Frugal Topology Construction for Stream Aggregation in the Cloud

Rachid Guerraoui[*], Erwan Le Merrer[†], Rhicheek Patra[*] and Bao-Duy Tran[*†]
[*]EPFL, [†]Technicolor,
Email: firstname.lastname@epfl.ch, erwan.lemerrer@technicolor.com, tranbaoduy@gmail.com

***Abstract*—Aggregation of streamed data is key to the expansion of the Internet of Things. This paper addresses the problem of designing a topology for *reliably* aggregating data flows from many devices arriving at a datacenter. Reliability here means ensuring operation without data loss. We seek a *frugal* solution that prevents wasteful resource consumption (over-provisioning). This problem is salient when building an aggregation service out of components (here aggregation nodes) that exhibit hard constraints on the amount of information they can handle per unit of time. We first formalize the problem and provide an analysis of the relation between monitored devices (plus information they send), and the operations performed at aggregation nodes, in terms of data rates. Building on this rate analysis, we devise a novel algorithm, which we call $\mathcal{CSA}$, that basically outputs an aggregation topology capable of handling those incoming data rates, preventing thereby empirical trial-and-error design. We analyze the algorithm, before validating it on the Amazon Kinesis platform, using a device dataset from a European telco operator.**

I. Introduction

The rise of the Internet of Things and the general migration of IT services to the cloud push for the adoption of practical low-latency processing solutions. Applications include the monitoring of potentially thousands of devices that stream information to a collection point in the background. Device Analyzer [30] for instance collects hundreds of thousands of data points per day per monitored device. It is crucial to process the data collected in real-time, in particular for scenarios like continuous analysis of application usage, or notifications for mention of specific keywords in social media.

Real-time computation is made possible by the progress of *stream processing* platforms [6], [18]. Underlying algorithms require small space and update time [24], [14]. The support for *stream aggregation* operations is required for advanced analytics since it allows advanced applications like the identification of heavy hitters [10] or anomaly detection [14]. Nevertheless, several technical challenges are incurred by the increasing amount of monitored resources. Clearly, no single machine can sustain millions of concurrent connections for aggregating device information. In systems where computation precision is essential (see *e.g.*, [27], [28] for gracefully degrading systems otherwise), computing units need to be organized in the form of a *topology* [6], [11], [18] so that the information received is reduced layer by layer in a scalable fashion, until the desired result is obtained [10], [35], [24]. In other words, data streams for a common aggregation operation are *shared* among processors for efficient distributed execution [17], [15].

Amazon Kinesis constitutes a celebrated example of a cloud-based platform to handle data streams in real-time [3], along with its open-source counterparts Kestrel [1] and Kafka [5]. The data *buffering* functionality proposed by such platforms is a required step to avoid packet loss [33]. However, such platforms impose physical or arbitrary data ingest limits. For instance, Kinesis limits the amount of data one can push to a *shard*[1] every second at 1MB. In practice, and as advertised by Amazon, packets causing rate excess at a given point in time are simply dropped (*i.e.*, lost). Multiple shards are thus required to sustain the load, and billing is precisely based on the number and usage duration of those shards.

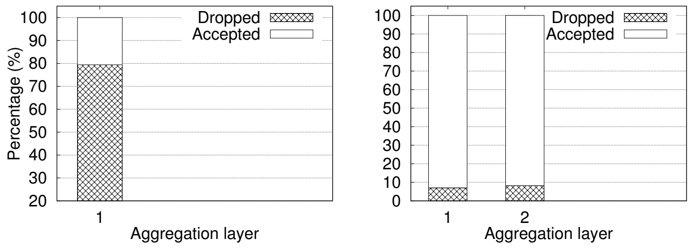

Fig. 1. Percentages of dropped data in an experiment on Kinesis (crosshatched), for a single shard (left), and for a 2-layer aggregation topology consisting of 13 and 1 shards (right)

Unfortunately, using such platforms for large-scale stream processing is challenging. Indeed there is no systematic way to design a streaming topology, other than empirically by trial and error. Research works [24], [14], [15], focusing on specific operations over data streams, use specific topologies for validating their algorithms. However, there is no mention regarding the methodology used for generating those topologies. Such empirically obtained topologies are likely to over-provision the number of shards: one can for instance provision a tangibly high number of those, eliminating data losses from devices. Operational costs are more than actually required by the application, which is a clear problem when considering the general will to consolidate costs in the cloud[2]. At the other extreme, if one designs a topology that is "too small" to ingest the load, device packets are dropped, resulting in aggregation imprecision.

Figure 1 presents the results of an experiment we conducted on the Kinesis platform, where the number of shards in the topology is under-provisioned with regard to the amount of information received. Five hundred processes continuously send 50KB packets at a rate of 0.5 packets per second to a service where aggregation is performed on some counters. The experiment utilizing a single shard (left sub-figure) exhibits

[1]A shard is the base throughput unit of an Amazon Kinesis stream.
[2]See *e.g.*, [18] for a practical over-provisioning example.

a high packet drop rate because the amount of received information is over the threshold set per shard by Kinesis (1MB/s). Dividing the total load by this threshold, to obtain a number of shards and align them in the first layer does not address the problem, as the sink in layer two is also overwhelmed by data coming out of layer one (right sub-figure); this means that more aggregation layers are required for a proper operation.

Contribution: Scalable Aggregation under Ingest Constraint

Acknowledging the commercial or physical limits of shards (or buffers), there is thus a need for a systematic approach to designing an aggregation topology on demand. This topology must be derived from the application settings (number of devices and their data sending rate) and on the aggregation operation to be achieved. Meanwhile, this topology should maintain an invariant that no shard must be overwhelmed by received data from its neighbors in the topology. Trivially deriving the number of shards from input load (divided by the imposed threshold) does not address the problem, as it does not provide any information on the shape of the topology, so that shards below the first layer can be overwhelmed, too (Figure 1, right sub-figure).

The contributions of this paper are threefold:

(i) We formalize the notion of *reliable* aggregation of streamed data with *ingest constraints*. We then introduce the Frugal Topology Design ($\mathcal{FTD}$) problem, for efficient aggregation in cloud environments. The problem consists in designing a topology, in which each shard has ingest rate constraints, for loss-free aggregation while using as few shards as possible for cost reasons.

(ii) We present *Conservative Sequential Assignments* ($\mathcal{CSA}$), an algorithm that solves the $\mathcal{FTD}$ problem. We analyze its convergence, its approximation of an optimal solution, and the characteristics of the output topology.

(iii) We experiment and validate $\mathcal{CSA}$ on the Kinesis platform and provide empirical demonstration of the effectiveness of $\mathcal{CSA}$ in a practical environment (using real and synthetic datasets from home gateways).

The remaining of the paper is organized as follows. We provide the background for stream aggregation in the cloud, an overview of the Amazon Kinesis service (as it is a concrete example of a buffering service) before introducing the problem statement in II. We provide an analytical result of the data rates implied in stream aggregation under ingest constraints in III. In IV, we propose our $\mathcal{CSA}$ algorithm for a reliable topology construction by building on the analysis. We validate $\mathcal{CSA}$ on the Kinesis stream processing platform in V. Finally, we discuss the related work in VI and conclude in VII.

II. Shared Aggregation In The Cloud

There are two ways to process streamed data in the cloud. The first one reads tuples from structured storage on disks, like in Photon [4] or Bolt [13]. In this paper, we follow the second trend, for we target real-time aggregation. We compute over data streams emitted from sources directly, as proposed by the Storm general framework [6]. To cope with the amount of data arriving at the datacenter, there is a need for buffers (sometimes named queues, or shards in Kinesis terminology) to temporarily store data (possibly in memory) before processors can catch up [33]. Examples of buffers include Kestrel [1] or Kafka [5] (platforms to deploy), and Amazon Kinesis [3] (available as an online service).

In this work, we adopt Kinesis as a standard building block for stream aggreagation, as it illustrates the hard constraints (here ingest rate constraints) faced by a designer when setting up a cloud-based computing solution. We provide below an overview of the Kinesis service, as a representative component of a stream processing platform. We then define the operations addressed in the paper (shared stream aggregation), before giving the problem statement.

A. Kinesis: A Buffering Service In The Cloud

In practice, the use of a buffering service before actual processing is mandatory to cope with high arrival rates and processing delays [33]. Kinesis implements such a service. It offers reliable buffers (shards) in the cloud. Kinesis accepts data records from *producers* (here devices) and releases them to *consumers* (here aggregation nodes) upon request. All producer and consumer applications are external to Kinesis and must be deployed elsewhere, albeit in-house or remote (*e.g.*, using Amazon EC2 [2]).

The considered streamed unit is a *data record* comprising the device identifier, an aggregation period identifier, and a free-format payload. Buffers in Kinesis are named *shards* where data actually reside. Every producer only indicates the desired target shard when pushing data to Kinesis, using a cryptographic hash function over the device identifier for instance. Data records are accessed on first-in-first-out (FIFO) and last-in-first-out (LIFO) consumption orders on a shard. Billing is conducted primarily on the basis of the active duration of used shards. Certain service limits concern data sizes (*e.g.*, maximum 50KB for a record payload). Others pertain to read/write throughput (each shard can support up to 2MB of data read per second, and up to 1MB data written per second).

Most importantly, Kinesis streams and associated application instances can be chained in a *topology* to realize more complex stream processing scenarios. In the sequel and for simplicity, a *shard* is renamed *node* as it is simply the first of a two-part component: the buffer-processor (*i.e.*, the aggregation unit).

B. Shared Stream Aggregation

Our work examines shared stream aggregation over non-overlapping windows [17], [15]. We employ a model where data sources emit streams of key-value tuples (T, v) where T is a time period (often called epoch) ID and v is a payload value: v in emitted tuple (T, v) is a data item drawn from an arbitrary but predefined domain, and compatible with an admissible aggregate function $\mathsf{agg}(.)$. This allows queries like "`return MAX(v) in T`" for operations over distributive and algebraic aggregates [12] (used for instance in [17], [15]).

We now reformulate properties of those aggregates for clarity and completeness. Let $\mathcal{V}^\star$ be a multiset of target values drawn from domain $\mathcal{D}$, and let $\{\mathcal{V}_i^\star \mid i \in \{1..N\}\}$ be a

TABLE I. TABLE OF NOTATIONS

θ	Ingest rate constraint at aggregation node
L	Number of aggregation layers
N	Total number of processors in the topology
n_0	Number of sources (considered as layer 0)
n_ℓ	Number of layer-ℓ nodes
λ_0	Mean Poisson emission rate at each source
Δt	Ideal regular key-change period for sources
$\lambda_{\ell k}^{(\text{in})}(t)$	Total incoming rate at k^{th} node in layer ℓ

partitioning of $\mathcal{V}^\star$ into N disjoint sub-multisets ($N \in \mathbb{N}^*$); an aggregate function $\text{agg}\left(\mathcal{V}^\star\right) \in \mathcal{D}$ is *admissible* if and only if:

$$\text{agg}\left(\mathcal{V}^\star\right) = \text{agg}\left(\biguplus_{i=1}^{N} \left\{\text{agg}\left(\mathcal{V}_i^\star\right)\right\}\right) \tag{1}$$

where $\biguplus$ denotes 'addition of multisets'.

Note that this definition can be generalized with $\text{agg}\left(\mathcal{V}^\star\right)$ returning a multiset of $\mathcal{D}$-valued target values (instead of a single one). Examples of generalized $\text{agg}(.)$ include finding top-K and bottom-K from a multiset of totally-ordered target values. Admissible aggregate function $\text{agg}(.)$ can be defined by *initializer* $\text{agg}(\emptyset)$ (or singleton-based $\text{agg}\left(\{v\}\right)$) and *accumulator* $\text{agg}\left(\mathcal{V}^\star \uplus \{v\}\right) = \text{accum}_{\text{agg}}\left[\text{agg}\left(\mathcal{V}^\star\right), v\right]$. Informally speaking, an aggregate function is admissible if it can be applied in phases, first on disjoint sub-multisets of target values to obtain intermediate results, then on the multiset formed by these intermediate results. In other words, such functions permit partial accumulation and parallelization of the aggregation process in a topology.

We finally give the definition of a complete shared stream aggregation work flow.

Definition 1 (Shared stream aggregation output) *For every time period ID T present in the streams, the aggregation system must output a single tuple (T, v_T) such that:*

$$v_T = \text{agg}\left(\mathcal{V}_T^\star\right) \quad (\forall T) \tag{2}$$

where $\mathcal{V}_T^\star$ is the multiset of target values v emitted in all *tuples (T, v) from* all *sources, and $\text{agg}(.)$ is the admissible aggregate function in use.*

C. Problem Statement: Frugal Topology Design

Our goal is to design a systematic approach to determining a sufficient topology for loss-free aggregation.

The Frugal Topology Design Problem ($\mathcal{FTD}$). *In the context of shared stream aggregation for non-overlapping time windows, under ingest rate constraints at aggregation nodes, derive a systematic approach to determining a reliable topology such that ingest rate saturation is avoided at all nodes, while minimizing the topology size (for reduced cost and operation latency).*

III. RATE TRANSFER ANALYSIS

Traffic rate analysis is a crucial step to design a topology that tightly matches imposed constraints [11]. We first formalize the problem before actually providing analysis results.

A. Problem Formalization

We now formalize a topology as being the multi-layer parallelization of shared stream aggregation. The concepts and notations presented are mandatory as a foundation for subsequent system analysis, as well as for the derivation of solutions to solve $\mathcal{FTD}$. The core notations are listed in Table I.

Definition 2 (Multi-layer parallelization) *The* multi-layer parallelization *of shared stream aggregation is a topology of n_0* sources *($n_0 \in \mathbb{N}^*$) in layer 0 and L aggregation layers ($L \in \mathbb{N}^*$). Aggregation layer ℓ ($\ell \in \{1..L-1\}$) comprises n_ℓ* processors *($n_\ell \in \mathbb{N}^*$) while the last layer L has a single* sink *($n_L = 1$). The k^{th} node in layer ℓ ($k \in \{1..n_\ell\}, \ell \in \{0..L\}$) is denoted as (ℓ, k). Consecutive layers $\ell - 1$ and ℓ ($\ell \in \{1..L\}$) form a complete bipartite graph where directed edges point from layers $\ell - 1$ to ℓ, representing directed flows of data streams.*

This topology for multi-layer parallelization is depicted in Figure 2. Next, we detail the characteristics and functionalities of various nodes in the topology.

Definition 3 (Source) *We assume that* source $(0, k)$ *($k \in \{1..n_0\}$) in the multi-layer parallelization emits data tuples according to a* homogeneous Poisson process *with same mean arrival rate $\lambda_{0k}^{(out)}(t) = \lambda_0 \in \mathbb{R}^+$ (items/sec), $\forall k \in \{1..n_0\}, \forall t \in \mathbb{R}$. At each source, tuples are uniformly routed to all processors $(1, m)$ in the next layer ($m \in \{1..n_1\}$),* i.e.*, a tuple is routed to processor $(1, m)$ with probability $\dfrac{1}{n_1}$. Furthermore, at time $t \in [t_{ki}, t_{k,i+1}]$, source $(0, k)$ emits data tuples in the form (T_{ki}, v). Key-change moment $t_{ki} = t_i^{(ideal)} + \Omega_k$ where independent random offset $\Omega_k \sim \mathsf{Normal}\left(0, \sigma_k^2\right)$ for source $(0, k)$ and $t_i^{(ideal)} = t_{i-1}^{(ideal)} + \Delta t, \forall i$ with constant ideal key-change period $\Delta t \in \mathbb{R}_+^*$ applicable to all sources.*

The core assumption for allowing analysis is that data arrivals from sources form a Poisson process. This is a classic and reasonable approximation for many applications [8], allowing to construct a model for understanding trends and implications. For instance, a constant mean for sending report is the norm in programmed devices, for updating their status to a collection point (*e.g.*, in the case of home gateways [25]). For each source, key change moments are normally distributed around the ideal moment in order to model clock skews [16]. We note that the model of a complete bipartite communication pattern between consecutive layers is chosen as the use of hash functions is now commonplace in cloud environments [6], [3], [4]. It specifically means that every processor (or source) sends a piece of information to *one among all* processors in the next layer whose selection is based on the outcome of the hash function employed.

Definition 4 (Aggregation node) *Every node in layers* 1 *to L ($L \in \mathbb{N}^*$) in the multi-layer parallelization is an* aggregation node*, which can either be a processor or the sink.*

Definition 5 (Processor) Processor (ℓ, m) *($\ell \in \{1..L-1\}, m \in \{1..n_\ell\}, L \in \mathbb{N}^*$) in the multi-layer parallelization receives data tuples in the form of stream*

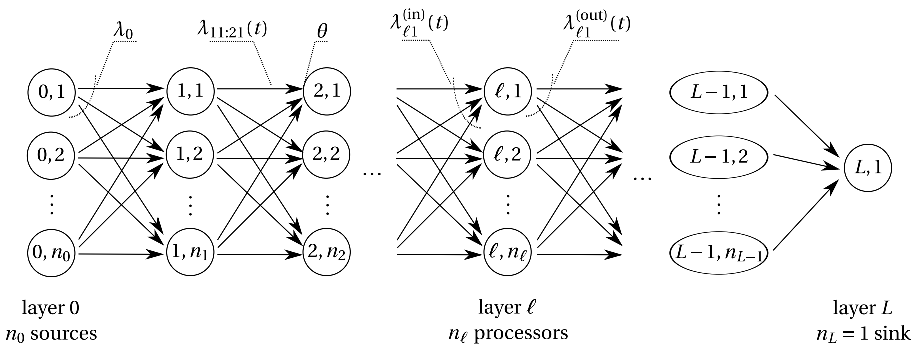

Fig. 2. General topology form for stream aggregation. Instantiation in practice often has the form of a "pyramid".

superposition from all nodes in the previous layer $\ell - 1$. Each processor runs Algorithm 1 (single-buffer, zero-coordination operation) using an internal time period ID cache T_0, a target value buffer b, as well as the initializer and accumulator of admissible aggregation function agg(.). *Output tuples are then uniformly routed to all processors $(\ell+1, w)$ in the next layer ($w \in \{1..n_{\ell+1}\}$),* i.e.*, a tuple is routed to processor $(\ell+1, w)$ with probability $\dfrac{1}{n_{\ell+1}}$.*

Algorithm 1: Single-buffer, zero-coordination operation at aggregation node

```
$T_0 \leftarrow$ null;    $b \leftarrow \text{agg}(\emptyset)$;
foreach tuple $(T, v)$ received do
    if $T_0 = null$ then
        $T_0 \leftarrow T$;    $b \leftarrow \text{agg}(\{v\})$;
    else if $T \neq T_0$ then
        emit tuple $(T_0, b)$;
        $T_0 \leftarrow T$;    $b \leftarrow \text{agg}(\{v\})$;
    else
        $b \leftarrow \text{accum}_{\text{agg}}(b, v)$;
```

Definition 6 (Sink) *The* sink $(L, 1)$ *($L \in \mathbb{N}^*$) in the multi-layer parallelization behaves in precisely the same manner as that of a processor, except that there is no uniform routing to the next layer (as there is no such layer). It constitutes the last operational node, containing all information needed to provide the aggregation results.*

The following definitions formalize the notations of incoming and outgoing rates incurred at various nodes, as well as the ingest rate constraint at these nodes (which is the fundamental condition to be satisfied in our problem statement). These are also illustrated in Figure 2.

Definition 7 (Rates in multi-layer parallelization) *In the context of multi-layer parallelization, data stream flowing from nodes $(\ell-1, k)$ to (ℓ, m) ($\ell \in \{1..L\}$) has rate $\lambda_{(\ell-1)k:\ell m}(t)$ (items/sec) at time t. Total incoming rate at time t at aggregation node (ℓ, m) ($\ell \in \{1..L\}$) is denoted as $\lambda^{(in)}_{\ell m}(t)$ (items/sec). Total outgoing rate at time t at source or processor (ℓ, m) ($\ell \in \{0..L-1\}$) is denoted as $\lambda^{(out)}_{\ell m}(t)$ (items/sec).*

Definition 8 (Ingest rate constraint) *Every aggregation node in the multi-layer parallelization imposes the same ingest rate constraint θ (items/sec) ($\theta \in \mathbb{R}^+$). Ingest rate saturation can be avoided if and only if:*

$$\lambda^{(in)}_{\ell m}(t) \leqslant \theta \quad (\forall \ell \in \{1..L\}, \forall m \in \{1..n_\ell\}, \forall t \in \mathbb{R}) \tag{3}$$

This ingest rate constraint can either be physical (*e.g.*, manufacturer hardware limit for a network peripheral) or commercial (*e.g.*, Kinesis case) in nature. In this work, rate constraint θ is assumed to be the same for all nodes participating in the aggregation operation.

Lastly, note that using the above notations, $\mathcal{FTD}$ is solved if a topology is found that given the number of sources n_0, the total emission rate λ_0 at each source, and the ingest rate constraint θ at each aggregation node has a sufficient numbers of processors n_ℓ at layers ℓ ($\ell \in \{1..L-1\}$), such that ingest rate saturation can be avoided at all aggregation nodes [Equation (3)] while employing the least number of nodes $\sum_{\ell=1}^{L} n_\ell$ possible and the least number of layers L possible.

B. Resulting Rate Transfers

Rate transfer analysis refers to the determination of an output rate given specific input conditions while the existence of outputs (which governs how the output rate turns out to be) depends solely on time period IDs T.

We modeled key-change moments at source $(0, k)$ to always exhibit random offset Ω_k from the ideal time point determined by constant ideal key-change period Δt. Depending on variance σ_k^2 of normally distributed Ω_k, it can happen, though rather unlikely, that an arbitrary number of distinct time period IDs can be present in a particular *consideration period*[3]. In practice, offsets are expected to be small with respect to time periods Δt, causing actual key-change moments to cluster around their respective ideal counterpart, rather than spreading infinitely on both sides of the latter. In such a practical scenario, there are at most two distinct time period IDs emitted by any sources within a consideration period. We assume this prerequisite for our analysis.

With all those notations and definitions in place, we are now ready to present a core result of the analysis of aggregation transfer rate over non-overlapping windows. We precisely derived bounds for incoming and outcoming rates occurring in

[3]Consider the superposition of all key-change moments t_{ki} of all sources $(0, k)$: the universal timeline can be split into several *consideration periods* demarcated by these key-change moments. Details are available in [29].

a topology of the form of Figure 2, at each node. Due to space constraint, please refer to [29] for detailed analysis; we only report on the main result, that is the bound on the incoming rate at any aggregation node.

Theorem 1: Total incoming rate at any node (ℓ, m) ($\ell \in \{1..L\}$, $m \in \{1..n_\ell\}$) is upper bounded by:

$$\lambda^{(\text{in})}_{\ell m}(t) \leqslant \lambda^{(\text{in})}_{\ell,\max} = \frac{n_0\lambda_0}{2^{\ell-1}n_\ell}\prod_{k=1}^{\ell-2}\left(1+\frac{1}{n_k(2n_{k+1}-1)}\right), \tag{4}$$

with n_k being the number of processors in layer k.

IV. Frugal Construction Algorithm

With a closed-form formula for the upper bound of total incoming rate at any aggregation node $\lambda^{(\text{in})}_{\ell,\max}$ ($\forall \ell \in \{1..L\}$) [Theorem 1], the ingest rate constraint of Definition 8 becomes:

$$\lambda^{(\text{in})}_{\ell,\max} \leqslant \theta \Leftrightarrow \psi_\ell(n_1, n_2, \ldots, n_\ell) \leqslant 0 \quad (\forall \ell \in \{1..L\}) \tag{5}$$

where function $\psi_\ell(n_1, n_2, \ldots, n_\ell)$ is defined as follows $\forall \ell \in \{1..L\}$:

$$\begin{aligned}\psi_\ell(n_1, n_2, \ldots, n_\ell) &= n_0\lambda_0\prod_{k=1}^{\ell-2}(2n_{k+1}n_k - n_k + 1)\\ &\quad -\theta 2^{\ell-1}n_\ell\prod_{k=1}^{\ell-2}n_k(2n_{k+1}-1)\end{aligned} \tag{6}$$

Using Equation (5) above, $\mathcal{FTD}$ can then be stated as:

$$\begin{cases}\min\limits_{L,n_1,n_2,\ldots,n_{L-1}}\left(\beta\sum_{\ell=1}^{L-1}n_\ell + \gamma L\right)\\ \text{subject to: } L, n_1, n_2, \ldots, n_{L-1} \in \mathbb{N}^*\\ \text{subject to: } \psi_\ell(n_1, n_2, \ldots, n_\ell) \leqslant 0 \quad (\forall \ell \in \{1..L\}),\end{cases} \tag{7}$$

with two 'independent' objective functions (number of nodes and topology diameter), arbitrarily weighted by introduced parameters $\beta, \gamma \in \mathbb{R}^*_+$.

Designing a topology boils down to finding a plausible assignment for $L, n_1, n_2, \ldots, n_{L-1}$ according to the requirements stated in Equation (7). We now introduce the $\mathcal{CSA}$ algorithm that solves $\mathcal{FTD}$.

A. The $\mathcal{CSA}$ Algorithm

We revisit the constraints (Equation (5)) which can be equivalently transformed as follows:

$$n_\ell \geqslant \frac{n_0\lambda_0}{\theta 2^{\ell-1}}\prod_{k=1}^{\ell-2}\left(1+\frac{1}{n_k(2n_{k+1}-1)}\right) = \tilde{\psi}_\ell(n_1, n_2, \ldots, n_{\ell-1}) \quad (\forall \ell \in \{1..L\}) \tag{8}$$

As can be seen from Equation (8), the criterion for n_ℓ at layer ℓ relies on the outcome of function $\tilde{\psi}_\ell(n_1, n_2, \ldots, n_{\ell-1})$ which *only* depends on the numbers of nodes at layers *preceding* ℓ. This makes sequential assignments of n_ℓ with incremental ℓ possible.

Algorithm 2: Conservative Sequential Assignments $\mathcal{CSA}$

```
Input: $n_0 \in \mathbb{N}^*$, $\lambda_0 \in \mathbb{R}^+$, $\theta \in \mathbb{R}^+$
Output: $L, n_1, n_2, \ldots, n_L \in \mathbb{N}^*$ (i.e., a topology)
for $\ell \leftarrow 1$ to $+\infty$ do
    $n_\ell \leftarrow \left\lceil \frac{n_0\lambda_0}{\theta 2^{\ell-1}}\prod_{k=1}^{\ell-2}(1+\frac{1}{n_k(2n_{k+1}-1)}) \right\rceil$;
    if $n_\ell = 1$ then
        $L \leftarrow \ell$;    return $L, n_1, n_2, \ldots, n_L$;
```

The minimization of $\sum_{\ell=1}^{L} n_\ell$ can then be achieved by *conservatively* selecting the smallest possible value for every n_ℓ. As $n_\ell \geqslant \tilde{\psi}_\ell(n_1, n_2, \ldots, n_{\ell-1})$ [Equation (8)] and $n_\ell \in \mathbb{N}^*$, this corresponds to picking $n_\ell = \left\lceil \tilde{\psi}_\ell(n_1, n_2, \ldots, n_{\ell-1}) \right\rceil$. The process terminates as soon as $n_\ell = 1$ at some ℓ (the single sink [Definition 6], with ℓ being the finalized number of layers L). This Conservative Sequential Assignment (denoted as $\mathcal{CSA}$) is proposed in Algorithm 2.

B. Analysis of $\mathcal{CSA}$

We first prove that Algorithm 2 converges, and thus outputs a topology solving $\mathcal{FTD}$. We finally give an approximation ratio to compare it to a theoretical optimum.

Theorem 2 (Convergence of $\mathcal{CSA}$) *In Algorithm 2, the conservative sequential assignment of n_ℓ converges with $\lim_{\ell\to+\infty} n_\ell = 1$ and the algorithm terminates at condition $n_\ell = 1$.*

Proof: At the ℓth iteration of Algorithm 2, the following assignment occurs:

$$\begin{aligned}n_\ell &= \left\lceil \frac{n_0\lambda_0}{\theta 2^{\ell-1}}\prod_{k=1}^{\ell-2}(1+\frac{1}{n_k(2n_{k+1}-1)}) \right\rceil\\ &= \left\lceil \frac{n_0\lambda_0}{2\theta}\times\frac{1}{2^{\ell-2}}\prod_{k=1}^{\ell-2}\left(1+\frac{1}{n_k(2n_{k+1}-1)}\right) \right\rceil \geqslant 1\end{aligned} \tag{9}$$

We have integers $n_k \geqslant 2$ and $n_{k+1} \geqslant 2$, $\forall k \in \{1..(\ell-2)\}$ (otherwise, $n_k = 1$ or $n_{k+1} = 1$ which implies the algorithm has terminated at an iteration earlier than the ℓth). Therefore:

$$\begin{aligned}& n_k(2n_{k+1}-1) \geqslant 6\\ &\Leftrightarrow 1+\frac{1}{n_k(2n_{k+1}-1)} \leqslant \frac{7}{6}\\ &\Leftrightarrow \prod_{k=1}^{\ell-2}\left(1+\frac{1}{n_k(2n_{k+1}-1)}\right) \leqslant \left(\frac{7}{6}\right)^{\ell-2}\\ &\Leftrightarrow \frac{n_0\lambda_0}{2\theta}\times\frac{1}{2^{\ell-2}}\prod_{k=1}^{\ell-2}\left(1+\frac{1}{n_k(2n_{k+1}-1)}\right)\\ &\leqslant \frac{n_0\lambda_0}{2\theta}\left(\frac{7}{12}\right)^{\ell-2} = \frac{72n_0\lambda_0}{49\theta}\left(\frac{7}{12}\right)^{\ell}\end{aligned}$$

$$\Leftrightarrow n_\ell = \left\lceil \frac{n_0\lambda_0}{2\theta} \times \frac{1}{2^{\ell-2}} \prod_{k=1}^{\ell-2} \left(1 + \frac{1}{n_k(2n_{k+1}-1)}\right) \right\rceil$$
$$\leqslant \left\lceil \frac{72 n_0 \lambda_0}{49\theta} \left(\frac{7}{12}\right)^{\ell} \right\rceil \tag{10}$$

With n_0, λ_0 and θ being positive constants and $\frac{7}{12} < 1$, consider function $f : \mathbb{N}^* \to \mathbb{R}_+^*$ where $f(\ell) = \frac{72 n_0\lambda_0}{49\theta}\left(\frac{7}{12}\right)^{\ell}$. We have $\lim_{\ell\to+\infty} f(\ell) = 0$ while $f(\ell) > 0, \forall \ell$, thus $\lim_{\ell\to+\infty} \lceil f(\ell) \rceil = 1$. From Equations (9) and (10), $1 \leqslant n_\ell \leqslant \lceil f(\ell) \rceil$, so $\lim_{\ell\to+\infty} n_\ell = 1$. ■

We next show that the algorithm terminates in a logarithmic number of steps with regard to input parameters, and thus outputs a topology with an also logarithmic diameter.

Lemma 1 (Topology diameter) *$\mathcal{CSA}$ outputs a topology whose diameter is logarithmic in the problem input values n_0, λ_0 and θ. In particular, $\mathcal{CSA}$ terminates in at most $L_{\max} = \left\lceil \log_b \frac{\eta\theta}{n_0\lambda_0} \right\rceil$ iterations, with $\eta = \frac{49}{72}$ and $b = \frac{7}{12}$.*

$L_{\max}$ is hence an upper bound for the number of layers L.

Proof: Condition $n_\ell = 1$ is attained at the latest when $\lceil f(\ell) \rceil = 1 \Leftrightarrow f(\ell) \leqslant 1 \Leftrightarrow \ell \geqslant \log_{\left(\frac{7}{12}\right)} \frac{49\theta}{72 n_0 \lambda_0} = \tilde{L}_{\max}$. Hence the algorithm terminates in at most $L_{\max} = \left\lceil \tilde{L}_{\max} \right\rceil$ iterations. ■

This bound on the diameter is of particular interest for practical deployments, in order to estimate for instance the end-to-end operation latency.

Let $N = \sum_{\ell=1}^{L} n_\ell$ be the total number of processors in the topology. We now bound N resulting from Algorithm 2.

Corollary 1 (Bound on the number of processors in the topology) *$\mathcal{CSA}$ outputs a topology composed of N processors: $N \leqslant \sum_{\ell=1}^{L_{\max}} \left\lceil \frac{n_0\lambda_0}{\eta\theta} b^\ell \right\rceil$.*

Proof: Direct application of Theorem 2 and Lemma 1. ■

We finally give the approximation ratio of Algorithm 2.

Theorem 3 (Approximation ratio of conservative sequential assignment) *An approximation ratio of $\mathcal{CSA}$ is $(1/\eta)\times\mathcal{OPT}$, plus $\log_b \frac{\theta}{n_0\lambda_0} + 1$.*

Proof: Let I be an instance of the frugal topology construction problem. Clearly, no algorithm can use fewer nodes in the output topology than $\mathcal{OPT} = \left\lceil \frac{n_0\lambda_0}{\theta} \right\rceil$. Let us denote this quantity as the optimal solution $N^*(I)$, and the quantity given by Algorithm 2 as $N^{\mathcal{CSA}}(I)$. Please note that $\mathcal{OPT}$ does not solve $\mathcal{FTD}$.

From Corollary 1, we have:

$$N^{\mathcal{CSA}}(I) \leqslant \sum_{\ell=1}^{L_{\max}} \left\lceil \frac{n_0\lambda_0}{\eta\theta} b^\ell \right\rceil$$
$$\leqslant \sum_{\ell=1}^{L_{\max}} \left(\frac{n_0\lambda_0}{\eta\theta} b^\ell + 1 \right)$$
$$\leqslant \frac{n_0\lambda_0}{\theta}\frac{1}{\eta} \sum_{\ell=1}^{L_{\max}} b^\ell + \sum_{\ell=1}^{L_{\max}} 1$$
$$\leqslant \frac{n_0\lambda_0}{\theta}\frac{1}{\eta} \left(\sum_{\ell=0}^{+\infty} b^\ell - b^0 \right) + \log_b \frac{\eta\theta}{n_0\lambda_0}$$
$$\leqslant \frac{n_0\lambda_0}{\theta}\frac{1}{\eta} \left(\frac{1}{1-b} - 1 \right) + \log_b \eta + \log_b \frac{\theta}{n_0\lambda_0}$$
$$\leqslant N^*(I) \times \frac{1}{\eta} + \log_b \frac{\theta}{n_0\lambda_0} + 1$$

■

This last result is a direct indicator of the maximum operational cost for a given problem instance. In Kinesis for example, user is billed on the number of shards (nodes) employed. In essence, this constructive proof for the minimal number of nodes required for aggregating data in the context of shared aggregation over non-overlapping windows provides a result that states that the number of processors in each topology layer is close to half the preceding layer, until reaching one (the sink).

V. Evaluation

In this section, we evaluate our $\mathcal{CSA}$ algorithm on real and synthetic datasets, using the Amazon Kinesis stream processing platform where practical ingest constraints apply. We first benchmark the platform for parameter settings. We then experiment an alternative scenario where lost packets are re-transmitted, to conclude that for scalability, a solution to $\mathcal{FTD}$ is more desirable than re-transmission. Finally, we evaluate the topology given by $\mathcal{CSA}$ on a home gateway scenario, before studying the elasticity of $\mathcal{CSA}$ instantiations.

A. Experimental setup

Datasets. We use two datasets: The first, EISP, is collected from real-world home gateways subscribed to a European ISP. The second, SYN, is a synthetic one based on the empirical parameters introduced in I.

EISP dataset. This Internet Service Provider dataset consists of data from 5789 devices collected across 240 distinct home gateways over a year (11/2013 to 10/2014) [25]. The dataset captures packets from 240 households which were subscribed to a (single) large European ISP during the collection period and were recruited to take part in the trial deployment. The home gateways send 30-50KB packets to the cloud around every 60 seconds (network latency and delays due to load on the gateways are observed in practice). For simulation purpose of this private dataset, one can use a Poisson process with emission rate $\lambda_0 = 1/60$ items/sec. It results in a Hellinger distance [22] of 0.55, which leads to a Bhattacharya coefficient (ρ) [7] of around 0.69. The distributions of the dataset and of the Poisson process are overlapping since $\rho > 0$.

SYN dataset. We synthesized this dataset by re-using the same parameters as in I: $n_0 = 500$ sources, $\lambda_0 = 0.5$ items/sec, $\Delta t = 1,200$ seconds, duration ≈ 1.1 hours.

Deployment. For evaluating the performance of $\mathcal{CSA}$, we use the Amazon Kinesis platform which is a fully managed, cloud-based buffering service. The deployment resembles the structure sketched in Figure 2. In essence, every aggregation layer comprises as many Kinesis shards as the number of aggregation nodes (processors), each being a single-worker consumer. Furthermore, data forwarding between layers is regulated by Kinesis shard partitioning scheme. We interacted with Kinesis through a Python script which instantiated the topology by 'chaining' shards according to the topology given by $\mathcal{CSA}$. After each shard, an Amazon EC2 instance computes the aggregation function presented in Algorithm 1.

B. Platform Benchmark: Rate Enforcement and Latency

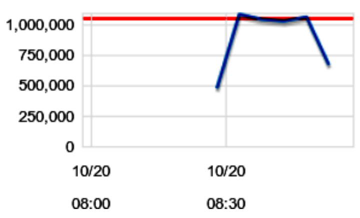

Fig. 3. Rate limit enforcement (1MB/s) at a given shard (plot obtained from Kinesis monitoring platform). When the limit is reached, packets are lost. θ should be set accordingly so that $\mathcal{CSA}$ can derive a topology without loss.

We first note that an ingest limit for a single shard is indeed enforced by Amazon, as seen in Figure 3, with no apparent tolerance over 1MB/s. Parameter θ should be set such that the actual rate recorded remotely at a given shard is below that enforced limit.

As an illustration for the need for prior service benchmarking, we ran the same experiment from a server in the Eu-central-1 zone ($n_0 = 20$, $\lambda_0 = 1$, $duration = 60$s). We sent data to shards located in different Amazon facilities (there are currently 7 available). Results of (write) interactions with shards are shown in Figure 4 (shown numbers are the average of 40 trials). This variability, due to the location of the server (source threads) with regard to the Kinesis facility targeted, is also of importance for the setting of parameter λ_0. For instance, if devices must sequentially send packets to multiples facilities for geo-replication, this latency constitutes a lower bound on their emission rate.

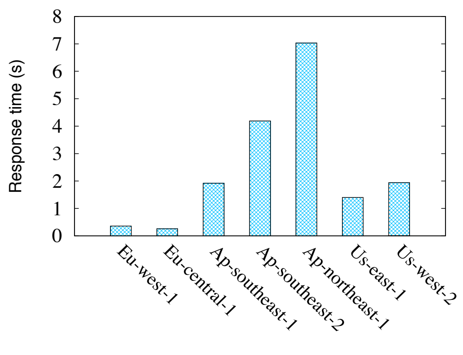

Fig. 4. Latency of write interactions with a single shard, in various facilities.

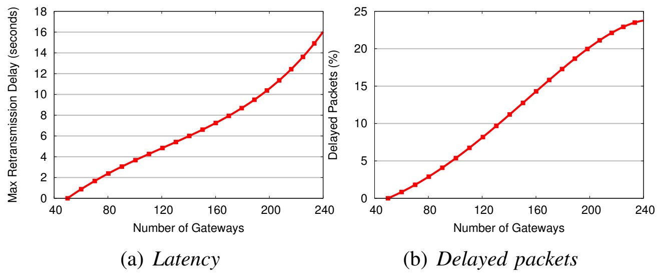

Fig. 5. Limitations of aggregation with increasing latency and count of delayed packets as the number of gateways increase on a single shard.

C. The Case For Avoiding Retransmissions

We highlight in Figure 5 the limitations of packet retransmissions upon loss, due to a topology that is not reliable regarding the service load. In particular, we replay streaming data from the EISP dataset to a single shard on Kinesis. As the amounts of data received at the shard are too high, packet loss occurs. A naive way to cope with this is to buffer lost packets on devices for later re-transmission. We observe from our results that this strategy is not scalable, meaning that loss-free aggregation should prevail. The EISP dataset packets are replayed by their original generation time-stamp, also around 60s at every gateway.

Latency. Whenever a packet loss occurs (`FailedRecordCount` field in the response from Kinesis to the device), the packet is stored in a queue maintained locally on each device. The packets from the queue are re-transmitted following the lognormal distribution to avoid congestion and bursty loss (*i.e.*, mimicking MAC layer re-transmissions [20]). Figure 5(a) demonstrates that the actual acceptance time for a re-transmitted packet increases as the number of gateways increases to a maximum of 18 seconds for 240 gateways (aggregation period $\Delta t = 60s$). This delay can lead to either incorrect aggregation results or longer waiting times which can have a significant impact in aggregation scenarios like algorithmic trading systems where one might want aggregates over 5-to-10-minute windows reported every 60-90 seconds [17].

Delayed packets. We also measure the fraction of delayed packets while increasing the number of gateways and observed that more packets get delayed as more streams arrive at the shard as shown in Figure 5(b). We notice that nearly 25% of the packets get delayed due to the traffic when there are 240 gateways streaming data to the shard.

Those two experiments at a small scale show that the use of a frugal topology, in order to avoid loss and re-transmission, should be preferred, illustrating the necessity for the consideration of $\mathcal{FTD}$.

D. $\mathcal{CSA}$ Algorithm Performance

We use the SYN dataset to validate the performance of $\mathcal{CSA}$ due to the limited number of gateways in the EISP dataset.

Limitation of an empirical topology. In addition to the empirically designed solution by trial-and-error already plotted in I, for instance arbitrarily designing a 3-layer topology $[13 \quad 7 \quad 1]^{\intercal}$ (Figure 6, left) yields in data drops at the sink node. Notice that the deeper drops occur in the topology, the

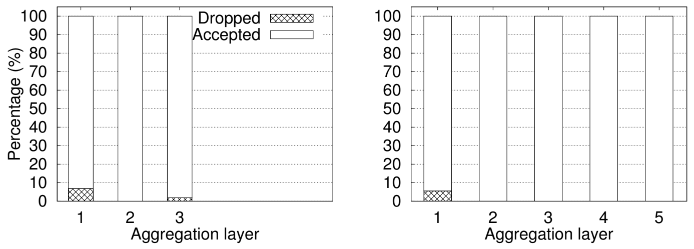

Fig. 6. A 3-layers under-provisioned topology (left), and the result of aggregation over a topology given by algorithm 2 (right). Trial & error causes data loss, design by $\mathcal{CSA}$ leads to no loss close to the sink.

higher the impact on the final aggregation results (as dropped packets are the aggregates of many previous packets). Next step for this empirical run would have been to add a third layer in between current layer 2 and the sink, with an also arbitrary number of nodes. As we shall see with our algorithm run, that solution would have probably failed again, increasing costs and time to operate the aggregation service.

$\mathcal{CSA}$ execution. Execution of $\mathcal{CSA}$ (Algorithm 2) yielded a 5-layer topology of the following form $[13 \quad 7 \quad 4 \quad 2 \quad 1]^{\intercal}$. Results are presented in Figure 6. For our designed topology, there were no packet drops except at the very first layer. Those drops are due to the fact that we operate at the limit of tolerance for a single shard (here $\theta = 20$) and also due to the non-Poissonian nature of the SYN dataset, as compared to our analysis leading to $\mathcal{CSA}$; the SYN scenario is more abrupt as all devices send their packets at the same clock tick, leading to bursty arrivals, and then to packet loss in first layer. A benchmark prior to execution would in practice lead to a safe-margin setting for instance $\theta = 18$. Besides this required initial benchmarking requirement, $\mathcal{CSA}$ succeeds in returning a topology for loss-free aggregation for all other layers.

Topology generation. In this section, we evaluate the elasticity of $\mathcal{CSA}$ in terms of latency to generate a topology. It is of interest when, for instance, the service operator is inserting a set of new devices to participate to the aggregation. $\mathcal{CSA}$ then must be invoked again, to derive the new topology; current topology in the cloud should then be updated, to handle the load variation. We measure the time needed to operate a modification on such a topology in production.

When a request is sent to Kinesis for the batch creation of a pre-defined number of shards, Kinesis replies with a response status `CREATING`. After the batch creation phase is over, status is `ACTIVE`, and shards are ready to receive data. We observed empirically that it takes roughly 30 seconds to create a batch of shards in Kinesis. Also, Kinesis limits the maximum number of parallel batch creations to 5. Hence, we use two approaches for the deployment of the topology generated by $\mathcal{CSA}$ on Kinesis.

`TopoGen-Par`. This approach for the topology deployment creates 5 batches in parallel and then waits for 30 seconds to create the next set of batches if there are more than 5 layers in the topology. Finally, after the last batch creation request, another wait time of 30 seconds is required for the last created batch of shards to be `ACTIVE`.

`TopoGen-Seq`. This approach for the topology generation waits 6 seconds after each batch creation. Hence, in a topology with more than 5 layers, when the 6^{th} batch needs to be created, the first batch of shards is already in the `ACTIVE` state. Finally, after the last batch creation request, another wait time of 30 seconds is required for the last batch to be `ACTIVE` similar to the other approach.

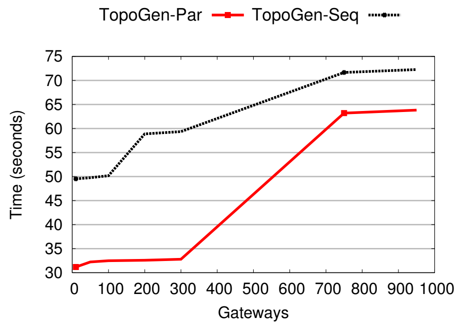

Fig. 7. *Latency for instantiating a topology in Kinesis.*

Figure 7 demonstrates the topology deployment (*i.e.*, creation from scratch) latency when the number of gateways varies from 10 to 950. This in essence captures the minimum time between two topology instantiations. We experimented with a maximum of 950 gateways, as this number triggers a topology of the form $[24 \quad 12 \quad 6 \quad 3 \quad 2 \quad 1]^{\intercal}$, thus composed of 48 shards[4]. We observe that there is a latency difference of around 30 seconds between 300 to 750 gateways for `TopoGen-Par` which results due to the limit (5 batches of shards) imposed by Kinesis on the number of batches created at same facility. We observe that the increase in time is sub-linear regarding the number of devices managed, yet this is totally dependant on the Kinesis platform implementation and current policy. As topology size tends to grow larger with the growth of the IoT phenomenon, we can expect platform providers to dimension their systems accordingly.

VI. Related Work

Large-scale aggregation is a must in modern computing systems [34], [4], [23], for it allows to build scalable and timely data-mining services. In case input data loads are unpredictable and data loss can be tolerated, techniques exist for graceful system performance degradation [27], [28], [26], [19]. In the shared stream processing context considered in this paper [17], [15], algorithms for solving various problems leverage aggregation topologies, for scalability and correctness reasons. It is not only the case for sketch-based querying over sliding windows [24], clustering [35], anomaly detection [14], but also very common in different contexts like querying sensor networks [9]. The crucial problem of building aggregation topologies was however not addressed. It is furthermore not clear how those approaches build their topologies, and whether they are barely sufficient or a large over-estimate of resources that are actually required for reliable service execution.

Recently, special attention has been devoted to optimizing the process of aggregating intermediate data residing in racks, for the reduce phase at a particular node in MapReduce jobs [31], [11]. Those approaches propose to configure the aggregation topology in the datacenter network, in order to

[4]We have a maximum limit of 50 shards currently on our Kinesis account.

minimize total network traffic. Algorithms apply to offline processing paradigms like MapReduce, specifically to traffic optimization, while we argue that a crucial first step in a stream context is to cope with rate constraints at processing nodes.

Other works considered constrained aggregation, but focused on memory limitations instead [36], [21]. Nadi *et al.* [21] proposed the creation of an aggregate tree, with the root being the input stream, and other nodes representing aggregates for an appropriate query plan. Sketching techniques [24], [14] constitute scalable approaches for aggregation, regarding the number of items considered; however, they do not cope with input rate limits. Finally, Xia *et al.* [32] considered the maximization of a distributed system's utility under CPU and bandwidth constraints, but for a fixed set of processing nodes, thus not focusing on system dimensioning.

VII. Conclusion

This paper addresses the frugal topology design problem, that aims at designing a reliable aggregation topology, tailored for shared operation with no data drop. We propose the $\mathcal{CSA}$ algorithm, that outputs a logarithmic-diameter topology. We prove an upper bound on the number of processing nodes for reliable execution; this number depends on problem inputs (number of sources, data sending rates, and ingest constraints). This type of informed design is needed to reduce operational costs, as promised by the cloud paradigm. Our evaluation in the cloud brings leanings on practical requirements for deployment, such as latency consideration due to location of facilities, and required safety margins for incoming rates.

We specifically address the problem of simple aggregation, as it is a necessary initial step. Interesting future work can include topology design for other operations like sketching and clustering where the use of topologies helps ensure scalability. If the upper bounds on their transfer rates are close to each other, this would allow to leverage this paper results, and to build a general framework for topology design algorithms.

References

[1] Kestrel. http://twitter.github.io/kestrel/, 2014.

[2] Amazon Web Services Inc. AWS | Amazon Elastic Compute Cloud (EC2) – scalable cloud hosting. http://aws.amazon.com/ec2, 2014.

[3] Amazon Web Services Inc. AWS | Amazon Kinesis. http://aws.amazon.com/kinesis, 2014.

[4] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, 2013.

[5] Apache Software Foundation. Kafka. http://kafka.apache.org/, 2014.

[6] Apache Software Foundation. Storm, distributed and fault-tolerant realtime computation. https://storm.incubator.apache.org, 2014.

[7] A. Bhattacharyya. On a measure of divergence between two multinomial populations. *Sankhya: The Indian Journal of Statistics (1933-1960)*, 7(4):401–406, 1946.

[8] S. Chakravarthy and Q. Jiang. *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[9] H. Chan, A. Perrig, and D. Song. Secure hierarchical in-network aggregation in sensor networks. In *CCS*, 2006.

[10] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in data streams. In *VLDB*, 2003.

[11] S. Das and S. Sahni. Network topology optimization for data aggregation. In *CCGrid*, 2014.

[12] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, 1996.

[13] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan. Bolt: Data management for connected homes. In *NSDI*, 2014.

[14] Q. Huang and P. P. Lee. LD-Sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams. In *INFOCOM*, 2014.

[15] R. Huebsch, M. Garofalakis, J. M. Hellerstein, and I. Stoica. Sharing aggregate computation for distributed queries. In *SIGMOD*, 2007.

[16] T. Kohno, A. Broido, and K. Claffy. Remote physical device fingerprinting. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):93–108, April 2005.

[17] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.

[18] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *SIGMOD*, 2015.

[19] Y. Mansour, B. Patt-Shamir, and D. Rawitz. Overflow management with multipart packets. In *INFOCOM*, 2011.

[20] Z. H. Mir, C. Suh, and Y.-B. Ko. Performance improvement of ieee 802.15. 4 beacon-enabled wpan with superframe adaptation via traffic indication. In *NETWORKING 2007. Ad Hoc and Sensor Networks, Wireless Networks, Next Generation Internet*, pages 1169–1172. Springer, 2007.

[21] K. V. M. Naidu, R. Rastogi, S. Satkin, and A. Srinivasan. Memory-constrained aggregate computation over data streams. In *ICDE*, 2011.

[22] M. Nikulin. Hellinger distance. *Hazewinkel, Michiel, Encyclopedia of Mathematics*, 1946.

[23] N. Pansare, V. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. In *VLDB*, 2011.

[24] O. Papapetrou, M. Garofalakis, and A. Deligiannakis. Sketch-based querying of distributed sliding-window data streams. In *VLDB*, 2012.

[25] I. Pefkianakis, H. Lundgren, A. Soule, J. Chandrashekar, P. Le Guyadec, C. Diot, M. May, K. Van Doorselaer, and K. Van Oost. Characterizing home wireless performance: The gateway view. In *INFOCOM*, 2015.

[26] G. Scalosub, P. Marbach, and J. Liebeherr. Buffer management for aggregated streaming data with packet dependencies. In *INFOCOM*, 2010.

[27] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003.

[28] N. Tatbul and S. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, 2006.

[29] B.-D. Tran. Systematic approach to multi-layer parallelisation of time-based stream aggregation under ingest constraints in the cloud. In *Master Thesis*. EPFL, 2014. https://infoscience.epfl.ch/record/214751.

[30] D. T. Wagner, A. Rice, and A. R. Beresford. Device analyzer: Large-scale mobile data collection. *SIGMETRICS Perform. Eval. Rev.*, 41(4):53–56, Apr. 2014.

[31] G. Wang, T. E. Ng, and A. Shaikh. Programming your network at run-time for big data applications. In *HotSDN*, 2012.

[32] C. Xia, D. Towsley, and C. Zhang. Distributed resource management and admission control of stream processing systems with max utility. In *ICDCS*, 2007.

[33] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: A real-time analytical data store. In *SIGMOD*, 2014.

[34] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP*, 2009.

[35] Q. Zhang, J. Liu, and W. Wang. Approximate clustering on distributed data streams. In *ICDE*, 2008.

[36] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *SIGMOD*, 2005.