THEMIS: Fairness in Federated Stream Processing under Overload

Evangelia Kalyvianaki
City University London
sbbj913@city.ac.uk

Marco Fiscato
Imperial College London
mfiscato@doc.ic.ac.uk

Theodoros Salonidis
IBM TJ Watson Research Center
tsaloni@us.ibm.com

Peter Pietzuch
Imperial College London
prp@imperial.ac.uk

ABSTRACT

Federated stream processing systems, which utilise nodes from multiple independent domains, can be found increasingly in multi-provider cloud deployments, internet-of-things systems, collaborative sensing applications and large-scale grid systems. To pool resources from several sites and take advantage of local processing, submitted queries are split into *query fragments*, which are executed collaboratively by different sites. When supporting many concurrent users, however, queries may exhaust available processing resources, thus requiring constant load shedding. Given that individual sites have autonomy over how they allocate query fragments on their nodes, it is an open challenge how to ensure global fairness on processing quality experienced by queries in a federated scenario.

We describe THEMIS, a federated stream processing system for resource-starved, multi-site deployments. It executes queries in a globally fair fashion and provides users with constant feedback on the experienced processing quality for their queries. THEMIS associates stream data with its *source information content* (SIC), a metric that quantifies the contribution of that data towards the query result, based on the amount of source data used to generate it. We provide the BALANCE-SIC distributed load shedding algorithm that balances the SIC values of result data. Our evaluation shows that the BALANCE-SIC algorithm yields balanced SIC values across queries, as measured by Jain's Fairness Index. Our approach also incurs a low execution time overhead.

1. INTRODUCTION

Federated stream processing systems (FSPSs) [14, 13] continuously process data streams using computation and network resources from several autonomous sites [9]. Submitted queries are split into *query fragments*, which can be deployed across multiple sites. For example, a cloud-based stream processing system may span more than one cloud provider to benefit from lower costs, higher resilience or closer proximity to data sources. In collaborative e-science applications, FSPSs such as OGSA-DAI [3] and Astro-WISE [1] pool resources from multiple organisations to provide a shared processing service for high stream rates and computationally expensive queries. Participatory sensing and smart city infrastructures [5, 31] require deployments of systems that combine independent domains with distinct data or processing capabilities for a large user base.

A challenge is that FSPSs are likely to suffer from long-term overload conditions. As a shared processing platform with many users, they can experience a "tragedy of the commons" [19] when users submit more queries than what can be sustained given the available resources. Instead of adopting a rigid admission policy, which rejects user queries when available resources are low, it is more desirable for an FSPS to use load-shedding techniques [33, 27]. Under load-shedding, the FSPS provides a best-effort service by reducing the resource requirements of queries through dropping a fraction of tuples from the input data streams.

Appropriate load shedding in an FSPS, however, is complicated by the fact that individual sites are autonomous and may implement their own resource allocation policies. For example, a site may prioritise queries belonging to local users at the expense of external query fragments. Without coordination of load-shedding decisions across sites, multi-site queries may experience significant variations in processing quality, depending on the load distribution across sites. *It is therefore an open challenge how to ensure that queries spanning multiple autonomous sites in an FSPS experience globally fair processing quality under overload conditions.*

Many stream processing systems support load shedding mechanisms to handle overload conditions. Load shedding mechanisms that operate at the granularity of individual nodes [33, 10, 35], however, cannot achieve fair shedding decisions for queries spanning multiple nodes. Proposals for distributed load shedding [34, 44] associate a utility function with query output rates and aim to maximise the sum of utilities, which is not a representative measure of fairness. In addition, they assume special structure and a-priori knowledge of utility functions. Load shedding decisions are controlled by a centralised entity or are based on pre-computed shedding plans—both of which are not practical in an FSPS in which domains retain control. Operator-specific semantic shedding approaches for, e.g. joins [21, 26, 17], aggregates [10, 35] or XML streams [38] cannot be applied in a federated context when users employ diverse sets of operators or customised, user-defined ones.

We describe a new approach for distributed load shedding in an FSPS that treats queries in a globally fair fashion. The key idea is to define a query-independent metric to measure the quality of processing that query fragments have experienced, and then to use this information for load shedding:

(1) We associate stream data with a metric called *source information content* (SIC), which represents the contribution of that data to the result in a query-independent way. The SIC metric quantifies the amount of source data that was used to generate a given query

result data item. Intuitively, data that was aggregated over many stream sources is considered to be more important to the final query result. The SIC metric thus decouples processing quality from the semantics of the operators and provides a query-independent way to capture the quality of query processing with respect to tuple shedding. This is particularly suited to accommodate a diverse set of user queries that executes operators of various semantics and even with user-defined operators.

(2) Overloaded nodes in the FSPS invoke a distributed semantic *fair load-shedding algorithm* that aims to balance the SIC values of query results across all queries, referred to as the BALANCE-SIC fairness policy. This policy balances the SIC values of query results (i.e. maximises the Jain's Fairness Index, a normalised scalar metric that quantifies balance). It effectively utilises the processing capacity of FSPS nodes, given the practical constraints of the placement of queries on sites and their autonomy.

When queries are assigned across FSPS sites, it becomes challenging to control per-node tuple shedding and yet provide global BALANCE-SIC fair processing. This stems from the fact that shedding tuples at a node affects its resource availability and also the processing quality of other queries. Since queries span across sites and share resources, such effects are spread across sites, affecting shedding decisions on the rest of the nodes. It is therefore non-trivial to control tuple shedding globally in a federated setting.

In our approach, each node takes independent yet informed shedding decisions about the overall processing quality of locally-hosted queries. Queries provide continuous feedback on their processing quality through the SIC metric. The shedding of tuples eventually converges to global fairness as each node continuously adjusts its shedding behaviour in response to that of other FSPS nodes.

To demonstrate the practicality of our fair load-shedding approach, we describe THEMIS, an FSPS for overloaded deployments.[1] Our evaluation of THEMIS shows that: (a) the SIC metric captures the result degradation across a variety of query types; (b) in contrast to the baseline of random shedding, THEMIS achieves 33% fairer query processing, according to Jain Fairness Index, even with skewed workload distributions; and (c) our approach has low overhead and scales well to the number of nodes and queries.

In summary, the contributions and the paper outline are:

1. a query-independent model and a metric called SIC for quantifying the quality of stream processing based on the amount of information contributed by data sources (§4) and a practical approximation for computing it (§6);
2. the definition of the BALANCE-SIC fairness in an overloaded FSPS based on the processing quality of queries; and a distributed algorithm for globally BALANCE-SIC-fair semantic load-shedding in an FSPS, which takes the loss of information suffered by queries into account (§5);
3. the design and implementation of THEMIS, an FSPS that implements efficiently the BALANCE-SIC fair load-shedding policy (§6); and
4. results from an experimental evaluation that demonstrate that the approach achieves fair query processing under various workloads in a federated setting (§7).

[1]According to the Greek mythology, THEMIS is the Titan goddess of law and order.

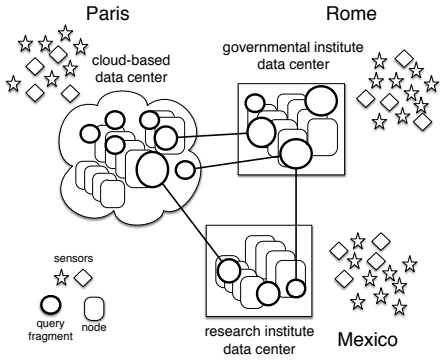


Figure 1: Example of a multi-site FSPS deployment for urban micro-climate monitoring

2. OVERLOAD IN FEDERATED STREAM PROCESSING

In this section, we describe the problem of fairness in query processing, which arises in an overloaded FSPS. We identify the key characteristics of an FSPS using an example application for query data processing over micro-climate sensor-generated data (§2.1). We then introduce the BALANCE-SIC fairness goal for an overloaded FSPS (§2.2) and discuss related work (§2.3).

2.1 Federated Stream Processing

Consider a use case of a globally-distributed FSPS for urban micro-climate monitoring. Figure 1 shows a deployment of such a system across three sites (i.e. Rome, Paris and Mexico), with environmental sensors as data sources. The FSPS collects data from a range of sensors, such as air temperature, humidity and carbon monoxide, and processes the data in real-time for analysis. Queries are issued, e.g. by government agencies for urban planning, transport authorities, citizens with respiratory problems and meteorological researchers. A sample high-level data streaming queries may continuously report: "*the 10 highest values of carbon monoxide concentration measurements on highways in Mexico every minute*" and "*the covariance matrix between measurements of (temperature, airflow) and (carbon dioxide, nitrogen) in Paris every 10 minutes*".

Each site consists of a data centre with physical nodes running a local distributed stream processing system, and we assume seamless integration across all these systems at the federated sites [12]. Below, we provide a high-level overview of data stream processing in an FSPS. Sources generate *tuples* for processing by *queries*. Queries are subdivided into *query fragments* and deployed at one or more sites. A query fragment consists of one or more *operators*, and each fragment of the same query is deployed on a different FSPS node. Query fragments use resources, i.e. CPU, memory, disk space and network bandwidth, to process incoming tuples and generate output tuples. Output tuples may be further processed by fragments of the same query, until result tuples are sent to the user issuing the query. Nodes share their resources among fragments belonging to different queries.

Below, we identify three main characteristics regarding user behaviour and resource utilisation in such an FSPS:

C1. Skewed query workload distribution. Sites primarily host queries of local users so the overall load distribution across sites may be skewed, with some sites being more loaded than others. In general, query fragments cannot be allocated uniformly across sites due to local policy constraints or the reliance on local sources. For example, queries using forecasting algorithms may be restricted to

running at a given site due to licensing constraints, which may limit the number of authorised users or remote sites using the system.

C2. Permanent resource overload. Due to the shared nature of an FSPS, we assume that the system is constantly overloaded, i.e. its resources are lower than required for perfect execution of all queries. In the above example, queries are issued by a large user population, leading to high demand. A common strategy for an FSPS to handle overload is to use *tuple shedding* [27, 33].

C3. Site autonomy. The collaborative nature of an FSPS means that a site should accept incoming queries, even under high load. However, sites belonging to different administrative domains are managed autonomously. It is therefore infeasible to assume centralised control over all tuple shedding decisions, enforced across all sites. Instead, sites elect to cooperate, having only a partial view of all resource allocation decisions across the whole FSPS.

2.2 Fairness in FSPS

The problem of how to implement *fair query processing* arises naturally in an overloaded FSPS. There exist many different approaches to address overload conditions. For example, admission control rejects incoming queries under overload [41, 40]. Such methods are not applicable in a federated context because the collaborative nature means that submitted queries must be accepted. Other approaches redistribute operators for load balancing [43, 40, 42, 11]. However, query placement in an FSPS is typically controlled by users, e.g. to leverage characteristics such as proximity.

We employ *distributed load shedding* to address overload conditions in an FSPS. By using load shedding, we assume that users agree to use the FSPS and receive *degraded query processing* for their queries. We assume that users submit queries whose results remain useful, even when their processing is degraded due to load shedding, such as aggregates [10], including averages, counts as well as top-k queries. Finally, we use *distributed* load shedding to comply with site autonomy (see C3 in §2.1).

There are two challenges to implement distributed load shedding. First, there is a need for a query-independent processing metric to capture the impact of shedding on the quality of query processing. Ideally, we require a measure for processing quality that quantifies the processing degradation under shedding but is *query-independent*, i.e. it does not have to be adapted manually to the semantics of specific queries. With such a measure, it becomes possible to compare the impact of tuple shedding across queries and hence guide shedding decisions according to a fairness policy. In §4, we introduce the SIC query-independent metric that captures the quality of processing by measuring the contribution of source tuples actually used for generating query results.

Second, depending on the deployment of query fragments to sites, some queries may get more penalised due to overload than others. We therefore want to achieve *global fairness* across all queries by enforcing load shedding at all sites so that all queries are equally penalised by the shedding. We achieve this by aiming to equalise a fairness measure of all queries after shedding. It is a challenge how to implement fairness across queries executing on overloaded, distributed and autonomous sites, regardless of their deployment. In §5, we present a new distributed load shedding algorithm that maintains BALANCE-SIC fairness of queries.

2.3 Related Work

The research community recognises the need for FSPSs, exploring relevant research challenges. Tatbul [32] argues for the integration of multiple stream processing engines for a variety of applications and pinpoints the challenges when dealing with heterogeneous query semantics. Botan et al. [12] present *MaxStream*, an FSPS for business intelligence applications. Our focus instead is on fairness in an overloaded FSPS using load shedding.

Centralised load shedding. Early proposals for load shedding focus on single-node systems [4, 33, 28]. A simple way to address overload is through *random shedding* [33] that discards arbitrary tuples. This baseline approach is easy to implement and has low overhead, however, it cannot be used to control the shed tuples.

In contrast, *semantic shedding* discards tuples using a function that correlates them with their contribution to the quality of the query output [33]. Tuples are discarded in a way that maximises result quality. Carney et al. [15] describe generic *drop-* and *value-based* functions to quantify the contribution of tuples on the result. A drop-based function specifies how the result quality of a query decreases with the number of discarded tuples. Many systems discard tuples as to maximise the output tuple rate [17, 34]. In some cases, a value-based function correlates the query output quality with the values of the output tuples [23]. In contrast, our goal is to maximise the contribution of the source tuples used for processing.

There exist semantic load shedding approaches for specific operator types, such as joins [21, 26, 17], aggregates [10, 35] and XML operators [38]. These approaches require domain knowledge of the operator semantics, while we treat operators as black-boxes.

Distributed load shedding. The problem of distributed load allocation has been studied for stream processing systems. Zhao et al. [44] consider the allocation problem for applications with tasks modelled as synchronous and asynchronous forks and joins, commenting that this approach can be applied to distributed stream processing. Their work emphasises a theoretical framework for convergence to an optimal solution and presents simulations over two queries and three output streams. In contrast, we provide a fair stream processing system based on the contributions of source tuples and evaluate a prototype implementation.

Tatbul et al. [34] employ distributed shedding to maximise the total weighted throughput of queries by computing the drop selectivity of random or window drop operators inserted at the input streams of a stream processing system. Shedding decisions are made sequentially by each node along a query, starting from the leaves and propagating through metadata up to the input nodes. They assume identical queries layouts to nodes (e.g. all root components are deployed on the same node), which is not applicable in a federated system. Finally, the scalability of the approach remains unclear, as the simulation-based evaluation only includes a handful of applications. In our prototype evaluation, we execute several hundreds of queries across tens of nodes.

Both approaches [34, 44] perform load shedding to maximise the sum of utility functions but sum maximisation does not achieve fairness. In addition, they require utility functions of special structure (either linear weighted functions [34] or concave functions [44] of rate), which does not capture query utility in practice. Finally, they require a-priori knowledge of the utility functions, which is challenging to estimate offline. In contrast, our approach targets fairness without assuming specific, a-priori utility functions. The only assumption is that the "utility" (as captured by the SIC metric) decreases with shedding and is implicitly modelled during system operation through the propagation and updating of the SIC metric in the data tuples.

An important issue for load shedding is the selection of drop locations in a query plan [33, 10]. The most efficient way is to discard tuples at upstream operators, close to sources [15, 20, 29]. This, however, is difficult to do in an FSPS because it requires global information about query plans that span multiple sites.

Metrics for query processing quality. The building block for a fairness mechanism is a metric that quantifies the processing degradation under shedding. A typical metric is the rate of query output tuples [17, 34, 44]. This metric intuitively increases when the rate of tuple shedding decreases. It remains unclear, however, how to compare performance degradation across queries with different semantics, which fundamentally may exhibit different output rates.

Our metric associates query performance with the contribution of source tuples required for perfect processing, which is related to the network imprecision metric (NI) [24] used in large-scale monitoring systems. The NI metric estimates the percentage of nodes available when calculating an aggregate value over a set of data sources. In contrast, our metric operates at the granularity of individual tuples and uses a time interval to reason about the impact of source tuples on the result.

Approximate query execution [22, 28, 15, 10, 7, 6] has the goal to reduce the resource footprint of queries by producing different, approximate answers compared to execution with abundant resources. Existing techniques in this area change operator semantics [28, 26, 17, 16, 30, 39]—instead, we approach queries as black-boxes and address overload by tuple shedding for fair processing.

Fairness. The notion of fairness in an FSPS remains largely unexplored. In early work, Babcock et al. [10] study the problem of load shedding across aggregation queries running over a single-node streaming engine. Although they do not discuss fairness directly, the goal of their work is to minimise, and hence equalise, the relative error caused by shedding for all queries. They employ a performance metric that is tailored for sliding window aggregation queries only. In a special case of their work, Zhao et al. [44] implement a weighted proportional fairness scheme on output stream rates. In contrast, we explore the concept of BALANCE-SIC fairness in the context of an overloaded FSPS.

3. FEDERATED PROCESSING MODEL

In this section, we describe our system model for an FSPS.

Data model. A tuple t is a set of three elements $(\tau, \text{SIC}, \mathcal{V})$ where τ is the tuple's logical timestamp; $\text{SIC} \in \mathbb{R}^+$ is the source information content meta-data, which we formally define in §4; and $\mathcal{V}$ is the set of payload values according to the tuple's schema. We use the notations t_τ, t_{SIC} and $t_\mathcal{V}$ to refer to the timestamp, meta-data and payload values, respectively, of a tuple t. A *stream* of tuples is an infinite, time-ordered sequence of tuples.

Query graph. A query q is a directed acyclic graph $q = (\mathcal{O}, \mathcal{M})$ where $\mathcal{O}$ is the set of operators and $\mathcal{M}$ is the set of streams. An operator takes one or more *input streams* and produces an *output stream* of *derived* tuples. Derived tuples are denoted by t^{out}. We assume that, for each operator $o \in \mathcal{O}$, there exists a time or count window that atomically emits tuples for processing by o.

In the query graph, the direction of edges indicates the direction of tuples flowing between operators. Certain operators in the query graph are connected to a finite set of *sources*, which are denoted by $\mathcal{S}$ and produce *source tuples* in time-variant rates. We assume sources of different rates per query. Every source is connected to a single operator in the query graph. We use t^s to denote that tuple t comes from source $s \in \mathcal{S}$. The domain of source tuples across sources $\mathcal{S}$ over time is given by $\mathbb{T}^S$. The timestamp t_τ indicates the time of tuple's t generation either by a source or by an operator in the case of a source or a derived tuple, respectively. There exists one *root operator* in the query graph to emit the *query result stream* containing result tuples for the user. Result tuples are denoted by t^r and the domain of result tuples is given by $\mathbb{T}^R$.

Query deployment. An FSPS consists of a set of nodes $\mathcal{D}$ with heterogeneous resources. Each node $d \in \mathcal{D}$ corresponds to a different autonomous site; without loss of generality, we focus on single-node sites as typically nodes within a site have similar capacities.

Upon deployment, the query graph is partitioned into *query fragments* that are disjoint sets of at least a single query operator. Each query fragment is deployed on a different FSPS node, and a query can span a subset or all of the FSPS nodes. We assume that query division to fragments and mapping of fragments to FSPS nodes are performed by the query user and remain the same throughout a query's execution. Finally, an FSPS node can host multiple fragments of different queries.

4. SOURCE INFORMATION CONTENT

We introduce the source information content (SIC) metric to quantify the quality of processing by accounting for the contributions of source tuples towards result tuples.

Query processing. We consider queries as black-boxes. For each query q, we use an abstract *query function* f_q to describe the collective processing by all operators in $\mathcal{O}$ over source tuples to produce result tuples, i.e. $f_q : \mathbb{T}^S \rightarrow \mathbb{T}^R$. A query function takes as input a set of source tuples T^S and outputs a set of result tuples $\mathcal{T}^R$ generated from operators processing $\mathcal{T}^S$, i.e. $f_q(\mathcal{T}^S) = \mathcal{T}^R$ where $\mathcal{T}^S \in \mathbb{T}^S$ and $\mathcal{T}^R \in \mathbb{T}^R$. Our goal is to enumerate the tuples of $\mathcal{T}^S$ and so to measure any loss of these tuples due to shedding.

Source information content. We now formally introduce the *source information content* (SIC) metric to quantify the information contribution from sources to query results.

The SIC value of a source tuple t^s_{SIC} measures its contribution to the result tuples in relation to the rest of tuples $\in \mathcal{T}^S$. We assign the SIC value of a source tuple as:

$$t^s_{SIC} = \frac{1}{|\mathcal{T}^\mathcal{S}_s||\mathcal{S}|}. \tag{1}$$

Equation (1) captures our consideration that all source tuples from a given source s, denoted by $\mathcal{T}^\mathcal{S}_s$, are equally important for the result. The more tuples a source generates, the less important each individual tuple becomes because it can be regarded as an *update*, e.g. multiple sensor readings at a higher sampling rate. Thus, the SIC value of an individual source tuple t^s is inversely proportional to $|\mathcal{T}^S_s|$ and is also normalised by the number of sources $|\mathcal{S}|$ in a query for a query-independent metric.

For a query q, the SIC value of its result tuples, q_{SIC}, is given by the sum of the SIC values of all source tuples used for the generation of the result tuples, i.e. tuples that are not dropped:

$$q_{\text{SIC}} := \sum_{t^s \in \widetilde{\mathcal{T}^S}} t^s_{SIC}, \tag{2}$$

where $\widetilde{\mathcal{T}^S} \subseteq \mathcal{T}^S$ is the set of source tuples *actually* processed for the result tuples. A source tuple is not used towards a result because it (or a tuple derived from it) was discarded due to load shedding.

The SIC value of the query result, q_{SIC}, lies in the range of $[0, 1]$. When $q_{\text{SIC}} = 1$, the result is *perfect*, and all source tuples are processed, $\widetilde{\mathcal{T}^S} = \mathcal{T}^S$. When $q_{\text{SIC}} \in (0, 1)$, the result is *degraded*, and a subset of source tuples are missing. The shed tuples are given by $\mathcal{T}^S / \widetilde{\mathcal{T}^S}$ and $f_q(\widetilde{\mathcal{T}^S}) = \widetilde{\mathcal{T}^R}$, where $\widetilde{\mathcal{T}^R} \subseteq \mathcal{T}^R$ is the set of result tuples derived from processing $\widetilde{\mathcal{T}^S}$. Finally, when $q_{\text{SIC}} = 0$, all source tuples were discarded, i.e. $\widetilde{\mathcal{T}^S} = \emptyset$.

However, it is challenging in practice to capture accurately the sets $\widetilde{\mathcal{T}^S}$, $\mathcal{T}^S$, $\widetilde{\mathcal{T}^R}$ and $\mathcal{T}^R$ because source tuples are successively transformed to derived tuples by operators and some are shed: de-

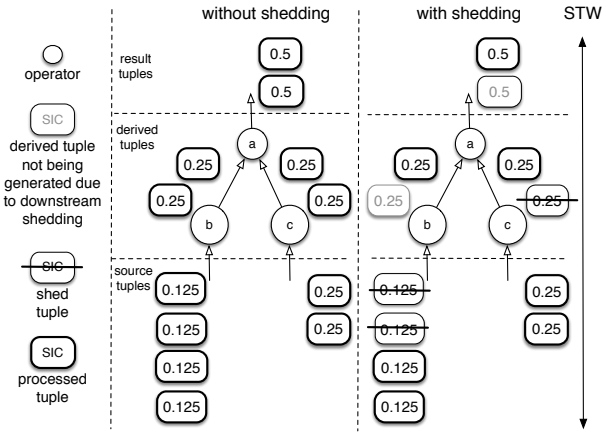


Figure 2: Example of SIC tuple values

rived tuples are "lost", e.g. due to filters and joins, which only select a subset of their input tuples. In other cases, tuples contribute to multiple derived tuples, e.g. sliding windows. Next, we introduce a practical way to (i) identify the source and result tuples in the corresponding sets of $\mathcal{T}^S$ and $\mathcal{T}^R$ such that $f_q(\mathcal{T}^S) = \mathcal{T}^R$ for perfect processing; and (ii) to measure the contribution of source tuples in $\widetilde{\mathcal{T}^S}$ for degraded processing.

Source time window. We introduce the concept of a *source time window* (STW), i.e. a period of time during which we account for all generated source tuples to contribute to certain result tuples. More formally, given a source tuple t^s and a result tuple t^r, $t^s \in \mathcal{T}^S$ and $t^r \in \mathcal{T}^R$ and $f_q(\mathcal{T}^S) = \mathcal{T}^R$ if and only if there exists a time index ρ such that $\rho \leq t^s_\tau, t^r_\tau \leq \rho + \text{STW}$. The duration of a STW should cover the *end-to-end processing latency* to allow enough time for source tuples to be processed by operators and produce result tuples. We always select a STW with significantly higher duration than the end-to-end latency, and it includes any network latencies for multi-fragmented queries.

SIC propagation from source to result tuples. Using the STW, we can calculate the query SIC value as in Equation (2). Rather than adding up the contributions of source tuples, we follow a bottom-up approach in which we propagate the SIC values of source tuples from $\mathcal{T}^S$ using derived tuples as they are processed by operators during the period of a STW. We then add the contributions of result tuples that contain the contributions of those source tuples *actually* processed towards the result tuples.

First, we calculate the SIC contribution of every derived tuple t^{out}:

$$t^{\text{out}}_{SIC} = \frac{\sum_{t \in \mathcal{T}^o_{in}} t_{SIC}}{|\mathcal{T}^o_{out}|}, \tag{3}$$

where T^o_{in} is the set of all input tuples processed atomically by operator o, producing a set of output tuples given by T^o_{out} during a STW. Recall that in our model, we always consider a time or count window that atomically emits input tuples for an operator to process. In this way, the SIC values of source tuples are gradually passed from source tuples through derived tuples to result tuples. As derived tuples are discarded, their SIC values are not accounted for because they are not part of the set of input tuples $\mathcal{T}^o_{in}$ for operator processing. Finally, the query SIC value of result tuples is:

$$q_{\text{SIC}} := \sum_{t^r \in \widetilde{\mathcal{T}^R}} t^r_{\text{SIC}}, \tag{4}$$

where we only consider result tuples $t^r \in \widetilde{\mathcal{T}^R} \subseteq \mathcal{T}^R$ that are derived from source tuples $\in \widetilde{\mathcal{T}^S}$. The use of STW captures the relationship between source and result tuples, i.e. $f_q(\widetilde{\mathcal{T}^S}) = \widetilde{\mathcal{T}^R}$.

SIC example. Figure 2 provides a numerical example of the use of SIC values on source, derived and result tuples, the use of a STW and how result SIC values are calculated in the case of perfect processing and shedding. It shows a query with three operators, a, b and c. During a STW, operator b receives 4 source tuples and outputs 2 derived tuples; operator c receives 2 source tuples and outputs 2 derived tuples; and operator a receives 4 derived tuples and outputs 2 result tuples. All SIC values for source tuples are normalised to the 2 sources. Without shedding, $q_{\text{SIC}} = 1$, $\mathcal{T}^S$ contains 6 tuples in total and $\sum_{t^s \in \mathcal{T}^S} t^s_{SIC} = 4 * 0.125 + 2 * 0.25 = 1$; and, $\mathcal{T}^R$ has 2 tuples and $\sum_{t^r \in \mathcal{T}^R} t^r_{SIC} = 2 * 0.5 = 1$. Assuming that operator b sheds two of its input tuples and operator a sheds one of its input tuples (non-shed tuples are shown with bold), then: $q_{\text{SIC}} = \sum_{t^r \in \widetilde{\mathcal{T}^R}} t^r_{SIC} = 0.5$ and $\sum_{t^s \in \widetilde{\mathcal{T}^S}} t^s_{SIC} = 0.125 + 0.125 + 0.25 = 0.5$ because some of the source tuples do not contribute to the result tuples $\widetilde{\mathcal{T}^R}$. Although not discarded, one of the input source tuples of c does not contribute to the result tuple since a derived output tuple of c is shed.

The SIC metric enables direct comparisons among queries using only their source tuples and without considering their semantics. Such a query-independent metric is key to compare shedding amongst queries and to define BALANCE-SIC fairness.

5. BALANCE-SIC FAIRNESS

Given a deployment of streaming queries in an overloaded FSPS, the problem arises how to perform load shedding in a fair manner. Shedding the same amount of data from all queries is not desirable because different queries derive different utilities from the same amount of data. In practice, such utilities may not be known a-priori before actual system deployment. In addition, FSPS sites are distributed and autonomous, and they must achieve global fairness across queries in a fully decentralised manner.

We address the fairness problem by introducing the BALANCE-SIC fairness objective. This objective seeks to equalise the queries' SIC values while fully utilising all node resources. The SIC metric represents the contribution of *processed* source tuples of a query to produce its result. It also represents the degradation of query result as opposed to the "perfect" result achieved with no load shedding. Equalising SIC values of different queries leads to utility fairness—each of the queries will generate degraded results to the same degree with respect to no shedding of its data.

We introduce the BALANCE-SIC fairness distributed algorithm (Algorithm 1) to enforce this objective. The algorithm performs load shedding at each FSPS node to comply with site autonomy. Depending on the deployment of query fragments to nodes, some queries may be more penalised due to overload compared to others. Given a deployment plan of queries to nodes, our goal is to make all SIC values of query results to converge to the same values. Note that any converged SIC values would depend on several, often time-changing, factors such as queries' arrivals and departures, tuples rates and operators processing demands and heterogeneous nodes' capacities. To this end, our algorithm strives to *opportunistically* converge to equal SIC values while respecting these constraints. Our algorithm drops tuples from input streams to achieve BALANCE-SIC fairness and does so by selecting the tuples with the highest SIC values to increase node's utilisation with fewer processed tuples.

Algorithm 1: BALANCE-SIC Stream Processing Fairness

```
1  /* all variables refer to a single node                    */
2  c := number of tuples a node can process
3  Q := set of running queries
4  T := set of tuples for a STW
5  foreach STW do
6      X = selectTuplesToKeep(c, Q);
7      shedTuples(T/X);
8  Procedure updateSIC(Q)
9      ∀ q ∈ Q update q_SIC
10 Procedure selectTuplesToKeep(c, Q)
11     while c > 0 do
12         q' = arg min_{q∈Q}(q_SIC)
13         Q' = Q/q'
14         q'' = arg min_{q∈Q'}(q_SIC, q''_SIC ≠ q'_SIC)
15         select set of tuples x from q' with aggregate SIC value
           x_SIC := Σ_{t∈x} t_sic,  such that:
16         min((q''_SIC − (q'_SIC + x_SIC)), max(x_SIC))
17             subject to  (|x| < c)
18         c = c − |x|
19         X = X ∪ x
20         updateSIC(Q);
21     return X
22 Procedure shedTuples(T/X)
23     shed all tuples ∈ T/X
```

We first describe the algorithm operation on a single FSPS node. We then describe how it achieves BALANCE-SIC fairness across multi-fragment queries running on a multi-site FSPS.

5.1 Single node Balance-SIC fairness

Consider a node running a set of single-fragment queries, denoted by $\mathcal{Q}$. Assume that the node's capacity c is known and is defined by the number of tuples that the node can process before overloading during a STW (Assumption 1), i.e. a node is overloaded when the total number of input tuples of all queries combined exceeds the node's capacity as defined by the number of tuples to process. In §6, we describe a cost model that estimates c in an online fashion. Assume that, for each query $q \in \mathcal{Q}$, the set of its source tuples $\mathcal{T}^S$ is known at the start of its STW (Assumption 2), which means that the set of all $\mathcal{T}^S$ from all $q \in \mathcal{Q}$ can be defined as $\mathcal{T}$. Note that these assumptions are made primarily for the description of the Algorithm 1. In §6, we discuss practical ways to satisfy these assumptions. Extensive evaluation results in §7 show that the implementation of our algorithm efficiently enforces the BALANCE-SIC fairness policy in an overloaded FSPS despite these assumptions.

The BALANCE-SIC Algorithm 1 is executed each STW (lines 5–7) and defines the set of tuples to shed so that all running queries $q \in \mathcal{Q}$ converge to equal result SIC values for the current STW. For simplicity, we consider all queries to have the same STW. The algorithm first executes the procedure `selectTuplesToKeep()` (line 6) to select the set of source tuples $\mathcal{X}$ to retain for processing so that all q_{SIC} values for all queries $q \in \mathcal{Q}$ converge to the same value. Then, the algorithm sheds all remaining tuples i.e. $\mathcal{T}/\mathcal{X}$ (line 7) that cannot be processed due to limited capacity.

The procedure `selectTuplesToKeep`$(c, \mathcal{Q})$ (lines 10–21) iteratively selects tuples to process from each query until the total number of tuples to keep reaches the node's capacity c. In each iteration, the algorithm selects the two queries with the least q_{SIC} values, i.e. q' and q'' (lines 12–14). Given the lowest result SIC query q' (line 12), it selects as many tuples as required from q' so that q'_{SIC} reaches the result SIC value of the second lowest query q'' (lines 15–16). If there is more that one query with the same minimum SIC values, the algorithm selects one randomly. In case the aggregate number of tuples to accept is higher than the node's capacity, the algorithm only accepts as many as possible without exceeding the node's capacity (line 17).

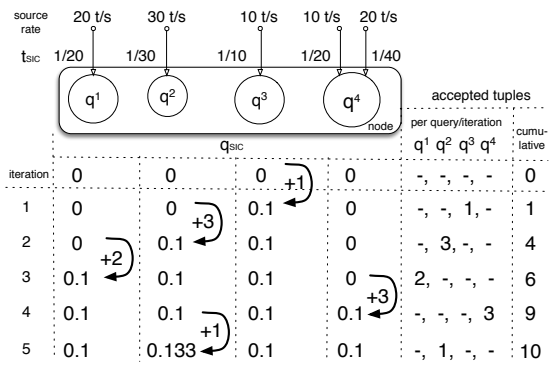


Figure 3: `selectTuplesToKeep()` on a single node with $c = 10$

For each query, the algorithm keeps the tuples with the highest SIC values, as shown in line 16 by $\max(x_{SIC})$. This maximises the aggregate SIC value of a query with the least possible number of tuples and so utilises nodes' resources efficiently. The accepted tuples are passed for processing and so q'_{SIC} is updated (line 20 and lines 8–9). We further explain in §5.2 the use of the $\mathcal{Q}$ argument in `updateSIC()` (line 20). The loop terminates when the number of accepted tuples reaches the node's capacity as measured by c. The remaining tuples $\mathcal{T}/\mathcal{X}$ are discarded (line 7) as shown by the function `shedTuples()` (lines 22–23).

The algorithm follows a gradient ascent approach to increase gradually the result SIC values of all queries while minimising the pairwise SIC differences of the two queries with the lowest SIC values. Iteratively, the algorithm minimises the pairwise SIC differences of the lowest two queries—eventually all queries' SIC values will converge as long as there are enough tuples from all queries to select (see Assumption 3).

Example of single node Balance-SIC fairness. Figure 3 shows an example of a single node with capacity c=10 with 4 single-fragment queries. Three of the queries have one source and query q^4 has two sources. Source rates and thus the normalised tuple t_{SIC} values are given. At the beginning, all queries have $q_{SIC} = 0$.

At iteration 1, the algorithm selects q^3 randomly to accept one tuple and so $q^3_{SIC} = 0.1$ At iteration 2, the lowest query is randomly selected to be q^2 and the next highest is q^3. The algorithm accepts 3 tuples from q^2 so that $q^2_{SIC} = q^3_{SIC} = 0.1$. At iteration 3, 2 tuples from q^1 are accepted to match $q^1_{SIC} = q^2_{SIC} = q^3_{SIC} = 0.1$. At iteration 4, the algorithm accepts 1 tuple from one of the q^4 sources and 2 tuples from the other source, i.e. $q^4_{SIC} = \frac{1}{20} + 2 * \frac{1}{40} = 0.1$.

After iteration 4, all queries' SIC values converge to the same value as they are all equal to 0.1. However, the node has remaining capacity ($9 < 10$) to process one more tuple, and so the algorithm accepts one more tuple (i.e. randomly from q^2) to fully utilise the node. The algorithm terminates after iteration 5 at which point the number of accepted tuples reaches 10 and equals the node's capacity. Eventually, three of the queries have exactly the same SIC values i.e. $q^1_{SIC} = q^2_{SIC} = q^3_{SIC} = 0.1$ and $q^4_{SIC} = 0.133$.

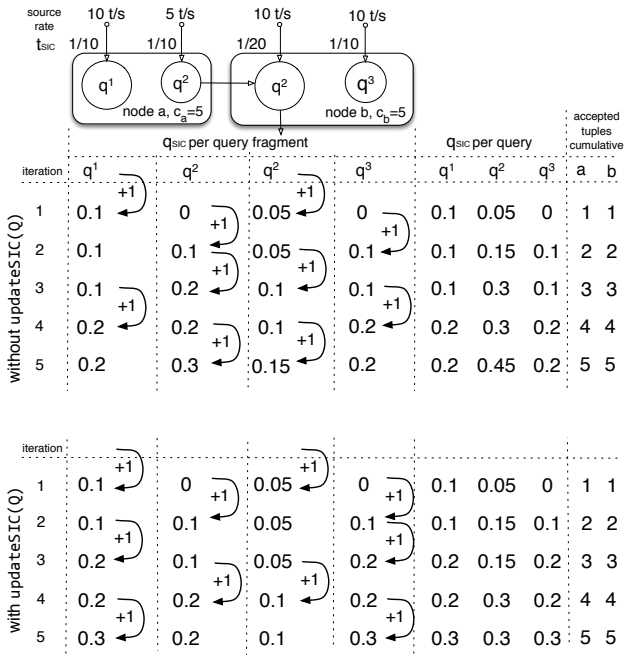


Figure 4: `selectTuplesToKeep()` on two nodes with c_a=5 and c_b=5 tuples

This example highlights that, in practice, the algorithm may not make all queries achieve exactly the same SIC values because of the involved node capacities and tuple rates, but it opportunistically tries to converge to equal SIC values.

5.2 FSPS Balance-SIC fairness

In the case of a multi-site FSPS, each node executes the same Algorithm 1 (lines 5–7). There are two main differences in the case of multi-fragment queries. First, the SIC value of a multi-fragment query is affected by the tuple shedding performed on other nodes. If such a query has the minimum SIC value among all co-located queries on the same node, the following problem emerges: all nodes running other fragments of these co-located queries attempt to maximise the SIC values of the same minimum SIC query, potentially overshooting and causing oscillations. To address this problem, the algorithm explicitly updates the result SIC value of the queries at each iteration so that all nodes eventually become aware of each other's shedding decisions (`updateSIC(`$\mathcal{Q}$`)` in line 20).

To better illustrate the use of `updateSIC(`$\mathcal{Q}$`)` for global convergence, we use the example in Figure 4 in which two nodes host three queries in total and query q^2 spans both nodes. The source rates of all queries and the nodes' capacities are shown in the figure. The top part of the figure illustrates the execution of the BALANCE-SIC without the use of the `updateSIC(`$\mathcal{Q}$`)` function. In each iteration, a node attempts to converge the SIC values of its running queries in isolation. In case of ties, the algorithm selects a query randomly. After iteration 5, the multi-fragment q^2 query has the highest result SIC value of all the single-fragment queries, and they all have different result SIC values ($q^2_{SIC} = 0.45 > q^1_{SIC} = 0.2 = q^3_{SIC} = 0.2$). This is because each node compares only the SIC values of the fragments of its hosted queries.[2]

[2]We consider that fragments of a query graph have the same structure. The SIC value of a fragment corresponds to the SIC value of the downstream operator, which outputs the tuples for the next downstream fragment in the case of multi-fragment queries, or the end-user in the case of a single-fragment query.

The bottom part of Figure 4 shows how the use of `updateSIC(`$\mathcal{Q}$`)` makes both nodes take informed shedding decisions for global convergence. At the end of each iteration, both nodes are updated with the result SIC values of their hosted queries and so become aware of each others' shedding decisions. In this case, the BALANCE-SIC algorithm converges and all result SIC values become equal ($q^1_{SIC} = q^2_{SIC} = q^3_{SIC} = 0.3$). The dissemination of the query SIC result values to the FSPS nodes hosting the query fragments for the use of `updateSIC(`$\mathcal{Q}$`)` is performed by the query coordinator described in §6.

In the description of the algorithm, we assumed no delays when updating the result SIC value of a query to the nodes hosting its query fragments: once a tuple is accepted by a query, its contribution to the result SIC value is assumed to be instantaneous (Assumption 3). This assumption is for the purposes of the explanation only—we provide a practical implementation in §6 to address delays, and our evaluation results that show that the algorithm works well with delayed updates in practice.

The second issue with multi-fragment queries is that an accepted tuple on a node from a multi-fragment query may be discarded by another node downstream, thus wasting processing resources at an upstream node. Rather than using back-pressure [44], our algorithm uses the actual SIC values of tuples to select the highest available value. Since, in our model, the SIC metric captures the importance of tuples, i.e. the higher the SIC value, the more important is the tuple, the algorithm thus always keeps the most valuable tuples ($\max(x_{SIC})$ in line 16).

The BALANCE-SIC algorithm aims to converge the SIC values of all queries by exploiting the overlap of queries fragments across nodes and by allowing each node to take informed shedding decisions based on the result SIC values of its hosted queries. In this way, nodes become aware of each others shedding decisions and converge to global fairness across the set of FSPS queries without centralised coordination. Our work considers FSPS in which all nodes are implicitly connected through the fragmentation of queries. This means that any accepted tuple on a node eventually affects the shedding on any other node transitively.

6. BALANCE-SIC FAIRNESS IMPLEMENTATION

Next we describe the integration of the BALANCE-SIC fairness approach with our FSPS system called THEMIS. THEMIS implements the performance model from §4 and associates stream tuples with meta-data about their SIC value. It uses this information for shedding decisions according to the BALANCE-SIC fairness. This section focusses on the practical aspects of the BALANCE-SIC fairness approach. In particular, we explain how to weaken our previous assumptions and, in §7, we will give experimental evidence on the convergence of the BALANCE-SIC algorithm. Figure 5 provides an overview of the architecture of a typical THEMIS node.

STW approximation. §4 introduced the STW concept as an interval, which captures the relationship between source and result tuples. The duration of the STW should exceed the end-to-end processing latency of the system in order to allow source tuples to be processed by all downstream operators. To model the continuous nature of stream processing and the generation of tuples from sources, THEMIS uses the concept of a *sliding window* to implement a STW, i.e. the STW logically slides continuously over time. This approximation allows us to handle the continuous generation of source tuples, relating them to result tuples.

There is a trade-off: the larger the STW size, the more unnecessary source tuples are part of it, i.e. ones that do not contribute to a

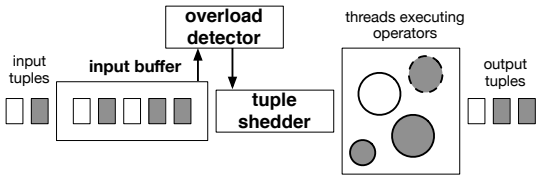


Figure 5: Architecture of a typical single THEMIS node

given result tuple. We find empirically that a STW that is an order-of-magnitude larger than the end-to-end processing delay and has a small slide, leads to an acceptable SIC precision. Furthermore, a large STW allows more tuples and therefore more queries to be selected by the BALANCE-SIC algorithm for global convergence.

SIC maintenance. In THEMIS, tuples are associated with SIC values. Based on a relational streaming model [8], each tuple has *fields* of a given schema. When an operator atomically outputs multiple tuples, they are grouped together into a *batch*. A batch contains a sequence of tuples preceded by a single header with the following fields: (a) the SIC value; (b) a unique identifier of the query that these tuples belong to; and (c) a timestamp of the tuple creation, in case of source tuples, or, of their generation, for derived tuples.

The assignment of SIC values to source tuples is performed as per Equation (1), which requires the number of source tuples per source per STW $\mathcal{T}_s^{\mathcal{S}}$ and the number of sources. Although the number of sources $\mathcal{S}$ per query is known a-priori—we consider queries with fixed sources—calculating the number of tuples in $\mathcal{T}_s^{\mathcal{S}}$ poses a challenge when source rates are unknown and time-varying. THEMIS uses the STW approximation of sliding windows to update the SIC values of all source tuples per slide online, before passing these for processing to downstream operators. By using a large STW and small slides, THEMIS can accurately capture time-changing source rates, thus providing a practical solution to relax Assumption 2 (see §5.1).

The assignment of SIC values to derived tuples is performed as per Equation (3), which requires the sets of input and output tuples. These groups are identified easily for tumbling windows before an operator emits tuples atomically for processing. In the case of sliding windows, we also provide a practical way to divide the SIC value of an input tuple across all its derived tuples per slide.

The dissemination of query result SIC values to nodes that host query fragments (i.e. `updateSIC()` in Algorithm 1) is performed by a logically-centralised *query coordinator* component. It is instantiated when a new query is deployed, and it is responsible for the query management during its lifecycle.

Overload detection. Each THEMIS node has an *input buffer* (IB) queue (Figure 5) in which all incoming tuples await processing. When the tuple arrival rate exceeds the node's capacity, the size of the IB grows, and the node overloads. To address overload, a node has an *overload detector* and a *tuple shedder* component.

The overload detector periodically checks the size of the IB. When its size exceeds a threshold c, the overload invokes the tuple shedder to discard excess tuples from the IB. The IB threshold represents the number of tuples that the node can process during a shedding interval. Its value changes and is estimated online based on a *cost model*. The tuple shedder discards batches until the size of the remaining tuples in the IB reaches c, as in Algorithm 1.

In case a node is momentarily not overloaded, THEMIS processes all tuples for all hosted queries. In this case, the BALANCE-SIC algorithm is independently invoked on all other overloaded nodes to balance SIC values across all FSPS nodes.

Cost model. The IB threshold depends on the properties of the operators. We adopt a cost model to calculate the average processing time spent on a tuple (see Assumption 1 in §5.1). This is calculated based on the number of processed tuples between successive invocations of the overload detector. We use a moving average over past estimations to calculate the average time required to process a tuple. Based on the time until the next invocation of the shedder, we estimate c. Our cost model works independently from the node's hardware capacity. It estimates online the processing capacity of any node as the average processing time per tuple. More complex cost models [37, 26] have been proposed, but they only apply to specific operator semantics.

Aggregate workload
AVG: average value of tuples every sec: `Select Avg(t.v) from Src[Range 1 sec]`
MAX: maximum value of tuples every sec: `Select Max(t.v) from Src[Range 1 sec]`
COUNT: no of tuples with values ≥ 50 every sec. `Select Count(t.v) from Src[Range 1 sec]` `Having t.v` ≥ 50
Complex workload
AVG-all: average CPU usage of nodes every sec. `Select Avg(t.v) from AllSrc[Range 1 sec]`, (13 ops)

TOP-5: top 5 nodes with largest available CPU and free memory ≥ 100 MB every sec.
`Select Top5(AllSrcCPU.id)`
`From AllSrcCPU[Range 1 sec],`
`AllSrcMem[Range 1 sec]`
`Where AllSrcMem.free` $>= 100,000$ and
`AllSrcCPU.id = AllSrcMem.id`, (29 ops)

COV: covariance of CPU usage of two nodes every sec. `Select Cov(SrcCPU1.value, SrcCPU2.value)` `SrcCPU1[Range 1 sec], SrcCPU2[Range 1 sec]`, (5 ops)

Table 1: Queries in CQL-like [8] syntax. `Src` represents a single source stream. `AllSrc` shows the union of multiple streams. For the *complex* queries, the number of operators/fragment is shown.

Tuple shedder. The tuple shedder executes Algorithm 1 in each shedding interval. The interval is a fixed configuration parameter and sets a bound on the maximum waiting time of tuples in the IB. For low latency processing, it should be set to a short interval, e.g. a few hundred milliseconds. Setting a very low threshold, however, incurs a higher overhead. In §7.2, we evaluate the impact of the shedding interval on BALANCE-SIC fairness.

To reduce the impact of delays when disseminating the result SIC values by the query coordinator to nodes hosting query fragments, the load shedder estimates the result SIC values of queries based on its local shedding. The shedder assumes that all batches in the IB are discarded and then estimates the effect of discarding them on the result SIC values. It subtracts the sum of the SIC values of batches from the current updated query result SIC. In this way, it projects the effect that its shedding will have on the result SIC value. Furthermore, the periodic invocation of the shedder enables it to make informed decisions across a STW in a step-by-step fashion. In this way, THEMIS captures the real-time nature of tuple processing and any further downstream shedding that must consider the query result SIC value. The above heuristics enables us to address Assumption 3 (see §5.2) on zero delays between shedding and updating the result SIC values. Evaluation results show that FSPS queries converge to BALANCE-SIC fairness.

7. EVALUATION

We now evaluate the BALANCE-SIC fair processing under overload in an FSPS. The goals of the evaluation are to demonstrate:

Local test-bed	
Server spec.	3 servers with 1.8 Ghz CPU, 4 GB mem, Linux 2.6.27-17-server, 1 Gbps LAN
THEMIS *deployment*	1 source node, 1 query submission node, 1 processing node
source rate	400 tuples/sec in 5 batches/sec of 80 tuples/batch per source
Emulab test-bed	
Server spec.	25 servers with 3 Ghz CPU, 2 GB mem, Linux RedHat 9, 1 Gbps LAN
THEMIS *deployment*	3 source nodes, 3 query submission nodes, up to 18 processing nodes
source rate	150 tuples/sec in 3 batches/sec of 50 tuples/batch per source

Table 2: Set-up of experimental test-beds

1. the **correlation of the SIC metric with result correctness** when query processing is degraded due to load shedding (§7.1). The results show that the SIC metric captures result correctness across different types of queries and data distributions;
2. the **effectiveness of the BALANCE-SIC fairness algorithm** in reducing the spread of SIC values across queries (§7.2). The results show that our approach is fairer than random shedding: for a mixture of multi-fragment queries deployed across 18 nodes, it has a 33% higher Jain's Fairness Index metric. We also demonstrate convergence for different shedding intervals;
3. the **scalability of the fairness algorithm** (§7.3). We show that the algorithm maintains fair query shedding with an increasing number of nodes and queries;
4. the **impact on burstiness and wide-area networks** (§7.4) in applying BALANCE-SIC fairness is minimal;
5. the **comparison against related work** (§7.5). We show that our approach outperforms [34] and works agnostically in terms of the output utility function as opposed to [44], which leads to fairer allocations in complex deployments; and
6. the **overhead of our fair shedder implementation** (§7.6). We provide evidence that the load shedder increases the execution time of a base random shedder by only 11%.

Experimental set-up. We use queries from two workloads, an *aggregate* and a *complex* workload as described in Table 1. Aggregate queries operate on a single source and perform basic aggregate functions; the complex workload implements a set of queries for monitoring the health of data center servers. It consists of monitoring queries over multiple sources, composed of various operators including average, max, top-k, group-by, filter, join and covariance.

The queries from the complex workload are split into fragments for multi-site deployment, as follows. Each fragment connects to sources and contains the same operators, performing equivalent processing as a single-fragment query in an incremental fashion. An AVG-all fragment connects to 10 sources; a TOP-5 fragment connects to 20 sources; and a COV fragment connects to 2 sources. The number of operators per fragment are shown in Table 1. For example, the most complex TOP-5 fragment consists of 29 operators, i.e. 10 CPU source data receivers, 10 memory receivers, 1 filter, 3 time windows, 2 averages, 1 join, 1 top-k and 1 output operator.

The query graphs, when running, are organised in two different ways: (1) in the case of the AVG-all query, a root fragment is connected to all other fragments and centrally aggregates partial results towards the final result forming a tree; and (2) for TOP-5 and COV queries, fragments form a chain, and the last fragment in the chain outputs the query result. Such graphs enable us to evaluate the effect of downstream shedding and the estimation of the result SIC values by the tuple shedder across two different query graphs.

Queries process either *synthetic* or *real-world* datasets. The data in the *synthetic* dataset follows either a *gaussian*, *uniform* or *exponential* distribution, with a mean of 50. We also use a *mixed* synthetic dataset, with values randomly chosen from any of the previous distributions. The *real-world* dataset are measurements of CPU and memory-related utilisation from PlanetLab nodes, as recorded by the CoTop project [36].

Table 2 describes our two *local* and *Emulab* [2] test-beds. We use the local test-bed for the SIC correlation to result correctness. We use the Emulab test-bed for the fairness and scalability experiments where all nodes are located within the same 100-Mbps LAN with a star topology and 5 ms of delay between nodes.

In all experiments, we set the STW duration to 10 secs and the shedding interval to 250 ms. The size of the STW should be set to an order of magnitude larger than the end-to-end processing delay across all queries, which emit new result tuples every 1 sec. We determined empirically that larger STW values do not effect the correct query SIC calculation. We deploy 10 TOP-5 queries with two fragments, and we measure their SIC values over STW set to 10 and 100 secs. In these underloaded cases, results show that the average SIC values across queries correctly measure perfect processing: 0.9700 ± 0.0064 and 1.0086 ± 0.0034, respectively. Therefore, we set the duration of the STW to 10 sec. In the rest of the paper, we report results over 5 minutes of execution after query deployment.

7.1 SIC Correlation

When using the SIC metric to provide feedback on processing quality, it is important that its value is correlated with the result error: a higher SIC value should indicate that the query result is *closer* to the perfect result.

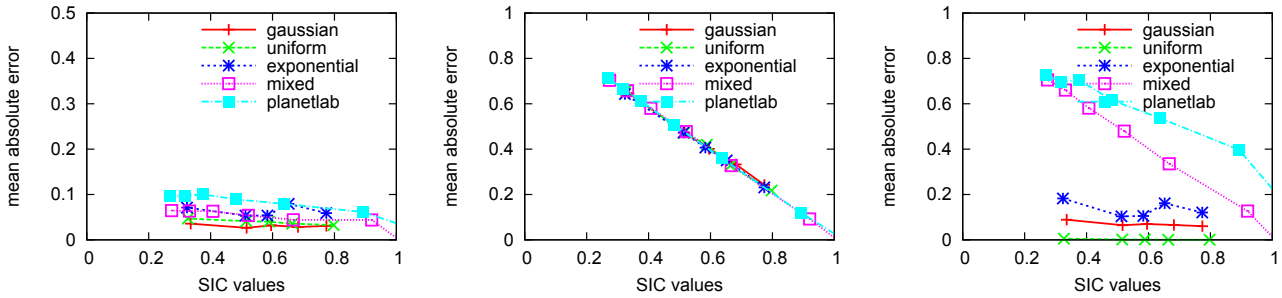
We evaluate this correlation for different types of queries and source data. We use the local test-bed to deploy queries from the aggregate workload and the TOP-5 and COV queries from the complex workload. For each query type, we increase the number of queries on a single node to emulate different degrees of overload. For the aggregate workload and the COV queries, the source rate is 400 tuples/sec and, for the TOP-5 query, it is 20 tuples/sec. The node uses a shedder that discards tuples randomly every 250 ms.

Figure 6 shows the correlation between the result correctness and the SIC values for the aggregate workload. We quantify correctness by comparing to perfect processing. We report the *mean absolute error*, which quantifies the relative distance of the degraded result values, $\tilde{t}^r_{\mathcal{V}}$ from the perfect values, $t^r_{\mathcal{V}}$ for all result tuples n:

$$\left(\sum_{\forall t^r} \left| \frac{\tilde{t}^r_{\mathcal{V}} - t^r_{\mathcal{V}}}{t^r_{\mathcal{V}}} \right| \right) / n$$

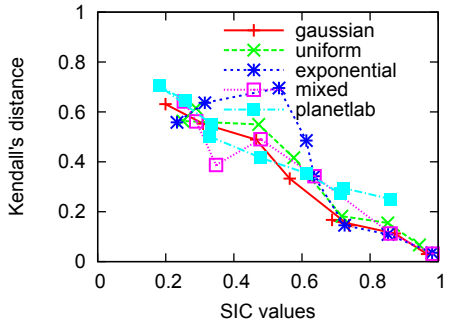
The results show that, when the SIC values increase, the error decreases, which indicates that the degraded results approach the perfect results. Depending on the query type, this correlation is stronger (e.g. for COUNT) or weaker (e.g. for AVG).

For the AVG query in Figure 6(a), we observe small changes in the error after shedding because the average value of the underlying data distribution does not significantly change. For the COUNT query in Figure 6(b), however, the error is substantially higher because degraded processing discards tuples proportionally to the overload condition. For the MAX query, there is a small error for the synthetic dataset, which increases for the mixed and real-world datasets. As for the AVG query, the maximum value of the synthetic dataset does not change significantly when some data is discarded, which is not the case for the real-world dataset.

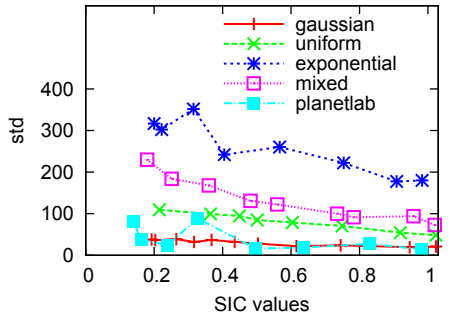


(a) AVG queries (b) COUNT queries (c) MAX queries

Figure 6: Correlation of SIC metric with result correctness for the aggregate query workload



(a) TOP-5 queries



(b) COV queries

Figure 7: SIC correlation with results for the complex workload

Next, we consider the SIC correlation for the complex workload. For the TOP-5 query, we calculate the error between perfect and degraded results using the Kendall's distance metric [18]. It counts the differences—i.e. permutations and elements in only one list—of pairs of distinct elements between two lists. We plot the normalised Kendall's distance with a maximum error of 1.

For the COV query, we discard source data at random, which produces a series of sample covariance values. Since these values are random, their expected value should match the real covariance, obtained through perfect processing. We can estimate the deviation of the values from the perfect value through the standard deviation.

The results in Figure 7 show that there is a significant correlation. As the result SIC value increases, the distance/error to the perfect result decreases. For example, the TOP-5 query shows similar behaviour across synthetic and real-world datasets. For the COV query, the error is more prominent in the real-world than the synthetic dataset because discarding values from the synthetic distribution does not significantly change the covariance of the distribution.

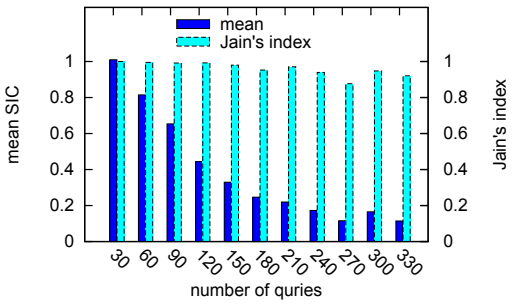


Figure 8: Single-node fairness

7.2 Balance-SIC Fairness

Next we evaluate the effectiveness of the BALANCE-SIC fairness approach to converge the result SIC values of queries. We start by validating that our approximation of the STW allows the algorithm to converge to BALANCE-SIC fairness on a single node. We also explore the behaviour of the algorithm for different shedding intervals. We continue to evaluate our approach in the case of a multi-node deployment, comparing to the baseline of random shedding. Finally, we study the effect of the ratio of multi-fragmented queries on the level of achieved fairness.

To measure the effectiveness of the BALANCE-SIC fairness approach, we use the *Jain's Fairness Index metric* [25]:

$$\text{Jain's index} := \frac{\left(\sum_{\forall q \in \mathcal{Q}} q_{\text{SIC}}\right)^2}{|\mathcal{Q}| * \sum_{\forall q \in \mathcal{Q}} (q_{\text{SIC}})^2},$$

where $\mathcal{Q}$ is the set of queries executed by the FSPS. Intuitively, the metric captures the proportion of queries whose differences in SIC values are small. The Jain's index values range from $1/|\mathcal{Q}|$ to 1; the higher the value, the fairer the system. When its value is 1, all queries have the same result SIC value.

Source time window approximation. We first explore the convergence of our algorithm to BALANCE-SIC fairness, validating the use of sliding window STW (see §6). We deploy queries of the complex workload on a local test-bed node. Over the course of this experiment, we did not observe any oscillation of SIC values and the standard deviation of values remained low.

Figure 8 shows that, with more queries, the load on the node increases, and the mean SIC values across queries decreases when more tuples are discarded. Our approach equalises the SIC values as shown by the Jain's index values close to 1. These results indicate that, even under extreme overload, i.e. when the total incoming input rate for all queries is 528,000 tuples/sec, the mean SIC value drops to values around 0.1 and the majority of the tuples are dropped, our approach can enforce fairness across queries.

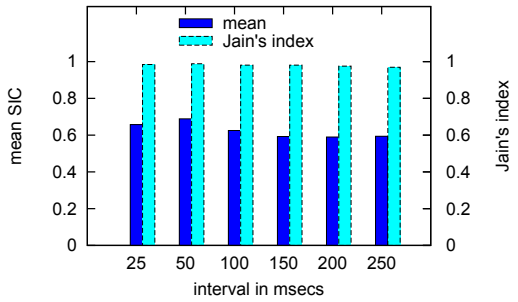


Figure 9: Shedding interval

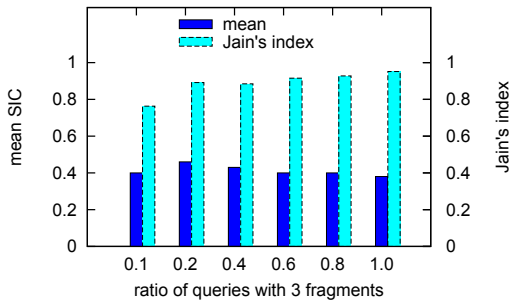


Figure 11: Fairness for multi-fragmented queries

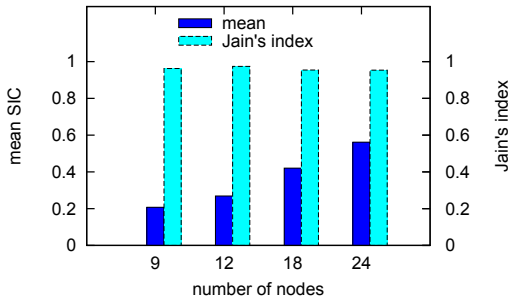


Figure 12: Fairness for increasing number of nodes

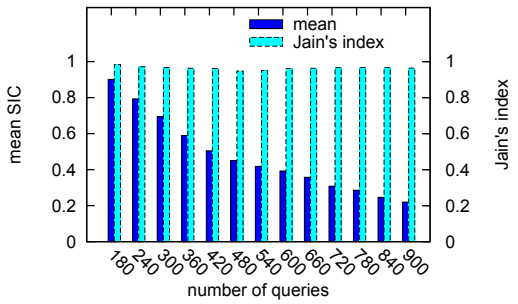


Figure 13: Fairness for increasing number of queries

Shedding interval. We study the effect of different shedding intervals, as shown in Figure 9. The shedding interval corresponds to the slide of the STW. In each case, we deploy 200 queries from the complex workload on 6 nodes, and each query has between 1 and 3 fragments. The results show that our algorithm achieves fair shedding regardless of the shedding interval, as shown by the close values for the mean and high values Jain's index metric.

BALANCE-SIC fairness across multiple nodes. Next, we deploy our fair shedding approach across 18 nodes. As we do not know the optimal fairness solution, we compare against random shedding as a practical baseline. We deploy multi-fragment, complex queries. We report results as we vary the number of fragments per query from 1 to 6 and when queries have a random number of fragments between 1 and 6 (mixed case). In all cases, the total number of fragments is the same, approximately 2,000.

Figure 10 shows that the BALANCE-SIC fair shedder provides a fairer solution in all cases compared to the random shedder. For the most realistic mixed workload, BALANCE-SIC is 33% better than the random approach, according to the Jain's index. In addition, BALANCE-SIC reduces the spread of SIC values across queries, as shown by the standard deviation (std) in Figure 10(b). The BALANCE-SIC shedder accepts tuples from the most degraded queries while it sheds tuples from the least degraded ones. In doing so, it better utilises resources towards processing more valuable source tuples, as supported by the increased mean SIC values in Figure 10(c). In contrast, the random shedder blindly sheds tuples regardless of their upstream processing.

Multi-fragmentation. We rely on query fragmentation to achieve our fairness. We study the effect that different ratios of multi-fragmented queries have. We vary the ratio of three-fragment over single-fragment queries from the complex workload across 10 nodes. The total number of fragments is about 2,000, and each node has roughly the same load. Figure 11 shows that, when more queries are multi-fragmented, the BALANCE-SIC fairness algorithm converges to a fairer system, as more queries span nodes.

7.3 Scalability

Next we evaluate the scalability of the BALANCE-SIC fairness algorithm. First we increase the number of nodes while hosting a constant number of 500 queries. We deploy queries from the complex workload and vary their fragments randomly from 1 to 6. Fragments are deployed according to a Zipf distribution.

As shown in Figure 12, BALANCE-SIC manages to utilise the increasing number of nodes as the mean SIC values increase. In all cases, our approach sheds tuples fairly, as the Jain's index metric approaches to 1. In Figure 13, we increase the number of queries for a fixed deployment on 18 nodes. The results show that, with more queries, tuples are discarded fairly. This is even the case when the large number of queries strain the resources of the system, decreasing the mean SIC value.

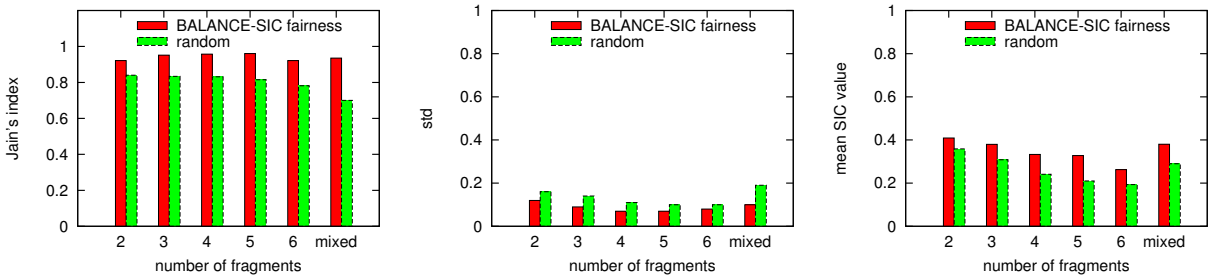
7.4 Burstiness and wide-area networks

We evaluate BALANCE-SIC fairness with bursty sources over a wide-area network, referred to as FSPS. We deploy THEMIS over a network with 4 nodes, and we emulate wide-area latencies between nodes of 50 ms. To introduce bursty source rates, we modify sources so that 10% of the time they generate tuples at 10× their normal rate. We also deploy queries on a LAN without burstiness as a baseline. We deploy queries from the complex workload with two query fragments, which are randomly assigned to nodes.

Figure 14 shows the average SIC values of queries after carrying out BALANCE-SIC shedding. The results show that, regardless of the deployment set-up, the average SIC values remain similar. This indicates that THEMIS can achieve BALANCE-SIC fairness in the presence of bursty sources and variation in nodes' latency.

7.5 Comparison against related work

We compare against the work on distributed data stream processing allocation from [34] and [44]. Both formulate a centralised optimisation problem: [34] maximises the sum of the outputs of weighted queries, and [44] maximises the sum of the utilities of query outputs. Both approaches rely on a-priori knowledge of query



(a) Jain's index (b) std (c) mean

Figure 10: Comparison of BALANCE-SIC fairness against random tuple shedding

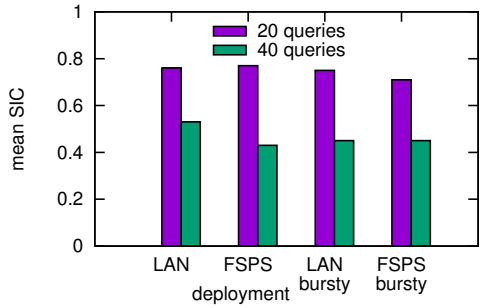


Figure 14: Fairness with source burstiness and wide-area latencies

parameter values. To address this limitation, we use the THEMIS cost model to calculate any unknown values.

We formulate the centralised problem of a fixed query deployment that closely matches the simple set-up used in the evaluation of [34]. It consists of two nodes and multiple two-operator queries in which all operators connecting to sources are collocated on the same node. We use 60 AVG-all queries of two fragments running on 2 nodes. As in [34], we use the GLPK tool to solve the optimisation problem. The optimal solution allows 3 out of the 60 queries to process all of their input tuples: one query discards a fraction of its input tuples; and all the other queries discard all of their tuples, which is clearly not a fair solution. This optimisation problem is not formulated to balance the throughput across queries (although all queries weights are set to 1).

To compare with [44], we first solve the problem for the same simple set-up of [34] as above, using the formulation of [44] and logarithmic output utility functions. For this simple set-up, [44] yields a fair solution as our approach. We further compare using a complex deployment of 20 AVG-all queries (3 fragments each), 20 COV and 20 TOP-5 queries (2 fragments each), and randomly deploying their fragments on 4 nodes. The solution of [44] is obtained using Matlab and the Jain's fairness index for the resulting utilities' distribution (normalised log-output rates) equals 0.87. This solution is less fair than our BALANCE-SIC solution where the Jain's fairness index for the resulting SIC values equals to 0.97, which is close to perfect fairness. Note that the solutions based on [34] are only applicable to concave utility functions.

Both approaches [34, 44] aim to solve the distributed shedding problem for specific deployments, and they rely on a-priori knowledge of values. Although [44] gives a fair allocation in simple deployments, with complex deployments and varying workloads, BALANCE-SIC provides a fairer allocation.

7.6 Overhead

We finally consider the overhead of BALANCE-SIC in terms of additional meta-data bytes and shedder execution time. In our prototype implementation, we use 10 bytes to store the SIC value per batch, which is a small amount compared to actual stream data.

In addition, our fair load shedder increases execution time only insignificantly compared to a random shedder. The average execution time per batch spent by the fair and the random shedders for the mixed workload (see Figure 10) are 0.088 ± 0.26 msec and 0.079 ± 0.32 ms, respectively. In summary, the BALANCE-SIC shedder incurs a 11% overhead.

Finally, each query coordinator submits the result SIC values to a query fragment as needed by `updateSIC()` in Algorithm 1. This creates a message of 30 bytes, which is sent at regular intervals to all query fragments. In the evaluation, we used intervals of 250 ms to match the shedding interval for update shedding. The additional data sent between nodes because of this operation is negligible when compared to high tuple rates in data stream processing.

8. CONCLUSIONS

It will be increasingly important to handle long-lasting overload in federated stream processing systems (FSPSs). This requires new decentralised designs to provide fairness in queries processing.

We describe a new fairness model for FSPSs under overload. We associate tuples with their *source information content* (SIC), i.e. the amount of information from data sources that they contain. This approach quantifies the perceived contribution of a tuple, without being dependent on processing semantics. We provide evidence that the contribution of source tuples is correlated with the quality of query results. We provide a new definition of BALANCE-SIC processing fairness in the context of FSPSs according to SIC values and describe a distributed algorithm that enables individual sites in an FSPS to perform BALANCE-SIC fair tuple shedding, reducing the skew in experienced processing quality.

Acknowledgments. This research work was partially supported by the EPSRC DISSP grant (EP/F035217/1), and the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-06-3-0001.

9. REFERENCES

[1] Astronomical Wide-field Imaging System for Europe. http://www.astro-wise.org/.

[2] Network Emulation Testbed. http://www.emulab.net.

[3] OGSA-DAI: Open Grid Services Architecture. http://ogsadai.org.uk/.

[4] D. J. Abadi, D. Carney, et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB*, 12(2), 2003.

[5] K. Aberer, M. Hauswirth, and A. Salehi. A Middleware for Fast and Flexible Sensor Network Deployment. In *VLDB*, 2006.

[6] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when You'Re Wrong: Building Fast and Reliable Approximate Query Processing Systems. In *SIGMOD*, 2014.

[7] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*, 2013.

[8] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB*, 15(2), 2006.

[9] A. Avetisyan, R. Campbell, et al. Open Cirrus: A Global Cloud Computing Testbed. *Computer*, 43(4), 2010.

[10] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *ICDE*, 2004.

[11] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based Load Management in Federated Distributed Systems. In *NSDI*, 2004.

[12] I. Botan, Y. Cho, R. Derakhshan, N. Dindar, L. Haas, K. Kim, and N. Tatbul. Federated Stream Processing Support for Real-Time Business Intelligence Applications. In *Enabling Real-Time Business Intelligence*, volume 41. 2010.

[13] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *PVLDB*, 3(1-2), 2010.

[14] I. Botan and Y. C. et al. Design and Implementation of the MaxStream Federated Stream Processing Architecture. Technical Report 632, ETH, 2009.

[15] D. Carney, U. Çetintemel, et al. Monitoring Streams: A New Class of Data Management Applications. In *VLDB*, 2002.

[16] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate Query Processing using Wavelets. *VLDB*, 10(2), 2001.

[17] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing over Data Streams. In *SIGMOD*, 2003.

[18] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. In *SODA*, 2003.

[19] H. Garrett. The Tragedy of the Commons. *Science*, 162(3859), 1968.

[20] B. Gedik, K.-L. Wu, and P. Yu. Efficient Construction of Compact Shedding Filters for Data Stream Processing. In *ICDE*, 2008.

[21] B. Gedik, K.-L. Wu, P. Yu, and L. Liu. GrubJoin: An Adaptive, Multi-Way, Windowed Stream Join with Time Correlation-Aware CPU Load Shedding. *TKDE*, 19(10), 2007.

[22] L. Golab and M. T. Özsu. Issues in Data Stream Management. *SIGMOD Rec.*, 32(2), 2003.

[23] A. Jain, E. Y. Chang, and Y.-F. Wang. Adaptive Stream Resource Management using Kalman Filters. In *SIGMOD*, 2004.

[24] N. Jain, P. Mahajan, et al. Network Imprecision: A New Consistency Metric for Scalable Monitoring. In *OSDI*, 2008.

[25] R. Jain, D. Chiu, and W.Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer System. Technical Report 301, DEC, 1984.

[26] J. K. Jeffrey, J. F. Naughton, and S. D. Viglas. Evaluating Window Joins over Unbounded Streams. In *ICDE*, 2003.

[27] B. B. Mayur, B. Babcock, M. Datar, and R. Motwani. Load Shedding Techniques for Data Stream Systems. In *MPDS*, 2003.

[28] R. Motwani, J. Widom, et al. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *CIDR*, 2003.

[29] C. Olston, J. Jiang, and J. Widom. Adaptive Filters for Continuous Queries over Distributed Data Streams. In *SIGMOD*, 2003.

[30] F. Reiss and J. M. Hellerstein. Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In *ICDE*, 2005.

[31] I. Schweizer, C. Meurisch, et al. Noisemap - Multi-tier Incentive Mechanisms for Participative Urban Sensing. In *PhoneSense*, 2012.

[32] N. Tatbul. Streaming Data Integration: Challenges and Opportunities. In *ICDEW*, 2010.

[33] N. Tatbul, U. Çetintemel, et al. Load Shedding in a Data Stream Manager. In *VLDB*, 2003.

[34] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In *VLDB*, 2007.

[35] N. Tatbul and S. Zdonik. Window-Aware Load Shedding for Aggregation Queries over Data Streams. In *VLDB*, 2006.

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

[37] S. Wang and E. Rundensteiner. Scalable Stream Join Processing with Expensive Predicates: Workload Distribution and Adaptation by Time-slicing. In *EDBT*, 2009.

[38] M. Wei, E. A. Rundensteiner, and M. Mani. Utility-Driven Load Shedding for XML Stream Processing. In *WWW*, 2008.

[39] M. Wei, E. A. Rundensteiner, and M. Mani. Achieving High Output Quality Under Limited Resources Through Structure-based Spilling in XML Streams. *PVLDB*, 3(1-2), 2010.

[40] J. Wolf, N. Bansal, et al. SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems. In *Middleware*, 2008.

[41] C. H. Xia, D. Towsley, and C. Zhang. Distributed Resource Management and Admission Control of Stream Processing Systems with Max Utility. In *ICDCS*, 2007.

[42] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing Resiliency to Load Variations in Distributed Stream Processing. In *VLDB*, 2006.

[43] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic Load Distribution in the Borealis Stream Processor. In *ICDE*, 2005.

[44] H. C. Zhao, C. H. Xia, Z. Liu, and D. Towsley. A Unified Modeling Framework for Distributed Resource Allocation of General Fork and Join Processing Networks. In *SIGMETRICS*, 2010.