Chapter 4

DISSP Design

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

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

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

4.1 Design Approach

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

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

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

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

4.2 Implementing the Model

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

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

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

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

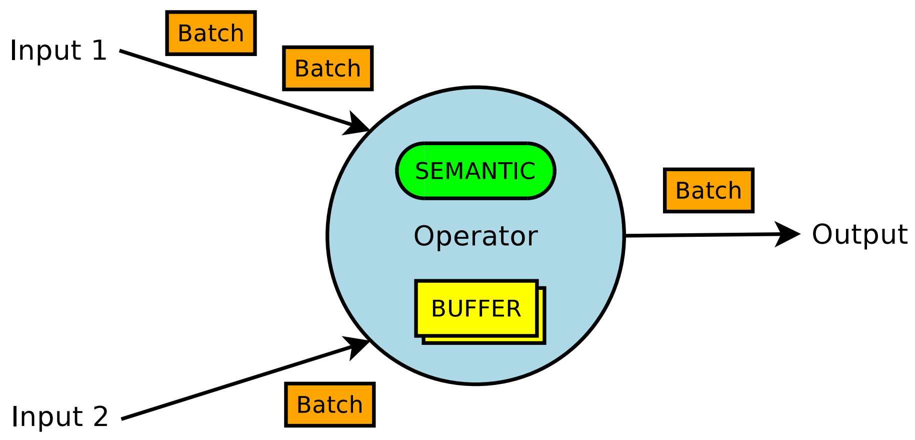

Figure 4.1: An operator with its internal structure.

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

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

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

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

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

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

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

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

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

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

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

To implement this part of the data model it would be necessary to know in advance which tuples will contribute to the creation of a future result, but because this result has not yet been created it is difficult to identify the source information tuple set. However, in practice, it is very difficult to accurately define this set due to time delays occurring during query processing, for instance processing delays, time windows, etc. To address this challenge, we consider a source information tuple set that includes all source tuples generated at sources during a predefined time period, referred to *Source Time Window (STW)*.

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

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

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

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

4.3 DISSP Architecture

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

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

System-wide Components

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

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

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

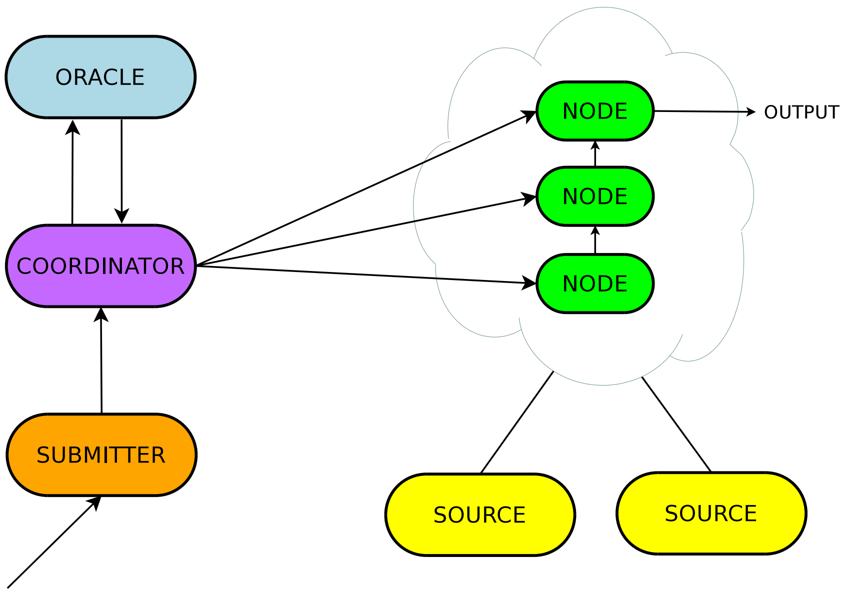

Figure 4.2: A system-wide overview of the components acting within the DISSP prototype.

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

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

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

Input Providers. An *input provider* is the component that allows input tuples to enter the system, acting as a collector of external data. The DISSP system is not concerned with the harvesting of input tuples. The data it processes is collected or generated elsewhere, for instance by a *sensor network* or through a *social media environment*. When the data is available for processing it is passed to the stream processing system so that queries can be computed over it. The input provider component acts as a *gateway*, where data is converted to the system representation and made available to queries for processing. Every source can be the entry point to many data sources, by employing multiple *InputDevice* operators. Each of them connects to an external source, for instance the sink of a sensor network or to the Twitter firehose.

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

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

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

Node Architecture

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

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

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

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

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

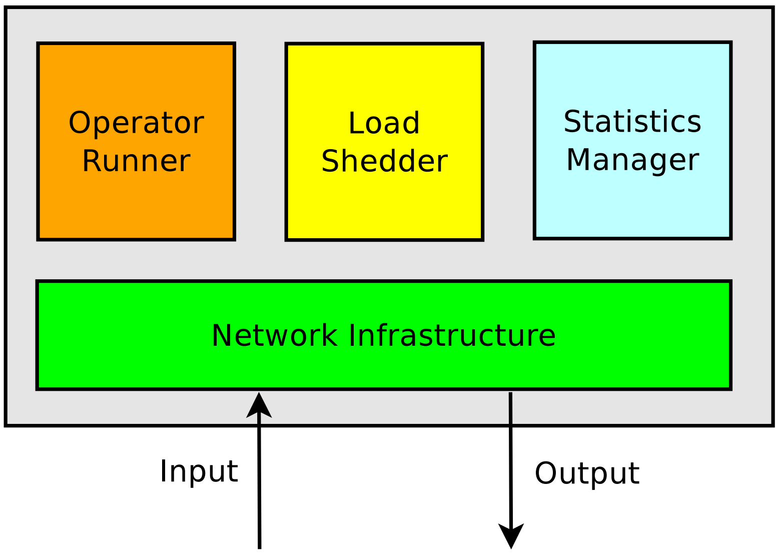

Figure 4.3: A high level view of a Processing Node, showing its internal logical components.

Network Layer

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

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

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

Tuples handling. Once a message has been determined as containing a tuple payload, it is directly passed to the *Operator Runner* component, without further processing. The content of the message stays in network format, thus being passed as a string. This follows the principle of *lazy deserialisation*, which states that the conversion from network format of tuples should happen as late as possible. Tuple objects are instantiated only when their are scheduled for processing, even their destination, the queries they belong to, is not known until then.

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

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

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

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

Web interface. The RequestHandler is also the gateway to the node *web interface*. When receiving a message it checks if it is an HTTP request and if so it replies with a web page containing information about the current status of the node, using the information obtained from the StatisticsManager. This includes the number of subqueries hosted together with their performance, as well as some global metrics describing the global status of the node, like its average throughput and latency. When connecting to the oracle, the web interface produces a report about the status of all processing nodes, together with a summary of the performance achieved by all queries, sorted by SIC value. The web interface of the oracle is useful to monitor the overall performance of the system and to evaluate the effectiveness of the shedding policy.

Operator Runner

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

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

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

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

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

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

Load-Shedder

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

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

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

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

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

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

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

Statistics Manager

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

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

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

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

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

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

Number of running subqueries. A number used to check if the deployment strategy leads to an even deployment or to a skewed distribution of the subqueries. This number is sent to the oracle and can be used to evaluate the goodness of the chosen deployment strategy.

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

4.4 Lifetime of a Query

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

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

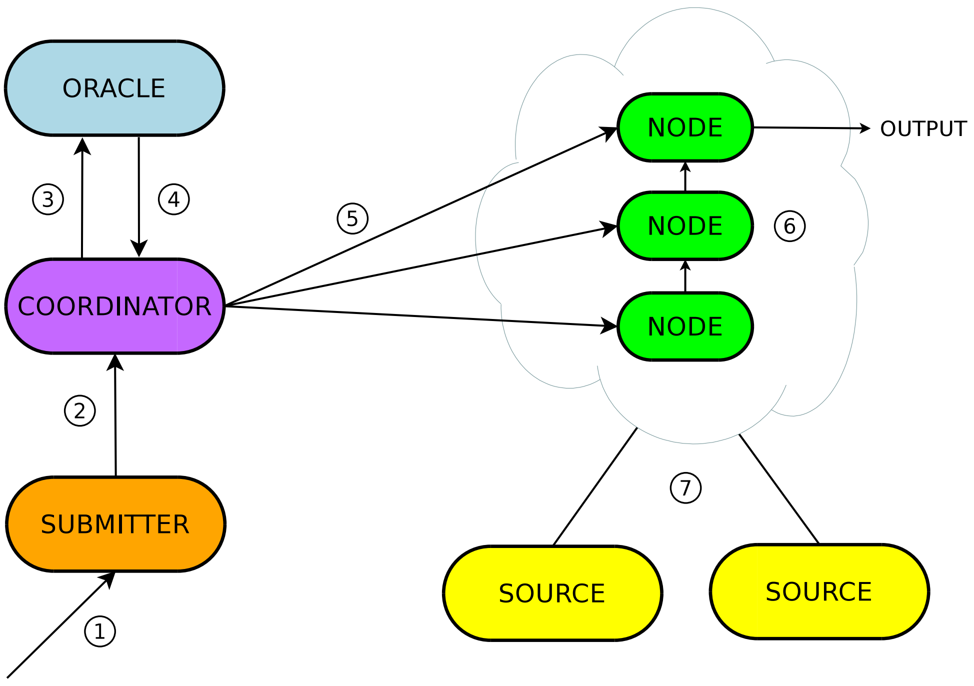

Figure 4.4: Steps involved in the deployment of a query.

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

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

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

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

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

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

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

Submission of Queries

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

Compiled Operators and Tuples

In DISSP tuples and operators are implemented as *compiled POJO objects*. This choice was made taking performance into consideration, since other options, based for instance on *reflection*, were considered too slow and cause of a potential performance bottleneck. To instantiate a custom tuple or operator object, based on the query requirements, a text *template* is completed with the information contained in the XML file describing the query producing a new .java file, which is then compiled to bytecode by a run-time compiler.

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

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

Listing 4.1: XML description of a Tuple

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

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

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

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

Listing 4.2: XML description of an Average operator

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

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

Partitioning of Queries

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

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

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

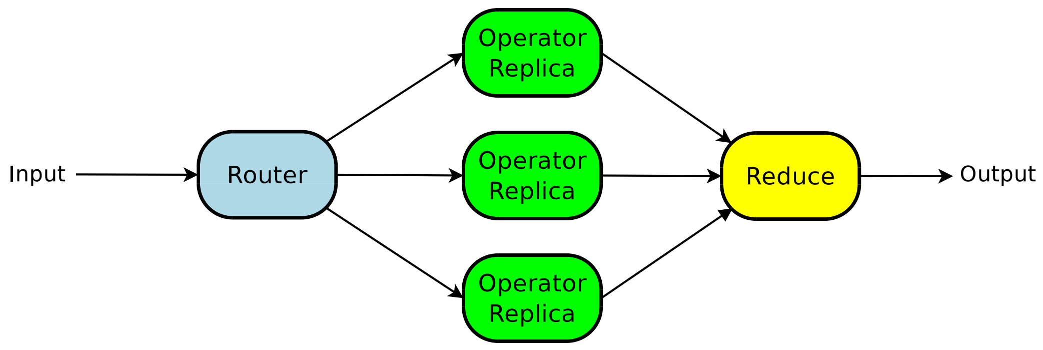

Figure 4.5: Map-Reduce decomposition of an operator.

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

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

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

4.5 Summary

This chapter presented the design of the DISSP prototype, a stream processing engine designed to implement and test the quality-centric data model described in Chapter 3. First, it presented the goals that drove the design of the system, such as the ability to perform efficiently under overload, the embedded calculation of the SIC metric and the possibility of tweaking the load-shedding policy according to the user needs. It explains how the theoretical entities presented in the description of the data model have been actually implemented in the basic components of a stream processing system.

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

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

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

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