CE-Storm: Confidential Elastic Processing of Data Streams

Nick R. Katsipoulakis, Cory Thoma, Eric A. Gratta,
Alexandros Labrinidis, Adam J. Lee, Panos K. Chrysanthis

Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA
{katsip, corythoma, eag55, labrinid, adamlee, panos}@cs.pitt.edu

ABSTRACT

Data Stream Management Systems (DSMS) are crucial for modern high-volume/high-velocity data-driven applications, necessitating a distributed approach to processing them. In addition, data providers often require certain levels of confidentiality for their data, especially in cases of user-generated data, such as those coming out of physical activity/health tracking devices (i.e., our motivating application). This demonstration will showcase Synefo, an infrastructure that enables elastic scaling of DSMS operators, and CryptStream, a framework that provides confidentiality and access controls for data streams while allowing computation on untrusted servers, fused as *CE-Storm*. We will demonstrate both systems working in tandem and also visualize their behavior over time under different scenarios.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing; Distributed Databases*

Keywords

Distributed Data Stream Management System; Continuous Queries; Confidentiality; Elasticity

1. INTRODUCTION

The convergence of computing / sensing / mobile devices is generating a plethora of diverse data streams and is one critical aspect of the *Big Data* landscape [12], often referred to as the *Velocity* dimension. Modern data stream management systems (*DSMSs*) have been proposed to address the needs of monitoring applications over incoming data streams. *DSMSs* have been around for many years, as academic prototypes [4, 5, 7, 23] and commercially [1, 2]. A lot of the systems research work in *DSMSs* has dealt with *scheduling* in single-server environments [8, 18, 13], *load shedding* in single-server environments [20, 19, 16, 15], and *fault tolerance* in distributed environments [9].

SIGMOD '15 May 31 - June 04, 2015, Melbourne, VIC, Australia

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2735357

The system that we will demonstrate, *CE-Storm*, has two distinct components that are working in tandem. First of all, we have *Synefo*, which provides a data/control routing infrastructure on top of the Storm engine [21], in order to enable scale-out and scale-in of Continuous Query (*CQ*) operators. Secondly, we have *CryptStream*, which provides confidentiality and access controls for streaming data, according to policies established by the data providers.

Motivating Application We assume an environment where personal health data (e.g., heart beats per minute) and physical activity data (e.g., number of steps walked, number of miles ran) of individuals, along with location and environmental data (e.g., barometric pressure), are being generated by monitoring devices such as Fitbit, Microsoft Band, Apple iWatch, etc. This data serves as input to a set of monitoring applications, which are implemented as *CQs*. Such applications can be running:

- on behalf of the user (e.g., notify me when I reach 10,000 steps in a day),
- on behalf of the user's primary care physician (who has been previously given full access by the user),
- by health insurance companies (to which the user may have provided only aggregate-level permissions in order to get a "healthy-living" discount),
- by scientists running experiments with micro-climate data (to which the user has previously provided only barometer reading access), and
- by the city (to which the user may have provided only aggregate-level permissions on location data), in order to help identify walk-friendly/bicycle-friendly areas.

Demonstration Overview This demonstration will visualize what is happening "under the hood" for CE-Storm and its components Storm, Synefo, and CryptStream. At first, we will showcase the different confidentiality levels that CryptStream supports, implementing access control policies for data providers, compute servers, or data consumers, for a workload where there is no need for scale-out / scale-in. Our demonstration will show which data tuples are transmitted encrypted (color-coded for the different encryption schemes), which tuples are special punctuation tuples, the load at each compute node, the underlying topology, the access control policies of each data provider, etc. Then, we will demonstrate CryptStream running over Synefo with a workload that requires scale-out, in order to avoid overload.

Over-time, the workload will return to normal levels and will showcase the scale-in operation.

Next, we present the CE-Storm System Model (Sec. 2), followed by a short introduction to *Synefo* (Sec. 3) and to CryptStream (Sec. 4). We conclude with a detailed description of the demonstration setup and scenarios (Sec. 5).

2. SYSTEM MODEL

Fig. 1 depicts the stack of CE-Storm. On the bottom layer, Storm's [21] engine is used. We preferred Storm over other systems, like Spark Streaming [23], mainly because the former is designed for per-tuple granularity during processing. Above Storm, *Synefo* is our proposed elastic shared–nothing approach for online scale–out/in stream processing. On top of that, *CryptStream* is responsible for ensuring confidentiality and access control of streaming data, according to data provider, server, or data consumer policies.

Synefo assumes a generic *data-flow oriented* computation model. This means that input is typically in the form of *data streams* and different *data processing tasks* can be executed at each processing node (or component, c_i). We assume a distributed system, so the processing nodes are part of a *topology*[1]. Each data processing task will receive input from one or more nodes; its output will be "fed" to another node in the topology, in a pipeline fashion.

Our generic model allows for different styles of computation. In particular, it can support *DSMS*-style computation: predefined *CQs*, consisting of multiple operators executing at one or more components, continuously processing the incoming streaming input and producing streaming output. This generic model can also support multi-stage computation that is typical with Map/Reduce and other modern analytics frameworks: each job j is broken down into a set of tasks $t \in T$, representing the consecutive stages of a data transformation process. Tasks are connected to form an execution tree (or a topology). A task is executed by one or more components c_i in parallel, and either disseminates data in the topology, or processes incoming data.

3. SYNEFO OVERVIEW

Load management in *DSMSs* has previously been addressed in stand-alone and distributed environments, primarily through *load shedding*, i.e., by dropping some of the tuples when there is an overload, in order to guarantee an acceptable Quality of Service (*QoS*). Such QoS is usually described in the form of the maximum acceptable response time or *delay target* [20, 19]. Load shedding is typically able to maintain latency and throughput at acceptable levels, however, this usually happens at the expense of the Quality of Data (*QoD*) produced. Load balancing becomes even more challenging when transaction processing or decision-making workloads are involved. Previously demonstrated work has shown how in–memory relational database management systems handle streaming data for the aforementioned types of workloads [11].

Unfortunately, most (if not all) of the previous approaches

[1]We make the distinction between physical and active topology; this is explained in the next section.

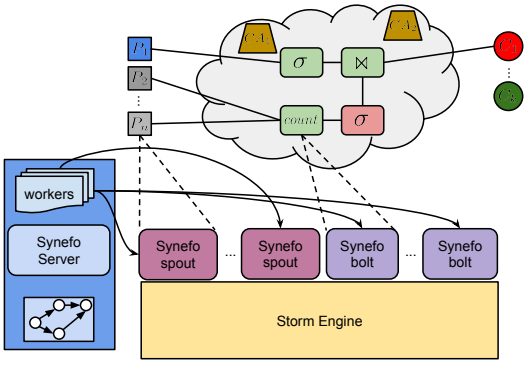


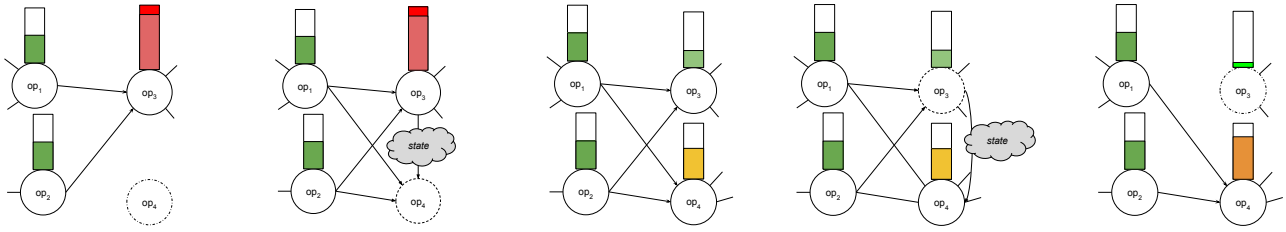
Figure 1: The proposed system stack.

assume that available processing components are known before the start of the computation and all of them remain active all the time during the computation, irrespective of whether they are used in the computation or not [21, 3, 23]. This requirement can impact a system's efficiency, in terms of cost, energy consumption and performance. On the one hand, if fewer components are provisioned, the system will soon reach its maximum capacity, and performance will deteriorate. On the other hand, if more components have been provisioned but are under–utilized, there will clearly be wasted resources, which would translate in extra energy costs, extra wear on computing nodes, etc.

Synefo[2] aims to provide an elastic execution environment for data–flow oriented computation, which can change the number of active processing components during runtime. Additional processing nodes can be included in the execution, if some of the active nodes are struggling, and under–utilized ones can be combined in order to keep the energy consumption of the cluster low. Our proposed system initiates execution by dividing the available components into *active* (i.e., components participating in processing) and *inactive* (i.e., components which are part of the cluster, but are not processing tuples). *Synefo*'s main role is to manage the activity status of available processing components, and route tuples to components accordingly. Our implementation works with Storm and it consists of (a) a Storm component API (*Synefo* bolt and spout), and (b) a coordinating entity for orchestrating the scale–out/in actions (*Synefo* server) (Fig. 1). Our design choice to have *active* and *inactive* operational components emanates from Storm's design limitations and our initial goal to have *Synefo* work with vanilla Storm clusters.

Online scale–out with shared state among processing components has been previously proposed in the work of Fernandez et al. [10]. Their prototype used virtual machine (*VM*) backups of upstream operators, which were responsible for disseminating state when operators were scaled out. However, the backup upstream operators will be a bottleneck in congested environments. Also, maintaining shared state among operators can become "painful" in a *DSMS*, where hundreds of operators exist. Our approach aims to eliminate shared state costs and achieve better scalability by following the share–nothing paradigm. Recent work from Wu

[2]*Synefo* comes from the Greek word $\sigma\upsilon\nu\nu\epsilon\varphi o$ which means cloud.



(a) op_3 is over–utilized and op_4 is inactive (b) op_3's state is replicated to op_4 (c) Both operators are active (d) *Synefo* triggers a scale–in (e) op_3 is inactive and op_4 is processing tuples

Figure 2: *Synefo*'s scale–out and scale–in scenario

et al. [22] has demonstrated ChronoStream, an elastic distributed stream processing engine. *Synefo* provides similar functionality, in terms of elasticity, but in a shared–nothing manner and is designed to work on top of Storm.

Every time a data processing task is given to the *DSMS*, an execution topology is created. There are two types of topologies in *Synefo*: the physical topology of components available in the distributed network of components, and the active topology of components participating in the data processing. Every time a scale–out/in command is given, components are added to (or removed from) the active topology.

Next, we give a brief overview of *Synefo*'s main parts:
(i) *Synefo* server, (ii) *Synefo* spout, and (iii) *Synefo* bolt.

3.1 Synefo Server

The *Synefo* server monitors resource utilization levels in the cluster and has an active role during the initiation of a data processing task in the *DSMS*. The server awaits for incoming connections of *Synefo* Storm components. A Coordinator thread is launched so that it synchronizes the execution of all worker threads, which are responsible for accumulating resource usage statistics from each component. By the time all components have connected and registered to the server, topology information is gathered and the execution of the topology initiates.

Our prototype follows a centralized management approach for the *Synefo* server. This can potentially lead to the server being the single point of failure of the application. We will consider fault tolerance as part of our future work.

3.2 Synefo Spout

The component responsible for disseminating data into the topology is the *Synefo* spout. This is just a sub–class of Storm's BaseRichSpout, having been extended to: (i) initiate operation by registering to the *Synefo* server, and (ii) perform scale–out/in operations. In addition, during execution the *Synefo* spout reports resource usage data periodically and is able to execute scale–out/in commands directed by the *Synefo* server. A downstream component of a *Synefo* spout is assumed to always be a *Synefo* bolt, because a spout represents a stream of incoming data in Storm. Every time a scale-out/in command is received by a *Synefo* spout, it updates its routing information accordingly. For instance, if a *Synefo* spout is directed to scale–out by adding a bolt in its active downstream tasks list, the following happens:

- Sends command tuples to all downstream bolts notifying them about the scale–out operation
- Updates routing table information, to include the newly added bolt in the active downstream components.

At this point we have to mention that as in the *Synefo* server, the *Synefo* spout has to maintain an active connection with its worker counterpart in the server.

3.3 Synefo Bolt

The *Synefo* bolt is the last part of our *Synefo* system and is similar to a *Synefo* spout. The main difference between bolts and spouts is that the former are extended to handle incoming command tuples for scaling out/in and exchanging state with other bolts.

Bolts are required to maintain operator state, such as intermediate results or join window data. When a *Synefo* bolt receives a command tuple for scaling–out/in, this state needs to be transferred. If an additional *Synefo* bolt is added to that stage of the computation, the new bolt has to receive the current state of the already active bolt. Similarly, when a bolt is scaled–in (meaning that is not going to be active any more) its state should be sent to all operating bolts performing a part of the same computation. *Synefo* bolts transfer state transparently from the operation which is currently executed in a component. It is the operator's implementation responsibility to utilize the received state accordingly.

Fig. 2 illustrates a scale–out and scale–in operation in *Synefo*. The circles represent active components executing data processing tasks and the bars indicate the load on each one. Components op_1 and op_2 are sending their output to op_3, which has exceeded its operational capacity (Fig. 2a). Therefore, *Synefo* decides that op_3 should be scaled–out and identifies op_4 as an inactive component available. Components op_1 and op_2 are directed to involve op_4 as their downstream component, and op_3 transfers its state to op_4 (Fig. 2b). After the scale–out finishes, we can see that the load has been balanced among components and, in fact, op_3 is underutilized (Fig. 2c). Hence, *Synefo* initiates a scale–in procedure (remaining state in op_3 is sent to op_4) and components op_1 and op_2 are directed to exclude op_3 from their active components (Fig. 2d). Finally, the components carry on with the execution and component op_3 is inactive (Fig. 2e).

Our prototype has been implemented as a general extension to the current Storm API. We decided to maintain the separation of data routing from computation as is the case for

Scheme	Type of Queries	Supported operators	Information Gained by Adversary
RND	None	None	Nothing
DET	Equality	Select, Project, Equi-Join, Count, Group By, Order by	Equality of attributes
OPE	Range	Select, Join, Count	A partial to full order of tuples
HOM	Summations	Aggregates over summations	Nothing

Table 1: Summary of what types of queries and operators are supported by each encryption scheme, as well as what each scheme could reveal to a potential adversary.

Storm. Therefore, it is the operators' responsibility to keep state consistent at a time of an imminent scale–out, so that a *Synefo* bolt transfers it to its peer bolts correctly.

4. CRYPTSTREAM OVERVIEW

Traditional database systems rely on login and user authentication to maintain access controls over their data. When these systems are outsourced, simple user authentication may not be enough since a third party now has access to the data, and is in control of the physical location of data storage. A malicious third party may wish to gain knowledge of their clients by snooping on their data. Systems like CryptDB [17] aim at allowing third party storage without leaking plaintext values through the use of encryption. While this helps ensure confidentiality for traditional database models, the techniques used by CryptDB do not extend to the data streaming paradigm. The main issue is that streaming data often does not have the data consumer controlling or owning any part of the data provider, and therefore cannot control how to encrypt data entering the system. To help alleviate this problem, we developed *CryptStream*.

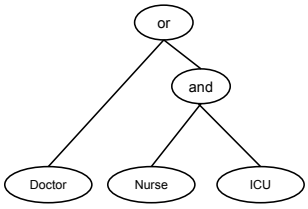


Figure 3: Simple ABAC policy stating that the Client be either a doctor or a nurse in the ICU.

CryptStream aims at enforcing access controls in *DSMSs* via cryptographic protocols which prevent unauthorized parties from accessing data. Working under the assumption of an honest-but-curious adversary, *CryptStream* protects a data provider's data from third party cloud service providers or other untrusted computing platforms. Access control is enforced through the union of Security Punctuations [14] and Attribute Based Access Controls (ABAC). ABAC policies grant or deny access based on what attributes the querier possesses. ABAC policies form a tree where each leaf node is an attribute and internal nodes are the "and" or "or" of the child nodes. An example is given in Fig. 3. This example illustrates a scenario where a data provider wants the data consumer to be a doctor or a nurse in the ICU ($doctor \vee (nurse \wedge ICU)$). ABAC policies are transmitted via a Security Punctuation (SP). A SP is a tuple emitted into a data stream wherein a data provider includes information on which data is being protected, how the data is being protected (i.e. what access control policy is being enforced) along with timestamps, ids, and other information.

Since *CryptStream* assumes an untrusted server, ABAC policies are enforced via Attribute Based Encryption (ABE). Both the security policy and data remain hidden to the server since only an encrypted string passes through the server. Work done prior to *CryptStream* utilized ABE for all data transmissions, which lead to prohibitively large performance overheads given the costly nature of ABE [6]. To improve upon this method, ABE is only used to encrypt cryptographic keys from faster protocols which allow computation to be preformed on the server without large overheads. *CryptStream* will only pay the cost of ABE whenever an access control policy is updated by data providers.

Similar to CryptDB, there are four encryption techniques used by *CryptStream*: *random* (RND), *deterministic* (DET), *order preserving* (OPE), and *Homomorphic* (HOM). Each encryption type enables some sort of computation on the server, but also may reveal some information to the server about each tuple. RND provides the strongest guarantee by leaking no information, but does so at the sacrifice of computation, since none can be done server-side. DET allows the server to decide if two values are the same, which enables equality selection and join queries, but will leak tuple equality. OPE provides more functionality than RND and DET since it holds the property that if $x < y$ then $OPE(x) < OPE(y)$. This allows the server to run range selections and joins, but it does provide the server with orderings amongst tuples. Finally, HOM allows the server to compute summations and does so without sacrifice in confidentiality since no information is leaked. Table 1 summarizes these techniques. It is important to note that the Data Provider chooses which level to encrypt their data based what they are willing to reveal to the server.

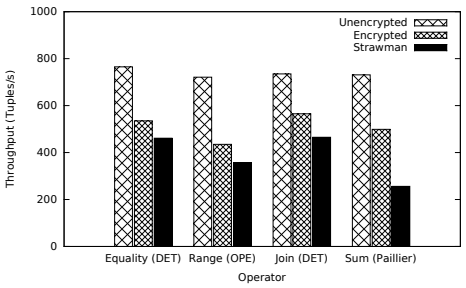


Figure 4: Throughput for each of the different operations supported for both unencrypted and encrypted streams. Join includes both equality and range oriented joins.

CryptStream allows for both data consumers and servers to provide security policies in addition to those from the data providers. Data consumer policies utilize Security Punctuations, but are not limited to ABAC policies as long as the server can support the intended policy. Server policies can be of any variety that is supported. *CryptStream* will first

check the server policy to make sure a data consumer is allowed access by their internal policy. The data consumer policy is then verified, and if both return true, the tuple is sent. ABAC and ABE enforce the data provider access control policies, so if the data consumer was not given permission via their attributes, the tuple is useless to them. *CryptStream* provides the above listed contributions with modest overheads. Fig. 4 shows the overheads of each encryption type above. Tests were run on a cluster of 10 Mac Minis running with 2 GB of ram and a 1.83 GHz Intel CPU 1400 processor. All components were programmed in Java and packaged in Jar files for distribution. The system accepted simulated twitter-like data from a workload generator which provided control over distribution and frequency of keywords. Throughput is not greatly affected by the increase in computation, and confidentiality remains preserved.

5. DEMONSTRATION OVERVIEW

Our demonstration will focus on showing: (i) *CryptStream*'s ability to enforce confidentiality policies on streaming data, (ii) *Synefo*'s scale–out and scale–in abilities, and (iii) our visualization tool for Storm/*Synefo*/*CryptStream*.

We will be using the Motivating Application described in Section 1 for our demonstration, where the input data streams are physical activity and health data of individuals, which come with different levels of confidentiality, depending on the type of data consumers.

The audience will be able to see *CryptStream*'s confidentiality enforcement and the way sensitive data can be protected from possible adversaries in a cloud infrastructure. Also, it will be made clear how *Synefo* makes elasticity feasible in a widely accepted stream processing framework like Storm.

5.1 Storm User Survey

The motivation behind the development of a visualization framework for Storm came from a user study we performed in the Fall of 2014. A survey about Storm's usability was presented to members of the Storm users mailing list in order to gauge the difficulty of using Storm. After one week, we have gathered 10 responses from a variety of end users, ranging from academic researchers to data analysts. We present some of those results, in aggregate form here. Table 2a shows that more than 30% of the users mentioned that debugging is the most disruptive task in Storm, followed by logging and monitoring an application. In addition, we wanted to get users' opinions on features of Storm that need to be improved (Table 2b). Most participants replied that the visualization techniques of Storm are poor and need to be enhanced with extra features.

Operation	Percent
debug	30.43
log	26.08
monitor	26.08
docs	13.04
deploy	4.34

(a) Operation that users found most disruptive in Storm

Feature	Percent
visuals	35.71
docs	21.42
new features	21.42
support	21.42

(b) Storm features that need improvement.

Table 2: Storm User Study Results

5.2 Visualizing CE-Storm

The purpose of the visualization component of our demonstration is to show what is happening "under the hood" for Storm, Synefo, and CryptStream. In particular, we will employ the familiar visual paradigm of a network of connected nodes to illustrate the topology of the underlying system. We will provide both topology "views": the physical topology, where each node of the network is a compute node, and the logical topology, where each node of the network is a data processing task or *CQ* operator.

For each node, we will primarily show the current load information and give the option to drill down to get more detailed information. This would be other useful statistics such as a breakdown of the different types of tuples going through the system (e.g., punctuation vs regular, unencrypted/RND/DET/OPE/SUM), amount of state stored, etc. Within each node we will be able to see a sample of tuples going through the system. Tuples will be coded according to their type, as well as an indication as to the encryption used for a given attribute in the tuple. We will provide aggregate statistics for each edge of the network. Our visualization will include a panel with summary information about the overall status of the system, including input and output rates. We will be able to make changes to the input workload using this panel.

We aim to make the demonstration fairly interactive. Towards that end we will have video-playback type functionality. In particular, we will also provide a *pause* button, that will then allow us to slow down the speed of the visualization, *reverse*, *fast-forward*, or skip to live mode. It should be noted that execution at Storm/*Synefo*/*CryptStream* will not be affected; we will only be able to see the outcome of the execution at our own pace. Another interactive feature will be the ability to drill down and get more information about different parts of the system (e.g., for nodes/edges, as mentioned above or the different access control policies). Finally, we will show changes to the system (e.g., changing policies, forcing scale-out/in, etc) through the user-interface.

5.3 Demonstration Setup

The testbed of our demonstration is going to be a cluster of EC2 instances, forming a distributed network of stream operators. The entire stack of Fig. 1 will be setup in each EC2 instance. For data input, we are going to use synthetic data produced by our own custom stream generator, which produces data that follow user–defined (i) frequency distributions, (ii) vocabulary, and (iii) word sampling methods.

5.4 Demonstration Scenarios

Our live demonstration will have four distinct phases:
(i) Simple: A simple demonstration of *CryptStream*, with two different policies, meant to familiarize the audience with the visualization interface; **(ii) Complex**: A demonstration of *CryptStream*, with a more complex scenario, having policies from data providers and from data consumers;
(iii) Scale–out: A demonstration of *Synefo*'s scale–out mechanism, by utilizing a workload where the incoming rate is too high for an operator to be handled by a single node;
(iv) Scale–in: A demonstration of *Synefo*'s scale-in mechanism, using a workload where a high input rate is reduced,

so that fewer nodes are needed for the processing of the operator that was previously scaled–out. We provide more details for the four different phases next.

Simple Scenario We will demonstrate different access control policies, as explained in the Motivating example. *CryptStream* will enforce different levels of access by deploying its set of encryption techniques (Section 4). This will illustrate the ability of *CryptStream* to provide different levels of access to different data consumers depending on the trust level for that data consumer.

Complex Scenario We will demonstrate a more complicated scenario, where access control policies come from data providers, data consumers, and compute servers. We will show detailed access control policies information and update the policies to see how *CryptStream* handles the changes.

Scale–out Scenario The third scenario is going to focus on the elastic ability provided by *Synefo* and the trade–off between elasticity and confidentiality experienced in complex scenarios of CE-Storm. We will extend the *Simple Scenario*, but have an increased data input rate above the system's capacity. This increase in the volume of incoming data might be caused by a "health scare" that requires more data to be accumulated to better monitor changes in personal well being, or by New Year's resolutions which prompt more people to join a gym and try to get fit. In order to avoid dropping tuples, *Synefo* will take action by scaling–out the congested operators. This scale–out will be triggered automatically by following user–defined usage thresholds (provided to *Synefo* at start–up). Our system will act proactively and if an operator reaches a usage percentage equal or above the user–defined threshold, it will be scaled–out. An important part of this scenario will be to demonstrate the cost of scaling–out, in terms of confidentiality. For instance, some operators with strict confidentiality policies might have to scale–out to machines that are not trusted. Therefore, we will demonstrate a number of trade–offs in terms of performance gain compared to confidentiality constraints relaxation.

Scale–in Scenario The last scenario will demonstrate a scale–in situation, where the data input rate falls below a certain limit, resulting in computing nodes being underutilized. This could happen because, for example, the health scare went away, or with the arrival of March (for those trying to stick to their New Year's resolutions).

Acknowledgments

This material is based on work supported by the National Science Foundation under grants CNS-1253204, IIS-0746696, and OIA-1028162.

6. REFERENCES

[1] IBM System S. `http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=2534`.

[2] SQLstream. `http://www.sqlstream.com/`.

[3] Summingbird. `http://github.com/twitter/summingbird`.

[4] D. J. Abadi et al. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[5] Y. Ahmad et al. Distributed operation in the borealis stream processing engine. In *Proc. of ACM SIGMOD*, pages 882–884, 2005.

[6] D. T. T. Anh and A. Datta. Streamforce: outsourcing access control enforcement for stream data to the clouds. In *CODASPY*, pages 13–24. ACM, 2014.

[7] A. Arasu et al. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[8] B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: Operator scheduling for memory minimization in data stream systems. In *Proc. of ACM SIGMOD*, pages 253–264, 2003.

[9] M. Balazinska et al. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1):3:1–3:44, Mar. 2008.

[10] R. Castro Fernandez et al. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proc. of ACM SIGMOD*, pages 725–736, 2013.

[11] U. Çetintemel et al. S-store: A streaming newsql system for big velocity applications. *PVLDB*, 7(13):1633–1636, 2014.

[12] H. V. Jagadish et al. Big data and its technical challenges. *Comm. of the ACM*, 57(7):86–94, Jul 2014.

[13] L. A. Moakar, A. Labrinidis, and P. K. Chrysanthis. Adaptive class-based scheduling of continuous queries. In *Proc. of IEEE SMDB Workshop*, pages 1–6, 2012.

[14] R. Nehme, E. A. Rundensteiner, and E. Bertino. A security punctuation framework for enforcing access control on streaming data. In *Proc. of IEEE ICDE Conference*, pages 406–415, 2008.

[15] T. N. Pham, P. K. Chrysanthis, and A. Labrinidis. Self-managing load shedding for data stream management systems. In *Proc. of SMDB Workshop*, pages 1–7, 2013.

[16] T. N. Pham, L. A. Moakar, P. K. Chrysanthis, and A. Labrinidis. Dilos: A dynamic integrated load manager and scheduler for continuous queries. In *Proc. of SMDB Workshop*, pages 10–15, 2011.

[17] R. A. Popa et al. CryptDB: protecting confidentiality with encrypted query processing. In *Proc. of ACM SOSP*, pages 85–100, 2011.

[18] M. A. Sharaf et al. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Trans. Database Syst.*, 33(2):5.1–5.44, 2008.

[19] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proc. of VLDB*, pages 159–170, 2007.

[20] N. Tatbul et al. Load shedding in a data stream manager. In *Proc. of VLDB*, pages 309–320, 2003.

[21] A. Toshniwal et al. Storm@twitter. In *Proc. of ACM SIGMOD Conference*, pages 147–156, 2014.

[22] W. Yingjun and T. Kian-Lee. Chronostream: Elastic stateful stream computation in the cloud. In *Proc. of IEEE ICDE Conference*, 2015.

[23] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proc. of HotCloud Conference*, pages 423–438, 2012.