, it is passed to the *operator runner* for processing. The operator runner employs a fixed number of threads, each executing a chain of operators at a 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, which can freeze the queue of pending jobs and analyse it to make the load-shedding 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. It 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 that it was assigned. The last operator of the subquery graph is a RemoteSender. It 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 frees its execution thread so that a new work unit can be processed.

**Thread Pool.** At the core of the *operator runner* is a *thread pool*, ready to execute work units as they are submitted. It is a subclass of ThreadPoolExecutor that augments its parent class with the capability of stopping and resuming execution, which is needed by the load-shedder to inspect the pending jobs queue. The number of threads is set to be equal to the number of available CPU cores. The pending jobs queue is a list of work units. As soon as one of the threads in the pool becomes 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. This increases the latency of processing and eventually leads to an exhaustion of memory. For this reason, the thread pool is interrupted periodically by the load-shedder, which inspects the pending batches and decides if there is the need of discarding a certain portion of them in order to overcome overload. The choice of which ones to discard is part of the shedding policy of the load-shedder.

**Work Unit.** A work unit is a container class that has 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 is executed by a thread from the pool. It starts by deserialising and instantiating the tuples contained in the message received from the network. It then checks to which query it should be delivered. In case there are multiple recipients, it creates a copy of itself for each query interested in its payload. After that, it calls the `process()` function on the first operator of the query graph. This function takes the incoming tuples and executes the operator logic, producing one or more batches as 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 continues processing as long as possible instead of creating a new work unit for each operator. This permits 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 is not affected by the load-shedder. An approach with one work unit per operator may result 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 recipient, such as in a fan-out query, the current work unit continues processing on a single branch, while creating a new work unit for each of the other operators.

### Load-Shedder

The *load-shedder* is the component that carries out the *overload management* of a processing node. When the amount of input tuples rises over a given threshold, the resources available at the node become insufficient for processing. More tuples are given as input to the node than what it can process, thus leading to the accumulation of jobs in the pending queue. This causes a growing increase in latency of the tuples, until the available memory is exhausted.

**Periodic Evaluation.** The *CheckOverload* thread periodically runs and evaluates the current load situation of a node. In the DISSP prototype, a *shedding interval* of 250 milliseconds was chosen, which allows a timely response to overcome an overload condition while keeping a low performance overhead. Every time the CheckOverload thread executes, it calculates the number of tuples that the node is 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 no tuples are discarded. If the number of waiting tuples is larger than the ones that the node manages to process, a certain amount needs to be discarded.

The system retains a record of its *output rate* (i.e. the amount of tuples processed within one shedding interval). It uses this to calculate the average time needed to process a single tuple called the *tuple cost*. Since the shedding interval is fixed, it is possible to calculate the number of tuples that the system can process before a new shedding iteration by dividing the *shedding interval* by the *tuple cost*. This provides the number of tuples *to keep*. Subtracting this number from the *total number of tuples waiting*, the system obtains the *number of tuples to be discarded*.

These numbers depend on many factors. For example, the number of tuples received by a subquery can be different in a given interval, or the query semantics may 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 for the node's processing capacity (i.e. the forecast number of tuples to process in the next interval) over a few prior iterations thus reducing the variance of these values. In summary, the periodic cycle followed by the load-shedder is as follows:

1. Pause the thread pool
2. Calculate the number of tuples to be discarded
3. Choose which 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 obtain the number of tuples that it contains in its pending queue at this moment. The processing of tuples cannot happen during the execution of the load-shedder so that it can choose what tuples to discard from a static set. After that, it estimates the number of tuples to be discarded, using the logic described in the previous paragraph. Once the number of tuples to retain is calculated, the load-shedder can make a decision about which tuples should be discarded out of the ones available. This choice depends on the *shedding policy*. A more detailed description of different shedding policies will be given in Chapter 5.

Then the actual tuples are discarded by removing the corresponding work units from the thread pool queue. In the DISSP prototype, the granularity of removal is at the level of *batches* because 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 is restarted and the processing of tuples continues. Before ending its cycle, the load-shedder calculates the new updated values for the current *tuple cost* and other internal metrics. Once it finished executing, it calculates the time that it took for processing, called the *shedding time*, which it subtracts from its fixed interval of execution, called the *shedding interval*, obtaining the amount of time to sleep before its next execution.

### 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 it are used to improve the overall performance of the system. The load-shedder can exploit its knowledge of the local utility of a query to implement a fair shedding policy, trying to provide the same processing quality 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 nodes that are already heavily loaded.

The statistics manager runs periodically, by default once a minute, and calculates a number of statistics, some of which are logged locally and some are propagated to the coordinator or the oracle. The following is a list of the most important 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 because this value is propagated from the coordinator to all the processing nodes. This metric is sent to the oracle and can be used during the deployment of a new query. For of a fair deployment strategy, a node already hosting parts of queries achieving a low global SIC value should not be chosen to host a new subquery. Adding further load to the node may lead to an overload condition, thus further reducing the processing performance of the currently running queries.

**Average tuple drop.** *The average number of tuples discarded by the node in a chosen time interval.*
This value helps quantifying the amount of overload experienced by a node. It is sent to the oracle and can be used by a deployment 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 quantify the spare processing capacity of a node. By recording the value of this metric when encountering an overload condition, it is possible to have an indication of the number of tuples per second that this node can process on average without loss. This value is sent to the oracle and allows a deployment strategy to choose the least loaded nodes out of those currently not experiencing overload.

**Average processing 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 to the next node. This value is sent to the oracle and can be used to discover performance bottlenecks and to evaluate the quality of the chosen deployment strategy.

**Number of running subqueries.** *The number of subqueries running on a node.*
This metric can be used to check if the deployment strategy leads to an even deployment or to a skewed distribution of subqueries. It is sent to the oracle and can also be used to assess the quality of the chosen deployment strategy.

**Number of connected subqueries.** *The number of subqueries fully connected.*
This metric can be used to check the correct deployment of a query. Every subquery contains at least one incoming network operator (RemoteReceiver) and one outgoing network operator (RemoteSender). If the query deployment was successful, the number of established connections should be equal to the number of these operators. A discrepancy indicates that one or more queries are disconnected either temporary or permanently, which may have been caused by a network partition or a neighbour node has crashed.

## 4.4 Life Cycle of a Query

This section describes the workflow of a query, from when it is submitted to when it starts processing. It shows how different aspects of the system are realised by describing 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. After a query has been instantiated, it is broken down into subqueries employing a flexible partitioning policy. Each subquery is deployed on a processing node, chosen from a set of suitable candidates provided by the oracle. Finally, the different subqueries are connected to each other, tuples start flowing from the input providers and the processing begins. Figure 4.4 shows the steps involved in the deployment of a query.

1. First, the query graph is submitted in XML format. It contains the query specification with a list of tuples and operators to be used. It also has information about the links among operators.
2. The *submitter* compiles and instantiates the XML query specification, creating the tuples and operator objects. Next, it starts a *coordinator*, which is the entity in charge of deploying and monitoring the status of the query.
3. The *coordinator* creates $N$ subqueries from the original query graph according to a user defined partitioning policy. Next, it contacts the *oracle* to obtain a set of $N$ nodes for deployment.

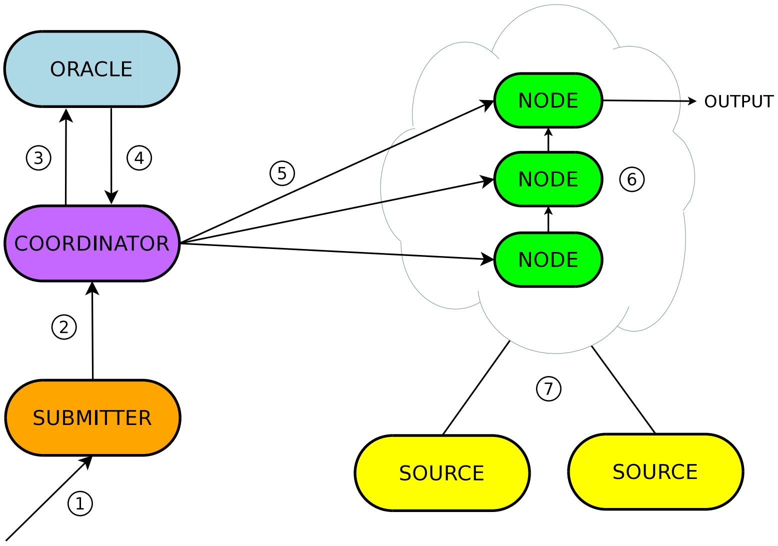

Figure 4.4: Steps involved in the deployment of a query.

4. The *oracle* returns a list of the $N$ most suitable nodes available for deployment. This choice is dictated by a system-wide policy, such as choosing the least loaded nodes or nodes with a low network latency between them.
5. Once the *coordinator* receives this information, it proceeds by deploying the individual subqueries onto the provided *processing nodes*.
6. Each *processing node* instantiates the assigned subquery and connects the query, establishing the required remote network connections between the 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.

### 4.4.1 Submission of Queries

The life cycle of a query begins with the submission of its query plan by a user. This is done through the *submitter* component. It 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 realised in the DISSP design.

#### Compiled Operators and Tuples

In the DISSP system, tuples and operators are implemented as *compiled Plain Old Java Objects (POJO)* [PJO12]. This choice was made taking performance into consideration because other options, based for example on *reflection*, are slow and can cause 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 contained in the XML query specification that describes the query producing a new Java source file, which is then compiled to bytecode by a run-time compiler.

**XML Query Representation.** Queries are submitted to the system in an XML representation. An XML query file contains the complete description of the query. It specifies what kind of tuples are processed and provides a description of the *tuple schemas* so that operators are aware of the number, type and name of the fields contained in the tuples that they process. It also includes the list of operators implementing the query semantics. Each operator is represented by an XML block, containing its description.

An operator needs to be specialised before it can be used in a query. A generic implementation is provided in a *template file*, which acts as a blueprint for the operator. It is then completed with the data included in the XML block, allowing the correct instantiation of the operator: its *name, parameters* and an indication of its *subsequent operator.* An operator either declares itself as a *terminal* operator by having no subsequent operators, or as an *intermediate* operator with its output used as input to another operator.

Based on this information, the system reconstructs the complete query graph. The XML query representation is used to generate a complete set of Java source files that define the customised tuples and operators that are used in the query. These files are then compiled and instantiated so that the query can begin processing.

**Templates.** Semi-complete Java source files represent the skeleton of a class and are used for the efficient instantiation of customised *tuples* and *operators.* All tuples share common characteristics but differ in the number of fields, their names and types. The same is true for all operators that belong to the same type. For example, all average operators have the same processing semantics but have different names and types of the fields to be averaged.

A generic blueprint for these operators comes from the generalisation of a specific instance, in which specific names and types are replaced by textual *placeholders.* Using the information contained in the XML query description, it is possible to substitute these placeholders with the correct details, transforming a template into a complete Java source file ready for compilation. Placeholders are all capitals keywords preceded by a dollar sign in the form "`$PLACEHOLDER`". They are replaced using the information provided in the XML file describing the query and the compiled into actual Java objects with the required characteristics. For this transformation, the system uses a *CharSequenceCompiler*, which takes as input a string with the content of a completed template file and produces an instance of a customised object. This means that the system operates on compiled bytecode, which combines a high execution efficiency with the flexibility of working with customised versions of tuples and operators tailored to the query requirements. Tuples are also implemented using a template file. A parent class called `Tuple` is provided that contains the basic processing logic common to all tuples, and every tuple class is derived from it. In the XML query description, the user specifies a schema and a name for the tuples that are used by the query, and the system creates a new tuple class with the required name and fields. The generic tuple template file is then completed, substituting the placeholders with the provided data, thus producing the Java source file of the new tuple class.

Listing 4.1 shows the XML description of a Tuple object with three fields: a `long` field for the timestamp named *ts* and two `double` fields, one with a numerical identifier *idx* and the other for a temperature reading *tmp*. These values are inserted in the tuple template in correspondence with the “`$FIELD`” placeholder, transforming it into a complete Java source file. As a result, the compiled object contains three public fields with the types and names from the XML listing.

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>
```

Listing 4.2 shows the XML description of an average operator. In the first line the operator is defined as being an instance of class `Average`, described in the corresponding template file, and it is given the name `MyAvgCpu`.

Next, there is the declaration of a *subsequent* operator, which means that this is not a terminal operator. Its output should be delivered to a single local operator named `MyOutput`. This is followed by three parameter definitions in the form of $\langle name, value\rangle$. The system considers the `$NAME` placeholder and replaces it with the string given by `value`. After this substitution, the template file becomes a complete Java source file and is 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>
```

The DISSP system provides a number of general-purpose operator templates: *window operators* provide transformations over windows, holding the input of an operator until a certain condition is reached; *I/O operators* act as gateways to the system and provide the conversion from external to a system tuple representation (and vice versa); *network operators* support inter-node communication so that a query can be partitioned and distributed across several nodes; and *data manipulation operators* are used to process the data and are equivalent to the *relation-to-relation* operators in CQL.

### 4.4.2 Partitioning of Queries

After a query was instantiated based on its XML representation, it is broken down into chunks called *subqueries*. These are computationally less intensive than the complete query and can be deployed across a set of *processing nodes*. The partitioning process can follow different strategies. This section presents two policies: one that takes into account the processing *load* of the newly created partitions and another that multiplexes particularly heavy partitions over several nodes, executing them in parallel according to a horizontal partitioning approach.

**Load-Aware Partitioning.** The partitioning of a query graph can follow a strategy that groups operators with similar processing loads. Such a *load-aware* strategy tries to obtain subqueries with similar costs in order to spread evenly the processing load. It is similar to the one described in [Xin+06]. Since the query partitioning occurs 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 is calculated for an operator by analysing its direct predecessors in the query graph and their window operators. The estimate can be more precise in the case of *aggregate operators*, such as average, because they produce a single output every time that they are triggered. For *filters*, the estimate is more difficult and must take their selectivity into account.

The load coefficient of an operator has to be determined offline and estimates the computational complexity of the operator. A synthetic load of fixed size is input to the operator on a testbed machine and its CPU load is measured.

By multiplying the input rate and the load coefficient, it is possible to estimate the future load of an operator. The initial partitioning can then divide the query graph evenly. Note that this should only be thought of as a starting point to be augmented with the run-time migration of operators or subqueries. This accounts for inaccuracies in the initial estimates.

**Horizontal Partitioning** The cost of a query is not only determined by the number of operators that it employs. Even queries with a simple query graph can be computationally expensive due to a high rate of incoming tuples or the complexity of an operator. In such cases, the partitioning of the

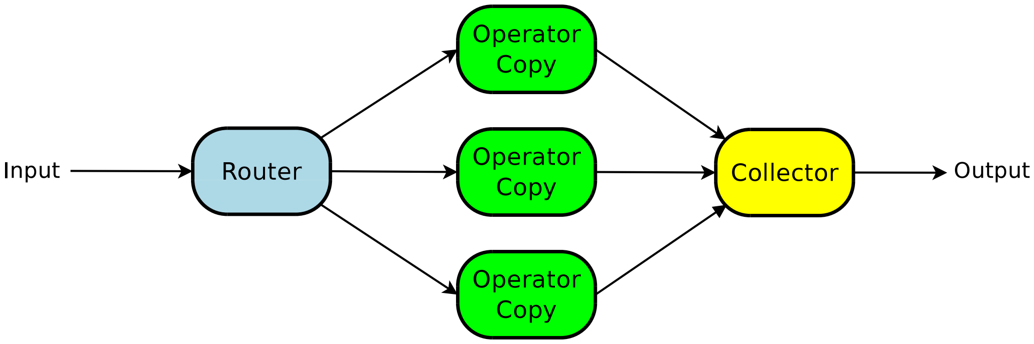

Figure 4.5: Horizontal partitioning of an operator.

query it is not limited to the grouping of operators into subqueries but includes the rewriting of the query graph. The costly operator, or subquery, is decomposed so that multiple copies of it can run in parallel on several processing nodes. The original input data is split by a *router* operator that sends the chunks in turn to the nodes hosting the decomposed operator copies. This approach performs a horizontal scale out of the operator. The output of these operator copies is sent to a collector operator, which finalises the processing to achieve the original semantics of the unpartitioned operator.

For example, a costly *average* operator could be decomposed into three copies, each calculating the average for a 1/3 of the input. The final average is then calculated by the collector based on the partial averages. Other operators, such as *filter*, do not need a further collector phase, and only take a union of the partial results to generate the final output.

Figure 4.5 shows the modified query graph of a query rewritten using the horizontal partitioning strategy. The original query with one expensive operator is transformed so that three copies of the operators run on different processing nodes. A round-robin router operator evenly splits the incoming tuples among them. A final replica of the operator, an aggregate in this case, collects the partial results and outputs the final value.

## 4.5 Summary

This chapter presented the design of the DISSP prototype system, a stream processing engine designed to realise 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 efficient processing under overload, the embedded calculation of the SIC metric and the support for adapting the load-shedding policy according to user needs. It explained how the theoretical concepts presented in the description of the data model can be implemented in the basic components of a stream processing system.

The chapter then described the *system-level components*, in particular their roles and their interactions.

In every DISSP deployment, queries run on one or more *processing nodes*. These process tuples provided by a set of *input providers* that convert external inputs to an appropriate system format. New queries are introduced into the system by a *submitter*. For each query a *coordinator* is spawned that is in charge of its deployment and management. An *oracle* oversees 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 processing nodes. Every node exchanges command messages and tuples with other nodes and the oracle through a *network layer*. An *operator runner* routes tuples to the assigned subqueries and manages their processing through the graph of operators. A *load-shedder* overcomes overload by selecting tuples to be discarded, while a *statistic manager* keeps track of important metrics about the performance of a node.

The chapter ended with the description of the deployment of a query. It used this workflow to describe the detailed steps involved from the submission of a query to when it starts processing. It explained how tuples and operators are compiled at run-time from an XML query description, and how the original query graph is partitioned into subqueries to be deployed on multiple processing nodes.

The next chapter focuses on the load-shedding process in more detail, presenting a *fair shedding* algorithm that exploits the SIC metric to allocate resources among queries evenly. The goal is for them to all achieve the same performance in terms of the quality of the computed results.

# Chapter 5

# Quality-Aware Load-Shedding

This chapter describes the overload management techniques employed by the DISSP system. In particular, it focuses on the *load-shedder* component and explains how it uses the SIC values to make semantic load-shedding decisions about which 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 which tuples to keep and which to shed leverages the information contained in the SIC values of tuples to implement an intelligent and fair load-shedding policy.

Using the SIC values, it is possible to implement a *fair shedding* policy that attempts to penalise all queries in a similar fashion. In this context, we define fairness based on processing quality (i.e. a system is fair if all queries achieve the same SIC values for their results). Before introducing the load-shedding algorithm, the main components of the overload management infrastructure are presented.

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

Load-shedding is carried out in a system that is already overloaded so that it can keep on processing even after some of the input was discarded. In a scenario in which the processing node runs 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 finishes with a description of the details of the load-shedding mechanisms employed by the DISSP system.

## 5.1 Fairness in Load-Shedding

A processing node is considered overloaded when it cannot process all the data that it receives as input in a timely fashion. If, in a given time interval, the amount of incoming tuples (i.e. the *input rate*) is larger than the number of tuples that the system can process (i.e. the *output rate*) the processing capacity of that node is exceeded. When this occurs, the internal queues of pending tuples increase in size. While such a condition can be sustained for a short amount of time, it leads eventually to an exhaustion of memory if it is not addressed. An overload condition can be caused by an increase in the rate of incoming tuples or by a change in the processing cost of tuples.

Augmenting streams with SIC values allows the system to reason about the quality of tuples. Our quality metric is designed to capture the amount of information lost during the creation of a tuple, and thus its contribution to the query results. As a result, the system can predict the impact of discarding a given tuple on the quality of the final results of a query. This allows for the implementation of an *intelligent shedding policy.* Section 5.3 explains the details of such a *fair shedding policy*, which attempts to equalise the resulting SIC values of all queries.

In a shared infrastructure, the available processing resources are divided among potentially 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 across all queries. Employing the SIC metric to quantify the reduction in information content in the results allows for 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 performs load-shedding is considered to be fair if it discards tuples in a way that minimises the difference in the SIC values of results for all running queries.*

This definition does not depend on the amount of resources consumed by a query or on the number of operators that it contains. Each query is treated equally and allocated an amount of resources that is proportional to its needs. This means that each query should achieve the same quality of processing. A policy that allocates the same amount of resources to all queries would favour small queries, while expensive queries would be penalised. This would lead to a large spread between the SIC values of the light and the heavy queries. The goal of a fair system, according to the above definition, is instead to equalise the SIC values of all queries, trying to reduce the differences among queries as much as possible.

## 5.2 Abstract Shedder Model

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

### Input Buffer

The tuples that are received by a node are stored in an input buffer waiting to be processed. This allows the system to continue receiving tuples from the network, even though it is unable to consume them at the same rate. The order in which tuples are stored 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 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, the overload detector invokes the tuple shedder to control its size.

An input buffer also allows the system to implement a *semantic shedding policy.* Instead of discarding tuples as they arrive, the system stores them first in the input buffer and leaves the decision about which tuples to discard to the tuple shedder. This component implements the shedding algorithm that chooses which tuples to discard among those awaiting processing in the input buffer.

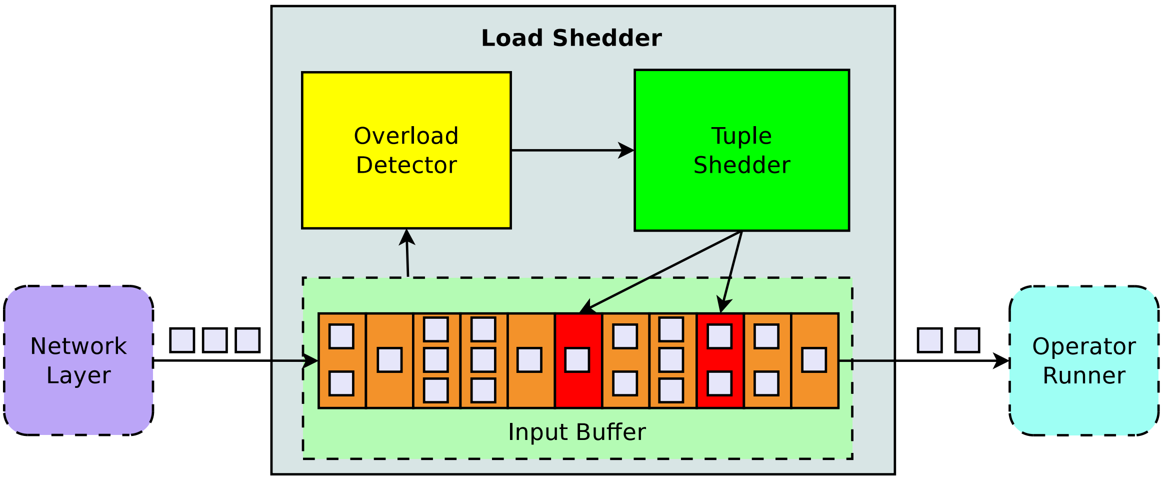

Figure 5.1: Abstract representation of a load-shedder, showing its internal conceptual model.

### Overload Detector

The overload detector is responsible for monitoring the load of a processing node. It detects an overload condition before it becomes critical and invokes the tuple shedder to discard a fraction of the input in order to contain the load within the operational limits of the node. Periodically, at fixed *shedding intervals* $\delta$, the overload detector checks the size of the input buffer. The shedding interval is set as 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 (e.g. 250ms). On the other hand, setting the value too low induces unnecessary overhead due to frequent shedder invocations.

After each shedding interval, the overload detector makes a prediction about the number of tuples that the system is able to process during the next interval, which is referred to as $N_{keep}$. In order to estimate this amount correctly, the overload detector uses a *tuple cost model*, as 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, $N_{keep}$, is the number of tuples to be discarded by the tuple shedder, $N_{discard}$.

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

If $N_{discard}$ is negative, there is no need for shedding during this interval. If its value is positive, the tuple shedder is invoked. According to a shedding policy, the tuple shedder then calculates which tuples to discard among those available in the input buffer.

### Tuple Shedder

After the overload detector identified an overload condition, it invokes the tuple shedder to discard a certain number 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 of which tuples to discard is made by the tuple shedder. It inspects the input tuples and selects 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 because load-shedding decisions are made later. Each tuple received by a node awaits processing in the input buffer until it gets either selected for processing and passed to the *operator runner*, or discarded by the *tuple shedder*. The *overload detector* decides if the amount of tuples currently in the input buffer can be managed by the operator runner or if instead some tuples should be discarded to avoid overload. When the overload detector estimates that $N_{keep}$ tuples should be discarded, it invokes the tuple shedder passing $N_{keep}$ as a parameter. At this point, the tuple shedder chooses $N_{keep}$ tuples among those currently available to be saved. This decision is made based on a *load-shedding policy*, which is implemented within the tuple shedder. The simplest policy is a *random* one, in which tuples are discarded arbitrarily.

Augmenting tuples with the SIC quality metric enables 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 permits the implementation of a *semantic shedding* algorithm, which can be used to maintain certain properties among queries. The next section describes the design of a *fair shedding* algorithm, which aims at penalising all queries in a similar fashion.

## 5.3 Quality-Aware Fair Shedding

This section describes the implementation of a *fair shedding* policy, following the principles stated in Section 5.1. The goal of this policy is to achieve a fair 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) result SIC values are numerically close. The tuple shedder addresses overload conditions by periodically discarding batches, enabling the node to retain a sustainable throughput of processed tuples with low queueing delays.

When the shedder is invoked by the *overload detector*, it selects $N_{keep}$ tuples to keep from the its input buffer queue. At first, the shedder assumes that all batches must be discarded. Next, it gradually admits batches that belong to 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.

Next, we present a *tuple cost model* to estimate the processing capacity of a node. The cost model is used by the overload detector to calculate if the node is overloaded and to predict the number of tuples that the system can process before the next shedding interval. We also present a description as pseudocode of the algorithm realising the fair shedding policy, providing a detailed explanation of the involved steps.

### Tuple Cost Model

A processing node is overloaded when the aggregate resource demand for executing hosted operators of queries exceeds the resource capacity of the node. When the rate of tuple arrival exceeds the processing capacity of the node, the size of the input buffer 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 a query. We then use the 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 the time that the node spent on average processing a tuple from a query. For an accurate estimation of $C_T$, we use the measured time interval $t$, instead of the fixed interval $\delta$, which 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 may differ from $\delta$. To compensate for any such differences and to have an accurate estimation of $C_T$, we use the measured elapsed time $t$ rather than the fixed interval $\delta$.

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

$$N_{keep} = \delta/C_T \tag{5.3}$$

where $\delta$ is the fixed time interval between shedding invocations, and $C_T$ is the average time spent processing a tuple. To estimate the value of $N_{keep}$, we use a sliding window to avoid transient rate fluctuations that would lead to under- or over-prediction of the true number of tuples that a node can process. Employing a sliding window allows for a smoother estimate, which is more resilient to sudden changes of the incoming tuple rate.

Figure 5.2 shows the two time intervals, $\delta$ and $t$ used to calculate the cost of a tuple and the average time needed to process an arbitrary tuple, respectively. Let us assume a shedding interval $\delta$ of 250 milliseconds. In practice, the *overload detector* triggers with a small delay, e.g. after a time $t$ of 300 milliseconds. 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 $\delta$ interval it will have the processing

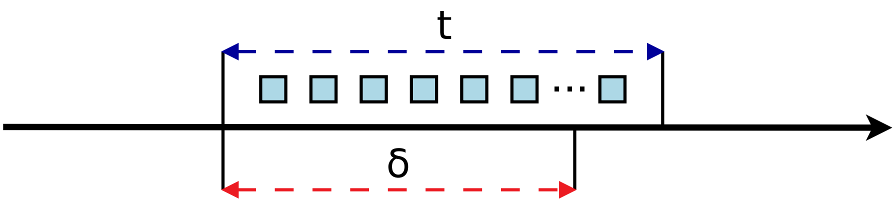

Figure 5.2: Graphical representation of the time intervals $\delta$ and $t$ used to calculate the *average tuple cost* $C_T$ needed to predict the *future processing capacity* $N_{keep}$ of a node.

capacity $N_{keep}$ of approximately 4100 tuples. Tthis value is smoothed by a sliding average.

More complex cost models were proposed in the past [WR09; JNV03]. These approaches are tailored for certain types of operators, such as joins, and require the tuning of 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, as some operators may require significantly more time to process tuples depending on their contents. A more precise cost model would also take the semantics of the query into account, e.g. considering the types of operators.

### Quality-Aware Fair Shedding Algorithm

The algorithm used by the tuple shedder to implement the fair shedding policy is specified using the pseudocode in Algorithm 1. The main idea behind it is to start by considering all tuples as discarded. Next, it estimates the reduction in SIC values experienced by each query, making a ranking. It iterates, adding one batch of tuples at the time, and chooses the one with the highest SIC contribution from the query with the lowest estimated SIC value. After each step, it recalculates the ranking and it retains tuples until it reaches the limit given by the overload detector. The rest of this section provides a more detailed description of the algorithm.

First, the input buffer queue is locked by the tuple shedder (line 1). This means that the system stops accepting new incoming batches and current batches in the queue are not processed by operators. While the shedder runs, new incoming batches are held temporarily in a secondary buffer, which is merged with the input buffer queue before the node resumes processing. This allows the *tuple shedder* to consider a snapshot of the current situation so that it reasons based 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, which reduces the number of iterations and thus the associated overhead.

First, it updates the current result SIC values 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). After that, it adds all active queries (i.e. ones that have at least one pending batch) to a priority queue sorted by increasing projected SIC values. In this way, it can quickly identify the query with the lowest projected performance so that a batch from it can be retained.

Next, the shedder removes all batches in the input buffer. It moves them to a temporary buffer and groups them according to the query that they belong to, while keeping an ordered list of queries and batches for each query (line 7). One by one, all batches are moved 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 values. Once the tuple shedder decides to save one more batch of a certain query, it can quickly select the one with the highest SIC value, and thus the most valuable one, by simply removing the head of a priority queue.

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

**Algorithm 1:** Tuple shedding with SIC fairness

```
   Input  : number of tuples to save ($N_{keep}$), input buffer (IB)
   Output: input buffer (IB) containing $N_{keep}$ 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 $Q_{SIC}$ as if $batch$ was discarded
 7     move $batch$ to priority queue of queries (increasing $Q_{SIC}$) and for each query keep batches
       in descending SIC order
 8     remove $batch$ from IB
 9 $n'_{\text{tuples}} = 0$
10 while $N'_{keep} < N_{keep}$ do
11     Q := pick query with the least $Q_{SIC}$
12     b := pick batch from Q with highest SIC
13     add b to IB
14     update ordered list of queries
15     Add tuples in b to $N'_{keep}$
16 unlock IB
```

Next, the shedder considers which tuples to admit from each query to achieve fairness lines 10–15). It first selects the query that would be penalised the most (i.e. the one with the lowest projected SIC value) (line 11). It then selects the batch with the highest SIC value for this query (line 12). By selecting the batch with the highest SIC value among those available for each query, the system maximises the projected SIC value, adding first the batches with the greatest contribution.

The shedder moves the selected batch back to the input buffer for processing (line 13). Since the batch is not discarded, the shedder also updates the projected SIC value of the query whose batch was saved and the priority queue of queries (line 14). The shedder repeats this process until it has accepted $N_{keep}$ tuples. Any remaining batches from the temporary buffer are discarded and regular tuple processing from the input buffer queue resumes (line 16).

## 5.4 Implementation of the DISSP Load-Shedder

This section provides implementation details about the overload management system that is part of the DISSP prototype system. It describes how the abstract components are realised in practice and outlines some of the mechanisms that have been employed to achieve robustness under overload.

**Input Buffer.** The input buffer 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 that each bundle a batch of tuples and a reference to their destination operators and represent future processing units. When the overload detector identifies a critical load situation, it calculates the maximum length of the input buffer and triggers the tuple shedder to remove the excess tuples from the input buffer.

**Overload Detector.** The overload detector is implemented by the CheckOverload class, which realises a monitoring thread that performs a check on the current system load at each shedding interval. This thread maintains certain statistics about the performance of the system and acts if it detects an overload condition. It calculates the average *tuple cost* (i.e. 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 that the system will be able to process during the next shedding interval, thus inferring the maximum number of tuples that should be present in the input buffer. If there are currently more tuples awaiting to be processed in the queue, it invokes the tuple shedder that chooses which tuples should be discarded in order to maintain a sustainable system load.

**Tuple Shedder.** The tuple shedder is the class implementing the load-shedding semantics. It contains the algorithm that chooses which tuples to discard from the ones currently awaiting processing in the input buffer. It exposes a simple interface to the overload detector, which includes a parameter with the number of tuples to keep in the buffer. The internal shedding policy analyses the input buffer queue and decides which tuples to discard. By hanging the class implementing the tuple shedder interface, it is possible to realise different shedding policies. In its simplest implementation, it discards tuples at random. However, by exploiting the SIC quality metric, it is possible to implement a semantic approach, such as the fair shedding algorithm 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, the data remain in network representation and needs to be converted first into concrete objects before it can be processed. This *deserialisation* step of the input tuples is costly and, therefore, delayed as much as possible in order to avoid spending CPU cycles on tuples that may never be processed. The system deserialises only the minimum required information, such as the query id to which the tuples belong and their SIC value. Using this information, the tuple shedder executes its shedding policy and decides if a batch of tuples is valuable enough to be preserved. 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 the WorkUnit from the input buffer.

**Sliding Windows.** The load experienced by a stream processing node can vary dramatically over time. As a consequence, when considering a few hundred milliseconds, the number of tuples delivered to the node can be highly variable. It is important that the overload detector does not mistake the arrival of an isolated large batch of tuples, which can be safely kept in the input buffer, with the beginning of a long term overload condition. For this reason, when implementing the tuple cost model, the system uses a moving average calculated over the last $N$ values of the metric. This adds slack in the reaction of the system and allows it to distinguish between a sudden load spike, which can be ignored, and a sustained change in the load, which must be addressed.

## 5.5 Summary

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

# Chapter 6

# Evaluation

This chapter presents the experimental evaluation to demonstrate the advantages of employing the SIC quality metric in a stream processing system. All experiments are 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 continuously overloaded stream processing system. The SIC metric is designed to be operator agnostic. It can be applied to a general purpose system supporting any type of query, providing an indication of the quality of the computed results. This chapter considers the following research questions. 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?

In addition, the SIC metric is used for the implementation of semantic shedding policies. We design a fair shedding policy and compare its performance to that of a random shedder. Does such a policy achieve better quality results? Is it more fair in the allocation of system resources?

We were also interested in evaluating the performance of such a fair shedding policy in terms of scalability, both when varying the number of nodes and the number of deployed queries. What is the behaviour of the fair shedder when the amount of processing resources changes? Does its 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 off the correctness of result with a reduction of costs, e.g. by deploying a larger number of queries on the same processing infrastructure. Is it possible to strike a balance between the achieved SIC value and the cost of paying for the processing infrastructure? Adding an increasing number of queries to the system 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 considering a range of sample queries using both synthetic and real-world input data.

## 6.1 Experimental Set-up

This section describes the experimental set-up used in this chapter. We use two testbeds: a local one and the Emulab testbed, as described in Table 6.1. The local testbed consists of three 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 workloads chosen for the experiments belong to two query classes: aggregate (i.e. AVG and COUNT) and complex (i.e. TOP-5 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 different operator semantics. The second class consists of more complex queries used to explore the properties of the SIC metric in workloads with a broader variety of operators, beyond just the aggregate domain. These query types are representative for a variety of data processing applications, such as sensor networks and social media analysis.

| **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 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.

| **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 cycles 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.

The input data generated by the sources contains either real-world load measurements or a specific synthetic workload. The real-world data streams are log traces of 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* distribution, as described in Table 6.3. Furthermore, a *mixed* synthetic workload is used that combines 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 the deployment of queries emulates 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 nodes according to a Zipf distribution [AH02]. Consider, for example, a scenario in which the number of nodes is 20, divided into 4 clusters of 5 nodes each, with a total number of query partitions of 150. According to the Zipf 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 receives 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 (see Table 6.2). The experiments use the synthetic workload as well as real load measurements collected on PlanetLab (see Table 6.3).

For this set of experiments the local testbed is used (see 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 more input tuples. The node uses a load-shedder that discards tuples randomly from each query. The experiment is repeated for every query class and for each of 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:

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

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 $\tau$ is defined as:

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

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 comes from perfect processing, and its 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 show 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 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.

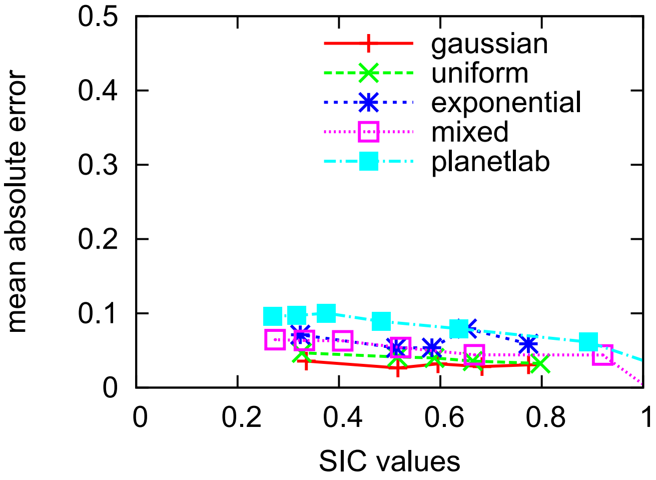

Figure 6.1: Correlation of SIC values with the query output quality for *average* queries.

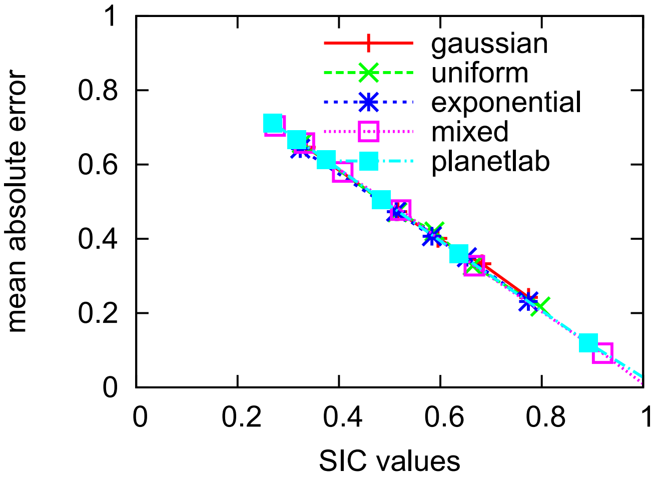

Figure 6.2: Correlation of SIC values with the query output quality for *count* queries.

**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 query.

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 of 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, although the SIC metric is designed as a generic quality metric, it provides a good indication of the quality of the computed results.

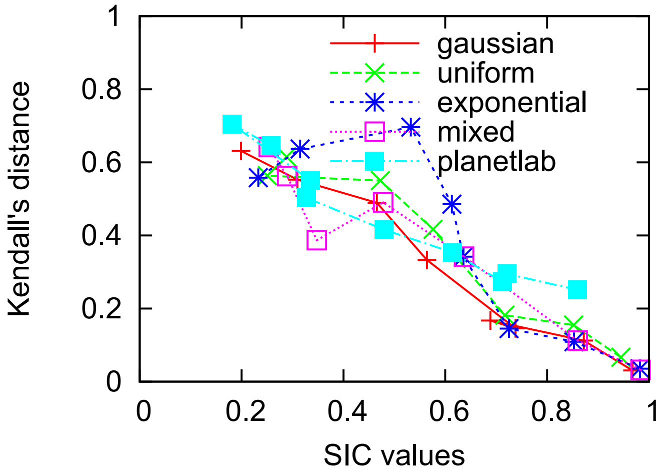

Figure 6.3: Correlation of the SIC values with the query output quality for *top-5* queries.

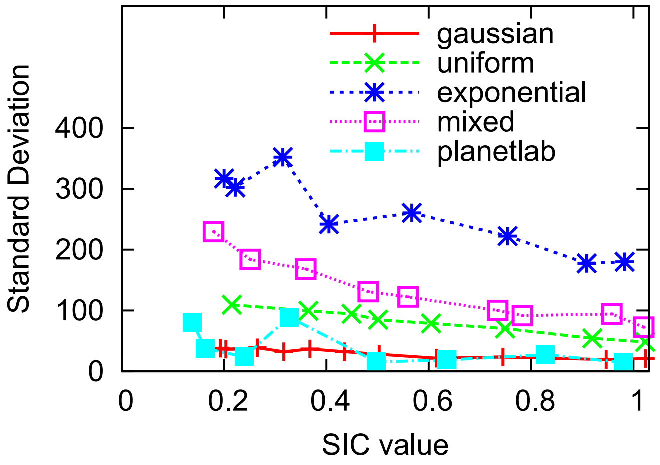

Figure 6.4: Correlation of the SIC values with the query output quality for *covariance* queries.

## 6.3 Load-shedding Policies

This section describes experiments that evaluate the use of the SIC quality metric for 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), which is designed to provide an equal allocation of system resources, with the goal of equalising the quality of processing (i.e. same SIC value) for all running queries under continuous 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 in a way that equalises 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. Our results 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 a sort of dispersion metrics (IQR, Q.95-Q.05 and STD). The experimental set-up for this set of experiments consists of 25 Emulab nodes, as described in Table 6.1. The query workload is a mix of three different queries: average, covariance and top-5, as described in Table 6.2.

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, if the number of partitions per query increases, the total number of queries decreases (see Table 6.4). A larger number of subqueries distributes the load

| Partitions | Queries | Total Subqueries |
|---|---|---|
| 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 partitions.

of each individual query 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 workload characteristics for each experimental run. The first column lists the number of subqueries that each query has been partitioned into, the second column shows the total number of deployed queries, 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 is 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, 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 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 of a data set [Rek69] is defined as the square root of 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; and 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. 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 Figures 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 achieves a lower value compared to the random shedder. This means that the fair shedder chooses a better set of tuples to be discarded, leading to 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.

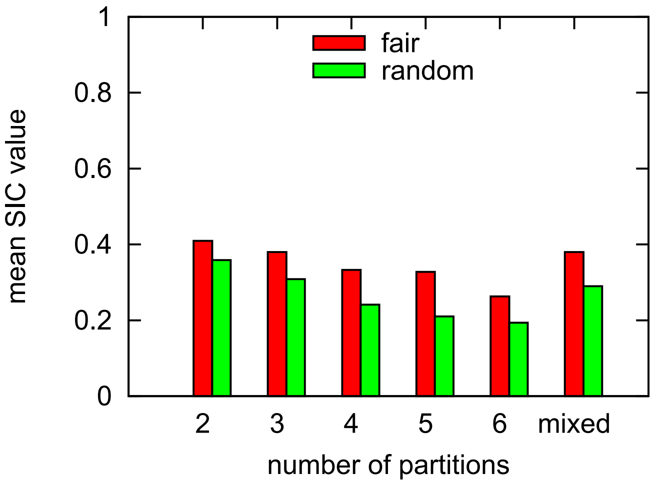

Figure 6.5: Comparison of fair and random load shedders (MEAN).

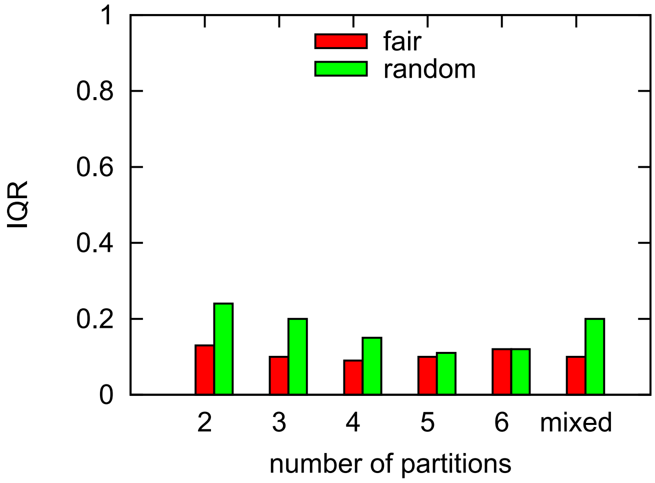

Figure 6.6: Comparison of fair and random load shedders (IQR).

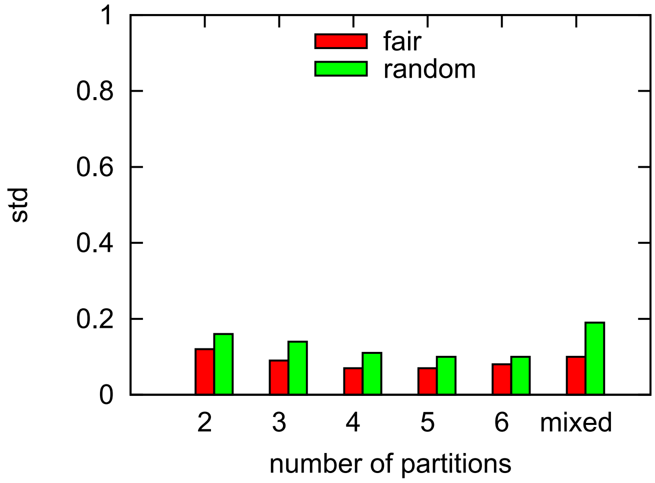

Figure 6.7: Comparison of fair and random load shedders (Standard Deviation).

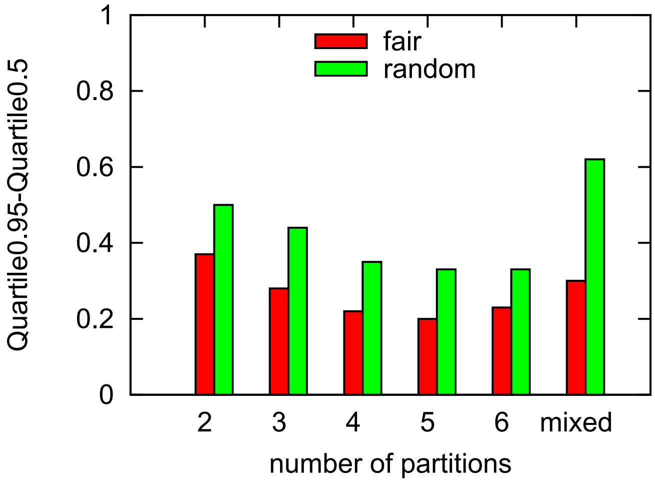

Figure 6.8: Comparison of fair and random load shedders (Q0.5-Q0.95).

### Scalability of Fair Load Shedder

The following set of experiments evaluate the scalability of the *fair load shedder* in terms of the number of nodes and queries. The aim is to observe the variation of the result SIC values if 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 result SIC values because the amount of required load-shedding 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, when varying the amount of processing resources. Increasing the number of nodes with a fixed number of queries, progressively reduces the overload on each processing node. The fair shedder should achieve a proportionally better processing quality in terms of the mean SIC value. Ideally, reducing the number of nodes by 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 the need for load-shedding. The higher amount of available processing resources leads to an increase in average SIC values of the output tuples. All dispersion measures show a reduction, which means that a better result is also achieved in terms of the variability of the output SIC values. The value of the dispersion measures grow together with the value of the mean. If the value of the mean doubles,

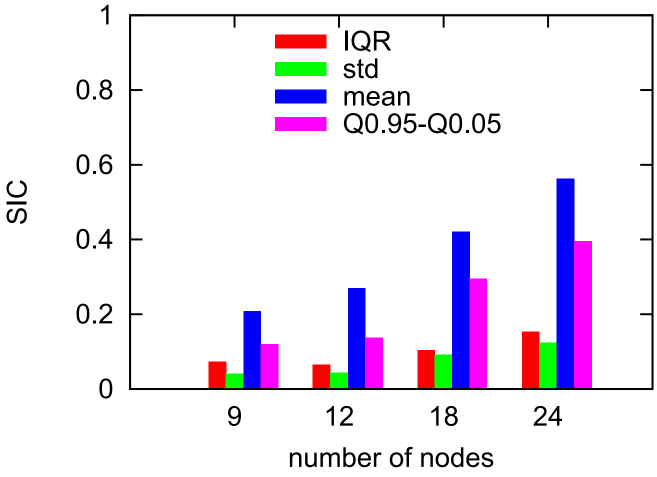

Figure 6.9: Fairness for an increasing number of nodes.

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, the dispersion measures should also be doubled. What should remain constant is the ratio between the dispersion measure and the mean, as observed this stable ratio between the standard deviation and the mean is referred to as *coefficient of variation* [Bla09].

### Increasing the Number of Queries

This set of experiments observes the processing quality in terms of the 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 (see Table 6.1). The nodes host 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 indicate 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 initially the percentile reduction in the SIC value is lower than the relative increase in the number of queries. At a certain point, this changes and the system behaviour becomes sublinear. The next section shows how it is possible to exploit this behaviour to expose a trade-off between the quality of results in terms of SIC values and the resource cost per query.

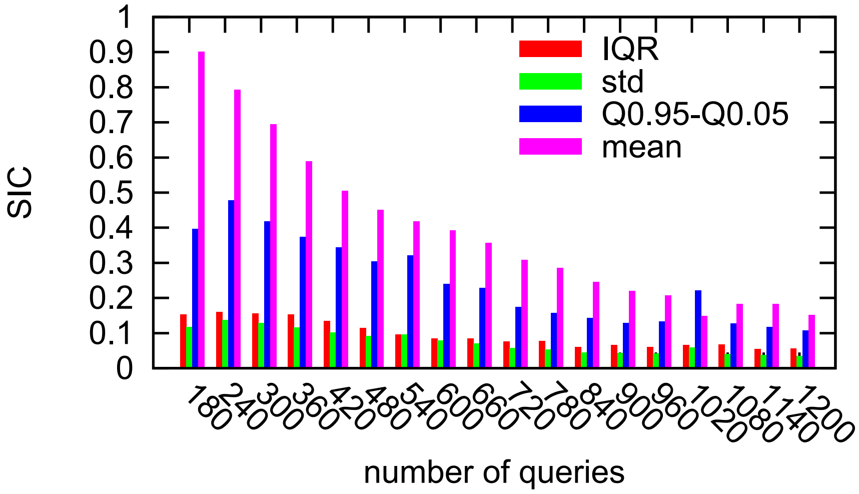

Figure 6.10: Fairness for an increasing number of queries.

## 6.4 Trade-off between Quality of Processing and Resource Cost

When the user is willing to accept approximate results, it is possible to use the SIC metric to expose a trade-off between the quality of processing and resource cost. If we maintain a stable amount of processing resources, increasing the number of deployed queries leads to a reduction of the resource cost per query. When the percentage of cost reduction is greater than the reduction in processing quality (i.e. SIC values) there is a cost advantage by 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 resource cost for the initial deployment and $C_x$ is the resource cost of deploying $x$ queries.

The difference in quality of processing, $\Delta(QoP)$, is calculated as:

$$\Delta(QoP) = \frac{QoP_{base} - QoP_x}{QoP_{base}} \tag{6.4}$$

where $QoP_{base}$ is the quality of processing value for the starting deployment and $QoP_x$ is the quality of processing achieved after deploying $x$ queries.

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

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

The difference between the reduction in cost and quality of processing is inversely proportional to the number of queries. The more queries are added, the lower the cost advantage, until a point is reached, at which increasing the number of queries is no longer beneficial. A user may add more queries until this point is reached or the difference is below a given threshold or minimal SIC value.

### Experimental Results

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

Figure 6.11 shows the mean SIC value as the quality of processing metric. The base case has 180 queries and, at each step, 60 more queries are deployed. The graph shows how, with 300 queries, there is almost an 8% difference between the reduction in quality of processing and the resource

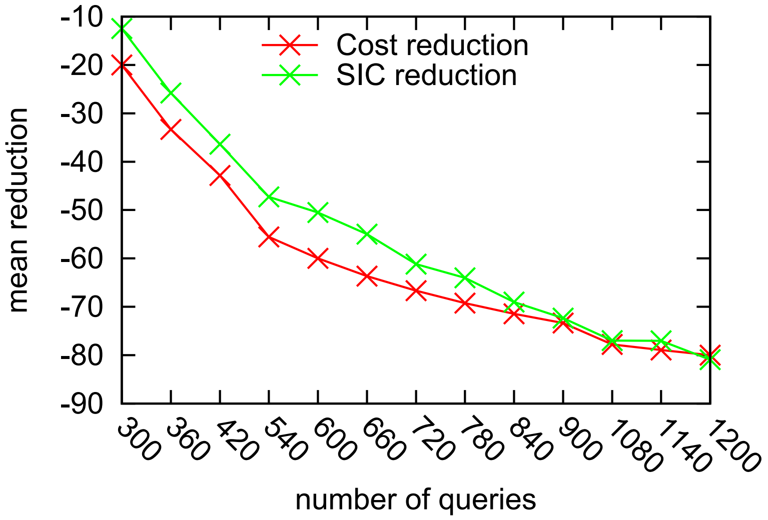

Figure 6.11: Trade-off between the reduction of resource cost per query and the reduction in quality of processing, as expressed by mean SIC values.

cost reduction. This difference grows smaller as more queries are deployed until a “tipping point” is reached with approximately 900 queries. From this point onwards, adding more queries does not result in further benefit.

## 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 carried out using the DISSP prototype system described in Chapter 4. The first set of experiments 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, our experiments showed that, for many classes of queries, there is a good correlation, confirming that a high value of the SIC metric indicates a high confidence in the correctness of results. The second set of experiments focused on the evaluation of the fair shedding policy, designed to exploit the SIC metadata to allocate resources among queries evenly and compares its performance with that of a random shedder. Our 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 resource cost and the processing quality of the results. They showed that it is possible to strike a balance between the achieved SIC values of the results and a reduced resource cost for the processing infrastructure.

# Chapter 7

# Conclusion

In a world in which the production of data continues to grow, data stream processing systems (DSPSs) play an important role by allowing the computation of continuous queries over streams of data in real-time. Their applications include, among others, the processing of environmental sensor data, allowing scientists to monitor a wide range of natural phenomena, and the analysis of user-generated content shared through social media, offering new ways of interpreting human interactions.

The large amount of available data makes the correct provisioning of DSPSs difficult. This is due to the unpredictable volatility of input data rates and the costs associated with the purchase or rental of computing platforms. A large amount of input data requires processing resources that may not be available to a user. To acquire resources, it is possible to use a federated resource pool, in which different parties contribute their resources to create a larger shared processing infrastructure, or to rent the required processing resources on demand in a cloud environment. In both cases, however, it may not be feasible to obtain the necessary resources, either due to practical issues or economical constraints.

As a consequence, overload may be a common condition in DSPS systems. Traditionally, overload is considered a rare, transient condition but our perspective is that it is a normal operating condition. Instead of trying to achieve perfect processing at all times, a DSPS under overload should account for its impact and report any processing degradation to users. For many applications, an imperfect result may still be valuable to the user.

Under overload, the lack of processing resources forces the system to discard a certain fraction of the input data, a process called load-shedding. The choice of which tuples to discard is critical to the correct functioning of the system. Instead of discarding tuples at random, we propose an intelligent load-shedding policy which selects tuples in a way so that all queries are penalised equally.

### Quality-Aware Stream Processing Under Overload.

This thesis presented a novel quality-centric data model for stream processing. It augments data streams with a quality metric called Source Information Content (SIC). The definition of this metric follows the reasoning outlined in a set of assumptions and considerations, making it applicable to a DSPS with generic operators. The SIC metric tracks the failure during processing, providing an indication of the quality of the computed results. It allows the user to reason about the result quality and the system to implement intelligent policies for overload management.

A prototype stream processing system called DISSP was developed to evaluate the benefits offered by the SIC metric. It was designed to process data efficiently under overload and provide continuous feedback to the user about the achieved quality of processing of queries. It exploits the SIC metric to degrade gracefully the quality of results for all queries. A particular focus is the efficiency of load-shedding, following a design that allows for the flexible implementation of semantic shedding policies. The algorithm that decides which tuples to keep and which to discard leverages the information contained in the SIC values of tuples to implement a fair load-shedding policy.

We defined fairness in a DSPS based on the SIC values achieved by queries (i.e. a system is fair if all queries achieve an equal quality of processing) as expressed through the SIC value. Based on this definition, the SIC metric was used to implement a fair load-shedding policy, which detects and handles an excess of processing load by discarding a portion of the input tuples. This allows all queries to achieve an equal quality of processing, reducing the differences in terms of the SIC values of their results. System resources are allocated fairly among queries, regardless of their size or characteristics. This is particularly useful in a shared processing infrastructure, such as a federated resource pool, so that all users receive a fair share of the processing resources.

We performed a set of experiment to explore the characteristics of the SIC metric and its applicability in the context of semantic load-shedding. The experimental results indicate that there is a good correlation between the error that occurs in the results and their associated SIC values. The fair shedding policy was compared with a random rejection of input data. Results show that the intelligent policy outperforms the random one, both in terms of the average SIC values as well as the dispersion of the values. The calculation of the SIC metric showed good scalability properties, maintaining a stable performance when varying the amount of processing resources and the number of deployed queries. Finally, we showed that it was possible to expose a trade-off between the reduction in cost per query and the reduction in SIC value when a user wants to reduce the processing costs and is willing to accept approximate results.

## Future work

Directions deserving further investigation include the application of the quality-centric data model to the dynamic allocation of the optimal amount of processing resources in a cloud deployment, the use of free-running replicas to increase the ability of the system to withstand failure and the augmenting of operators with custom error functions to better correlate SIC values and the accuracy of results.

**Dynamic Resource Allocation.** In a cloud deployment, the quality-aware data model could be used to scale the amount of rented resources at run-time, detecting a temporary shortage of resources in case of a sudden increase of the input rates, and also signaling an excess of processing capacity causing unnecessary costs. Based on the SIC value of results, it is either possible to *scale-out*, increasing the amount of rented processing nodes when the SIC values of results do not reach a given threshold, or to *scale-in*, reducing the amount of processing nodes in order to reduce the rental cost, when the amount of processing resources is excessive.

Scaling-out resources allows to find the optimal amount of processing nodes to rent in order to achieve a given quality of processing. When a query is first deployed, an horizontal operator decomposition starts by allocating a modest amount of processing resources, decomposing the main operator into a small number of copies. It then checks if the number of allocated processing nodes is sufficient to achieve result SIC values that are above a user specified threshold. If the result SIC values are below this threshold, more processing nodes are allocated, increasing the number of parallel subqueries of the decomposition. This process is repeated until the achieved result SIC values go above the minimum threshold. In this way, it is possible to correctly provision the amount of resources needed to run a certain query, thus minimising the processing rental cost.

The SIC metric could also be employed to scale-in resources, deallocating processing nodes when they are not needed. When the achieved result SIC values are well above the acceptable minimum value, the system can estimate if there is an advantage in reducing the number of processing nodes allocated for the current computation. For example, if the amount of input data is reduced, the current allocation of resources may become excessive and too costly. The system could start migrating queries from a node to the rest of the infrastructure and finally release it, reducing the resource costs. It would repeat this process as long as the result SIC values remain above a given threshold.

**Replica Management.** The proposed quality-centric data model can be also used to increase the system ability to withstand failure with the use of *free-running replicas* [MW06]. This thesis was mainly concerned with issues arising in a system failing due to overload. Other kinds of failure that may occur in a DSPS include node crashes and network partitions. To mitigate the impact of failure caused by overload or other means, a DSPS should replicate the most important operators of a query. By running multiple copies of the same operators in parallel, more copies of the same result are computed at different sites. In case of failure of a node, the system can use the result calculated at another replica. In case of overload, the system can choose the result with the highest processing quality.

Under failure, it is typical for different replicas to produce different results. The internal state of operators may not be consistent due to the missing inputs. A standard approach is to employ a *eventual consistency* model, in which the internal state of the failed replica is restored after failure by replaying the missing tuples. However, this may not be feasible, due to the lack of resources.

We propose instead to adopt a more relaxed consistency model, in which operators continue processing without any attempt of reconcile the internal state of a failed replica. If this relaxed consistency model is employed, the DSPS system must choose among the different outputs produced by the replicas, deciding which one that to deliver to the user. In this situation, the SIC metric offers a valuable indication about the quality of the output computed by the replicas. By capturing the amount of failure that occurred during the creation of tuples, it allows the system to select the correct output to deliver by choosing the one with the highest quality of processing (i.e. that has the higher SIC value) and thus is closer to what would have been produced in absence of failure.

**Custom Error Functions.** A design choice that was made for the quality-centric data model is to ignore the processing semantics of the individual operators. Being operator agnostic allows the model to be applicable to any type of queries, without being restricted to a limited set of operator. Even though a link between the quality of processing and the accuracy of results was shown in the evaluation of the model, there is no direct correlation between the result SIC values and the error caused by overload.

An extension to the model would be to allow users to submit custom error functions for operators. In this way, the model could calculate the exact error for the computed results. For a range of generic operators the error function could be already provided by the system, leaving the option of modifying it to the user. For custom operators instead, it would be necessary to submit it together with their implementation.

# Bibliography

[Aba+03] D. J. Abadi et al. “Aurora: a new model and architecture for data stream management”. In: *The VLDB Journal.* Vol. 12. 2. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003, pp. 120–139.

[Aba+05] D. J. Abadi et al. “The Design of the Borealis Stream Processing Engine”. In: *Second Biennial Conference on Innovative Data Systems Research CIDR 2005.* Asilomar, CA, 2005.

[ABW06] A. Arasu, S. Babu, and J. Widom. “The CQL continuous query language: semantic foundations and query execution”. In: *The VLDB Journal* 15.2 (2006), pp. 121–142.

[AH02] L. A. Adamic and B. A. Huberman. “Zipf’s law and the Internet”. In: *Glottometrics* 3 (2002), pp. 143–150.

[AHS06] K. Aberer, M. Hauswirth, and A. Salehi. *Global Sensor Networks.* Tech. rep. EPFL, 2006.

[Ami+06] L. Amini et al. “Adaptive Control of Extreme-scale Stream Processing Systems”. In: *ICDCS ’06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems.* Washington, DC, USA: IEEE Computer Society, 2006, p. 71.

[Ara+04] A. Arasu et al. *STREAM: The Stanford Data Stream Management System.* Technical Report 2004-20. Stanford InfoLab, 2004.

[Arm+10] M. Armbrust et al. “A view of cloud computing”. In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58.

[Bab+03] B. Babcock et al. “Chain: operator scheduling for memory minimization in data stream systems”. In: *SIGMOD ’03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data.* San Diego, California: ACM, 2003, pp. 253–264.

[Bal+04] Balazinska et al. *Availability-Consistency Trade-Offs in a Fault-Tolerant Stream Processing System.* Tech. rep. Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, 2004.

[Bal+05] M. Balazinska et al. "Fault-Tolerance in the Borealis Distributed Stream Processing System". In: *ACM SIGMOD Conf.* Baltimore, MD, 2005.

[Bal+07] M. Balazinska et al. "Data Management in the Worldwide Sensor Web". In: *IEEE Pervasive Computing* 6.2 (2007), pp. 30–40.

[BBS04] M. Balazinska, H. Balakrishnan, and M. Stonebraker. "Contract-based load management in federated distributed systems". In: *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation.* San Francisco, California: USENIX Association, 2004, pp. 15–15.

[BDM04] B. Babcock, M. Datar, and R. Motwani. "Load Shedding for Aggregation Queries over Data Streams". In: *In ICDE.* 2004, pp. 350–361.

[BGS01] P. Bonnet, J. Gehrke, and P. Seshadri. "Towards Sensor Database Systems". In: *MDM '01: Proceedings of the Second International Conference on Mobile Data Management.* London, UK: Springer-Verlag, 2001, pp. 3–14.

[Bla09] K. Black. *Business Statistics: Contemporary Decision Making.* Wiley Plus Products Series. John Wiley & Sons, 2009.

[Blo70] B. H. Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Commun. ACM* 13 (7 1970), pp. 422–426.

[BMZ10] J. Bollen, H. Mao, and X.-J. Zeng. "Twitter mood predicts the stock market". In: *CoRR* abs/1010.3003 (2010).

[Bot+09] I. Botan et al. "Federated Stream Processing Support for Real-Time Business Intelligence Applications". In: *VLDB International Workshop on Enabling Real-Time for Business Intelligence (BIRTE'09).* Vol. 41. 2009.

[BW01] S. Babu and J. Widom. "Continuous Queries over Data Streams". In: *International Conference on Management of Data (SIGMOD 2001).* Vol. 30. 3. 2001.

[BWH12] BrandWatch: Social media monitoring. 2012. URL: `http://www.brandwatch.com/`.

[Car+02] D. Carney et al. "Monitoring streams: a new class of data management applications". In: *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases.* Hong Kong, China: VLDB Endowment, 2002, pp. 215–226.

[Cha+01] K. Chakrabarti et al. "Approximate query processing using wavelets". In: *The VLDB Journal* 10 (2-3 2001), pp. 199–223.

[Cha+08] M. Cha et al. "Characterizing social cascades in Flickr". In: *Proceedings of the first workshop on online social networks.* WOSN '08. Seattle, WA, USA: ACM, 2008.

[Che+03] M. Cherniack et al. “Scalable Distributed Stream Processing”. In: *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*. Asilomar, CA, 2003.

[CK09] J. H. Chang and H.-C. M. Kum. “Frequency-based load shedding over a data stream of tuples”. In: *Inf. Sci.* 179 (21 2009), pp. 3733–3744.

[CoD10] CoDeeN Project. *CoTop, A Slice-Based Top for PlanetLab*. `http://codeen.cs.princeton.edu/cotop/`. 2010.

[CSM12] COSM: Where the Internet of Things is being built. 2012. URL: `https://www.cosm.com/`.

[Dea09] J. Dean. *Designs, Lessons and Advice from Building Large Distributed Systems*. 2009.

[ERS12] Earthscope project. 2012. URL: `http://www.earthscope.org`.

[FB99] A. Fox and E. A. Brewer. “Harvest, yield, and scalable tolerant systems”. In: *Hot Topics in Operating Systems, (HotOS-VII)*. 1999, pp. 174 –178.

[FBK12] Facebook: A social utility that connects people with friends, others who work, study, and live around them. 2012. URL: `http://www.facebook.com/`.

[FKS03] R. Fagin, R. Kumar, and D. Sivakumar. “Comparing Top k Lists”. In: *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. 2003, pp. 28–36.

[FLK12] Flickr: An online photo sharing platform. 2012. URL: `http://www.flickr.com/`.

[FS112] *Where the Young and Tech-Savvy Go*. 2012. URL: `http://blogs.wsj.com/digits/2011/05/19/a-week-on-foursquare/`.

[FS212] *How we found the rudest cities in the world Analytics @foursquare*. 2012. URL: `http://engineering.foursquare.com/2011/02/28/how-we-found-the-rudest-cities-in-the-world-analytics-foursquare/`.

[FSQ12] Keep up with friends. Discover new places. Get deals and discounts. 2012. URL: `http://www.foursquare.com/`.

[Gal+10] W. Galuba et al. “Outtweeting the Twitterers - Predicting Information Cascades in Microblogs”. In: *Proceedings of the 3rd Workshop on Online Social Networks (WOSN 2010)*. Boston, Massachusetts, USA, 2010.

[GBH09] A. Go, R. Bhayani, and L. Huang. “Twitter Sentiment Classification using Distant Supervision”. In: *Processing* (2009), pp. 1–6.

[Ged+08] B. Gedik et al. “SPADE: the system s declarative stream processing engine”. In: *SIGMOD ’08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. Vancouver, Canada: ACM, 2008, pp. 1123–1134.

[Gib+03] P. B. Gibbons et al. “IrisNet: An Architecture for a Worldwide Sensor Web”. In: *IEEE Pervasive Computing* 2.4 (2003), pp. 22–33.

[GM04] J. Gehrke and S. Madden. “Query Processing in Sensor Networks”. In: *IEEE Pervasive Computing* 3.1 (2004), pp. 46–55.

[GO03] L. Golab and M. T. Özsu. “Issues in data stream management”. In: *SIGMOD.* Vol. 32. 2. New York, NY, USA: ACM, 2003, pp. 5–14.

[Har68] G. Hardin. “The Tragedy of the Commons”. In: *Science* xx (1968), pp. 1243–47.

[HCZ07] J.-H. Hwang, U. Cetintemel, and S. B. Zdonik. “Fast and Reliable Stream Processing over Wide Area Networks”. In: *ICDE Workshops.* 2007, pp. 604–613.

[HcZ08] J.-H. Hwang, U. Çetintemel, and S. Zdonik. “Fast and Highly-Available Stream Processing over Wide Area Networks”. In: *Proceedings of the 24th International Conference on Data Engineering(ICDE).* 2008, pp. 804–813.

[Hwa+05] J.-H. Hwang et al. “High-Availability Algorithms for Distributed Stream Processing”. In: *ICDE ’05: Proceedings of the 21st International Conference on Data Engineering.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 779–790.

[Jai+08a] N. Jain et al. “Network Imprecision: A New Consistency Metric for Scalable Monitoring”. In: *OSDI.* 2008.

[Jai+08b] N. Jain et al. “Towards a streaming SQL standard”. In: *Proc. VLDB Endow.* 1.2 (2008), pp. 1379–1390.

[JNV03] J. K. Jeffrey, J. F. Naughton, and S. D. Viglas. “Evaluating Window Joins over Unbounded Streams”. In: *Proceedings of the 19th International Conference on Data Engineering (ICDE).* 2003, pp. 341–352.

[Kan+07] A. Kansal et al. “SenseWeb: An Infrastructure for Shared Sensing”. In: *Multimedia* 14.4 (2007), pp. 8–13.

[KS05] J. Krmer and B. Seeger. “A temporal foundation for continuous queries over data streams”. In: *In Proc. 11th Int. Conf. on Management of Data (COMAD).* 2005, pp. 70–82.

[Li+08] X. Li et al. “Real-time storm detection and weather forecast activation through data mining and events processing”. In: *Earth Science Informatics* 1.2 (2008), pp. 49–57.

[LLS08] G. T. Lakshmanan, Y. Li, and R. Strom. “Placement Strategies for Internet-Scale Data Stream Systems”. In: *IEEE Internet Computing* 12.6 (2008), pp. 50–60.

[Ma+08] L. Ma et al. “Semantic Load Shedding over Real-Time Data Streams”. In: *Computational Intelligence and Design, International Symposium on* 1 (2008), pp. 465–468.

[Mcl+05] D. J. Mclaughlin et al. “Distributed Collaborative Adaptive Sensing (DCAS) for Improved Detection”. In: *in Proc. American Meteorological Society Annual Meeting.* 2005.

[Mis+08] A. Mislove et al. “Growth of the flickr social network”. In: *Proceedings of the first workshop on Online social networks.* WOSN ’08. Seattle, WA, USA: ACM, 2008, pp. 25–30.

[Mot+02] R. Motwani et al. *Query Processing, Resource Management, and Approximation ina Data Stream Management System.* Technical Report 2002-41. Stanford InfoLab, 2002.

[MPH12] Mappy Health: A tracking disease trends and terms on twitter 140 characters at a time. 2012. URL: `http://www.mappyhealth.com/`.

[MW06] R. N. Murty and M. Welsh. *Towards a dependable architecture for internet-scale sensing.* Seattle, WA: USENIX Association, 2006, pp. 8–8.

[MZ10] B. Mozafari and C. Zaniolo. “Optimal load shedding with aggregates and mining queries”. In: *Data Engineering (ICDE), 2010 IEEE 26th International Conference on.* 2010, pp. 76 –88.

[NEO12] National Ecological Observatory Network. 2012. URL: `http://www.neoninc.org`.

[Pie+06] P. Pietzuch et al. “Network-Aware Operator Placement for Stream-Processing Systems”. In: *ICDE ’06: Proceedings of the 22nd International Conference on Data Engineering.* Washington, DC, USA: IEEE Computer Society, 2006, p. 49.

[Pie08] P. Pietzuch. “Challenges in dependable internet-scale stream processing”. In: *SDDDM ’08: Proceedings of the 2nd workshop on Dependable distributed data management.* Glasgow, Scotland: ACM, 2008, pp. 25–28.

[PJO12] POJO: An acronym for: Plain Old Java Object. 2012. URL: `http://www.martinfowler.com/bliki/POJO.html`.

[Pla+06] B. Plale et al. “CASA and LEAD: Adaptive Cyberinfrastructure for Real-Time Multiscale Weather Forecasting”. In: *Computer* 39.11 (2006), pp. 56–64.

[PLB12] The PlanetLab Distributed Testbed. 2012. URL: `http://www.planet-lab.org/`.

[PWB07] E. Pinheiro, W.-D. Weber, and L. A. Barroso. “Failure trends in a large disk drive population”. In: *Proceedings of the 5th USENIX conference on File and Storage Technologies.* FAST ’07. San Jose, CA: USENIX Association, 2007, pp. 2–2.

[PZYD11] G. Pallis, D. Zeinalipour-Yazti, and M. Dikaiakos. “Online Social Networks: Status and Trends”. In: *New Directions in Web Data Management 1.* Ed. by A. Vakali and L. Jain. Vol. 331. Studies in Computational Intelligence. 10.1007/978-3-642-17551-$0_8$. Springer Berlin / Heidelberg, 2011, pp. 213–234.

[Rek69] K. Rektorys. *Survey of applicable mathematics.* M.I.T. Press, 1969.

[Rog03] E. M. Rogers. *Diffusion of Innovations, 5th Edition.* 5th. Free Press, Aug. 2003.

[RTW12] *FAQs About Retweets (RT).* 2012. URL: `https://support.twitter.com/articles/77606-faqs-about-retweets-rt`.

[Sbz+03] S. Z. Sbz et al. “The Aurora and Medusa Projects”. In: *IEEE Data Engineering Bulletin* 26 (2003), pp. 3–10.

[Sce+11] S. Scellato et al. “Track globally, deliver locally: improving content delivery networks by tracking geographic social cascades”. In: *Proceedings of the 20th international conference on World wide web.* WWW ’11. Hyderabad, India: ACM, 2011, pp. 457–466.

[SCM12] Social Mention: Real-time search. 2012. URL: `http://www.socialmention.com/`.

[ScZ05] M. Stonebraker, U. Çetintemel, and S. Zdonik. “The 8 requirements of real-time stream processing”. In: *SIGMOD Rec.* 34.4 (2005), pp. 42–47.

[SHB04] M. A. Shah, J. M. Hellerstein, and E. Brewer. “Highly available, fault-tolerant, parallel dataflows”. In: *SIGMOD ’04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data.* Paris, France: ACM, 2004, pp. 827–838.

[SKS12] Sloan Digital Sky Survey. 2012. URL: `http://www.sdss.org`.

[SMW05] U. Srivastava, K. Munagala, and J. Widom. “Operator placement for in-network stream query processing”. In: *PODS ’05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* Baltimore, Maryland: ACM, 2005, pp. 250–258.

[Str08] Streambase. *Real-Time Data Processing with a Stream Processing Engine.* Tech. rep. Streambase.com, 2008.

[SWX12] The Swiss Eperiment. 2012. URL: `http://www.swiss-experiment.ch`.

[SYC09] N. Sastry, E. Yoneki, and J. Crowcroft. “Buzztraq: predicting geographical access patterns of social cascades using social networks”. In: *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems.* SNS ’09. Nuremberg, Germany: ACM, 2009, pp. 39–45.

[Tat+03] N. Tatbul et al. "Load shedding in a data stream manager". In: *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*. Berlin, Germany: VLDB Endowment, 2003, pp. 309–320.

[Tat+08] N. Tatbul et al. "Load Management and High Availability in the Borealis Distributed Stream Processing Engine". In: *GeoSensor Networks: Second International Conference, GSN 2006, Boston,MA, USA, October 1-3, 2006, Revised Selected and Invited Papers* 1 (2008), pp. 66–85.

[TBN12] Comprehensive Test Ban Treaty Organization. 2012. URL: `http://www.ctbto.org`.

[Tea06] B. Team. *Application Programmer's Guide*. Tech. rep. Brown University, 2006.

[Tha+02] N. Thaper et al. "Dynamic multidimensional histograms". In: *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. Madison, Wisconsin: ACM, 2002, pp. 428–439.

[Tur+10] D. S. Turaga et al. "Design principles for developing stream processing applications". In: *Softw., Pract. Exper.* 40.12 (2010), pp. 1073–1104.

[TWA12] Twitrratr: tracking opinions on Twitter. 2012. URL: `http://twitrratr.com/`.

[TWF12] TweetFeel: Real-time Twitter search with feelings using insanely complex sentiment analysis. 2012. URL: `http://www.tweetfeel.com/`.

[TWS12] Twitter Sentiment: A Twitter sentiment analysis tool. 2012. URL: `http://twittersentiment.appspot.com/`.

[TWT12] Twitter. 2012. URL: `http://www.twitter.com/`.

[TWZ12] Twendz : exploring Twitter conversations and sentiment. 2012. URL: `http://twendz.waggeneredstrom.com/`.

[TZ06] N. Tatbul and S. Zdonik. "Window-aware load shedding for aggregation queries over data streams". In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 799–810.

[UC96] G. Upton and I. Cook. *Understanding Statistics*. Oxford University Press, 1996.

[Ucl+01] A. A. Ucla et al. *Fundamental Concepts of Dependability*. 2001.

[USV12] US National Virtual Observatory. 2012. URL: `http://www.us-vo.org`.

[VGS05] S. Voulgaris, D. Gavidia, and M. V. Steen. "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays". In: *Journal of Network and Systems Management* 13 (2005), p. 2005.

[Vis+09] B. Viswanath et al. “On the evolution of user interaction in Facebook”. In: *Proceedings of the 2nd ACM workshop on Online social networks*. WOSN ’09. Barcelona, Spain: ACM, 2009, pp. 37–42.

[VS05] S. Voulgaris and M. van Steen. “Epidemic-Style Management of Semantic Overlays for Content-Based Searching”. In: *Euro-Par*. 2005, pp. 1143–1152.

[WDF12] WildFire: Complete Enterprise Social Media Marketing Software. 2012. URL: `http://www.wildfireapp.com/`.

[WR09] S. Wang and E. Rundensteiner. “Scalable Stream Join Processing with Expensive Predicates: Workload Distribution and Adaptation by Time-slicing”. In: *Proceedings of the 12th International Conference on Extending Database Technology (EDBT)*. 2009, pp. 299–310.

[Xin+06] Y. Xing et al. “Providing resiliency to load variations in distributed stream processing”. In: *VLDB ’06: Proceedings of the 32nd international conference on Very large data bases*. Seoul, Korea: VLDB Endowment, 2006, pp. 775–786.

[XZH05] Y. Xing, S. Zdonik, and J.-H. Hwang. “Dynamic Load Distribution in the Borealis Stream Processor”. In: *ICDE ’05: Proceedings of the 21st International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 791–802.

[Yin+09] L. Ying et al. “Distributed Operator Placement and Data Caching in Large-Scale Sensor Networks.” In: *INFOCOM*. IEEE, Mar. 20, 2009, pp. 977–985.

[YTB12] YouTube: Broadcast Yourself. 2012. URL: `http://www.youtube.com/`.

[Zho+06] Y. Zhou et al. “Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System”. In: *International Conference on Cooperative Information Systems*. Montpellier, France, 2006, pp. 54–71.