Effective Access Control in Shared-Operator Multi-tenant Data Stream Management Systems

Marian Zaki, Adam J. Lee*, Panos K. Chrysanthis*

College of Science and Engineering, Computer Science, Houston Baptist University, Texas, USA
mzaki@hbu.edu

*Department of Computer Science, University of Pittsburgh, Pennsylvania, USA
{adamlee, panos}@cs.pitt.edu

Abstract. The proliferation of stream-based applications has led to the widespread use of Data Stream Management Systems (DSMSs), which can support the real-time requirements of these applications. DSMSs were developed to efficiently execute continuous queries (CQs) over incoming data. Multiple CQs can be optimized together to form a query network by sharing operators across CQs. DSMSs are also required to enforce access controls over operators according to data providers' policies. In this paper, we propose the first solution to satisfy access control policies at run-time in shared-operator networks in an non-disruptive, efficient manner. Specifically, we propose a new set of low overhead streaming operators, coined as Privacy Switches (PrSs), which are strategically placed in the operator network to dynamically allow or deny the flow of data in certain branches of the network based upon the current state of access control permissions. Our experimental evaluation confirms that our approach introduces low overheads in the shared operator networks while achieving high savings in the overall network performance.

Keywords: access control · operator networks · stream processing engines

1 Introduction

Nowadays, an increasing amount of data is produced in the form of high velocity data streams. This has led to the proliferation of *stream-based applications* such as sensor-based monitoring (surveillance, car traffic, air quality), financial applications (stock markets, fraud detection), and health care applications. At the same time, it has led to the widespread use of *Data Stream Management Systems* (DSMSs) [1, 8, 4, 10, 5, 15], developed to efficiently execute *continuous queries* (CQs) over streaming data to support these classes of applications. DSMSs are also referred to as Stream Processing Engines (SPEs) and stream processing systems.

CQs are stored queries that execute continuously over data streams as data arrives, on the fly. Since CQs are long-running, multiple CQs can be optimized

together to form a query (operator) network, in which multiple queries may share one or more physical operators. The intermediate tuples produced by a shared operator are placed in a shared input queue for the downstream operators of the individual queries involved in the sharing. These optimizations increase the throughput of processing multiple CQs simultaneously and minimize the memory usage and computation times.

In DSMSs that host sensitive data, e.g., patient monitoring data, financial data, etc., data providers restrict data accesses via access control policies that describe the conditions under which users are permitted access to specific data streams. Accordingly, the CQs registered by the system users may be granted or denied access to specific data streams during the execution of these queries.

The most commonly used ways of applying access control over operators in query networks enforce access control either before (pre-filtering) or after (post-filtering) the execution of the network [17]. In both techniques, the fixed placement of the access control filters may considerably limit query performance. Pre-filtering means cutting off the streams in case any of the users lose access, which will affect the input streams feeding into downstream shared-operator networks. Post-filtering on the other hand, causes all operators to execute until the end and then denying access to the query results for the users who lost access. This wastes resources by processing query results that are never accessed.

An alternative solution is to isolate the queries that have temporarily lost access from the rest of the shared-operator network. This requires modifying the shared-operator network at runtime whenever access control changes are encountered. This can cause extensive overheads on the system that can potentially affect the system throughput and overall performance.

Under these challenges, it is vital for DSMSs to satisfy the access control policies at run-time in shared-operator networks in a non-disruptive and efficient manner, reinforcing both the *need to share* and *need to protect* design models. To balance these properties, we propose a cost-effective way for shared-operator networks to enforce access control within the network, thereby eliminating the need to perform any changes to the structure of the shared-operator networks in case of intermittent changes in the access control.

Specifically, in this paper we make the following contributions:

- We propose a new set of low overhead streaming operators that we refer to as *Privacy Switches* (PrSs). These switches dynamically allow or deny the flow of data streams in shared-operator networks based upon the current state of the access control permissions. Accordingly, certain branches of the query operator networks can be halted and spared execution to accommodate intermittent loss of access to different users.
- We propose a placement algorithm to identify the best placement points for PrSs in an optimized shared-operator network. The strategic placement of the switches ensures seamless execution of CQs and reduces overheads in the networks while honoring all changes in access permissions.
- We experimentally evaluate our approach by generating shared-operator networks with a controlled set of generation rules and input parameters. Our

experiments show that PrSs introduce low overheads in shared-operator networks and achieving high savings in overall network performance.

The remainder of the paper is organized as follows. Section 3 describes prior work on access control in DSMSs and introduces the concept of security punctuations. Section 4 presents our system model, assumptions, and preliminaries. Section 5 describes our proposed approach, which is evaluated in Section 6. Section 7 concludes the paper.

2 Motivating Example

In the smart healthcare industry [13, 9], wearable smart devices are equipped with various bio sensors that are used to measure and monitor diverse health data of individuals such as blood glucose levels, blood pressure, oxygen saturation, heart rate, etc. This data makes it possible for different health care applications to continuously query these data stream and provide an alarm service, notifying in the risk of health issues based on individual activities of daily living.

Even though a large collection of health data is a valuable asset to the smart healthcare field, serious concerns of data privacy are being raised. That is, indiscriminate collection of personal health data can cause significant privacy issues. Hence, most users do not agree to their health data being collected for the purposes of data analysis which presents a major obstacle for the development of smart healthcare services.

The following listing demonstrates three different data streams and Figure 1 illustrates two example CQs that can be used in smart healthcare applications.

```
Stream1: streamid, location, heartRate, timestamp
Stream2: streamid, location, speed
Stream3: streamid, location, screentime, category
```

Listing 1.1. Data streams generated by smart devices

```
SELECT s1.streamid, s1.heartRate
  FROM Stream1 as s1 [RANGE 5 min, SLIDE 1 min],
       Stream2 as s2 [RANGE 5 min, SLIDE 1 min]
  WHERE s1.location = s2.location
  AND s1.timestamp > 6:00am
  AND s1.timestamp < 7:00pm
  AND s2.speed < 30;
```

```
SELECT s1.streamid, s1.heartRate
  FROM Stream1 as s1 [RANGE 10 min, SLIDE 1 min],
       Stream2 as s2 [RANGE 10 min, SLIDE 1 min],
       Stream3 as s3 [RANGE 10 min, SLIDE 1 min]
  WHERE s1.location = s2.location
  AND s2.location = s3.location
  AND s1.timestamp > 8:00am
  AND s1.timestamp < 7:00pm
  AND s2.speed < 0
  AND s3.screentime > 5h;
```

Fig. 1. Continuous queries $CQ1$ and $CQ2$

When executing these CQs in DSMSs, the queries optimizer will be able to identify the common sub-expressions (both `SELECT` timestamps within the same interval and have the same `JOIN` conditions on the input streams) found in the queries and accordingly construct a shared-operator network as shown in

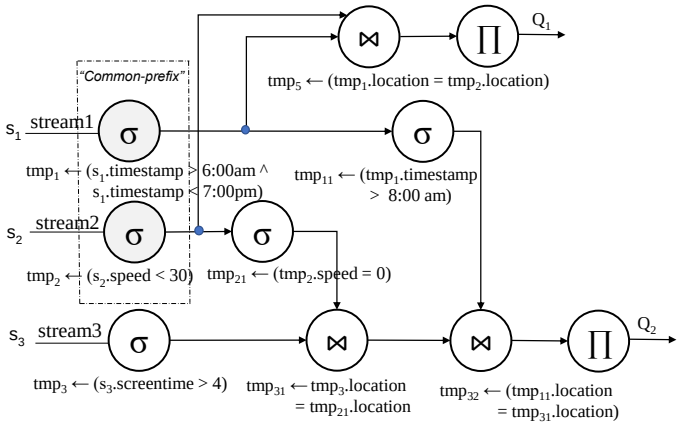


Fig. 2. Shared-operators network for $Q1$ and $Q2$

Figure 2. The figure excludes the window specifications, for clarity since they are the same for all three data streams. Furthermore, in the figure, the highlighted vertices are annotated as "common-prefix" operators since those are the shared operators between both queries CQ_1 and CQ_2 (i.e., the output of these operators feed into two different operators belonging to the two queries).

3 Related Work

Prior work on access control enforcement in DSMSs can be divided into two different categories: *cryptographic* solutions and *non-cryptographic* solutions. Cryptographic solutions such as those presented in Streamforce [2], utilize Attribute-Based Encryption (ABE) to enforce access control which requires the data provider to be directly involved in the querying process. PolyStream [21] allows untrusted third-party infrastructure to compute on encrypted data, allowing in-network query processing and access control enforcement. PolyStream uses a combination of security punctuations, ABE, and hybrid cryptography to enable flexible (ABAC) access control policy management. Sanctuary [20] uses Intels SGX as a trusted computing base for executing streaming operations on untrusted cloud providers.

Non-cryptographic solutions such as those presented by Carminati et al. [7, 6], provide access control via enforcing Role Based Access Control (RBAC) and replacing the operators with secure versions which determine whether a client can access a stream by referencing an RBAC policy. Lindner et al. [14] utilize limited disclosure by using filtering operators and applying them to the stream query processing results to filter the output based on relevant access control policies. Ng et al. [18] also use the principles of limited disclosure to limit who can access and operate on data streams, requiring queries to be rewritten to match the level at which they can access the data.

Most non-cryptographic solutions require changing the underlying DSMS either by modifying the traditional operators to become security aware or by rewriting the queries, and therefore they are not globally applicable. Limited disclosure applies basic filtering to the query outputs which means that query operators execute at all times regardless of whether the output is used or not.

Unfortunately, all the cryptographic and non-cryptographic solutions that have been proposed to maintain privacy through access control focused mainly on independently executing queries. There has been no research to date that allows access control policies on the input data streams and queries to be applied over shared-operator networks to maintain both privacy and high performance. Our proposed privacy switches approach is the first work in this context.

4 Preliminaries and System Model

In this section we will introduce our system model, assumptions and the preliminaries used in our proposed solution.

4.1 SYSTEM MODEL

The system consists of the following four main classes (entities) as shown in Figure 3):

- *Data Providers* generate and distribute data streams to DSMSs. They have the choice of creating and updating the access control policies for the data streams that they emit.
- *Data Consumers* can be individuals or applications who submit CQs to the DSMSs Data consumers must submit credentials to satisfy the policies protecting access to their registered queries.
- *DSMSs* handle all the incoming data streams and submitted queries. They execute query optimizers to generate shared-operator networks for the interleaved execution of queries and schedule the operators' execution. They enforce the access control policies on the data streams and the corresponding query results.
- *Authorization Servers* can be separate entities or an integral part of the DSMSs. They validate users' credentials to check authorizations, and inject relevant SPs into the incoming data streams. They keep track of system state changes such as new data consumers signing up, changes in access control policies, and changes in the users' credentials (e.g., revoked or expired). These changes can trigger the injection of new SPs into impacted data streams to alter which users are able to see the results of CQs.

4.2 Operators and Operator Networks

The following are the main defining elements of data streams, CQs, and operator networks that will be used throughout the remainder of this paper:

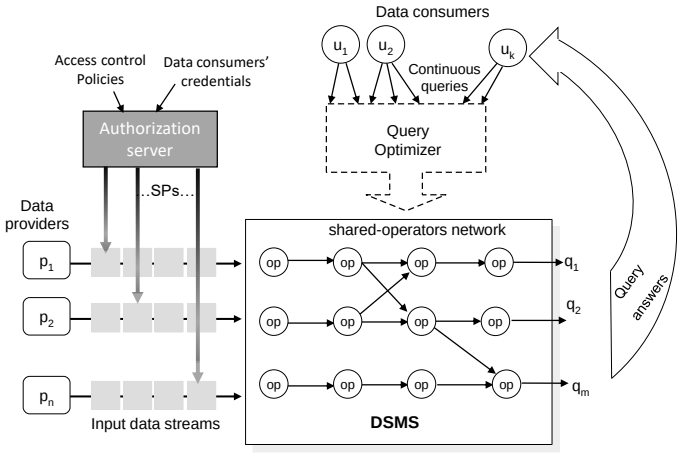


Fig. 3. DSMS system model

- DSMSs are capable of processing long running set of continuous queries $\mathcal{CQ}s = q_1, q_2, \ldots, q_n$ executing over data streams in the system.
- $\mathcal{S}$ denotes the set of all input data streams in the system, while a continuous data stream $s \in \mathcal{S}$ is a potentially unbounded sequence of tuples that arrive over time.
- Tuples in a data stream are of the form $t = [sid, tid, A]$, where sid is the stream identifier, tid is the tuple identifier, and A is the set of attribute values of the tuple.
- Queries are comprised of a set of relational query operators $o_1, o_2, \ldots, o_n$.

A relational operator o_i is presented as $operator_{predicate}$ with an associated predicate. This may be a `SELECT` operator σ over one data stream, a `JOIN` operator $\bowtie$ between multiple data streams, a `PROJECT` operator Π that reduces the number of attributes included in a single data stream, or a `GROUP AGGREGATE` operator with predicate over a single attribute of a data stream to perform some algebraic aggregate function (e.g., `MAX`, `COUNT`, `SUM`, etc.).

For example, the following are valid query operators

$expr_1$: $o_1 = \sigma_{s_2.speed<30}$

$expr_2$: $o_2 = \bowtie_{s_1.location=s_2.location\ \&\ s_1.timestamp>6{:}00am\ \&\ s_1.timestamp<7{:}00pm}$

The query optimizer takes as input a set of continuous queries $\mathcal{CQ}$ and identifies groups of queries that share sub-expressions. The optimizer then generates a shared-operator network, arranged in a Directed Acyclic Graph (DAG) format, for each group of queries.

Definition 1 *An interleaved execution plan for a group of queries is a DAG network $\mathcal{N} = (V, E, L)$. V, E, and L being the set of vertices, edges and set of labels, respectively, and are defined as follows:*

- *A vertex v_i is introduced for every operator o_i in query q_i. If the results of o_i are used by more than one operator belonging to different queries, then the vertex v_i will be annotated as "Common-prefix"*
- *If the results of o_i are used in o_j, an edge $(v_i \rightarrow v_j)$ is introduced*
- *The label $L(v_i)$ is the processing done by the corresponding operator o_i (i.e., $operator_{predicate}$)*

4.3 Access Control

We assume that DSMSs can enforce *query-based* access restrictions that can be specified by both the data stream providers and/or the DSMSs over the entire data stream. For example, if a user does not renew his subscription to access certain query results in a DSMS, then this user could be temporarily denied access to any of his registered queries—by denying access to their corresponding data streams—until the subscription is renewed. Similarly, stream data providers can define policies to identify the different data consumers who are allowed to query those data streams. The work presented in this paper is very generic in that it can accommodate a wide range of access control models (e.g., RBAC [11], ABAC [12], or DAC [19]).

In general, let $\mathcal{P}$ denote the set of all authorization policies, each authorization policy $P : P \in \mathcal{P}$ enforced by a DSMS is defined as: $P \subseteq \mathcal{CQ} \times U$, where U is the set of users or user roles/attributes. Let function m be a mapping such that $m : \mathcal{CQ} \rightarrow S$, that is, m identifies the set of data streams that are being accessed by a continuous query.

For each policy P, the authorization server will perform the following:

1. evaluate the proofs of authorization for each (q_i, u_i) pair in P
2. execute the mapping function m to identify the set of streams S that are involved in each query q_i in the case of query-based policies
3. construct the necessary SPs to identify the access privileges of each user or group of users and inject them in the corresponding streams

According to the outcome of the proofs of authorizations, the *Sign* field in the SP will be set to $Sign =$ '+', if a user may access the data stream tuples for a given query at any time $ts_{access} \geq ts$ or $Sign =$ '-' if a user is denied access to the data stream at any time $ts_{access} \geq ts$. Accordingly, the injected SPs are used by our proposed algorithms to trigger the PrSs to turn *on* or *off* in the shared-operator networks to enforce access control.

4.4 Security Punctuations

We adopt the notion of *Security Punctuations* (SPs) [17, 16] proposed to enforce access control in operator networks. SPs are considered meta-data in the form of predicates that are injected into data streams in the order of their timestamps and describe the access control privileges of each query. SPs are comprised of the following fields (see Figure 4):

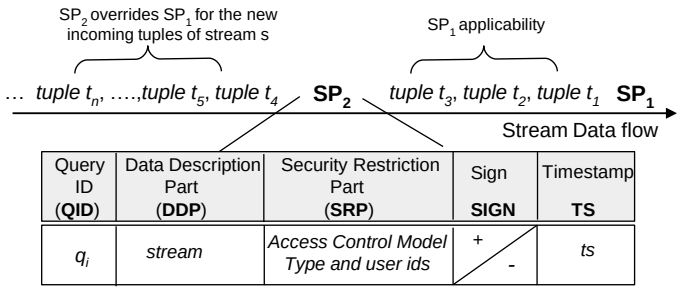


Fig. 4. Security Punctuations injected in a data stream

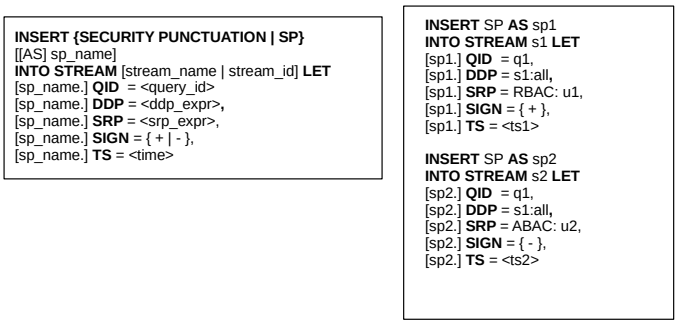


Fig. 5. SP syntax (right) and sample SPs injected in two separate data streams (left)

- *Query ID*: the identifier of the query that the SP is defining access for.
- *Data Description Part*: indicates the schema fields of a data stream tuple that are protected by the policy. This can be at the granularity of an entire stream, or specific tuples or attributes within the tuples.
- *Security Restriction Part*: defines the access control model type and the data user(s) authorized by the policy.
- *Sign*: specifies if the authorization is positive (+) or negative (-).
- *Timestamp*: the time at which the SP is generated.

Figure 4 shows an example input data stream with two SPs. Figure 5 shows an extension to the CQL syntax [3] to support the specification of SPs in data streams. The figure illustrates the syntax of an SP as well as an example of two SPs injected into two data streams. Each SP indicates the access privileges of a separate user over the tuples of the input data stream.

5 Privacy-Aware Shared Operator Networks

We now present the details of our proposed approach to achieve a cost-effective way of handling access control in shared-operator networks. As mentioned earlier in Section 1, the naive approaches of applying pre- or post-filtering may considerably limit the shared-operator network performance. The alternative approaches

of constantly changing the interleaved shared-operator networks to isolate the queries that are no longer accessible by any of the users, or even maintaining different copies of shared-operator networks are both considered very costly approaches for handling access control.

To overcome these limitations, our proposed solution involves embedding a new set of operators called *Privacy Switches*(PrSs) in the networks. At a high level, the main idea is to strategically place these switches in the network to shut off branches of operators in the case of total access loss to certain query outputs or by filtering out the query results in the case of partial access loss to queries. By doing so, the shared-operator networks can execute disruptively and efficiently.

5.1 Privacy Switches

Privacy Switches (PrSs) are the novel set of operators that will be integrated within the shared-operator networks. These switches allow shared-operator networks to execute the queries impeccably while performing access control-based filtering. For this purpose, three different types of PrSs are introduced:

- **initial-switches**: placed at some of the input streams to each operator network and perform the traditional pre-filtering operations.
- **in-network switches**: embedded within the operator network and are capable of temporarily shutting off certain branches in the network to save the unnecessary execution of some operators in certain access permission cases.
- **terminal-switches**: placed at the query outputs and they act as multiplexers that can selectively filter query output to multiple users.

All three types of PrSs operate just like any other conditional query operator. They are similar to `SELECT` or `PROJECT` operators' that filters data input streams based on the security predicates determined by the SPs injected in the streams.

To better understand how the PrSs operate, assume there is a shared-operator network that interleaves the execution of multiple queries and each query could possibly be shared by multiple users in the DSMS. The three different types of the PrSs cover the following access scenarios:

- *Case I: partial loss of access*: In this case, only a subset of users lose access to the data streams processed by one or more of the interleaved queries in the shared-operator network. In this case, terminal switches will be in charge of granting access of the query results to the subset of users who did not lose their access privileges.
- *Case II: total loss of access*: In this case, all users lose access to one or more of the interleaved queries in the shared-operator network. Accordingly, instead of operating terminal switches and having possibly unnecessary operators executing, both initial and in-network switches will shut off the isolated branches of operators in the network that are not shared by any other queries. In this case, a considerable amount of unnecessary work will be saved and performance gains will be achieved.

Algorithm 1: PrS execution algorithm

```
   input: Stream
 1 set PrS.AccessCounter = 0;
 2 new SPs_batch arrives;
 3 foreach SP ∈ SPs_batch do
 4 |  if SP.TS < ts_access then
 5 |  |  discard SP;
 6 |  else
 7 |  |  if PrS.Type = "in-network" OR "initial" then
 8 |  |  |  if SP.Sign = '+' AND PrS.QueryID = SP.QID then
 9 |  |  |  |  increment PrS.AccessCounter;
10 |  |  |  if SP.Sign = '-' AND PrS.QueryID = SP.QID AND PrS.AccessCounter
   |  |  |  != 0 then
11 |  |  |  |  decrement PrS.AccessCounter;
12 |  |  |  if PrS.AccessCounter >0 then
13 |  |  |  |  send stream to PrS output;
14 |  |  |  else
15 |  |  |  |  discard stream;
16 |  |  else
17 |  |  |  if PrS.Type = "terminal" then
18 |  |  |  |  if SP.Sign = '+' AND PrS.QueryID = SP.QID then
19 |  |  |  |  |  add SP.SRP.u_id to PrS.U;
20 |  |  |  |  |  discard SP;
21 |  |  |  |  |  send stream to output of SP.SRP.u_id;
22 |  |  |  |  else
23 |  |  |  |  |  if SP.Sign = '-' AND PrS.QueryID = SP.QID then
24 |  |  |  |  |  |  remove SP.SRP.u_id from PrS.U;
25 |  |  |  |  |  |  discard stream;
```

Initial and in-network PrSs are defined as: $PrS =< Type, QueryID, AccessCounter >$, where $Type$ defines whether this is an initial or in-network PrS, $QueryID$ is the query that the PrS is governing access to, and $AccessCounter$ is a counter that changes during the execution of the PrS to identify the number of users who have access to that particular query. The counter will have a value greater than zero in the case of partial loss of access, and will be set to zero in the case of total loss. Terminal switches are defined as: $PrS =< Type, QueryID, U >$, where U is the set of users that have access to the query $QueryID$ governed by this switch.

Algorithm 1 shows the pseudocode for executing the different types of PrSs. The input to each PrS is a data stream with embedded SPs. The PrSs execute this algorithm only when new SPs arrive in the data streams. The algorithm shows how the PrSs will allow or prevent the data streams from flowing through the network based on the output of the authorization predicates indicated by the *Sign* value in each SP.

Both initial and in-network PrSs operate in a similar manner. Each of these PrSs increment their *AccessCounter* whenever an SP with a matching QueryID and a '+' sign is encountered in the input stream and decremented each time a matching SP with a '-' sign shows up. In the case of partial access loss, the *AccessCounter* will be greater than zero (i.e., there is at least one SP in a stream that grants access to any user). In this case, the PrS will be switched *on* and the data stream tuples will flow normally through the network (lines 14 and 15). Note that the assumption made here is that the authorization server will

re-inject SPs for the same user or set of users only if there are changes in the access control permissions for those users.

In the case of total loss of access, the *AccessCounter* will decrement down to zero (i.e., the last user who had access to the query lost that access). Accordingly, the PrS will be switched *off* and the flow of the tuples will halt temporarily (lines 16 and 17) until new SPs show up with positive access signs.

Terminal switches operate slightly different. They multiplex the final query output tuples to the users that have positive access as defined by their SPs (lines 20 - 24). Note that for privacy preservation, the default setup of terminal switches is deny the output to all users and only grant access when explicitly granted by an SP (i.e., they start by an empty set of users U). Similarly, initial and in-network switches have their *AccessCounter* initialized to zeros, which by default will deny any access to the query outputs.

The operation of the PrSs is very similar to the well known notion of counting semaphores that are typically used by many systems to coordinate access to different resources. By looking at the status and counter of each PrS, the DSMSs can collect statistics about how many users in the system are currently allowed access to a certain query output. The PrSs present an effective and simple solution for enforcing access control in shared-operator networks. They allow the on-the-fly adjustment of the network status as changes in access control take place without the need to re-direct the input streams to different networks or the need to restructure the operator network.

5.2 Placement of Privacy Switches

Algorithm 2 shows the pseudo-code for the PrSs placement algorithm. This algorithm extends the query optimizer, i.e., after a query optimizer constructs a shared-operator network strategically embedding the switches in the network.

The input to the algorithm is a shared-operator network pre-computed by the queries optimizer in the DSMSs. The placement of terminal switches is straightforward, at the end of each query output a terminal-switch will be inserted (line 2). In-network switches placement requires some analysis of the network graph. The main idea is to find the operators that are shared by multiple queries. Those shared-operators annotated are the ones annotated as "Common-Prefixes". The in-network switches will be placed along the outgoing edges of the last set of operators that belong to the common-prefixes (line 15). Finally, the initial-switches are placed at the input streams (lines 7 and 8) to apply pre-filtering of the input streams in the case of total access loss.

5.3 Example Execution of Privacy Switches

Figure 6 shows the same shared-operator network as that of the motivating example from Section 4 with the PrSs embedded in the network according to the proposed placement algorithm.

Assume that this shared-operator network is being executed by multiple users who initially have access granted to all three data streams, hence all users can

Algorithm 2: PrSs placement algorithm

input: A shared-operator network $\mathcal{N} = (V, E, L)$
Data: s defines a stack

foreach *query* q_i *output traverse* $\mathcal{N}$ *backwards (DFS-search)* **do**
  insert $PrS = <Type = \text{“}terminal\text{”}, QueryID = q_i, U = null>$ at the output of q_i;
  s.push(v);
  while s *is not empty* **do**
    $v = s$.pop();
    if v *is NOT annotated “Common-prefix”* **then**
      if $\mathcal{N}.adjacentEdges(v)==\phi$ **then**
        insert $PrS =$ $< Type = \text{“}inital\text{”}, QueryID = q_i, AccessCounter = 0 >$ at the input stream;
      else
        foreach *edge from* v *to* $w \in \mathcal{N}.adjacentEdges(v)$ **do**
          s.push(w);
    else
      insert $PrS =$ $< Type = \text{“}in-network\text{”}, QueryID = q_i, AccessCounter = 0 >$ at outgoing edge from v;

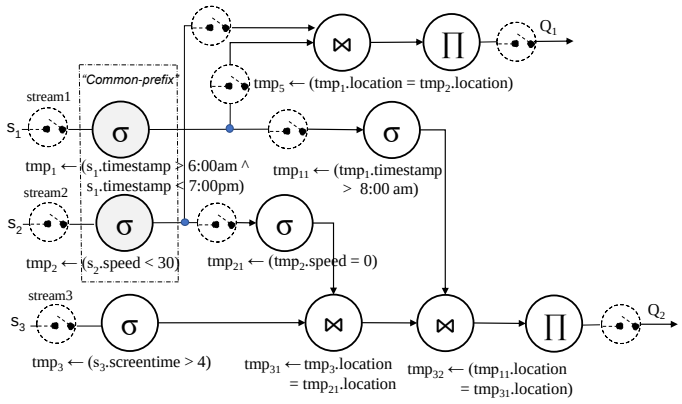


Fig. 6. Example of a privacy-aware shared-operators network

view the output of both queries. From the figure, the dotted box highlights the “Common-prefix” zone of the operators shared by both executing queries. According to the switches placement Algorithm 2, the in-network privacy switches are placed right after those operators. Also at the output of both queries, terminal switches are placed, and at the front of the streams initial PrSs are placed.

In this particular example, if all users accessing $Q2$ lose access to the input streams feeding into this query, the PrSs will shut off six out of the ten operators in the network saving unnecessary processing and bandwidth. When any of these users gain their access back, the PrSs will resume operating all the nodes. This shows that the different switches orchestrated together are capable of enforcing the access control policies in shared-operator networks in a cost-effective way.

6 Evaluation

In the following sections we will present the details of the shared-operator networks simulator we implemented to evaluate the performance of our proposed privacy-aware shared-operator networks, as well as the experimental results.

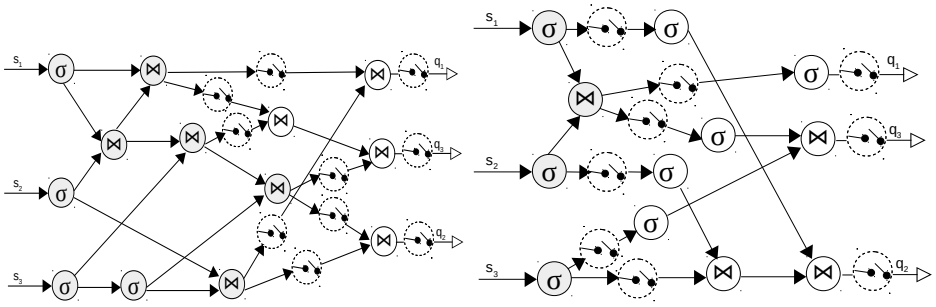
6.1 Configurations and Experimental Setup

Generating synthetic shared-operator networks gave us control over the input parameters and the different scenarios of access control. The simulator takes as input the following parameters: *number of input streams*, *number of interleaved queries*, *number of users executing those queries*, *number of query operators*, and *degree of sharing*. The degree of sharing input parameter indicates the percentage of the query operators that will be included in the "common-prefix" of the network (i.e., how many query operators will be shared across the executing queries). An assumption is made that all shared operators are defined over the same window specifications which are omitted for simplicity of the analysis. Given these parameters the simulator generates shared-operator networks arranged as DAGs.

Some heuristics were enforced to assure the correctness and validity of the generated networks. For example, the "join-push-down" approach of operators was enforced (i.e., `SELECT` and shared `SELECT` nodes appear before `JOIN` and shared `JOIN` nodes). This is a common practice for queries optimizer to execute the `SELECT` operators first to filter out the tuples as early as possible and improve queries processing times. `SELECT` and `PROJECT` operators take one input stream and produce a filtered output stream, while `JOIN` operators take two different input streams and produce a single joined output stream.

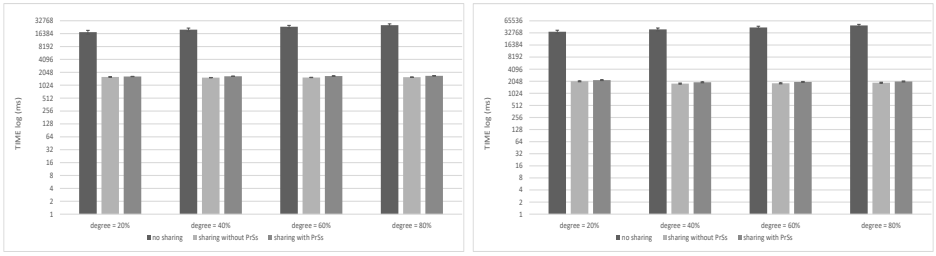
After the random shared-operator networks are generated, our placement Algorithm 2 executes to identify the locations of the PrSs. The number of PrSs is dependent on the topology of the generated network. Finally, the users are randomly assigned to the query outputs of each generated graph. Figure 8 illustrates two randomly generated shared-operator networks by the simulator. Note that the final total number of query operators generated could be slightly higher than the initial input parameter.

In the experiments, the average processing time per tuple for each `SELECT` and `PROJECT` operator in the network was set to 0.1ms, and the average processing time per SP for each PrS to 0.1ms as well. `JOIN` operators average processing time per tuple was set to 0.3ms. Since these simulations are intended to compare and contrast the performance of different operator networks, these values were chosen as rough estimates and defined as constants throughout the execution of the networks. In reality, the processing times of operators would be different from one another. The upper bound on the processing time of each network to process 1000 input tuples was computed. Note that the reported times are upper bound since not every single operator in the network will process all 1000 tuples, as the input tuples get filtered by the `SELECT` operators, fewer tuples will be processed in the network. The arrival rate of SPs was set to be 100 tuples



(a) Degree of sharing = 20% (b) Degree of sharing = 80%

Fig. 7. Samples of randomly generated shared-operator networks[input streams = 3, queries =3, total operators =12]



(a) Execution time for a network with 15 operators (b) Execution time for a network with 25 operators

Fig. 8. Experiment 1 results

(i.e., for every 100 tuples, new SPs will show up in the input streams for each user in the system). The PrSs will process all SPs but will only take action for those SPs that match with the $SP.QID = PrS.QueryID$.

To better understand the security enforcement overheads in different shared-operator networks, the upper bound of the network execution times were compared for the following three cases: i)*no-sharing* – each user is executing a separate operator network for each submitted query (base case), ii)*shared without PrSs* – shared-operator network with only post-filtering applied, iii)*shared with PrSs* – shared-operator network with security enforcement using PrSs.

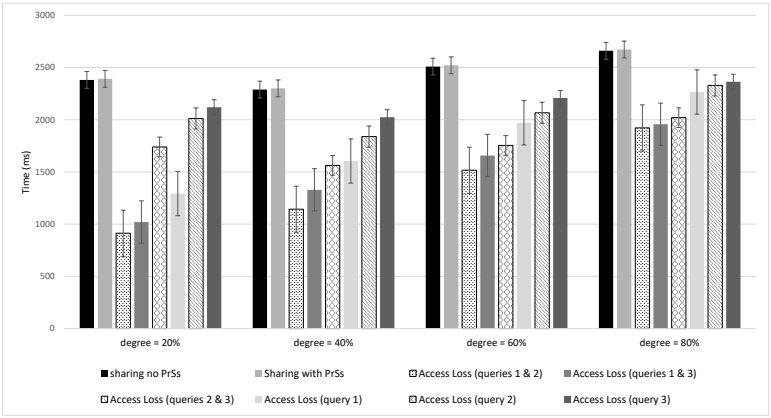


Fig. 9. Network execution time for varying degrees of sharing and access loss patterns

6.2 Experiment 1: Varying Degree of Sharing

These set of experiments attempt to answer the following question: *Q1: what are the security enforcement overheads in the shared-operator networks induced by the PrSs*?

To answer this question, the experiments computed the average processing times of 1000 input tuples for different degrees of sharing. Two different experiment settings were used, once for 3 interleaved queries with 15 operators and another for 5 interleaved queries with 25 operators. For each degree of sharing the network execution times were averaged for 10,000 randomly generated operator networks.

Figures 8a and 8b show the reported execution times (in log_2 ms) of the networks. The figures show that the average execution time of shared-operator network outperforms the non-shared networks with approximately 92%. This validates the benefits of executing shared-operator networks in DSMSs. The experiments show that the PrSs add negligible overheads to the network performance (an average of 0.5% increase in the execution time). This behavior is justified by the fact that PrSs execute infrequently (only when they encounter a new SPs in the streams that match the query id). The experiments also show that both the number of PrSs inserted in the networks by the placement algorithm and the execution times were insensitive to the degree of sharing.

6.3 Experiment 2: Varying Access Control

These experiments were designed to answer the following two questions: *Q2: how much cost savings in the networks can be achieved as users start losing access to queries*? and *Q3: how much does the overlap between queries affect the execution times as users start losing access to queries*?

To answer both questions, the experiments measured the execution times of the networks as users start losing access to query outputs. To cover all cases,

all different possible combinations of access loss were examined. For each degree of sharing examined, the network execution times were averaged for 10,000 randomly generated shared-operator networks. For each generated network, the execution times were measured in the following scenarios: shared network without PrSs and only post-filtering, shared network with PrSs and full access to all queries and all possible combinations of queries' losses. Figure 9 shows the results for networks generated with 3 input streams and 3 interleaved queries with an average total of 20 operators. On average 11 PrSs were inserted in the generated graphs.

From Figure 9, and consistent with the previous experiments, the difference in execution times between shared networks with and without PrSs was minimal. As the queries start losing access, the savings in the execution times were noticeable when compared to the cases of shared with only post-filtering of query outputs and shared with PrSs and full access. The average execution time savings between the shared with PrSs and shared without PrSs was approximately 38% in the case of two out of three queries lose access, and 19% in the case of one query access loss.

Another observation is that PrSs improve the network performance when the queries have lower degrees of sharing (i.e., less common subexpressions). The reason is, with fewer shared-operators, the in-network switches can shut off more isolated (unshared) query operators in the case of total users loss of some query outputs. This behavior of the privacy-aware shared operator networks show that even with low degrees of sharing, the performance benefits would be even bigger when intermittent total access control loss is encountered.

7 Conclusions

In this paper we presented a novel solution for enforcing access control over shared-operator networks in DSMSs. The solution presented allows DSMSs to interleave the execution of multiple overlapping queries shared by multiple users while applying access control restrictions on a per user or group of users basis without disrupting the operation of the shared-operator networks. The solution introduces a new set of operators, Privacy Switches (PrS), that are capable of seamlessly configuring the network to allow or deny access to certain query results complying with the access control policies defined in the system. The experimental evaluations showed that the proposed technique induces minimal overheads on the shared-operator networks while achieving great gains in the network performance in the cases of intermittent access loss to some streams and queries. The technique proved to perform better in the case of total access loss with queries that have less common subexpressions, which shows greater benefits of the privacy-aware shared-operator networks even in lower degrees of sharing. A dynamically configurable shared-operator network saves not only time, but also bandwidth consumption, and several consequential monetary costs associated with configuring and executing shared-operator networks in DSMSs that could possibly be operating in cloud environments.

Acknowledgments

This work was produced while the first author was a PhD student at the University of Pittsburgh. This work was supported in part by the NSF award CNS1253204 and the NIH award U01HL137159. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NSF and NIH.

References

1. Abadi, D.J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: A new model and architecture for data stream management. The VLDB Journal pp. 120–139 (Aug 2003)
2. Anh, D.T.T., Datta, A.: Streamforce: Outsourcing access control enforcement for stream data to the clouds. In: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy. pp. 13–24. CODASPY '14, ACM, New York, NY, USA (2014)
3. Arasu, A., Babu, S., Widom, J.: The cql continuous query language: Semantic foundations and query execution. The VLDB Journal pp. 121–142 (Jun 2006)
4. Cangialosi, F.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The design of the borealis stream processing engine. In: Second Biennial Conference on Innovative Data Systems Research (CIDR '05) (Jan 2005)
5. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**, 28–38 (2015)
6. Carminati, B., Ferrari, E., Tan, K.: Enforcing access control over data streams. In: Proceedings of the 12th ACM Symposium on Access Control Models and Technologies. pp. 21–30. SACMAT '07, ACM, New York, NY, USA (2007)
7. Carminati, B., Ferrari, E., Tan, K.L.: Specifying access control policies on data streams. In: Proceedings of the 12th International Conference on Database Systems for Advanced Applications. pp. 410–421. DASFAA'07, Springer-Verlag, Berlin, Heidelberg (2007)
8. Çetintemel, U., Abadi, D.J., Ahmad, Y., Balakrishnan, H., Balazinska, M., Cherniack, M., Hwang, J., Madden, S., Maskey, A., Rasin, A., Ryvkina, E., Stonebraker, M., Tatbul, N., Xing, Y., Zdonik, S.: The aurora and borealis stream processing engines. In: Data Stream Management - Processing High-Speed Data Streams, pp. 337–359 (2016)
9. Chen, C.M., Agrawal, H., Cochinwala, M., Rosenbluth, D.: Stream query processing for healthcare bio-sensor applications. In: Proceedings. 20th International Conference on Data Engineering. pp. 791–794 (April 2004)
10. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: Niagaracq: A scalable continuous query system for internet databases. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. pp. 379–390. SIGMOD, ACM (2000)
11. Ferraiolo, D.F., Barkley, J.F., Kuhn, D.R.: A role-based access control model and reference implementation within a corporate intranet. ACM Transactions on Information and System Security pp. 34–64 (Feb 1999)

12. Hu, V.C., Kuhn, R., Ferraiolo, D.F., Voas, J.: Attribute-based access control. Computer **48**, 85–88 (Feb 2015)
13. Kim, J.W., Jang, B., Yoo, H.: Privacy-preserving aggregation of personal health data streams. PLOS ONE **13**, 1–15 (11 2018)
14. Lindner, W., Meier, J.: Securing the borealis data stream engine. In: 2006 10th International Database Engineering and Applications Symposium (IDEAS'06). pp. 137–147 (Dec 2006)
15. Liu, X., Buyya, R.: D-storm: Dynamic resource-efficient scheduling of stream processing applications. In: 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS). pp. 485–492 (Dec 2017)
16. Nehme, R.V., Lim, H.S., Bertino, E.: Fence: Continuous access control enforcement in dynamic data stream environments. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy. pp. 243–254. CODASPY '13, ACM, New York, NY, USA (2013)
17. Nehme, R.V., Rundensteiner, E.A., Bertino, E.: A security punctuation framework for enforcing access control on streaming data. In: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering. pp. 406–415. ICDE '08, IEEE Computer Society, Washington, DC, USA (2008). https://doi.org/10.1109/ICDE.2008.4497449, https://doi.org/10.1109/ICDE.2008.4497449
18. Ng, W.S., Wu, H., Wu, W., Xiang, S., Tan, K.L.: Privacy preservation in streaming data collection. In: 2012 IEEE 18th International Conference on Parallel and Distributed Systems. pp. 810–815 (Dec 2012)
19. Sandhu, R.S., Samarati, P.: Access control: Principle and practice. IEEE Communications Magazine **32**(9), 40–48 (Sep 1994)
20. Thoma, C., Lee, A., Labrinidis, A.: Behind enemy lines: Exploring trusted data stream processing on untrusted systems. In: Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy. pp. 243–254. CODASPY '19, ACM, New York, NY, USA (2019)
21. Thoma, C., Lee, A.J., Labrinidis, A.: Polystream: Cryptographically enforced access controls for outsourced data stream processing. In: Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies. pp. 227–238. SACMAT '16, ACM, New York, NY, USA (2016)

Appendix A CORRECTNESS OF PRIVACY SWITCHES

In this appendix we will present proof of correctness of the PrSs operation to ensure that the data streams privacy are enforced at all times. The proof will show that the PrSs will only allow tuples of a particular stream to be accessed by the users that are specified in the SPs, otherwise they will enforce denial-by-default.

Theorem 1 (PrSs Correctness). *During the execution of the PrSs as described by Algorithm 1, for any tuple t, PrS will allow the flow of t only if its preceding SP has a '+' access sign, otherwise PrS will block the flow of t.*

Proof. To prove this claim, the following must be asserted:

(i) terminal PrSs will only allow users specified by the SPs to access the tuples.
(ii) in-network and initial PrSs only allow tuples to flow through the network if there is at least one user that has access granted.

Note that the terminal PrSs are the switches that have the ultimate control of which users can access the tuples of a particular data stream, even if the initial and in-network PrSs allow all the tuples to flow through the network. As such, proving the privacy of a shared-operators network is only dependent on proving that assertion (i) is true. Yet, to prove that a shared-operators network not only denies access to the data streams to those users who are not allowed access, but also ensures that access is granted to those users who are allowed access, then both assertions (i) and (ii) need to be true.

For the base case, assume that $\exists$ authorization policy P that specifies that u_1 can access q_1 and SP is used to encode this policy and is injected in all data streams used in processing q_1. Let $t \in T$, where T is a set of tuples, be a tuple that belongs to the data stream used in processing q_1. There are two cases to be considered to assert both (i) and (ii):

1. SP arrives prior to tuple t,
2. tuple t arrives prior to SP.

Case 1: if $SP.q_1$ is '+', then all initial and in-network PrSs processing this SP will increment their $PrS.AccessCounter$ and allow the following data tuple t and $SP.q_1$ to reach to the terminal switches. Terminal switches will in turn add $SP.SRP.u_{id}$ to the list $PrS.U$ and open up the access channel for this user to allow tuple t to flow into query output.

If $SP.q_1$ is '-', then all initial and in-network PrSs will decrement their counters. If the counter value goes down to zero, this means this user was the last user to have access and the PrSs will not allow the tuples to flow to the output (total loss case). If the counter is greater than zero, then the tuples will flow to the terminal switches. For each terminal switch, if $SP.SRP.u_{id} \in PrS.U$, then this user will be removed from the list, and the access channel to this user will be blocked.
Case 2: tuple t will only flow to the channel assigned to user u_1 iff $u_1 \in PrS.U$. Since users are only added to the PrS users list and given access if an explicit SP with a '+' sign is encountered at time $SP.TS < ts_{access}$, then if t arrives prior to the SP, the denial-by-default will be enforced.

The above cases account for the possible scenarios of a set of tuples T and their equivalent SPs showing up in the data streams. By proving that both assertions (i) and (ii) are true, it is shown that Theorem 1 holds in the base case.

Observe that an argument similar to that used in the base case shows that the same behavior of PrSs would apply to all policies. Furthermore, the inductive hypothesis can be used to prove that the value of the $SP.Sign$ is in charge of activating or deactivating the users channels of the terminal PrSs. As such, only those tuples that are preceded with a positive SP sign will flow to the users specified by the SPs, and Theorem 1 holds for all policies.