Chapter 2

Background

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

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

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

2.1 Stream-based applications

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

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

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

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

Environmental Monitoring

The first domain of applications I will take into consideration is large-scale scientific sensing, in particular I will describe projects dealing with large-scale environmental monitoring. In recent years, the availability of cheap micro-sensing technology has unleashed the capability of sensing at an unprecedented scale. We can expect in the future to see everything of material significance being *sensor-tagged* and report its state or location in real-time [Gib+03; GM04; Bal+07]. At the present time, a great number of large-scale sensing projects are being developed and we can expect even more to come soon into place [ERS12; NEO12; Pla+06; SWX12].

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

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

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

Environmental sensing is a natural application for stream processing systems, as a constant flow of measurements is generated by a large number of sensors. Stream processing offers an intuitive and flexible paradigm to harness the complexity of mining such a high volume of data. It also give the possibility of obtaining real-time picture of the measured phenomena, enabling quick reactions to possibly disruptive events. The core components of these projects are sensing stations geographically widely distributed and often subject to harsh conditions. In this scenario, the failure of these sensing devices is not going to be uncommon and often their replacement is going to be problematic. Long running continuous queries need to be able to deal with some missing data, adapting to the new conditions and reporting the achieved quality of service. To collect and process all the data at all times is most probably infeasible and it is important for the computing backbone of such projects to be able to withstand a certain degree of failure within the system [MW06].

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

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

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

The integration of these two systems offers an invaluable infrastructure to atmospheric scientists [Pla+06]. First of all, it allows meteorologists to directly interact with data from the instruments as well as control the instruments themselves. Also, unlike traditional forecasting approaches, it allows the establishment of interactive closed loops between the forecast analysis and the instruments. This sensing paradigm, based on numerous input devices and the possibility of directly process their data into a forecasting model, is changing the way meso-scale weather events are detected and will help to greatly mitigate their impact. Stream processing has the potential of enhancing the CASA/LEAD infrastructure, extending its real-time monitoring capabilities.

Real-time Social Media Analysis

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

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

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

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

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

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

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

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

An attempt to characterise the spread of social cascades focuses on improving the caching of contents exploiting the geographic information contained in the cascades [Sce+11]. This approach leverage the observation that a social cascade tends to propagate in a geographically limited area, since social connections often reflect real life relationships. It easy to see how a video spoken in a particular national language would tend to propagate within the national borders of the country in which that language is native. This study analyses a corpus of 334 millions messages shared on Twitter, extracting about 3 millions single messages with a video links. It was found that about 40% of steps in social cascades involve users that are, on average, less than 1,000 km away from each others.

2.2 Query Languages

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

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

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

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

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

CQL: Continuous Query Language

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

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

Data Types

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

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

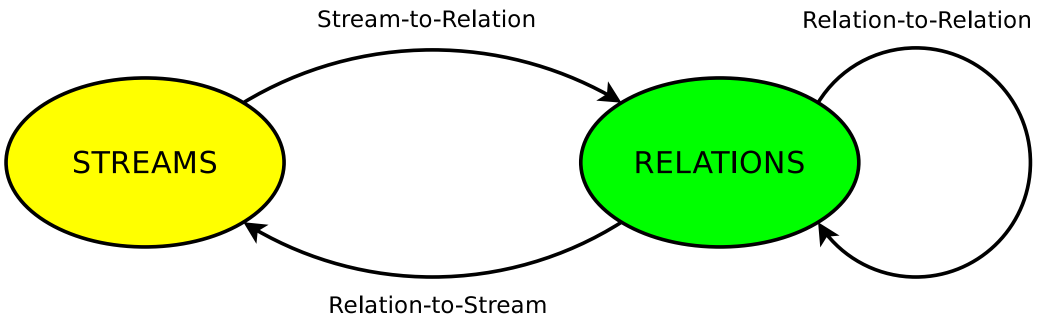

Figure 2.1: Interaction among the three classes of operators in CQL.

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

EXAMPLE: In our temperature monitoring example there would be only one base stream with schema: ROOMTMPSTR(ROOMID, ROOMTMP). Attribute ROOMID identifies the room where the sensor is located, while ROOMTMP contains the current detected temperature.

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

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

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

Operator Classes

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

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

Three kinds of window operators exists in CQL: time-based, tuple-based and partitioned. A *time-based* window contains all the tuples in the stream with timestamp within the specified boundaries. For example it could hold all the tuples arrived in the last minute. In a *tuple-based* window instead, the number of tuples is specified and fixed, so it only contains the last N tuples of the stream. A *partitioned* window is a special kind of the tuple-based. It allows the user to specify a set of attributes as a parameter and splits the original stream based on these arguments, similarly to the SQL GroupBy, and computes a tuple-based window of size N independently creating a number of substreams.

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

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

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

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

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

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

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

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

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

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

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

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

<u>Example</u>: Figure 2.2 illustrates a simple CQL query implementing out temperature monitoring example. It signals all rooms where a temperature above 30 degrees celsius is detected in the last 5

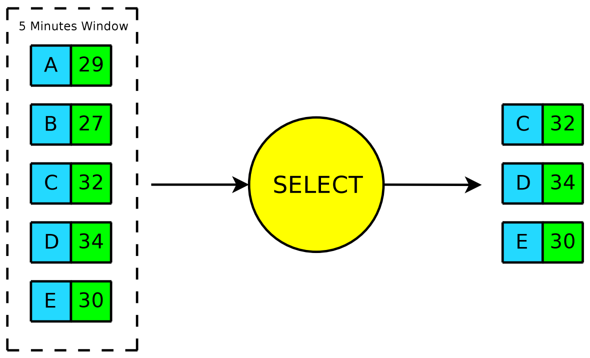

Figure 2.2: This query outputs all tuples having a temperature reading ≥ 30 in the last five minutes.

minutes. The input stream is called RoomTmpStr and carries tuples formed by a room identifier (RoomId) and a temperature reading (RoomTmp). A time-based window id applied to it, to obtain a snapshot that contains only fresh measurements generated in the last five minutes. Once this relation has been created a conditional Select is applied to it, filtering out all tuples with a temperature value lower than 30 degrees. Finally, a new stream, called HotRoomsStr, is created by an IStream relation-to-stream operator, containing tuples coming from over heated rooms, which are then fed to the air conditioning control subsystem.

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

Boxes-and-Arrows Model

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

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

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

Aggregate is a typical example of such an operator. It computes an aggregate function, like a sum or an average, over a window of tuples. A wide range of aggregate functions is provided, and others can be specified by the user. It accepts a single input stream, and it is usually preceded by a Union operator.

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

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

Example:

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

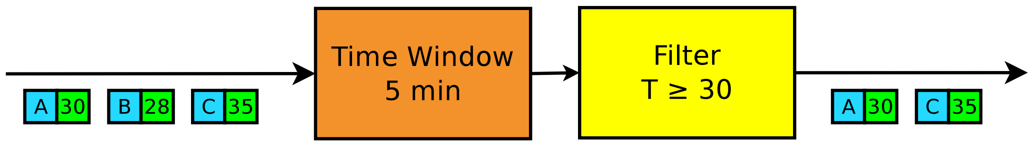

Figure 2.3: Temperature monitoring query example, expressed according to the boxes-and-arrows representation.

2.3 System Model

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

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

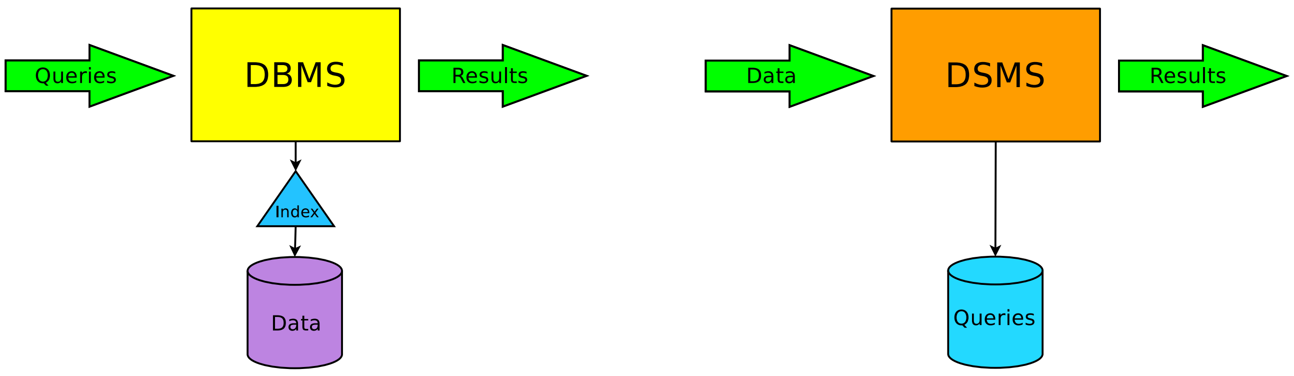

Figure 2.4: Comparison between a traditional Data Base Management System and a Data Stream Management System. On the left queries are issued over stored data, while on the right data flows through continuous queries.

Centralised

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

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

Distributed

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

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

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

Operator Placement and Load Balancing

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

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

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

When processing nodes are instead distributed over wide-are networks, a central coordinator becomes ineffective and a decentralized approach to the problem have to be employed. These algorithms make decisions based on local workload and resource information. Pietzuch et al. [Pie+06], for instance, employ a distributed algorithm based on multiple spring relaxation, trying to globally minimize a metric called network usage, based on both bandwidth and latency. Another approach has been taken by Amini et al. [Ami+06], in this case the algorithm maximises weighted throughput of processing elements, while ensuring stable operation in the face of highly bursty workloads. Zhou et al. [Zho+06] propose an algorithm in which the initial deployment is determined by minimising the communication cost among processing nodes, and then adapts to the changes in the data and workload within the system. Ying et al. [Yin+09] propose to also account for the state of operators when performing migration decisions. When the resources are administered by multiple authorities, the degree of trust and collaboration among them must be taken into account. The algorithm from Balazinska et al. [BBS04] achieve this by the mean of of prenegotiated load-exchange contracts between the nodes.

2.4 Overload Reduction

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

Load Shedding

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

Overload Detection

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

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

Load-shedding Location

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

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

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

Load-shedding Selection

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

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

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

Distributed load-shedding

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

In Borealis [Tat+08] the load-shedding problem is solved using linear optimisation. The goal is to maximise the output rate at the query end-points. The goal is to insert a number of load-shedding operators and to choose their drop selectivities, the percentage of tuples to be discarded, so that the total throughput is maximised. The solution of the load-shedding problem is broken down into four steps:consists of four steps: (i) advanced planning, (ii) load monitoring, (iii) plan selection, and (iv) plan implementation. In the first step, a series of load-shedding plan is computed, one for each predicted overload condition. Then, during the lifetime of the query, the load is constantly monitored at each node. When an overload condition is detected, one of the previously computed shedding plans is selected and implemented at the various nodes, installing the corresponding set of load-shedding operators. The idea is to prepare the system against all possible overload conditions before hand, and to modify the query at run-time depending on the overload scenario. This approach has been implemented in two ways: *centralised* and *distributed.*

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

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

Approximated Operators

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

2.5 Failure

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

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

Consistency Model

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

Borealis [Bal+05] adopts a strict consistency model which aims at eventual consistency. This means that in case of tuple loss or misordering, the system always tries to obtain the correct final result. It achieve this by marking as *tentative* a stream on which an error has occurred. Later it tries to restore the correct result by employing a revision process which allow the final result to be correct. This, of course, assumes a scenario where fault is more an exception than the rule, as the cost for this revision process can become really high. There is no mechanism to quantify the error to the user. As the error is thought to be transient and recoverable the only feedback given is that the

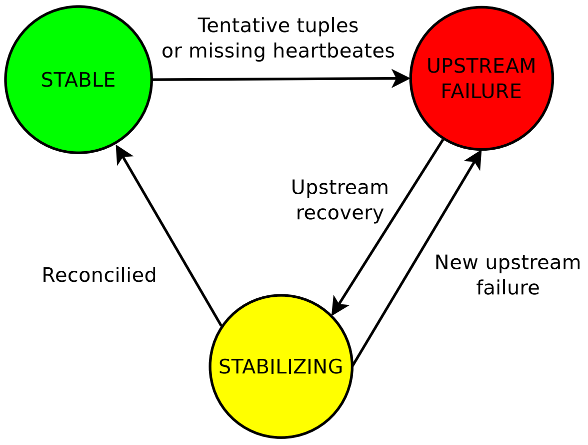

Figure 2.5: State machine describing Borealis fault tolerance mechanism.

stream content is invalid and cannot be trusted until recovery. We developed a quality-centric data model that allows the system to quantify the impact of the error and to provide the user with better feedback on the actual performance of the system. The revision mechanism used in Borealis also allow to perform "time travel". It means that it's possible to recalculate tuples of the past and even in the future, by providing a prediction formula. Rolling back to a past configuration is a heavy-duty process and needs a large amount of storage which is not always feasible due to the nature of stream processing. Figure 2.5 shows a state machine depicting the fault tolerance mechanism employed by Borealis. When failure occurs upstream an operator is alerted by the receival of tentative tuples or by the lack of heartbeat messages. It then marks his tuples as tentative, while waiting for stabilization. In the end, the failure is overcome, the state is reconciled and the system becomes stable once again.

Another more relax approach to consistency has been proposed by Murty and Welsh [MW06]. In this model, operators are considered to be *free-running*, updating their internal state independently as incoming tuples arrive. This means that two replicas of the same operator might produce different outputs, because of their difference in internal state due to failure. Instead of trying to achieve eventual consistency, they propose to deliver imperfect results, but augmenting them with two quality metrics called *harvest* and *freshness*, to provide the user with a hint about the quality of the results. Harvest represents the coverage of a query result in terms of sources and its value is reduced by failures during the query processing. An harvest of 100% indicates that the result has been computed using data coming from all sources. This metric is not directly related to correctness, aiming at providing an indication about the confidence the user should have about the delivered results. A high value for harvest indicates that the provided result represent a large number of the input sources. Freshness represents the age of the input data that generated the result. A high network latency or a high system load affect negatively the freshness value of the delivered tuples. The lower bound for freshness is the network diameter.

Replication Model

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

Early works on replication in a DSPS have been focusing on masking software failure [Hwa+05], by running multiple copies of the same operators at different nodes. Others [SHB04] strictly favoured consistency over availability, by requiring at least one fully connected copy of the query tree to be correctly working in order for the computation to proceed.
In Borealis, the approach has been to provide a configurable trade-off between availability and consistency [Bal+05]. This is done by letting the user specify a time threshold, within this time window all tuples are processed regardless of whether failures occur or not on input streams. This increase availability, since the operator continues to process even in presence of failure on its input streams. Output tuples though, are then marked as *tentative*, in order to signal the incorrectness of these results. Once the system has heal from the failure, it tries to reconcile the state of operators which processed tentative tuples, by re-running their computation with the stable data tuples. This also means that every source or operator has to buffer all their outgoing tuples, at least for a certain amount of time, to replay them in case of failure. The goal, in Borealis, is to achieve *eventual consistency*, or the guarantee that eventually all clients will receive the complete correct results.

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

2.6 Summary

This chapter presented the relevant background work useful to understand the work presented in this thesis. It started with a description of some stream processing applications, in particular those that have to deal with large data sets and can tolerate a certain degree of approximation in their results. Environmental monitoring is crucial for the timely prediction of possibly disruptive weather phenomena and for the long term understanding of world-wide climate change. Social media analysis deals with the processing of user generated content in online social networks, a valuable tool for researchers that investigate the spread of information in social cascades and for online social marketers. Then there has been a presentation of the most important query languages used to express queries in stream processing systems. The continuous query language (CQL), that extends the relational model for the processing of unbounded streams, and the boxes-and-arrows representation that employs a visual paradigm to compose queries by creating a network of operators. The next section introduced the system models for stream processing, presenting examples of systems that pioneered the centralised as well as the distributed approach. When dealing with a distributed environment the correct allocation of resources across processing sites is essential. In this regard, there has been a description of efficient resource allocation techniques for both data-center and wide-area deployments. A major focus of this work is on techniques for the management of overload, in particular load-shedding. An overview of the most relevant strategies to efficiently discard input data was provided, emphasising the different issues related to load-shedding. Finally, there has been a description of techniques used to avoid and recover from failure, with a discussion on the possible consistency and replication models. The next chapter will describe the data model developed in the scope of this work, deriving a quality metric that can used to augment streams and allow the system to detect and measure the effect of overload.