Chapter 3

Quality-Centric Data Model

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

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

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

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

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

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

3.1 Model Definitions

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

Schema

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

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

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

<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 processing node. Each field is named so that it can be referred to when processing the tuple and is defined using an abstract type which will be translated to a system type once the tuple is instantiated.

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

Table 3.1: Notation used in the model definitions.

Tuple

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

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

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

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

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

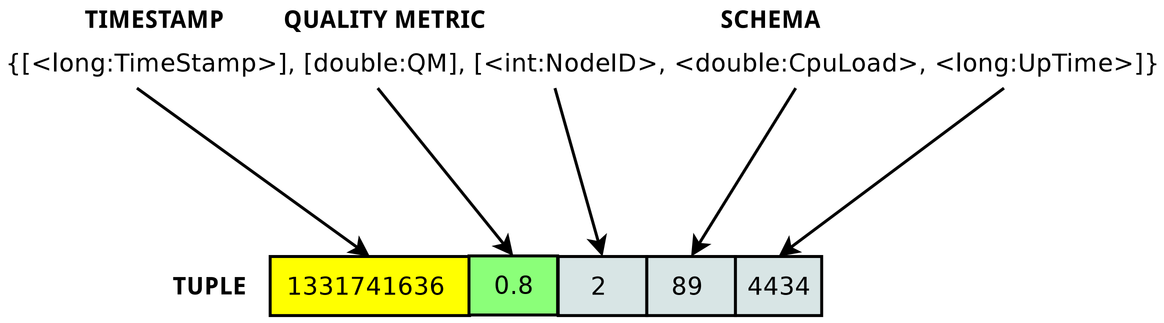
TIMESTAMP
QUALITY METRIC
SCHEMA
{[<long:TimeStamp>], [double:QM], [<int:NodeID>, <double:CpuLoad>, <long:UpTime>]}
TUPLE
1331741636
0.8
2
89
4434

Figure 3.1: A simple tuple with the relative schema.

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

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

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

Stream

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

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

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

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

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

Batch

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

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

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

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

Operator

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

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

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

Operators are seen as *black boxes* in our system, meaning that their internal semantic is not taken into account when calculating the quality metric value for the newly generated derived tuples. We can divide operators based on their behaviour when dealing with input streams into two categories: *blocking* and *non-blocking*.

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

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

Query

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

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

A query describes the processing to transform a set of input streams into a set of output streams. In the boxes-and-arrows model, queries are depicted as a directed acyclic graph (DAG) where arcs represent streams and vertices represent operators. One or more sources produce streams of tuples in

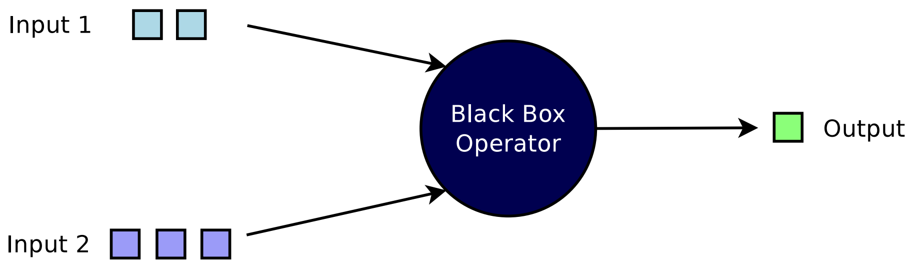

Figure 3.2: A black-box operator with two inputs and one output.

various time-variant rates, which are fed in input to the system. Then one or more operators process these tuples, either in sequence or in parallel.

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

3.2 Model Assumptions

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

1. FAILURE IS EVERYWHERE

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

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

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

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

2. APPROXIMATE PROCESSING

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

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

In sensor networks it is common to have a lot of failure at the source level. Sensors can suddenly stop working because of hardware failure or can become temporarily unreachable. In this cases a query processing their readings delivers results that are incomplete. Nevertheless the quality of the query results can be good enough to be meaningful for the final user. Especially when dealing with a large set of sensors the lack of input from a certain number of them does not usually mean that the computation should be considered void as a whole. Instead the results simply become less accurate due to the missing input data. If we consider queries aggregating readings of sensors scattered over a certain area, like an average of pollutants in a city area, it is possible that a failing sensor is located close to others that will record a similar reading. Thus the missing information might not produce a great variation in the final result, that is approximate but still meaningful.

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

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

3. A GENERIC METRIC

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

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

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

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

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

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

4. SOURCE EQUALITY

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

In our model sources are all considered equally important, regardless of their tuple production rate.

If we consider a sensor network deployment it is common to observe different rates of input readings. This is due to different settings or different battery constraint of the sensors. A unit with a depleted battery would usually reduce the rate at which it propagates information to save energy. Another reason could be the position of sensor, the energy cost of propagating a reading in fact grows with distance, as it requires a higher transmission power.

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

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

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

5. TUPLES EQUALITY

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

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

Let us consider a simple query, composed by one source and one average operator producing a single result every minute. The source produces 1000 tuples per second, the system is overloaded and is not able to process all of them. If the distribution of values carried by the source tuples is uniform it does not make any difference which tuples are dropped. If instead the distribution is highly skewed, presenting a small number of outliers, the dropping of one of them could sensibly change the aggregated result. Since the system employs an operator independent model it cannot know which tuples to drop in order to reduce the error in the results, so it treats all source tuples as equals and makes a random decision.

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

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

6. CONSERVATION OF INFORMATION

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

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

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

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

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

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

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

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

7. INFORMATION IS VALUE

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

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

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

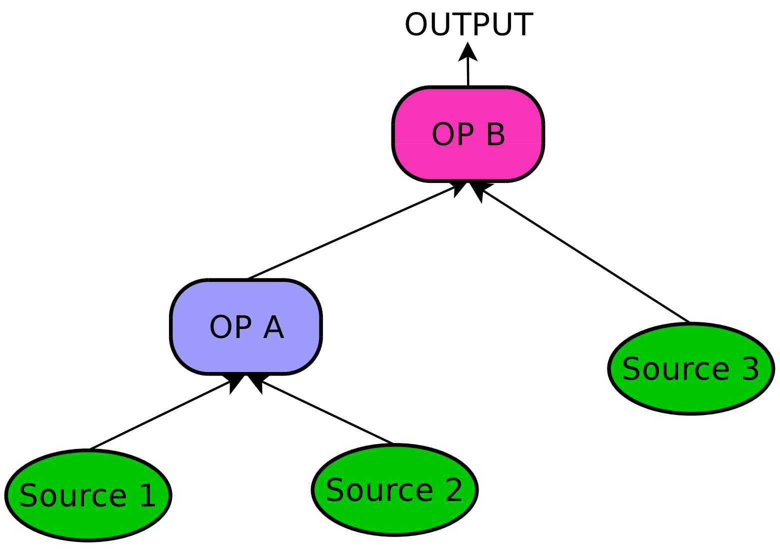

Figure 3.3: An unbalanced query tree, showing the different information contents of the tuples flowing through it.

8. TOTAL INFORMATION CONTENT

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

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

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

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

9. QUERIES ARE DAGs

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

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

The first case deals with queries computing multiple results. These are called *fan-out* queries and will be further analysed in Section 3.3. They compute multiple results based on some common input information, but are logically seen as a single query. The reason for treating a query with multiple results as a single query is to allow the system to be fair when making shedding decisions under overload. Let us consider a system with 2 running queries, submitted by 2 users. The first user submitted a query with a single terminal operator, while the query submitted by the other user have 9. If the system considered each result computed as a single query, the second user would be allocated 90% of the available resources, leaving only 10% to the first. If the system instead considered the query submitted by the second user as single computation yielding 9 different results, it would evenly shed tuple among the two queries, leading to a fair allocation of the available computing resources. Fair shedding will be explained in more detail in Section 5.3.

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

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

3.3 Types of Queries

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

Fan-in queries

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

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

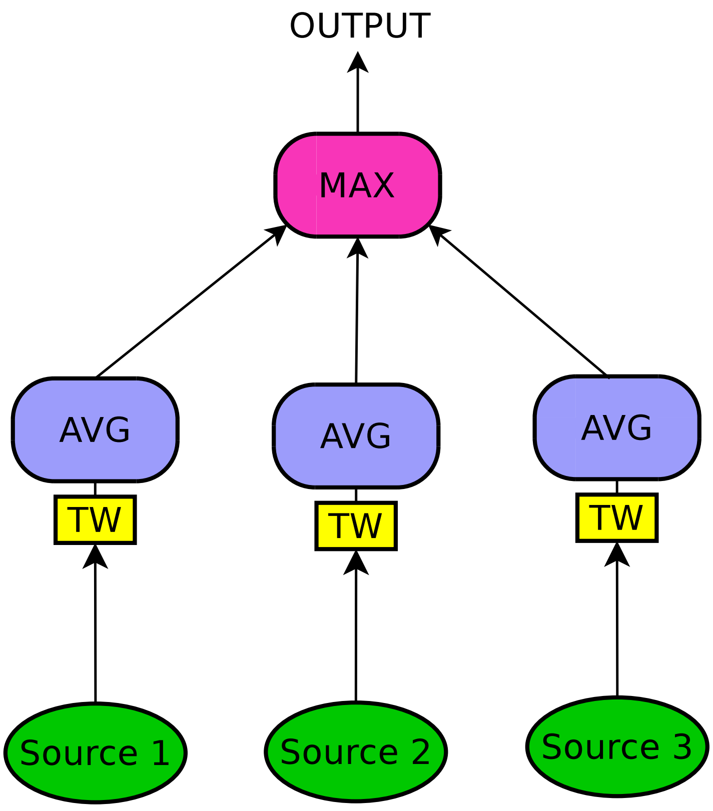

Figure 3.4: An example of fan-in query, input values for each source are first averaged over a certain time window, then the maximum is selected.

Fan-out queries

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

Split computation queries. The semantic of these queries is similar to the map-reduce processing paradigm. Since one or more operators are computationally too heavy to be hosted on a single processing node, their computation is split among a certain number of copies, running at different sites. These copies are all composed by the same set of operators, but process only a subset of the original data. Splitting the computation allows the system to spread the processing cost of a set of operators over several nodes and thus overcome the overload condition at the original hosting node.

EXAMPLE: Figure 3.9 shows a query processing Twitter data in real-time. A stream of Twitter

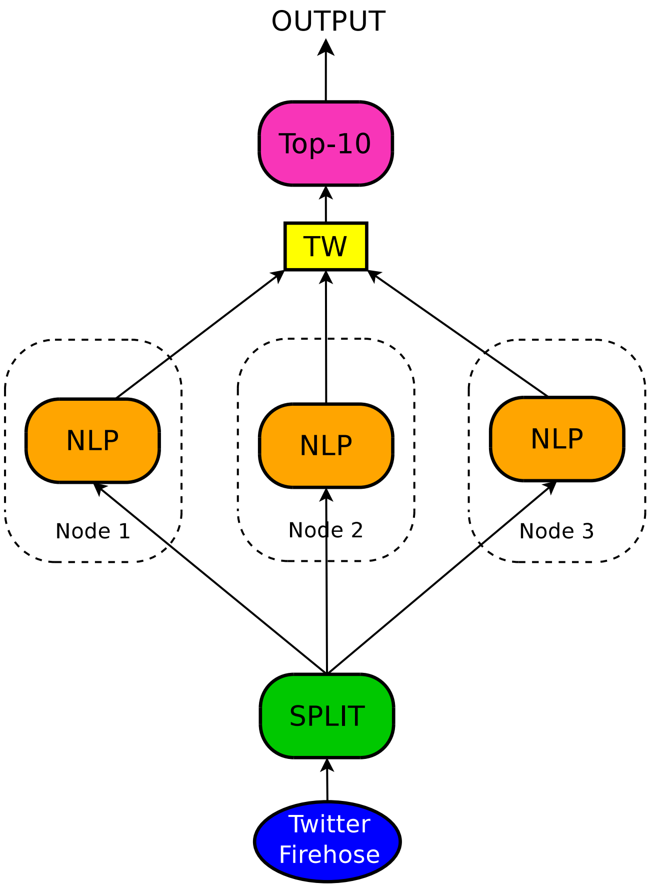

Figure 3.5: An example of fan-out query processing Twitter data implementing the map-reduce paradigm.

messages can be analysed to calculate the "rudest" status updates. Every message contains a lot of information in addition to the message text itself, like the location where it was sent from and some information about the author.

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

Multiple results queries. These queries produce several outputs from the same set of input data. These can be seen as multiple single output queries, with a partial overlap in their computation. When analysing a stream of data, a user might be interested in obtaining several different results. Instead of submitting N queries though, it can submit a single fan-out query with multiple ending points. This is conceptually simpler than having to design and submit several almost identical queries and directly exploits the processing redundancy by reusing a number of streams and operators.

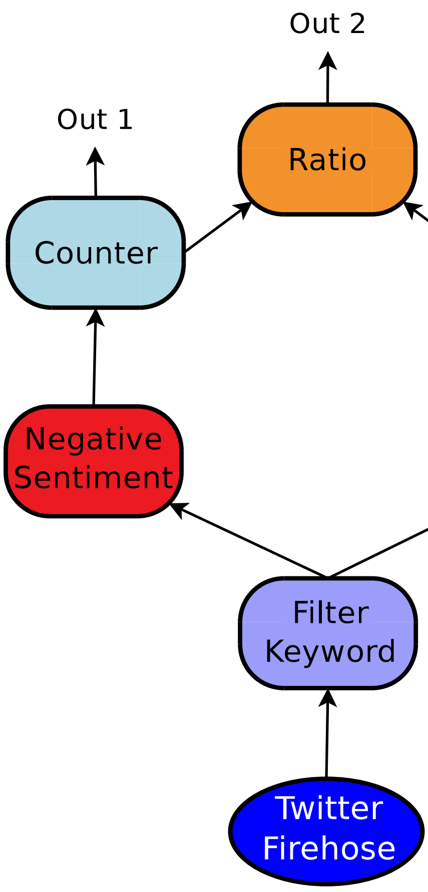

Figure 3.6: An example of fan-out query processing Twitter data. It counts the occurrence of positive and negative mentions of a certain keyword, calculating also their ratio.

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

The next section will describe the *Source Information Content* (SIC) quality metric developed in the scope of this project, derived following the reasoning of all the model assumptions made in Section 3.2. In Section 3.5 there will be a further analysis of these sample queries, which will be used as running examples of real-world applications of the SIC metric.

3.4 Source Information Content

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

Introduction

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

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

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

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

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

Data Model Formalisation

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

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

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

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

This concept is central to the definition of our quality centric metric called *Source Information Content* (SIC). The main idea behind it is that a result tuple is considered to be perfect when no information is lost during its creation, meaning that all tuples in its source information tuple set or their derivatives are correctly processed. If one of the tuples from the source information tuple set or a derivative is lost, either because of failure or deliberate shedding, the information contained in the result tuple is not perfect and its SIC value is decreased accordingly.

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

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

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

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

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

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

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

SIC propagation from source to result tuples

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

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

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

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

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

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

SIC relation to query performance

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

Model Benefits

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

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

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

A user can monitor the status of a computation by observing the changes in SIC values delivered by a query at different times. A sharp decrease in SIC values means that a lot more information is been

discarded and there might be the need to act upon it. Reasons for a sudden reduction could be:
1) the failure of one or more processing nodes, requiring the migration of some operators and maintenance of the failed infrastructure,
2) a sharp increase in input rates, requiring the increase of processing resources to maintain the delivered SIC value with acceptable boundaries.

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

When the system experiences *overload* and needs to discard a certain amount of tuples, it can use SIC values to determine which tuples to drop and which tuples to spare aiming at the equalisation of result SIC values among all queries. This process takes the name of *fair-shedding* and will be further explored in Section 5.3.

3.5 SIC Use Cases

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

Fan-in

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

According to Assumption 4 (Source Equality) all sources equally contribute to the creation of the final result. Assumption 5 (Tuple Equality) then states that all tuples from a source in a *source information tuple set* contain the same amount of information. Figure 3.7 show that each source assigns a total value of 1 to the tuples belonging to the same source information tuple set window, and that the individual SIC values are different according to the number of tuples contained in it. In this example the first source produces only 2 tuples and thus the individual SIC value is 1/2, the second source

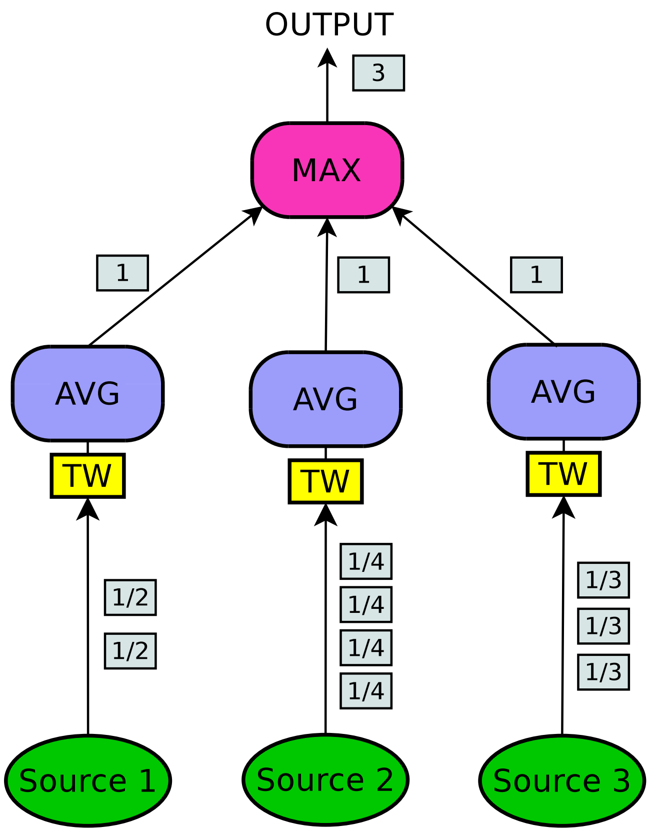

Figure 3.7: An example of fan-in query, input values for each source are first averaged over a certain time window, then the maximum is selected. Source Information Content values are shown for each tuple.

produces 3 tuples with an individual value of 1/3 and the third 3, each with a value of 1/3.

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

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

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

Fan-out

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

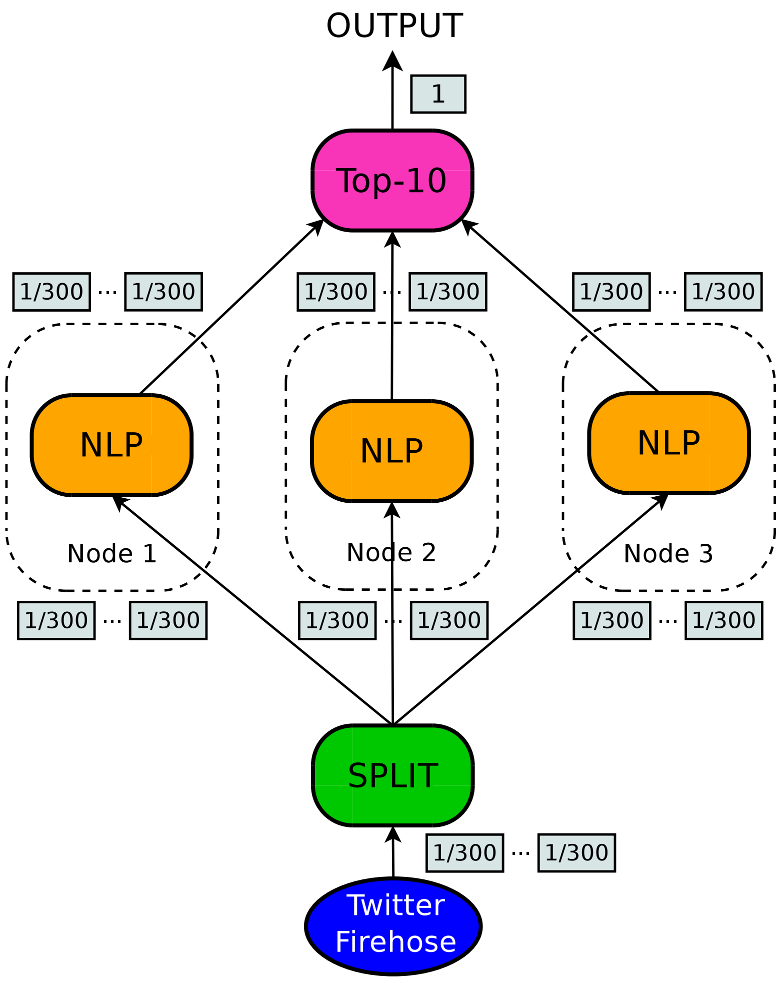

Figure 3.8: An example of fan-out query processing Twitter data implementing the map-reduce paradigm. Source Information Content values are shown for each tuple.

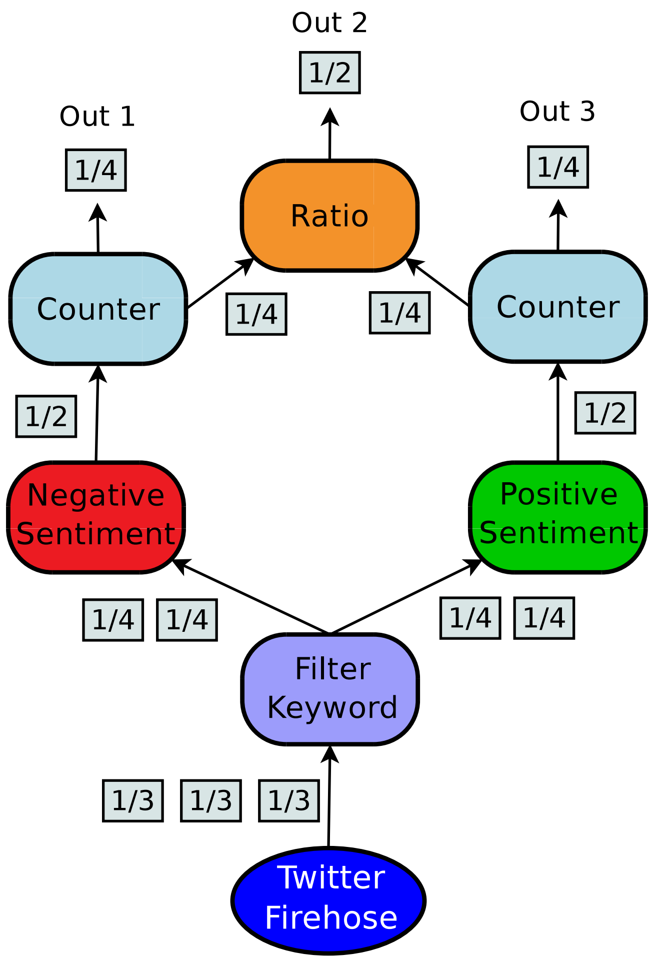

Figure 3.9: An example of fan-out query processing Twitter data. It counts the occurrence of positive and negative mentions of a certain keyword, giving also their ratio. The numbers shown on tuples are the individual SIC values.

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

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

Assumption 9 (Queries are DAGs) tells us that when such a multiplication of the output occurs, the total SIC value has to be distributed over all the output tuples. This is also in accordance to Assumption 6 (Information Conservation), as the amount of information in input is equal to the amount in output. Each output stream of the Keyword Filter operator then contains an identical set of 2 tuples, each having an individual SIC value of 1/4. The positive and the negative filter then each outputs a single tuple, with a SIC value of 1/2.

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

This query computes 3 different results: the number of messages with a positive mention of a keyword, the number of with a negative one and their ratio. The result tuples have different SIC values, 1/4 the ones produces by the counter operator and 1/2 the one produced by the Ratio operator. If we sum together these value though we obtain a total SIC value of 1 for the query, which indicates that no failure occurred during the processing.

3.6 Summary

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