A Quality-Centric Data Model for Distributed Stream Management Systems

Marco Fiscato
Department of Computing
Imperial College London
London, United Kingdom
mfiscato@doc.ic.ac.uk

Quang Hieu Vu
Department of Computing
Imperial College London
London, United Kingdom
qhvu@doc.ic.ac.uk

Peter Pietzuch
Department of Computing
Imperial College London
London, United Kingdom
prp@doc.ic.ac.uk

ABSTRACT

It is challenging for large-scale stream management systems to return always perfect results when processing data streams originating from distributed sources. Data sources and intermediate processing nodes may fail during the lifetime of a stream query. In addition, individual nodes may become overloaded due to processing demands. In practice, users have to accept incomplete or inaccurate query results because of failure or overload. In this case, stream processing systems would benefit from knowing the impact of imperfect processing on data quality when making decisions about query optimisation and fault recovery. In addition, users would want to know how much the result quality was degraded.

In this paper, we propose a *quality-centric* relational stream data model that can be used together with existing query processing methods over distributed data streams. Besides giving useful feedback about the quality of tuples to users, the model provides the distributed stream management system with information on how to optimise query processing and enhance fault tolerance. We demonstrate how our data model can be applied to an existing distributed stream management system. Our evaluation shows that it enables quality-aware load-shedding, while introducing only a small per-tuple overhead.

1. INTRODUCTION

Today's distributed stream management systems (DSMSs) must support a class of applications that process continuous queries over a geographically-distributed set of data stream sources. Applications in many domains fall under this pattern. In healthcare, a DSMS may monitor behaviours of patients and elderly citizens across a metro area and signal emergency attention in real-time when necessary [22]. In supply chain management, DSMSs may supervise manufacturing chains to detect shipping delays before they affect production [12]. In an urban-scale sensing infrastructure [26], such systems may collect and analyse weather data to generate real-time notifications about air pollution levels and severe weather conditions affecting road users. More detail on a variety of sensor network applications can be found in [34].

Previous research on DSMSs focused primarily on high-volume financial data processing in single data centres [1, 4]. Such applications require perfect [5] and highly-available [16] data processing. They benefit from resource over-provisioning in terms of computational nodes and high-speed networks to cope with failure and workload peaks. In contrast, the large-scale DSMSs described above face a more hostile environment. Data sources are widely distributed and therefore only interconnected through unreliable wide-area network links. A set of heterogeneous processing nodes may be spread around the infrastructure at various locations, with different failure behaviour and under different administrative control [27].

In such a deployment environment, stream processing failures occur frequently due to faulty hardware, software bugs, overloaded nodes and network faults or partitions. The DSMS may not have sufficient resources to recover all lost processing after failure. While users in many domains can accept incomplete query results, they want to know about the degree of quality degradation due to imperfect processing. Answering this question is actually a challenge for existing query processing methods since they aim for perfect processing, masking the effects of failure through redundant processing or re-processing of missed tuples [15]. Imperfect stream processing usually indicates a catastrophic failure of the system.

To address this problem, we propose a new *quality-centric* stream data model. In this model, streams have associated meta-data about *weight*, *recall* and *utility* that estimates how imperfect processing has affected the quality of tuples in a stream. This model is independent of specific query semantics and can be used with existing relational DSMSs to provide continuous feedback to users on the achieved processing level. It also enables the DSMS to identify important data streams and, for example, replicate them in advance to mask future failure or optimise query processing under resource shortage by dropping least important data tuples first.

The paper makes the following three main contributions: (1) We describe a quality-centric data model that provides users with useful information about incompleteness of query results due to failures. The data model is also designed to give the system feedback on the importance of data streams when making resource allocation decisions. (2) We demonstrate how to apply our data model to estimate the correctness of query results, optimise query processing and replicate important data streams to mask failure. (3) We present our implementation of the data model as part of Borealis [1], an existing state-of-the-art DSMS, and evaluate its performance.

The rest of the paper is organised as follows. In Section 2, we discuss related work. In Section 3, we state our assumptions about the DSMS. We describe the quality-centric data model in Section 4. Three use cases for the data model are introduced in Section 5. In Section 6, we describe how to implement the data model over an existing DSMS. Our experimental evaluation that highlights benefits and overheads is presented in Section 7. The paper finishes with a discussion of future work (Section 8) and conclusions (Section 9).

2. RELATED WORK

Data stream processing has been extensively studied in the database research community. A considerable number of research projects, such as STREAM [3], Aurora [4], Gigascope [9], NiagaraCQ [8], TelegraphCQ [7] and Borealis [1], exist in this research area. All of them focus on how to process data stream queries efficiently within the limited resource capability (e.g., memory, network bandwidth and CPU) of nodes given the high rate of incoming data streams. For example, the authors of [20] propose a unit-time-basis cost model to evaluate the performance of join query processing algorithms from which they propose suitable query processing strategies for different scenarios. When tuples must be dropped due to memory limitations, the authors of [10] suggest that tuples should be dropped selectively to minimize the error of query results. The authors of [32] introduce eight requirements that need to be satisfied for real-time stream processing, handling stream imperfection being one of them.

Sensor networks carry out a specific type of distributed data stream processing and often suffer from a high rate of failure [14]. Since failure is unavoidable in many cases, it is necessary to cope with failure in sensor networks. Several solutions have been proposed to provide fault tolerance in sensor networks by detecting failure and performing failure recovery as soon as it happens [36, 28]. To support real-time query processing, most solutions allow the system to continue processing queries in case of failure [35, 30, 14]. However, none of the solutions are able to report to users (or the system) about missed data due to failure. Since missing data in query processing has an impact on the result quality, answering this question is important, especially when the scale of sensor networks becomes larger leading to global-scale sensing infrastructures [13, 25, 2, 6, 23]. Such large-scale system for stream processing continuously suffer from failures [27, 29].

Data stream models. There have been multiple data models proposed for distributed stream systems [3, 31], in general, and sensor networks [21, 11], specifically. They cover different aspects of distributed data stream systems. The authors of [21] propose a model for aggregating and routing data over sensor networks that reduces data routing cost. Deshpande et al. [11] suggest a model to obtain statistics in sensor networks, which is used to optimise data collection from sensors in query processing. On the other hand, Saleh et al. [31] describe how to use quality-of-service constraints together with queries to guarantee the quality of query results. The work by Jain et al. [18] introduces "network imprecision", a consistency metric, that allows the system to raise alerts when it becomes unstable. Our proposal for a quality-centric data model is build on top of the relational stream data model in [3]. The authors describe a stream query processing language called CQL and a mechanism for executing queries. None of the existing stream data models address the issue of missing data in query processing due to failure and the impact of missing data towards the final query result.

3. STREAM PROCESSING MODEL

Our work takes the streaming relational model proposed in [3] as a starting point. In this model, a stream S is defined as an infinite multiset of elements $\langle t, \tau \rangle$, where t is a tuple consisting of attributes belonging to a fixed schema of S and τ is the timestamp of the element. There are two types of streams: *base streams* that are the stream originating from data sources and *derived streams* that are intermediate or result streams produced by operators in queries.

Next we briefly describe our assumptions about how query operators are executed. In a relational DSMS, tuples are processed by operators for each *query window* whose size is specified as part of the query. The size of a query window can be a time interval (time-based window) or a number of tuples (tuple-based window). With a time-based window, an operator is executed at each time interval, while with a tuple-based window, it is executed after a sufficient number of input tuples was received [19].

As an example, consider an urban sensing infrastructure [26], in which sensor sources of environmental data are deployed pervasively throughout a city to obtain weather information. These sensors are interconnected using a wired backbone network with stream processing nodes at different locations. In our model, we regard source nodes to be backed by an actual sensor or a wireless sensor network. In this example, a source node can be either a *temperature sensor* that provides temperature readings in a region $T_{\text{Sensor}}(\text{temperature}, \text{region})$ or a *rainfall sensor* that measures rainfall $R_{\text{Sensor}}(\text{rainfall}, \text{region})$. Based on the above model and the data streams provided by sensors, users can issue different relational stream queries to obtain real-time information such as:

Q_1**:** get the 5-min average temperature in the city.
  Select AVG(temperature)
  From T_{Sensor} [**Range** 5 mins]

Q_2**:** get the total hourly amount of rainfall in the city.
  Select SUM(rainfall)
  From R_{Sensor} [**Range** 1 h]

Q_3**:** get all regions with snow in the past hour (assuming that a region has snow if it rains with the temperature below zero).
  Select T.region
  From T_{Sensor} T [**Range** 1 h], R_{Sensor} R [**Range** 1 h]
  Where T.region = R.region
    and T.temperature < 0 **and** R.rainfall > 0

Since we plan to combine our data model with current query processing methods, we assume existing techniques for creating query execution plans from declarative query specifications. Therefore, we will not discuss how to discover sensor sources and processing nodes, assign operators to nodes, and build or optimise execution plans for queries.

In addition, we assume that the DSMS only drops tuples deliberately. In other words, tuples are discarded by the system on purpose when there is a shortage of network or processing resources or downstream node failure. As a result, the DSMS knows the number of tuples that it dropped due to resource shortage or failure.

4. QUALITY-CENTRIC DATA MODEL

A design goal of our quality-centric data model was to be query agnostic, which is why we add meta-data to streams and keep the process of meta-data management independent from query processing. In our quality-centric model, we employ the same concepts as used in the streaming relational model above except that we introduce three additional properties to elements of a stream: *weight* w, *recall* r and *utility* u. Before describing these quality metrics in detail, we define a quality-centric stream as follows:

DEFINITION 4.1. *A quality-centric stream S is an infinite multiset of elements $\langle t, \tau, w, r, u \rangle$, where τ, w, r, and u represent the timestamp, weight, recall and utility of tuple t, respectively.*

Throughout this section, we use the query execution plan shown in Figure 1 as a running example of how to calculate values of

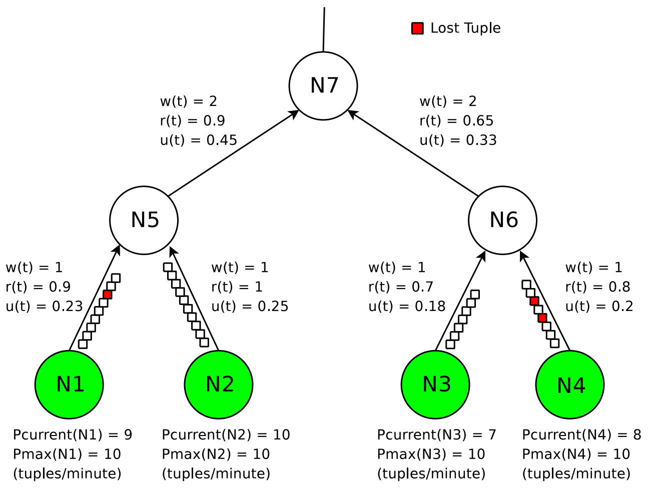


Figure 1: A sample query execution plan showing the values of the quality metrics. Since tuples are lost, the recall and utility values are lowered.

weight, recall and utility. In this example, four sensor sources N_1, N_2, N_3 and N_4 generate basic streams s_1, s_2, s_3 and s_4 at a rate of 10 tuples/minute. N_5, N_6, and N_7 are processing nodes that execute query operators once every minute (i.e., the query window size is one minute). They generate derived streams s_5, s_6 and s_7, of which s_7 is the final result of the query returned to users.

4.1 Weight

The *weight* of a tuple t, denoted as $w(t)$, describes the "importance" of data contained in t relative to the data in other tuples. In our model, we define the weight of a tuple based on the number of streams that contributed to the tuple and the weight of tuples in these streams. The intuition behind this definition is that the weight of a tuple generated from important tuples or a large number of streams should be high and vice versa. We compute the weight of tuples for different types of streams, as follows.

Weight of tuple in base stream. If we know nothing about the tuples in a base stream of a query, we treat them as equally important and assign them the same weight value. Similarly, when we know nothing about the relative importance of different base streams, the weights of all tuples in base streams are set to the same value.

In cases when we know more about the relative importance of tuples, we can assign different weights to different tuples. For example, if we know the location of sensors that output base streams, we may assign higher weights to tuples in sparsely-populated sensor locations than in densely-populated ones. The rationale behind this is that the former type of sensors provides more precious information than the latter. Alternatively, if we know the quality/accuracy of individual samples generated by a sensor, we can assign higher weights for higher quality/accuracy tuples.

In this paper, we assume that we know nothing about the sensors. Hence we set the weight of all tuples in base streams to be 1. Given that the weight of tuples in stream S at node N is denoted as $W(S, N)$, this means that in Figure 1, we have

$$W(s_1, N_1) = W(s_2, N_2) = W(s_3, N_3) = W(s_4, N_4) = 1. \quad (1)$$

Weight of tuple in derived stream. Given n input streams, denoted as $\{X_1, X_2, \cdots, X_n\}$, used to generate a derived stream S at a node N, $w(t)$ is calculated as follows.

$$w(t) = \sum_{i=1}^{n} w_{\text{avg}}(X_i, I, N) \quad (2)$$

where $w_{\text{avg}}(X_i, I, N)$ is the average weight of tuples in the i^{th} input stream in query window I. This formula states that the weight of tuples in a derived stream is the sum of the average weights of tuples in the input streams that are used to generate the derived stream. When the weights of all tuples in base streams equal to 1, the weight of tuples in a derived stream is equal to the total number of base streams that were used to generate the derived stream, up to this point. For example, according to Figure 1, we have

$$\begin{aligned} W(s_5, N_5) &= W(s_1, N_1) + W(s_2, N_2) = 2 \\ W(s_6, N_6) &= W(s_3, N_3) + W(s_4, N_4) = 2 \\ W(s_7, N_7) &= W(s_5, N_5) + W(s_6, N_6) = 4 \end{aligned} \quad (3)$$

The weight value of tuples is specific to each query. This means that tuples in the same base stream may have different weights corresponding to different queries based on their contributions to the query result.

4.2 Recall

The *recall* of tuple t in a stream specifies the ratio between the actual amount of data that was used to generate tuple t over the maximum amount of data that could have been used to generate tuple t. Details of how to compute the recall of tuples in base and derived streams are as follows.

Recall of tuple in base stream. Recall of a tuple t in a base stream describes the ratio between the amount of tuples emitted versus the maximum amount that the source node could have generated in perfect conditions. Note that while the first value is measured at runtime, the second value can be known based on the properties (and configuration parameters) of the sensor source. Usually the recall of a base stream is 1, but in some cases, for example, when the source node is congested, it may emit fewer tuples, thus lowering its recall value.

Formally we define recall as follows. Given a tuple t of a base stream S produced at a source node N, recall of the tuple t, denoted as $r(t)$, is determined as the ratio between the current amount of tuples produced by S, $P_{\text{current}}(S)$, and the maximum amount that N can generate when N is in perfect condition, $P_{\max}(S)$.

$$r(t) = \frac{P_{\text{current}}(S)}{P_{\max}(S)} \quad (4)$$

According to the above formula, when the source node N producing data stream S is in good condition achieving its maximum productivity, $r(t)$, $t \in S$, should be 1. When the source node does not work well (e.g., when its network link is interrupted or congested), $P_{\text{current}}(S)$ can be less than $P_{\max}(S)$, and hence $r(t)$, $t \in S$, can be less than 1. In our data model, $r(t)$ is calculated at each time interval (e.g., every hour) or when there is a change in the productivity of S because, say, N is running out of resources.

Let us assume that the current data rates for base streams s_1, s_2, s_3, and s_4 in Figure 1 are 9, 10, 7 and 8 tuples/minute, respectively. Since the maximum data rate of these streams is 10 tuples/minute, the recalls of tuples in s_1, s_2, s_3, s_4 are

$$\begin{aligned} t \in s_1 : r(t) = \frac{9}{10} = 0.9, t \in s_2 : r(t) = \frac{10}{10} = 1.0 \\ t \in s_3 : r(t) = \frac{7}{10} = 0.7, t \in s_4 : r(t) = \frac{8}{10} = 0.8. \end{aligned} \quad (5)$$

Recall of tuple in derived stream. Given a tuple t in a derived stream S generated at a node N by executing a query operator over a query window I from n input streams $\{X_1, X_2, \cdots, X_n\}$, recall of the tuple t, $r(t)$ is calculated as

$$r(t) = \frac{\sum_{i=1}^{n} w_{\text{avg}}(X_i, I, N) \cdot r_{\text{avg}}(X_i, I, N) \cdot F(X_i)}{\sum_{i=1}^{n} w_{\text{avg}}(X_i, I, N)} \quad (6)$$

where $w_{\text{avg}}(X_i, I, N)$ is the average weight of tuples from stream X_i received at node N in query window I, $r_{\text{avg}}(X_i, I, N)$ is the average recall value of tuples from stream X_i in query window I. $F(X_i)$ is the ratio between the amount of data sent from the upstream node generating X_i and the amount of data of X_i that is received and available for processing at the downstream node N.

$$F(X_i) = \frac{\sigma_{\text{recv}}(X_i, N)}{\sigma_{\text{send}}(X_i, N)} \quad (7)$$

In the recall formula, $F(X_i)$ refers to the amount of data that was not lost. Ideally, when processing and network transmission is perfect, $F(X_i)$ should be 1. However, under failure, $F(X_i)$ can be less than 1. In the worst case, when the network link is disconnected, $F(X_i) = 0$ since there is no received input from X_i. When some of the tuples were lost (e.g., due to load-shedding or limited capacity of a network link), we assume that the tuples were explicitly discarded by the DSMS, which therefore knows the count of lost tuples and can provide this count to downstream nodes.

By calculating $r(t)$ from $r_{\text{avg}}(X_i, I, N)$ and $F(X_i)$, $r(t)$ measures both failures that affected input tuples earlier in the query and failures that have just occurred on the last hop. The use of $w_{\text{avg}}(X_i, I, N)$ in the formula is to allow higher weight tuples to have more effect on the calculation of $r(t)$. Note that if $F(X_i) = 1, i = 1 \ldots n$, $r(t)$ is the average recall value of input streams, normalised by the weight (i.e., importance) of input tuples.

Let us assume that N_5 and N_6 in Figure 1 receive 8 tuples of s_1, 10 tuples of s_2, 7 tuples of s_3, and 6 tuples of s_4 from source nodes, respectively. In that case, we have

$$\begin{aligned} F(s_1, N_5) &= \frac{8}{9} = 0.89, F(s_2, N_5) = \frac{10}{10} = 1.00 \\ F(s_3, N_6) &= \frac{7}{7} = 1.00, F(s_4, N_6) = \frac{6}{8} = 0.75 \\ t \in s_5, r(t) &= \frac{1 \cdot 0.9 \cdot 0.89 + 1 \cdot 1.0 \cdot 1.00}{1+1} = 0.90 \\ t \in s_6, r(t) &= \frac{1 \cdot 0.7 \cdot 1.00 + 1 \cdot 0.8 \cdot 0.75}{1+1} = 0.65 \end{aligned} \quad (8)$$

Reasons for reduced recall. In general, the recall of output tuples from an operator is lower than 1 if any of the input tuples in a processed window have reduced recall. A tuple may acquire a reduction in recall due to several reasons.

(1) The first reason is *load-shedding*: some tuples need to be dropped deliberately because a node is overloaded and cannot keep up with the processing. In this situation, the rate at which tuples flow in the input streams is too high and needs to be lowered by discarding tuples. In Figure 2, node N_5 is not able to process all incoming tuples and decides to drop one of them. This affects the recall of the outgoing tuples and eventually it is also reflected in the final results. In this case, the tuples delivered to the user by N_7 have a recall value of $7/8$ instead of 1.

(2) The second case in which recall is lowered is node or network *failure*. In this situation, no tuple is output at all. The downstream node treats this as an extreme case of load-shedding — it sets the recall for the missing stream to 0 while retaining an estimated value for weight. Figure 3 shows this scenario. Here node N_4 crashed

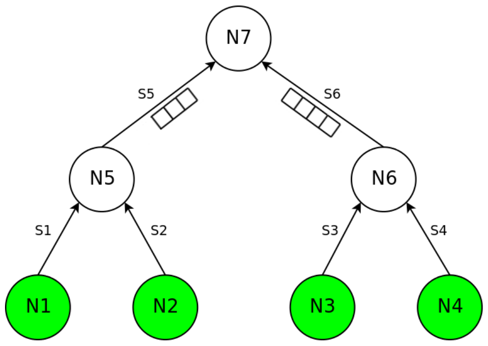


Figure 2: Node N_5 dropped a single tuple due to load-shedding. Recall for stream S_5 is consequently lowered to $3/4$ and it is finally $7/8$ at node N_7.

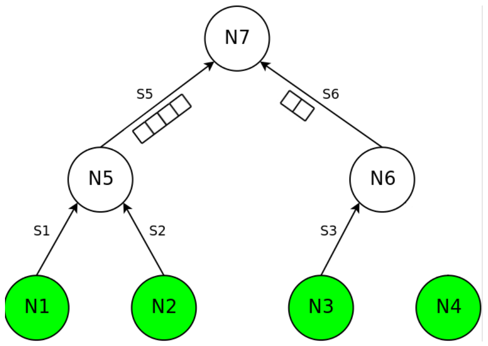


Figure 3: The network link to node N_4 failed. Recall for stream S_5 is then lowered to $1/2$ and it is finally $3/4$ at node N_7.

and does not output any tuples. Downstream node N_6 calculates the recall value of new tuples considering a recall of 0 for the missing stream S_4, assigning a recall of $1/2$ for S_6. The final recall value perceived by the user is $3/4$.

As discussed above, the recall value of tuples in a stream can vary over time. For base streams, it is re-calculated only when there is a variation in the stream data rate produced by the sources. Recall for tuples in a derived stream, instead, is calculated for each query window. This means that when a node performs a query operation over a window, it also calculates the new recall value, which is assigned to all produced tuples for this window.

4.3 Utility

The *utility* of a tuple t, denoted as $u(t)$, specifies the contribution of the tuple towards the final query result, i.e., the result returned to the user issuing the query. The utility of a tuple t in a stream S with respect to a query q is computed as

$$u(t, q) = \frac{w(t) \cdot r(t)}{|W_B(q, I)|} \quad (9)$$

where $|W_B(q, I)|$ is the maximum possible weight value calculated as the sum of the average weights of tuples in all base streams that exist in the query execution plan of q. In practice, $|W_B(q, I)|$ can be computed by a periodic two-phase data propagating process. In the first phase, the average weights of base stream tuples are forwarded up the query tree, from sources to the root node in the query plan, for computing $|W_B(q, I)|$. In the second phase, the result value is sent downwards from the root node to all intermediate nodes in the query plan.

A property of utility is that its value always lies between 0 and 1. A tuple has the maximum utility value of 1 when it is part of the result stream of a query with all sources producing data for that query in good condition and no loss occurred during query processing (i.e., the recall values of all input streams used to compute the final output are 1). When considering the value of $u(t, q)$, the closer the value is to 1, the more important t is towards the final result of q and vice versa. In other words, $u(t, q)$ is a measure of the contribution of a tuple to the whole query computation.

While the concept of utility appears to be similar to weight, it is actually different. The weight $w(t)$ specifies the importance of t measured by how many base streams have been used to produce t. In contrast, $u(t, q)$ emphasises the contribution of t towards the final query result. $u(t, q)$ is useful when we want to compare the importance of two tuples t_1 and t_2 that are used to generate the result of different queries q_1 and q_2. For example, if q_1 is a query that aggregates data from thousands of sources, even though $w(t_1)$ can be 100, it does not really contribute much to the final result. On the other hand, if q_2 is a query that operates on a smaller number of sources, even though $w(t_2) = 5$, it contributes more towards the final result. In this case, a comparison between $u(t_1, q_1)$ and $u(t_2, q_2)$ highlights this difference.

In the example in Figure 1, the utility of tuples in s_1, s_2, s_3, s_4, s_5 and s_6 towards the query q in the query execution plan are calculated as follows

$$\begin{aligned}
u(t,q) &= \frac{1 \cdot 0.90}{4} = 0.23, \quad t \in s_1\\
u(t,q) &= \frac{1 \cdot 1.00}{4} = 0.25, \quad t \in s_2\\
u(t,q) &= \frac{1 \cdot 0.70}{4} = 0.18, \quad t \in s_3\\
u(t,q) &= \frac{1 \cdot 0.80}{4} = 0.20, \quad t \in s_4\\
u(t,q) &= \frac{2 \cdot 0.90}{4} = 0.45, \quad t \in s_5\\
u(t,q) &= \frac{2 \cdot 0.65}{4} = 0.33, \quad t \in s_6 \qquad (10)
\end{aligned}$$

In contrast to weight and recall, utility does not need to be carried by tuples as meta-data in streams. Instead, it can be calculated by operators when needed because it is a function of the other two metrics and the total number of base stream weights in a query. As explained above, we assume that the number of base stream weights is known or can be computed at runtime. Tuples in the result stream at the root of the query tree have the same recall and utility values so that utility does not need to be calculated.

4.4 Discussion

Adding the three properties of weight, recall and utility to data streams helps derive useful information that can benefit both users and the system. Looking at the weight value of tuples in a stream, a DSMS is able to know how many base streams are used to generate a tuples, and hence can conclude the importance of the tuple in general. On the other hand, while a data stream can be used to answer different queries, its role in different queries may be different. In this case, the utility value of a stream specifically tells us the importance of the stream with respect to a specific query. In addition, since the utility is always between 0 and 1, this value can reveal how close the current stream is towards the final result (e.g., if the value is close to 1, the current stream is close to the final result and vice versa). Finally, the recall value of a stream quantifies the fraction of the overall amount of data that is used to generate the stream, and can be used to infer the amount of missing data.

Note that the definition of these properties is query-agnostic, in that it does not make assumptions about the specific semantics of query operators. This has the advantage that it makes the model applicable to any relational processing operator. However, it has the drawback that it can only estimate the true utility of a tuple to a user and will provide incorrect estimates for some queries. To improve the accuracy of estimation, custom equations for computing weight, recall and utility may be provided for given operators when their exact processing semantics is known.

5. APPLICATIONS OF DATA MODEL

Since the quality-centric data model provides information about "importance" of data streams (in terms of *weight* and *utility*) and incompleteness of query results (in terms of *recall*), it can applied in several ways. Next we describe three use cases: (1) to optimise query processing under resource shortage by carrying out quality-aware load-shedding; (2) to replicate important operators (and therefore associated streams) to deal with node and network failures and (3) to estimate the correctness of query results.

5.1 Quality-aware load-shedding

When a node in a DSMS has a shortage of computational or network resources while executing query operators, it becomes overloaded. Overload conditions are often transient since they may be caused by variations in stream rates or resource consumption by other processes executing on the node. For example, a sensor source may increase its sampling rate after it discovered a significant observation resulting in increased bandwidth consumption.

To relieve an overload condition, a node may have to drop tuples in streams. Mechanisms for *load-shedding* can be found in many practical DSMSs [24]. Different strategies and algorithms were proposed in the past (a) to decide *where* to drop tuples in the query to relieve overload, (b) to choose *how many* tuples to drop and (c) to select *which* particular tuples to drop from the stream.

Without information about the significance of tuples in a stream, most load-shedding mechanisms instruct nodes to drops tuples simply at random until the processing load falls below a given threshold. Using the quality-centric model, a node can prefer to discard tuples with low utility values from one or more streams. Since low utility tuples contribute less to the final query result, the quality of query results does not change significantly after such tuples were dropped. As a result, the DSMS is able to achieve better performance with limited processing resources. We describe the implementation of such a quality-aware load-shedder as part of the Borealis system in Section 6.1.

A DSMS that attempts to load-balance operators across a set of processing nodes may decide to compensate for load-shedding by assigning additional resources to the overloaded operator. For example, the DSMS may monitor the performance of each operator in a query and decide to migrate or replicate an operator if the utility of its output tuples decreases below a given threshold after load-shedding. This threshold can be specified by the user or set by the system given the amount of available resources that can still be allocated for query processing.

When assigning additional resources to a poorly-performing operator, the DSMS may either (a) migrate the operator to a new node or (b) replicate it. When the operator is replicated, the system can start a competition between the old node and the new replica and observe which node produces streams with higher utility. This competitive approach for operator replication has been proposed previously to improve overall system reliability [16, 17].

5.2 Quality-aware fault tolerance

When processing long-running, continuous queries, DSMSs must cope with network and node failures. A typical technique for achieving fault-tolerance in a DSMS is to use redundant processing of streams by replicating operators on multiple physical nodes. Previous research in this space explored the practicalities of stream replication in terms of replication strategies and consistency guarantees for replicated streams [16]. A hidden assumption in previous work is that the system has enough spare resources to support redundant processing. In a large-scale deployment environment with limited resource availability, this may not be the case and the DSMS will have to be strategic when deciding which operators to replicate.

The quality-centric model can enhance fault tolerance of the system. Since the utility values of tuples in a stream specifies the contribution of the stream towards the final query result, a DSMS can use this information to decide where to spend resources for redundant processing. If the utility values in a stream are high, users should benefit more from replicating this stream. If failure occurs affecting a single replica of a high utility stream, the system can still keep the quality of the final query result unchanged.

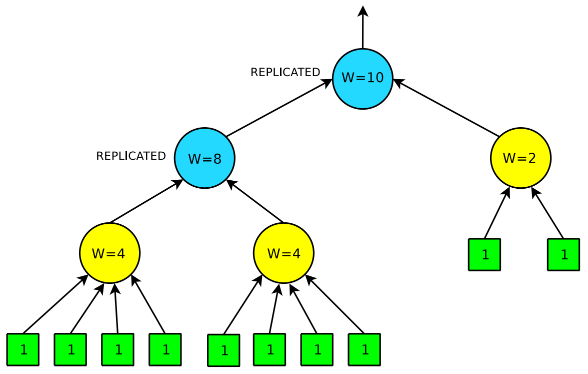
Next we describe two mechanisms for selecting operators for replication given a limited budget for additional replicas. In a *proactive mechanism* (Section 5.2.1), decisions about replication are made statically at query deployment time before failure occurs. The *reactive scheme* (Section 5.2.2) uses the change in calculated tuple utilities at runtime to decide whether to compensate for failures after they have happened.

5.2.1 *Proactive replication*

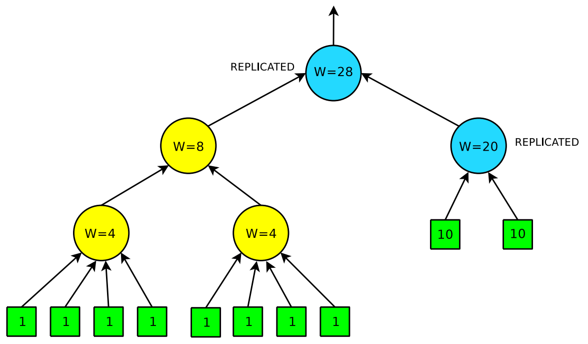
Queries in a DSMS are represented logically as abstract query trees (see Figure 1). When the DSMS has to make a proactive decision on replication, it is natural to look at the query tree to evaluate the importance of streams. This is particularly important in a resource-constraint environment since not all operators can be replicated: it is crucial to spend resources on the most valuable ones. In general, the importance of tuples increases as they moves towards the root of the tree since they contain more information and more computational resources have been spent on their processing. A tuple in a base stream contains only simple sensor data but a tuple close to the root may carry information computed after aggregating data from a large number of sources. Intuitively, when failure occurs at a node close to the root, the impact of data loss is greater than when it happens close to the leaves.

A simple replication strategy would be to start replicating operators beginning with the root of the tree, then traversing the tree in a breadth-first fashion. To achieve a dependable query processing service, as many operators as possible should be replicated given the available resources. However, such a replication strategy would only provide an optimally outcome when the query tree is perfectly balanced and all sources make an equal contribution towards the query result.

In our quality-centric data model, we can provide a proactive replication mechanism based on the *weight* of tuples in streams. Weight can be estimated at query deployment time. The DSMS first orders operators according to the weights of their output streams and then replicates as many operators as possible starting with the ones having maximum weight. The weight metric naturally captures the increase in importance of tuples flowing through the query tree. The weight of tuples in a derived stream is calculated as the sum of the average tuples weights in its input streams, thus being minimum at the leaves and maximum at the root. Tuples become more important as they propagate up the query tree. Streams with the highest weight are likely to carry tuples with the highest utility



(a) All base streams have a default weight of 1.



(b) The base streams in the right branch have higher weights.

Figure 4: Unbalanced query plan with a replication budget of 2. Different weight assignment result in different replicated nodes.

at execution time. By choosing these for replication, we increase the ability of the system to deliver tuples with maximum utility to the user.

An example of replication based on stream weight is shown in Figure 4. In the case of an unbalanced query tree in Figure 4a, replicating nodes just based on their distance from the root could lead to bad decisions. Some operators may be equi-distant from the root but carry tuples with different weights. By considering weight, the right decision can be made. Figure 4b illustrates how a DSMS may make different replication decisions, as the weights of tuples in base streams are assigned differently.

5.2.2 *Reactive recovery*

When node or network failures occur in a DSMS, the system may take action to recover from those failures. This usually means creating new instances of the failed operators on unaffected processing nodes with spare resources and updating the query accordingly. Since the system may only have limited resources available for new operator instances, it must make decisions about which operators to recover and which to ignore. Especially in the case of multiple node failures, the system has to decide which operators of which queries should be recovered with higher priority.

Using the quality-centric data model, the utility value of tuples in a stream can be used to decide in which order operators should be recovered. The higher the average utility value of the stream emitted by an operator, the higher its priority for recovery should be. Therefore, if there are not enough spare resources, the system should consider the historical average of utility values produced by a failed operator. If this value is above a given threshold, it should recover that operator. Otherwise it will deem the failure of this operator acceptable. Overall this will help maximise the utility of result tuples received by users.

5.3 Estimating correctness of query results

Based on information about the incompleteness of query results and the semantics of query operators, it may be possible to estimate the actual correctness of results for simple queries such as single aggregations. These queries have a direct relationship between the amount of missing information and the accuracy of the final result. For example, if the recall value of Q_2 from Section 3 is 0.6 and the returned value is 60 mm of rainfall, we may estimate that the correct result without failures should have be $60/0.6 = 100$ mm of rainfall (given the assumption that the amount of missing rainfall in the result is proportional to the number of failed base streams in the system). If the utility value of a query result is 0.9, the DSMSs can notify users that even though failures occurred and the result is incomplete, the true numerical value of the result is unlikely to be significantly different from the returned value.

Although utility can be used to estimate the error for simple queries, it is not a metric that can provide an estimate of the exact error of any query. In general, the utility value of result tuples has no direct correlation to the difference between the correct query result and the returned one. This is because utility is defined in a query-agnostic way. Instead, utility is an indicator of the overall processing performance of the DSMS. When the utility values of result tuples are lower than 1, the user is made aware that some failure occurred during the computation and the amount of failure is quantified as $1 - U$.

6. IMPLEMENTATION

To evaluate the quality-centric data model, we implemented it within the *Borealis* stream processing engine [1]. Borealis is a DSMS based on the *Aurora* processing engine [4] with advanced features, such as adaptive query optimisation, support for load-balancing and load-shedding and fault-tolerance mechanisms for achieving high availability. Queries are presented using a boxes-and-arrows model, in which boxes are query operators and arrows represent streams between them.

The aim of our implementation was to extend any Borealis query so that the system computes the metrics required by the quality-centric data model. This is done by modifying the original query, while retaining the same semantics. We implemented a query pre-processor that parses the original query and generates a modified version. First, the pre-processor changes the schema of tuples in streams so that they carry weight and recall values. This is done by adding two fields named `weight` and `recall` to the tuple schema. Utility is not propagated in streams since it is calculated only when needed. Next, the pre-processor updates the operators in the query. Each Borealis operator has a specific way of updating the quality metrics of tuples. We differentiate between three classes of operators:

1. Single input, single tuple: First, we consider operators that have a single input stream and only handle a single tuple at a time. Borealis' Map and Filter operators fall within this category. Such operators are trivially supported because nothing needs to be done as the recall and weight values are not modified.

2. Multiple inputs, single tuple each: A second class of operators generates an output stream based on multiple input streams, processing exactly one tuple from each input stream window at a time. This is the case for the Join, AuroraJoin and Union operators in Borealis. For Join and AuroraJoin, the quality metrics of output tuples have to be re-calculated based on the values of two input tuples. Since exactly two tuples contribute to each new output tuple, the new values can be calculated easily from within the operator simply by adding the `weight` of the two tuples and averaging their `recall`. This is done based on Equations 2 and 6.

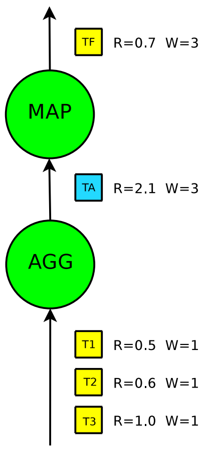


Figure 5: New weight and recall values are calculated for 3 tuples passing through an Aggregate operator. Tuple TA holds intermediate values that are corrected by an additional Map box. The final output tuple TF contains the correct new values.

The Union operator creates a single combined stream from several input streams of the same type. Since it only passes on tuples without modification, only the weight of the tuples is updated according to Equation 2.

3. Single input, multiple tuples: The third class contains the Aggregate operator, which computes an output tuple based on all tuples within a window. Here the computation of weight and recall values is more complicated. It requires a modification of the structure of the query plan by introducing additional operators. The necessary modifications are twofold: first the pre-processor adds two expressions to the aggregate operator that calculate intermediate values. Then it attaches a Map box on the output to compute the final recall value and restore the correct weight value. A Map is necessary because we cannot calculate the new recall value directly with one aggregate function. We illustrate the computation for the Aggregate operator in Figure 6.

We adopted this two step approach in the computation due to the different nature of the quality metrics. Recall is dynamic and changes at query execution time based on the amount of dropped tuples, while weight in our example never changes and can be calculated during the pre-processing step. We exploit this observation to minimise the overhead introduced by our extension by not recalculating weight dynamically. We investigate the overhead of this as part of the evaluation in Section 7.1.

6.1 Load-shedding

In Borealis, load-shedding is done by inserting *drop boxes* into the query plan. When the load manager in Borealis detects an overload condition at a processing node, it determines which operator is causing the overload and adds a drop box operator on its input streams. The drop box operator reduces the rate of incoming tuples by discarding a given number of them. Two types of drop box operators exist, RandomDrop and WindowDrop: the former discards a percentage of tuples at random, and the latter discards an entire window worth of tuples, as specified by a set of parameters.

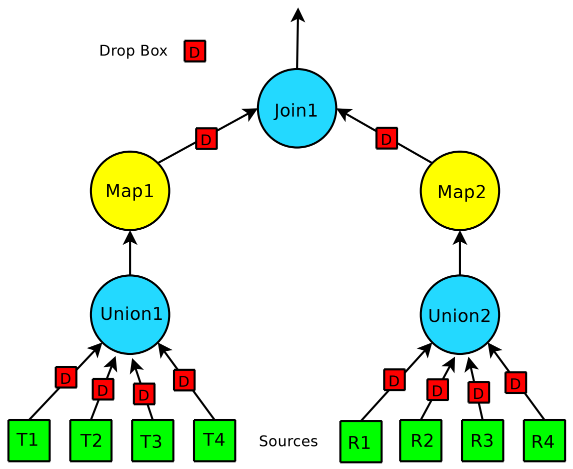


Figure 6: Three-operator Borealis query used for experiments, consisting of 8 stream sources, 2 union operators and 1 join with associated drop boxes. The 2 additional map operators are needed for the computation of the quality-centric metrics.

The main issue with these drop box operators is that they are not aware of the potentially different importance of tuples in a stream.

We extended Borealis with a new load-shedding operator called UtilityDrop that discards tuples selectively according to their utility values. It is a modified version of RandomDrop but inspects tuples to extract the quality metrics and makes decisions based on their *utility*. For every window of tuples, it calculates the amount of tuples to drop N as

$$N = \text{drop\_rate} \cdot \text{window\_size} \qquad (11)$$

where drop_rate is the fraction of tuples in the window to be dropped and window_size is the window size. The UtilityDrop operator then discards the N tuples with the lowest utility values in the window. By doing so, it manages to maximise the utility of the preserved tuples in the window.

7. EVALUATION

In this section, we describe the experiments and results of the preliminary evaluation of our quality-centric data model as an extension to the Borealis stream processing engine. The goals of our evaluation were two-fold: First, we wanted to investigate the overhead introduced by adding quality metrics to regular Borealis stream queries (Section 7.1). Second, we wanted to observe the operation of the quality-aware load-shedding mechanism and compare the quality of result tuples with and without our mechanism (Section 7.2).

We ran all of the experiments on a Intel dual core 2.1 Ghz Linux machine with the Summer 2008 distribution of Borealis. Our extensions of the Borealis code base were minor and only involved writing in the order of a hundred lines of code to implement the data model and custom drop box operators.

7.1 Operator overhead

In this experiment, we studied the performance reduction due to our quality-centric model in terms of the increase in tuple delay. As a micro-benchmark, we measured the performance decrease for each operator after adding the quality metrics. For this, we executed a synthetic query that chained 10 operators of the same type together. We then measured the increase in tuple delay with our quality-centric data model.

Operator	Increase in tuple delay
Join	1%
Map	1%
Aggregate	18%

Table 1: Micro-benchmark that shows the increase in tuple delay for each operator due to the quality-centric calculation.

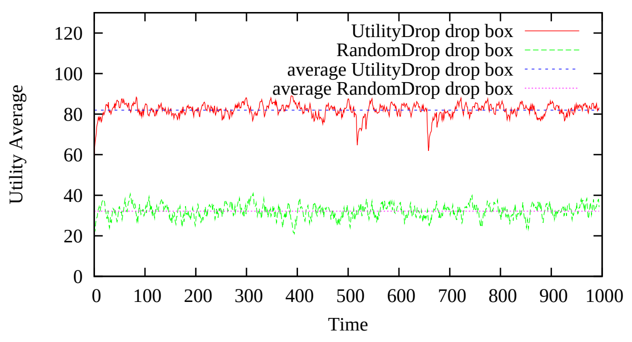


Figure 7: Comparison of load-shedding with utility drop boxes and with random drop boxes under varying tuples loss.

We show the results for each type of operator in Table 1. As expected, there is almost no impact for stateless operators. In the case of the Map operator, it only needs to propagate two extra fields. Since the Join operator in Borealis always processes two tuples at a time, the performance impact of an additional calculation is also negligible. The only significant overhead occurs for the Aggregate operator. This is due the insertion of an extra Map box, which is needed to finalise the calculation of recall and to restore the correct weight value, as described in Section 6.

7.2 Load-shedding

The next two experiments compare quality-aware load-shedding to the regular Borealis load-shedding mechanism. In both experiments, we executed the query shown in Figure 6. The query takes the union of 4 temperature sensors $T1$–$T4$ and 4 rainfall sensors $R1$–$R4$ and after that computes a join. Drop boxes at the inputs of the union and join operators can discard tuples when nodes becomes overloaded. The additional map operators are needed to update the quality metrics for the join, as explained in Section 6.

(1) In the first experiment, we varied the recall values of tuples coming from source nodes uniformly at random between 0.5 and 1. This emulated imperfect data collection from the sensor sources. The date rate of tuples from the sources was 10 tuples/second. Next we performed runs with RandomDrop and UtilityDrop drop boxes. All drop boxes were discarding tuples with a drop_rate of 0.5 emulating an overloaded system.

The graph in Figure 7 shows the observed utility values of result tuples with RandomDrop and UtilityDrop drop boxes over time. The utility values give an indication of the completeness of the result data returned to the user. As can be seen from the graph, the utility of result tuples for UtilityDrop remains higher on average, compared to RandomDrop. Due to its drop strategy, UtilityDrop discards lower utility tuples thus raising the average utility value of the stream.

(2) In the second experiment, we investigate the impact of quality-

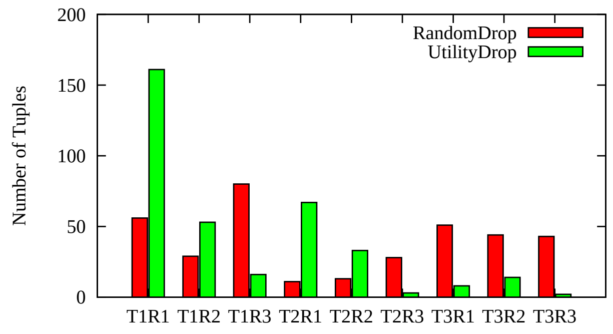


Figure 8: Distribution of result tuples after joining samples from two types of sensors. We consider the pairing with the highest number of tuples to have the highest accuracy and therefore be most desirable by the user.

aware load-shedding on the quality of the result tuples from a user's perspective. In this experiment we show that a quality-aware load-shedding leads not only to an increase in the perceived utility but also to more accurate results.

In the query in Figure 6, we assume that the temperature sensors $T1$–$T3$ and the rain sensors $R1$–$R3$ redundantly measure the same phenomenon. Sensors of each of these two types compete with each other to deliver the highest number of readings. Each sensor is impacted by adverse conditions, which we emulate by varying the drop rates of the bottom layer of drop boxes. Samples from temperature and rain sensors are combined using the Join operator. The user is interested in maximising the number of joined temperate/rain $TxRy$ tuples that originated from the same sensor. We assume that only samples obtained from the same temperature (or rain) sensor can be compared in a meaningful way.

We present the results in Figure 8. The graph shows the number of tuples in the join result for each pairing of temperature/rain sensors. The quality of the query result from the perspective of the user is proportional to the number of tuples for the temperature/rain pairing with the largest number of tuples.

We observe that in case of RandomDrop, the distribution is fairly uniform, while for UtilityDrop the $T1R1$ tuples are represented in greater numbers in the result stream compared to the other pairings. This means a more accurate result, as there are more measurements coming from the same two sensors.

The increase in user-perceived result quality is also reflected by our quality metrics. Due to the more frequently occurring pairing of measurements, we can observe an increase in average utility in the result stream. Using RandomDrop, we achieve an average utility of 0.49, while with UtilityDrop the value is higher, namely 0.73.

8. FUTURE WORK

As part of future work, we will explore the potential of our data model to estimate the correctness of query results in more depth. Since correctness estimation requires not only meta-data about data streams but also an understanding of stream and operator semantics, it relies on assumptions about the underlying query and is harder to achieve for any generic query. Given that non-trivial queries involve multiple types of operators, we plan to examine how users can provide additional input about error propagation in their queries. The DSMS can then use this information to estimate the correctness and accuracy of result tuples under the constraints of imperfect processing.

In addition, we want to explore how our quality-centric data model can support custom processing operators. In many application domains of DSMSs, relational query operators alone are not sufficient to express the custom processing needs of users. Here it would be beneficial for users to define custom operators in queries and specify how these operators may affect weight, recall and utility of output tuples.

Finally, we are building a global-scale stream processing engine that we intend to deploy on the *PlanetLab* test-bed [33]. This prototype deployment is meant to show-case our ideas on quality-aware, imperfect data processing on a large scale. We believe that the resource-constraint PlanetLab test-bed will be a good environment to explore the benefits of intelligent load-shedding and fault-tolerance given a limited set of computational resources and constant node and network failures.

9. CONCLUSION

In this paper, we have described a quality-centric stream data model that gives explicit feedback to the system and users on how much data was missed in query processing due to failure and resource shortages. The data model tracks the importance of each data stream towards the final query result. The basic idea of our data model is to add meta-data, in the form of *weight*, *recall* and *utility* metrics, to data streams. In general, weight and utility describe the importance of tuples in streams: weight measures the importance of data based on the number of streams that are used to generate an output stream, while utility evaluates the importance of data in a stream with respect to the contribution of that stream towards the final query result. The recall metric captures the amount of data that was lost during query processing.

Based on our data model, we have sketched simple solutions that perform quality-aware load-shedding when resources are limited, provide fault-tolerance with both proactive replication and reactive failure recovery, and estimate the correctness of query results. We have implemented our proposed data model as part of Borealis, a mature stream processing engine, and performed initial experiments to evaluate the efficiency and effectiveness of the data model. Our results indicate the benefits of a quality-centric stream processing model in terms of quality-aware load-shedding and show that these benefits can be achieved with an acceptable performance overhead.

10. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proceedings of the 2nd Conference on Innovative Data Systems Research (CIDR)*, 2005.

[2] K. Aberer, M. Hauswirth, and A. Salehi. Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks. In *Proceedings of the Mobile Data Management (MDM)*, 2007.

[3] A. Arasu, B. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Jounal*, 15(2):121–142, 2006.

[4] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *VLDB Journal*, 13(4), 2004.

[5] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Distributed

Stream Processing System. In *Proceedings of the SIGMOD Conference*, 2005.

[6] M. Balazinska, A. Deshpande, M. J. Franklin, P. B. Gibbons, J. Gray, M. Hansen, M. Liebhold, S. Nath, A. Szalay, and V. Tao. Data Management in the Worldwide Sensor Web. *IEEE Pervasive Computing*, 6(2):30–40, 2007.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the 1st Conference on Innovative Data Systems Research (CIDR)*, 2003.

[8] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the SIGMOD Conference*, 2000.

[9] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *Proceedings of the SIGMOD Conference*, 2003.

[10] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing over Data Streams. In *Proceedings of the SIGMOD Conference*, pages 40–51, 2003.

[11] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-driven Data Acquisition in Sensor Networks. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB)*, 2004.

[12] L. Evers and P. Havinga. Supply Chain Management Automation using Wireless Sensor Networks. *IEEE International Conference on Mobile Adhoc and Sensor Systems Conference*, 0:1–3, 2007.

[13] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An Architecture for a Worldwide Sensor Web. *IEEE Pervasive Computing*, 02(4):22–33, 2003.

[14] R. Gummadi, N. Kothari, T. Millstein, and R. Govindan. Declarative Failure Recovery for Sensor Networks. In *Proceedings of the 6th International Conference on Aspect-oriented Software Development (AOSD)*, pages 173–184, 2007.

[15] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 2005.

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

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

[18] N. Jain, P. Mahajan, D. Kit, P. Yalagandula, M. Dahlin, and Y. Zhang. Network Imprecision: A New Consistency Metric for Scalable Monitoring. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 87–102, 2008.

[19] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Cetintemel, M. Cherniack, R. Tibbetts, and S. B. Zdonik. Towards a Streaming SQL Standard. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB)*, 2008.

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

[21] B. Krishnamachari, D. Estrin, and S. Wicker. Modelling Data-centric Routing in Wireless Sensor Networks. In *Proceedings of IEEE INFOCOM*, 2002.

[22] H.-Y. Kung, C.-Y. Hsu, and M.-H. Lin. Sensor-based Pervasive Healthcare System: Design and Implementation. *Journal of High Speed Networks*, 16(1):35–49, 2007.

[23] D. Logothetis and K. Yocum. Wide-scale Data Stream Management. In *Proceedings of the USENIX Annual Technical Conference on Annual Technical Conference*, 2008.

[24] B. B. Mayur, B. Babcock, M. Datar, and R. Motwani. Load Shedding Techniques for Data Stream Systems. In *Proceedings of the Workshop on Management and Processing of Data Streams (MPDS*, 2003.

[25] S. F. Midkiff. Internet-Scale Sensor Systems: Design and Policy. *IEEE Pervasive Computing*, 2(4), 2003.

[26] R. Murty, G. Mainland, I. Rose, A. R. Chowdhury, A. Gosain, J. Bers, and M. Welsh. CitySense: An Urban-Scale Wireless Sensor Network and Testbed. In *IEEE International Conference on Technologies for Homeland Security*, 2008.

[27] R. N. Murty and M. Welsh. Towards a Dependable Architecture for Internet-scale Sensing. In *Proceedings of the 2nd conference on Hot Topics in System Dependability (HOTDEP)*, 2006.

[28] L. Paradis and Q. Han. A Survey of Fault Management in Wireless Sensor Networks. *Journal of Network and Systems Management*, 15(2), 2007.

[29] P. Pietzuch. Challenges in Dependable Internet-scale Stream Processing. In *Proceedings of the 2nd Workshop on Dependable Distributed Data Management (SDDDM)*, pages 25–28, 2008.

[30] I. Saleh, H. El-Sayed, and M. Eltoweissy. A Fault Tolerance Management Framework for Wireless Sensor Networks. *Innovations in Information Technology*, 2006.

[31] S. Schmidt, B. Schlegel, and W. Lehner. QDM: A Generic QoS-Aware Data Model for Real-Time Data Stream Processing. In *Proceedings of the 2nd International Conference on Digital Telecommunications (ICDT)*, 2007.

[32] M. Stonebraker, U. Cetintemel, and S. Zdonik. The 8 Requirements of Real-time Stream Processing. *SIGMOD Record*, 34(4):42–47, 2005.

[33] The PlanetLab Consortium. PlanetLab. `http://www.planetlab.org`, 2004.

[34] N. Xu. A Survey of Sensor Network Applications. Technical report, University of Southern California, 2003.

[35] Y. Yao and J. Gehrke. Query Processing for Sensor Networks. In *Proceedings of the 1st Conference on Innovative Data Systems Research (CIDR)*, 2003.

[36] M. Yu, H. Mokhtar, and M. Merabti. A Survey on Fault Management in Wireless Sensor Networks. In *PGNET Conference*, 2007.