Secure Shared Continuous Query Processing

TR-20100916-1

Raman Adaikkalavan and Thomas Perez
Computer and Information Sciences
Indiana University South Bend
1700 Mishawaka Ave
South Bend, Indiana, USA
raman@cs.iusb.edu

September 16, 2010

Abstract

Data stream management systems (DSMSs) process continuous queries over streaming data in real-time adhering to quality of service requirements. The data streams generated from sensors, mobile phones, and other devices are processed by continuous queries to monitor situations and to detect and react to events in (near) real-time. Though data stream management systems are being used in multiple application domains (e.g., stock trading) the need for processing data securely is becoming critical to several stream applications (e.g., patient monitoring, battlefield monitoring). Existing systems that provide access control to maintain data confidentiality use various techniques such as query rewriting, post-processing, and security punctuations. These approaches have several limitations such as modifying query plans, non-shared continuous query processing, checking a tuple multiple times, and introduction of new security operators.

In this paper, we introduce a novel framework that uses three stages (preprocessing, query processing, and post-processing) to enforce access control in DSMSs. We do not modify the query plans nor introduce any new security operators, and check a tuple only once irrespective of the number of active continuous queries. As opposed to existing systems, our approach allows continuous queries to be shared when they have either same or different privileges. In addition, our approach does not affect the DSMS quality of service improvement mechanisms as query plans are not modified. We discuss the prototype implementation using the MavStream DSMS. Finally, we discuss experimental evaluations using the MavStream DSMS to demonstrate the low overhead and feasibility of our approach.

1 Introduction

Data Stream Management Systems (DSMSs) [1, 2, 3, 4, 5, 6] process continuous queries over stream data in real-time. Some of the DSMS applications are: network traffic management [7], health care [8], and network fault management [9]. Quality of Service (QoS) plays a major role in data stream processing and some of the QoS improvements supported by various DSMSs include [6, 10] capacity planning, scheduling strategies, and load shedding.

Several stream applications (e.g., situation monitoring applications such as patient monitoring, airport monitoring, battlefield monitoring, and border security) require DSMSs to process data securely and provide stream data confidentiality. For example, consider the patient monitoring application where a patient's critical condition requires an immediate response. Assume that data streams are generated from continuous monitoring devices (e.g., heart rate monitor) attached to patients (i.e., they can be in their home, or driving a car). The DSMS needs to detect abnormal scenarios in an online fashion and take various actions (e.g., alert hospital emergency room). On the other hand, if the DSMS cannot process the patient data securely, it will lead to violation of data confidentiality and privacy laws (e.g., US Health Insurance Portability and Accountability Act, EU Data Protection Directive). Nevertheless, if the DSMS cannot satisfy the application's quality of service requirements it can even lead to loss of lives.

Access control [11] models and mechanisms such as role-based [12], mandatory, and discretionary, specify and enforce authorization policies (i.e, *who* can access *what*, *when* and *how*), preserving data confidentiality. Existing systems [13, 14, 15] that provide access control in a DSMS modify the underlying DSMS in a substantial way affecting scheduling, capacity planning, and other runtime QoS improvement mechanisms. Also, these systems do not support sharing of queries which in turn reduces the memory footprint and resource usage that are critical to the DSMS to maintain quality of service. The limitations of these approaches are discussed in detail in Section 8. As DSMSs process high-speed data in real-time, access control enforcement should not introduce a lot of overhead. That is, the enforcement should not modify query plans which can lead to the usage of more resources such as memory or processor time, affect query sharing, and affect existing QoS optimizations.

In this paper, we introduce a novel three stage framework to enforce access control in a DSMS. The first stage is the *preprocessing* stage where tuples are checked for access control before entering the query processor. The second is the *query processing* stage where tuples are processed by privileged queries. The third is the *post-processing* stage where tailored access control policies are enforced and results are delivered to query creators. Our framework does not introduce any special security operators. It supports sharing of complete queries (i.e., all operators in the query plan are shared) created by two or more users active in the same role (*user-level sharing*), or active in different roles (*role-level sharing*). We do not discuss sharing queries partially (*system-level sharing*) where only a set of operators in a query plan are shared, and is outside the scope of this paper. We discuss the issues involved in access control enforcement and present our framework. We discuss the enforcement of Role-Based Access Control [12] (RBAC), in this paper, using our framework. We also discuss the prototype implementation and experimental evaluations using the MavStream [3, 5, 6] DSMS to demonstrate the low overhead and feasibility of our approach.

Overview: We discuss DSMS and RBAC in Section 2. Access control enforcement issues are discussed in Section 3. User-level sharing is presented in Section 4, and role-level sharing is presented in Section 5. Prototype implementation is discussed in Section 6, and experimental evaluations are discussed in Section 7. Related work is discussed in Section 8. Conclusions and future work are presented in Section 9.

2 Background

In this section, we briefly discuss data stream management systems (DSMSs) and role-based access control (RBAC).

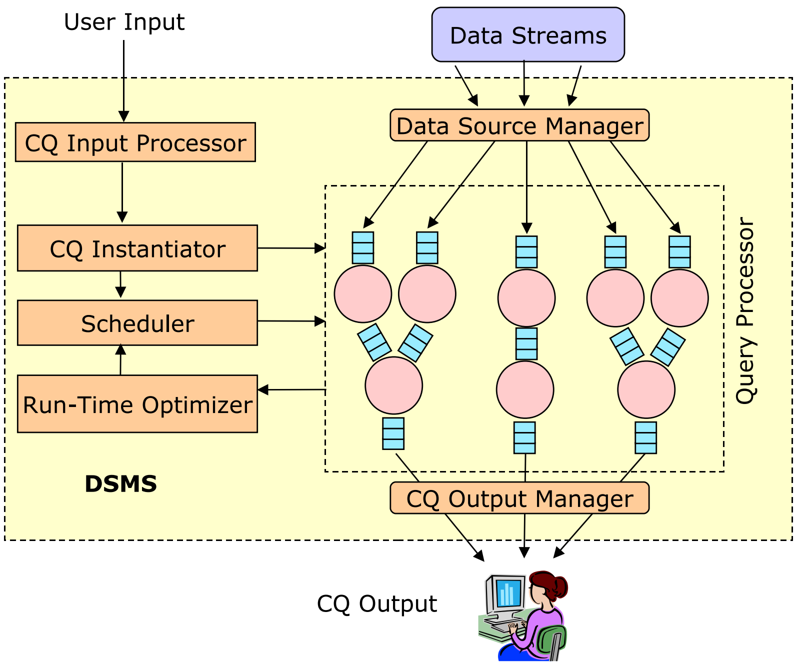


Figure 1: Data Stream Management System

2.1 Data Stream Management System (DSMS)

We have used the MavStream [3, 5, 6] DSMS to enforce role-based access control. A typical DSMS [6, 4, 16, 17, 18] architecture is shown in Figure 1 and is used in the paper.

A Continuous Query (CQ) can be specified using specification languages [19], or as query plans [4]. The CQs defined using specification languages are processed by the input processor, which generates a query plan. Each *query plan* is a directed graph of operators (e.g., Select, Join). Each operator is associated with one or more input *queues*[1] and an output queue. One or more *synposes*[2] [19] are associated with each operator (e.g., Join) that needs to maintain the current state of the tuples for future evaluation of the operator. The generated query plans are then instantiated, and query operators are put in the ready state so that they can be executed.

Based on a scheduling strategy (e.g., round robin, chain [20]), the scheduler picks a query, an operator, or a path, and starts the execution. A query is not scheduled (or executed) when there is no data, and is stopped when the user explicitly stops it. This is in contrast to a relational database system, where a query is executed once over the snapshot of the current data set, and the results are returned. The run-time optimizer monitors the system, and initiates load shedding as and when required. Scheduling strategies and load shedding are part of the QoS improvement mechanisms that minimize resource usage (e.g., queue size) and maximize performance and throughput. In addition, other QoS improvement mechanisms such as static and dynamic approximation techniques [10] are used to control the size of synopses.

All the input tuples arriving to the DSMS from the stream sources are first processed by the Data Source Manager. The number and type of attributes between the input tuples and the stream schema must match. The Data Source Manager enqueues the incoming tuples to input queues of *all* the leaf operators associated with the stream. In the directed graph of operators, the

[1]Queue data structures are used by the operators to propagate tuples. A queue is associated with an input operator and one or more output operators.

[2]Synposes are temporary storage structures used by the operators (e.g., Join) that need to maintain a state. They also act as the sliding window for that operator.

HRStr	BPStr
$t_h^1 - 10:00:00, 1, X12U, 85$	$t_b^1 - 10:00:00, 1, Y23K, 130, 80$
$t_h^2 - 10:00:30, 1, X12U, 84$	$t_b^2 - 10:00:30, 1, Y23K, 130, 80$
$t_h^3 - 10:01:00, 1, X12U, 84$	$t_b^3 - 10:01:00, 1, Y23K, 132, 82$
$t_h^4 - 10:01:30, 1, X12U, 95$	$t_b^4 - 10:01:30, 1, Y23K, 136, 90$
$t_h^5 - 10:02:00, 2, X44C, 71$	$t_b^5 - 10:02:00, 2, Y21B, 120, 75$

Table 1: Data Stream Tuples

data tuples are propagated from the leaf operator to the root operator. Each operator produces a stream (can also be a relation) of tuples. After a processed tuple exits the query plan, the output manager sends it to the query creators (or users).

The five major parts of a DSMS that are affected by access control enforcement are briefly described below. These components will be used to discuss the issues and our approach in the rest of the paper.

- *Query Specification:* Allows users to define queries.
- *Query Plan Generation and Query Sharing:* When an user specifies a CQ, the DSMS generates a query plan for that query. When more than one user creates the same query, the DSMS shares the operators or query plan to minimize computations and resource usage [19].
- *Input:* All the arriving input tuples are enqueued to the leaf operators associated with the stream.
- *Query Processing:* Each operator that is part of the query plan dequeues tuples from the input queues, stores the tuples in the synopses (if required), processes the tuples, and enqueues the result to the output queue. An operator can have more than one processing algorithm. In operators, such as the Join operator, new combined tuples are created.
- *Output:* Each tuple enqueued by the root operator of a query plan is propagated to the user who created the query.

2.2 Role-Based Access Control (RBAC)

Access control models and mechanisms allow subjects to access only authorized objects. There are multiple access control models [11] - discretionary, mandatory, and role-based. *Discretionary* allows subjects to own objects, and grant permissions to other subjects to access those objects. It allows subjects to access objects if the authorized rules are satisfied. *Mandatory* allows subjects to access objects using security axioms based on sensitivity level and categories.

Role-based access control assigns object (e.g., tuple, file) permissions to roles (e.g., doctor, nurse, programmer, system administrator) that can then be assigned to more than one subject (e.g., users). In any given session, a subject can activate one or more assigned roles. After activating a role, whenever a subject creates an object, the active role is associated with that object. Similarly, subjects are allowed to access objects only if the active role(s) have the required permissions.

RBAC allows for better management of permissions, because if a person changes job roles then it is easier to revoke roles from or grant roles to a user, rather than an entire set of permissions for each user. RBAC does not provide a complete solution for all access control issues, but with its rich specification [12], it has shown to be cost effective [21]. ANSI RBAC Standard [12] has four functional components: Core, Hierarchical, Static Separation Of Duty, and Dynamic Separation of Duty. In this paper, we will discuss the enforcement of Core or Flat RBAC, where a user is assigned one or more roles and the user can activate the assigned roles to access objects.

3 Access Control Enforcement Issues

To enforce access control, objects are set with permissions, subjects are granted privileges, and subjects are allowed to process only authorized objects. Assume, file `payments.dat` has the permission (`read to manager role`), and user `Bob` is assigned roles (`programmer`, `manager`). When RBAC is used, `Bob` is allowed to access the file `payments.dat` only when he is active in role `manager`. Similarly, in a DSMS, subjects are the *continuous queries* that process data on behalf of users, and objects are the incoming *data stream* tuples.

In this section, we will first discuss an example from the patient monitoring domain that will be used in the rest of the paper. We then discuss the issues that need to be addressed when enforcing access control.

Example: Patients have mobile medical devices that stream (i.e., send a tuple) their vitals every 30 seconds. We define two data streams, and a continuous query. The schemas for the data streams `HRStr` (heart rate) and `BPStr` (blood pressure) are shown below (timestamp (`ts`), patient id (`patID`), and device id (`devID`)):

```
HRStr (ts, patID, devID, pulseRate)
BPStr (ts, patID, devID, systolic, diastolic)
```

The continuous query CQ1 shown below computes the average pulse rate and blood pressure over a sliding window of 2 minutes. A set of input tuples are shown in Table 1.

```
CQ1: SELECT   HRStr.patId, AVG(pulseRate), AVG(systolic), AVG(diastolic)
     FROM     HRStr [Range 2 Minutes], BPStr [Range 2 Minutes]
     WHERE    HRStr.patID = BPStr.patID
     GROUP BY patID
```

Assume the following users and roles: User `Bob` is active in role `doctor`, and user `Alice` is active in role `administrator`. There are two patients with patId 1 and patId 2, and their access control policies are:

Patient 1 Policy: Allow access to `doctors`

Patient 2 Policy: Allow access to `nurses`

Below, we discuss the enforcement issues in detail.

- The stream schema definition should be modified to include security policies. For example, patient 1 should be allowed to set his/her own security policy (e.g., policy 1) on their own data. The granularity (stream, tuple, attribute) at which the policies are specified, and who (system, data owner) can set the data policies should be analyzed.

- In the above example, `CQ1` should be allowed to process only tuples that authorize `CQ1`. For example, only queries created by users who are doctors should be allowed to access patient 1's data. Thus, to enforce RBAC, a CQ must be associated with roles and these roles can then act as that query's privileges. The query specification component needs to be modified to handle the role association.

- For each CQ specified, the DSMS generates a query plan consisting of operators, queues, and/or synopses. The ways in which roles can be assigned to queries and the granularity (query-level, operator-level) at which roles are assigned needs to be analyzed.

  Multiple users who are active in the same role or different roles can create the same continuous query. When a query is created by users active in the same role, all the instances produce the same result, and we term this *user-level sharing*. When the users are active in different roles then the queries produce different results based on the associated roles, however, they have the same query plan. We term this *role-level sharing*. Let us assume that query $\mathtt{CQ}_1$ is defined by two users active in role $\mathtt{R}_1$ and one user active in role $\mathtt{R}_2$. Since the queries are the same, it is ideal to share the query plan between roles $\mathtt{R}_1$ and $\mathtt{R}_2$. If they are not shared then each tuple will be processed by all the operators in each query separately. In order to support sharing, query plan generation component should be analyzed and modified (if required).

- Each operator within a query plan processes tuples using one or more algorithms. The required access control checks for each tuple can be done at different places (at each operator, within the query, or only once for a tuple), and each has different costs associated with it. This plays a major role as the access control checks must be performed for each tuple and each query. In the above example, tuples from both the streams can be sent to `CQ1` irrespective of the roles associated with the query. Tuples can be dropped by enforcing access control using a special filtering leaf operator. But this can be expensive as all tuples are processed by all the associated queries. In addition, this requires all the unauthorized queries to receive and drop unauthorized tuples.

  This is further complicated with role-level sharing. For example, when a query is shared between different roles (e.g., doctor and nurse), the Join operator needs to combine tuples based on the roles in addition to the join condition. Moreover, the storage of tuples in the synopses by Join operator must also be analyzed.

- When a query outputs a tuple it must be sent to the user who created the query. With role-level sharing, the query can emit tuples with different roles, and the tuples must be sent to appropriate users. Thus, output component needs to be modified to handle access control and query sharing.

4 Access Control Enforcement Framework: User-Level Sharing

In this and Section 5, we will discuss solutions for all the issues raised in Section 3. We discuss user-level sharing in this section, and role-level sharing in Section 5.

Our proposed access control enforcement framework is shown in Figure 2, and is used in the below discussions. All the modified components, in comparison to Figure 1, are shown using boxes with white background and *italics* text.

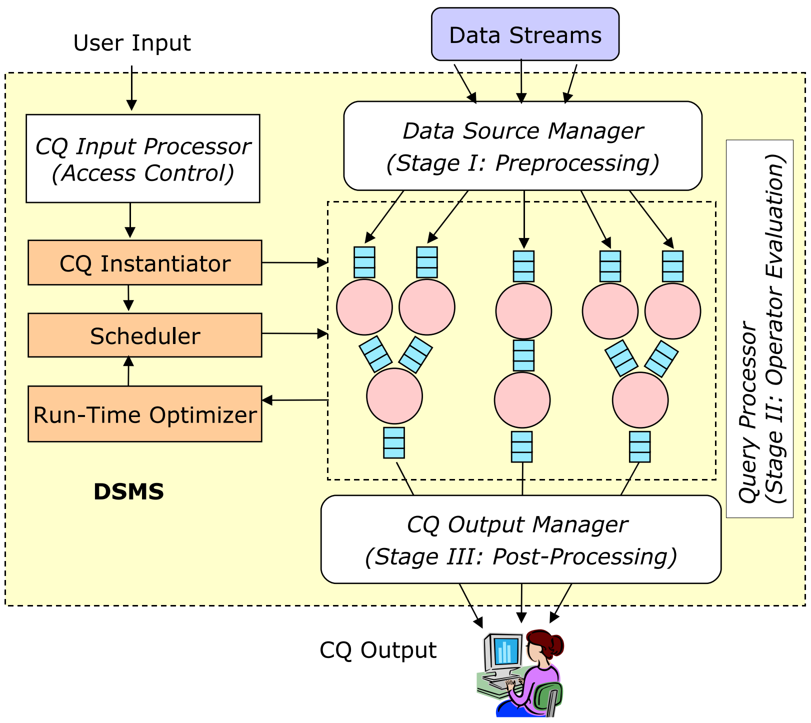

User Input
Data Streams
CQ Input Processor (Access Control)
Data Source Manager (Stage I: Preprocessing)
CQ Instantiator
Scheduler
Run-Time Optimizer
DSMS
Query Processor (Stage II: Operator Evaluation)
CQ Output Manager (Stage III: Post-Processing)
CQ Output

Figure 2: Access Control Enforcement Framework

4.1 Query Specification and Plan Generation

The specification of `CQ1` shown in Section 3 is modified as shown below. The `AS` clause associates the role doctor with `CQ1`. This authorizes the query to access any tuple that allow role doctor. On the other hand, the system should verify that the user submitting the query has the role doctor assigned and activated. With systems where users can send a query plan object, the above approach does not work. Below, we discuss how we associate roles to CQs.

```
CQ1: AS       doctor
     SELECT   HRStr.patId, AVG(pulseRate), AVG(systolic), AVG(diastolic)
     FROM     HRStr [Range 2 Minutes], BPStr [Range 2 Minutes]
     WHERE    HRStr.patID = BPStr.patID
     GROUP BY patID
```

Whenever a user creates a CQ, an active role of the user is associated with the CQ. For example, if $\mathtt{CQ}_1$ is created by a user active in role $\mathtt{R}_1$, then $\mathtt{CQ}_1$ is associated with role $\mathtt{R}_1$. This allows the query $\mathtt{CQ}_1$ to process data streams, tuples or attributes that permit role $\mathtt{R}_1$. On the other hand, situations where a user is active in more than one role are handled by asking the user to choose one or more roles. We have added security catalogs to store the relationship between the users, roles, and CQs.

In order to support sharing we assume that the underlying DSMS's *query plan generator* can identify two queries that are same [19], while generating the query plans. Once the system identifies, we use the user-role-query catalog shown in Figure 4 (discussed in Section 4.3) to associate roles and handle sharing.

4.2 Data Stream Input

Different approaches can be used to specify which roles have access to a particular stream, tuple or attribute [13, 14, 15, 22, 23].

1. Access control policies can be pre-determined (e.g., based on the stream source). Each input device can be associated with roles that can access the stream. This approach will be useful when data providers use devices from the service providers. For example, HRStr defined in Section 3 can be streamed from devices attached to inpatients.

2. Access control policies can be embedded within a tuple using a security attribute. It can also be streamed together with each tuple using meta tuples. Each tuple in the data stream can be associated with a role or set of roles specifying who can view this tuple or its attributes. This approach is appropriate when data providers need to control their data. For example, patients transmitting their data can control who can access their data. But this approach requires data providers or their devices to have the knowledge of a particular organization's roles.

   DSMS applications can have many different data providers and each provider can define access permissions for their data using security attributes. We assume that there is some mechanism available to convert provider's permissions into a set of roles (or other appropriate structure). We would also like to point to ontologies as a solution to this problem, but that is outside the scope of this paper.

In this paper, we assume that each tuple contains a security attribute, which includes a set of roles that define permissions for that *tuple*. Multiple roles can be assigned with both conjunction ($\wedge$) and disjunction ($\vee$) of roles. For example, stream HRStr and tuples $\mathtt{t}_\mathtt{h}^1$ and $\mathtt{t}_\mathtt{h}^5$ (from Table 1) are modified as shown below. The tuple $\mathtt{t}_\mathtt{h}^1$ allows access to role R1, and tuple $\mathtt{t}_\mathtt{h}^5$ allows users who are active in both roles R1 and R2.

$\mathtt{HRStr(ts, patID, devID, pulseRate, roles)}$
$\mathtt{t_h}^1 : (\mathtt{10 : 00 : 00, 1, X12U, 85, R1})$
$\mathtt{t_h}^5 : (\mathtt{10 : 02 : 00, 2, X44C, 71, R1} \wedge \mathtt{R2})$

4.3 Stage I: Preprocessing

Figure 3(a) has Join and Select queries. The role-to-query mappings are shown in Figure 4. Query CQ1 is shared by users U1 and U4 active in role R1. Query CQ2 is associated to user U2 active in role R2. Assume that both CQ1 and CQ2 queries are exactly the same, but submitted by users active in different roles. Since we discuss only user-level sharing, in this section, there are two instances of the same query plan. In Figure 3(a), all tuples from stream HR (or HRStr from Section 3) are sent to the queues associated with $\sigma 1$, $\sigma 3$, and $\sigma 5$, when there is no access control. Similarly, tuples from stream BP (or BPStr) are sent to $\sigma 2$ and $\sigma 4$.

When access control needs to be enforced, queries should be allowed to process those tuples that they have permissions for. As an example, $\sigma 5$ should process only tuples from stream HR that allow role R2. This should be carried out with minimal overhead and without affecting QoS optimizations. For example, introducing special filter operators to perform access control checks, requires modification to the existing query plans. This affects the capacity planning and scheduling of operators and CQs. Below, we discuss our approach that enforces access control in DSMSs.

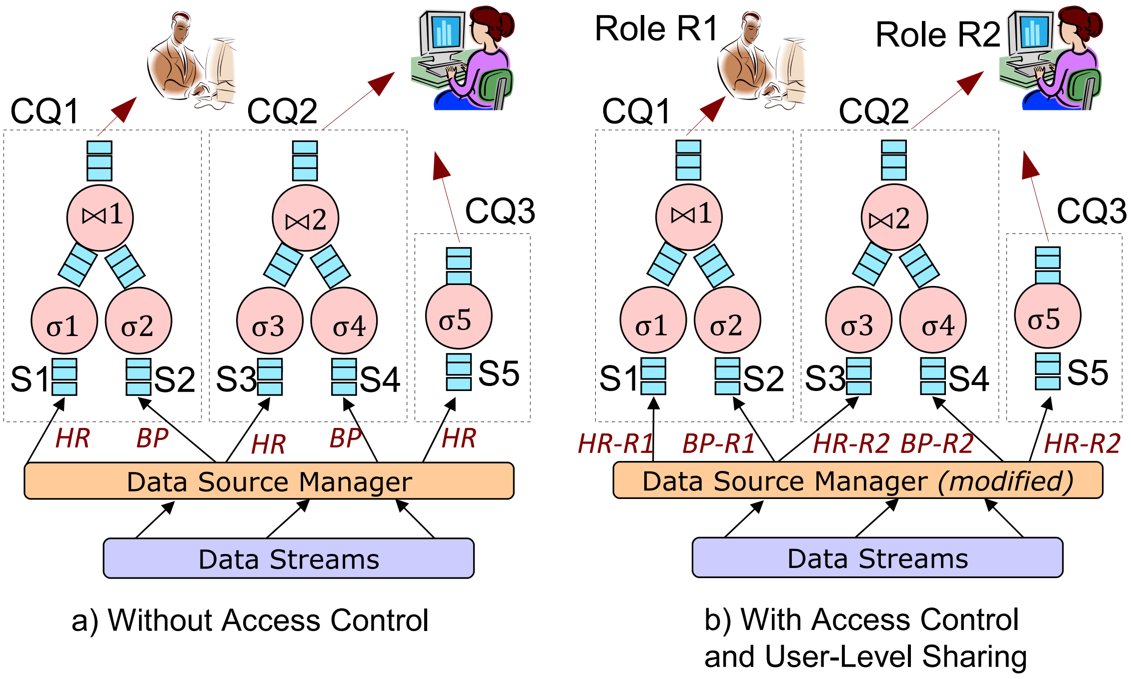


Figure 3: Access Control Enforcement with User-Level Query Sharing

Users	Role	Queries
U1, U4	R1	CQ1 ($\bowtie$1,σ1,σ2)
U2	R2	CQ2 ($\bowtie$2,σ3,σ4)
U3	R2	CQ3 (σ5)

Figure 4: User-Role-Query Catalog

Continuous queries and incoming tuples have many-to-many relationship. Assume that `m` CQs are associated with stream `HRStr`, `n` data items arrive each second via `HRStr`, and `k` (where $\texttt{k} \leq \texttt{m}$) CQs have the privileges to process each incoming tuple.

To enforce access control, either the set of `k` authorized CQs needs to be determined for each incoming data tuple, or each tuple should be sent to all CQs and unauthorized tuples should be filtered by the CQs. This is the most important step, as it enforces access control in two different ways: *"send tuples to authorized queries only"* or *"send all tuples and let the queries filter"*. The former is more advantageous than the later as the access check is performed only *once* per tuple. In addition, with the latter approach, if each operator within a query checks for access permissions then the number of checks will increase according to the total number of operators in the system.

Our framework follows the first approach. If authorized CQs can be determined and tuples can be propagated from the data source manager to only authorized leaf nodes, as shown in Figure 3(b), there is no need for special filter operators at each leaf node. This protects tuples from underprivileged CQs and operators, and reduces resource usage. For example, only tuples with role `R1` from streams `HR` and `BP` should be propagated to operators $\sigma 1$ and $\sigma 2$, respectively. It is also critical that this access check operation, in the data source manager to determine which queries can access the incoming tuple, should be carried out only once for each tuple.

In our framework, we have created a input routing structure to maintain query and role asso-

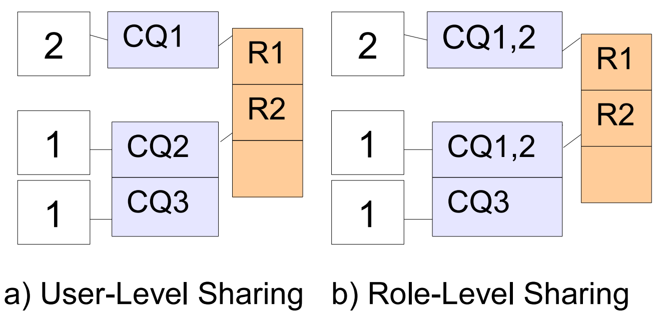


Figure 5: Preprocessing: Input Routing

ciations. The input routing structure design supports efficient insertions, modifications, deletions, and retrievals. This is critical as the data source manager determines authorized CQs for every arriving tuple using this structure. In addition, the input processor has to update the structure every time a new CQ is created. The main difficulty in designing the structure arises due to the many-to-many relationship between roles, CQs, and users

We have designed and developed a *input routing* structure shown in Figure 5(a), for storing and maintaining role-to-query mappings and to support user-level sharing. The routing structure is a hash of a hash set. The first hash's key is the role and the value is the set of associated queries. The value for the second hash is the count of active users who require results from that query. For example, when tuple $\mathtt{t}_\mathtt{h}^\mathtt{1}$ (from Section 4.2) with role `R1` arrives, the data source manager retrieves all the queries that are mapped to role `R1` using the routing structure in Figure 5(a). It retrieves query `CQ1`, and enqueues the tuple $\mathtt{t}_\mathtt{h}^\mathtt{1}$ to the input queue of operator $\sigma 1$ with role `R1` as its permission.

This design is efficient, because there is a role that maps to a set, and then each key in the hash set is quickly retrieved. When a role is associated with a query, the count is incremented for each user that is executing the query, and has the said role activated. If a user deactivates the role or stops the CQ, the count is decremented. When a count is zero, the query can be disabled.

Thus, this stage supports user-level sharing and enforces access control by determining the set of authorized CQs for each incoming tuple and by propagating the tuples to the authorized CQs.

4.4 Stage II: Query Processing

All the queues and sliding windows associated with operators store tuples with only one role due to the user-level sharing. For example, input queue S1 shown in Figure 3 stores tuples with role `R1`. Since tuples are enqueued to only authorized CQs, all the other operators in that CQ can process the incoming tuple without any further checking of role permissions. Thus, if tuples can be propagated from the data source manager to only authorized leaf nodes as shown in Figure 3(b), no additional checks are required to propagate this tuple to the internal nodes, and finally to the authorized user. This is due to the fact that there is only user-level sharing, and the entire query plan has the same role.

The preprocessing stage allows the DSMS to enforce access control prior to the propagation of a tuple to leaf nodes. It moves access control enforcement outside the query plan and, therefore, outside the query processor. As there is no modification to the query plan it neither modifies the query operator semantics nor affects query processing. When compared to existing approaches, our approach performs the security comparisons only once per tuple.

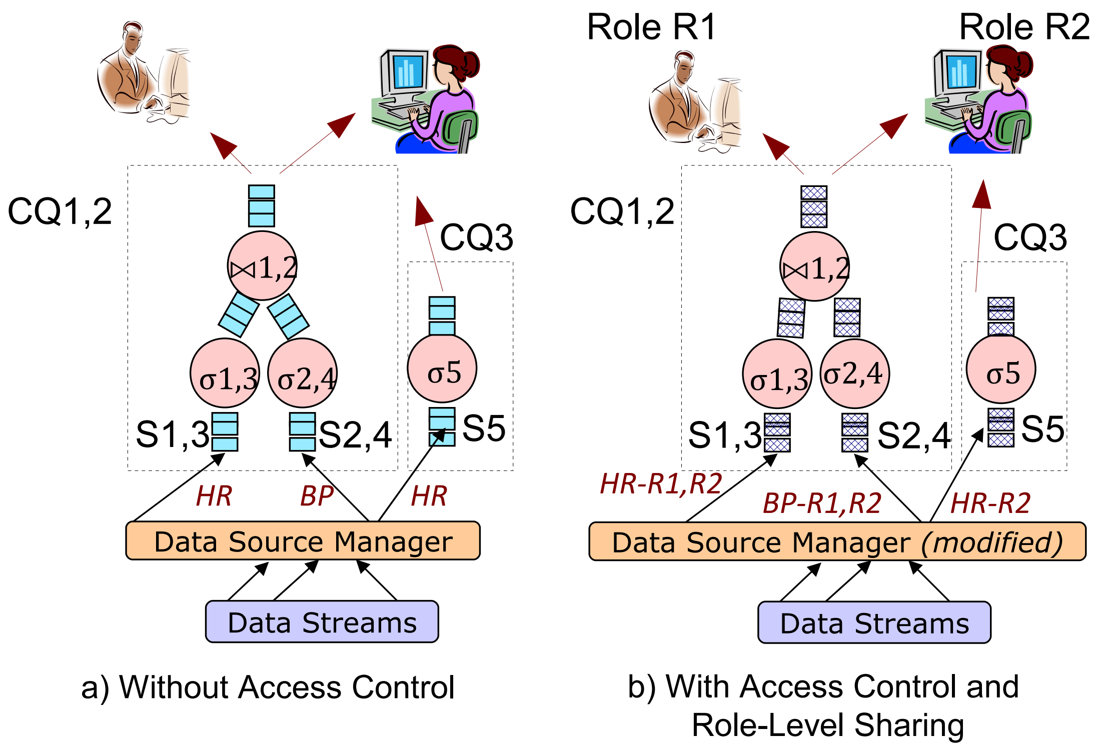


Figure 6: Access Control Enforcement with Role-Level Query Sharing

4.5 Stage III: Post-Processing

The root operator of each CQ enqueues the final output tuple to its output queue. Once enqueued these tuples are handled by the post-processing stage. In this stage, the tuples are sent to the users who have created the queries. Similar to the role-query catalog discussed in the preprocessing stage, we use a query-role-user catalog to find the users or query creators. For example, when the operator $\bowtie 1$ enqueues a tuple to the output queue, the post-processor dequeues the tuple and determines that it should be sent to users `U1` and `U4` (see Figure 4). Since each query has only one role, the roles need not be checked in this stage.

5 Access Control Enforcement Framework: Role-Level Sharing

In this section, we discuss RBAC enforcement when CQs are shared between users with different roles. When role-level sharing needs to be supported, all the query operators that are part of the query plan should be able to handle the tuples with one or more roles. Below, we discuss all the components except query specification and data stream input, which were discussed in Section 4, as they handle role-level sharing without any further modification.

Figure 6 illustrates role-level sharing of queries `CQ1` and `CQ2` from Figure 3. As shown, operators $\sigma 1$ and $\sigma 2$ are combined to form the operator $\sigma 1, 3$[3]. As discussed earlier, we assume the underlying DSMS's query plan generator supports and provides this sharing.

[3]We have used ',' to illustrate that two queries are combined.

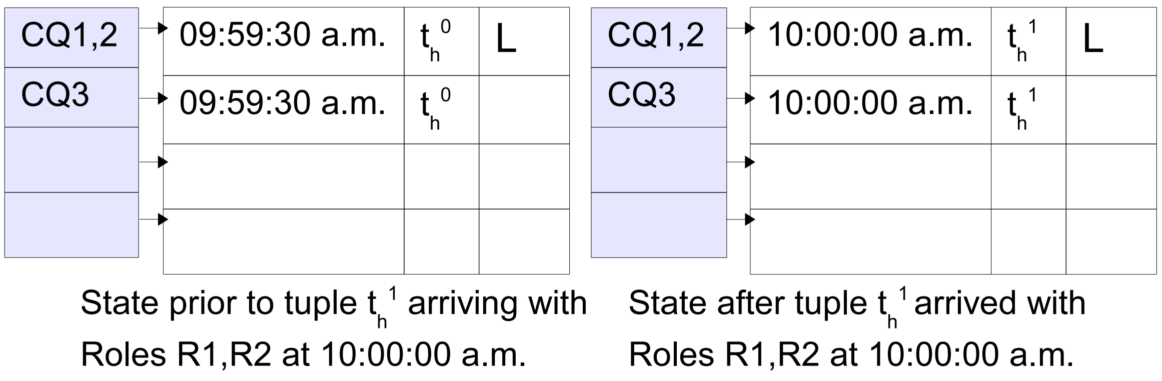

CQ1,2
CQ3
09:59:30 a.m.
t_h^0
L
09:59:30 a.m.
t_h^0
CQ1,2
CQ3
10:00:00 a.m.
t_h^1
L
10:00:00 a.m.
t_h^1
State prior to tuple t_h^1 arriving with Roles R1,R2 at 10:00:00 a.m.
State after tuple t_h^1 arrived with Roles R1,R2 at 10:00:00 a.m.

Figure 7: Tuple-Query Timestamp Cache

5.1 Stage I: Preprocessing

Even with role-level sharing, the data source manager can propagate tuples to appropriate CQs using the techniques discussed in Section 4.3. For example, consider Figure 6(b) where tuples from `HRStr` with roles `R1` or `R2` are propagated to operator σ1,3. On the other hand, the same tuple cannot be sent to the same query more than once, even when multiple roles satisfy the access control checks. This is possible when queries are shared by different roles, as each stream tuple can authorize multiple roles. The newly created *tuple-query timestamp cache* shown in Figure 7 is used by the data source manager to prevent duplicate propagation. This structure and Figure 5(b) are used to retrieve the set of authorized CQs for a tuple.

The state before the arrival of tuple $\mathtt{t}_\mathtt{h}^1$ is shown in the left side in Figure 7. Assume that $\mathtt{t}_\mathtt{h}^1$ enters the DSMS at 10:00:00 a.m. with roles `R1` and `R2`. The input routing structure from Figure 5(b) is used to determine the authorized queries. Since $\mathtt{t}_\mathtt{h}^1$ allows access to `R1` and `R2`, first `R1` is processed. This retrieves `CQ1,2` from Figure 5(b). Now, the cache shown in Figure 7 is accessed with `CQ1,2` as the key. Since the timestamp stored there is less than the $\mathtt{t}_\mathtt{h}^1$'s timestamp (i.e., this tuple has not been propagated earlier), the cache is updated as shown in the right side in Figure 7. Now, `R2` is taken for processing. When the input structure is invoked with `R2` as the key, it retrieves `CQ1,2` and `CQ3`. When the timestamp cache is invoked with `CQ1,2` as the key, the stored timestamp is same as the tuple's timestamp and the tuple id also matches. When `CQ3` is used, the timestamp on the left side is less than the tuple's timestamp and the cache is updated as shown on the right side. Since all the roles authorized by tuple $\mathtt{t}_\mathtt{h}^1$ have been processed, it is enqueued to left leaf operator of `CQ1,2` with roles `R1,R2` and `CQ2` with role `R2`, using the cache.

Thus, in our framework, access control is enforced before a tuple enters the query processing stage using both the routing structure and timestamp cache.

5.2 Stage II: Query Processing

In order to support role-level sharing we have not modified any of the operators except Join ($\bowtie$) and Aggregate. Below, we discuss the modifications made to the Join ($\bowtie$) operator processing.

5.2.1 Join ($\bowtie$) Query Operator

Assume a sliding window[4] of size `one tuple`, and the following tuples (from Table 1):

[4]Sliding windows allow the blocking operators such as Join to produce continuous output.

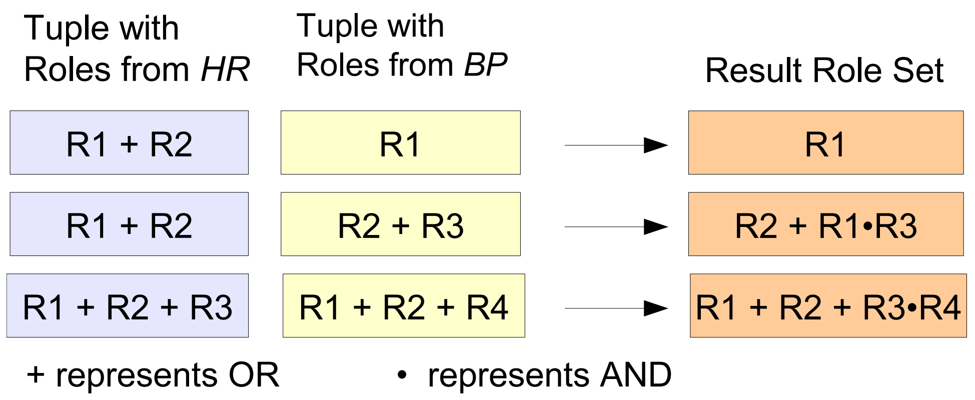


Figure 8: Reduction of Joined Security Attributes

$\mathtt{t}_{\mathtt{h}}^{1}$ *(*R1*, 10:00:00am)* & $\mathtt{t}_{\mathtt{h}}^{2}$ *(*R2*, 10:00:30am)*
$\mathtt{t}_{\mathtt{b}}^{1}$ *(*R2*, 10:00:00am)*, $\mathtt{t}_{\mathtt{b}}^{2}$ *(*R1*, 10:00:30am)*, & $\mathtt{t}_{\mathtt{b}}^{3}$ *(*R2*, 10:01:00am)*

When $\mathtt{t}_{\mathtt{b}}^{1}$ arrives, it is propagated to σ2,4 shown in Figure 6(b) and then to right synopsis of $\bowtie$1,2. Since the sliding window size is one tuple, when $\mathtt{t}_{\mathtt{b}}^{2}$ arrives, *should it replace* $\mathtt{t}_{\mathtt{b}}^{1}$*?* Both these tuples have two different roles, and replacing one with the other can lead to *unintended* query results. We address this, in our framework, by partitioning the sliding window [19] based on the roles. A tuple with multiple roles resides in multiple partitions. For example, sliding window (synopses) attached to the $\bowtie$1,2 will have two partitions with roles R1 and R2. An arriving input tuple with roles R1 and R2 will go to both the partitions R1 and R2. Thus, whenever a tuple arrives it replaces tuples that have the same permission (i.e., in the same partition). The size of the sliding window can be maintained for each partition or for the entire set of tuples.

On the other hand, when $\mathtt{t}_{\mathtt{h}}^{1}$ arrives, it is enqueued to σ1,3 and finally to the left synopses attached to node $\bowtie$1,2, shown in Figure 6(b). When $\mathtt{t}_{\mathtt{b}}^{1}$ arrives, it is propagated to σ2,4, and then to right synopsis of $\bowtie$1,2. Though $\bowtie$1,2 has tuples from both the sides, it cannot still join them, as both tuples have different permissions.

In our framework, we introduce two different approaches to join tuples: 1) *Exact Match*: Wait till all the tuples with matching permissions arrive in appropriate partitions, 2) *Cumulative*: Join the existing tuples with cumulative permissions.

In the *exact match approach*, when tuple $\mathtt{t}_{\mathtt{h}}^{1}$ arrives it is propagated to σ1,3. After processing, σ1,3 enqueues it to $\bowtie$1,2 and is finally placed in the role R1 partition within the left synopses. When $\mathtt{t}_{\mathtt{b}}^{1}$ arrives it is propagated to σ2,4, then to $\bowtie$1,2 and is placed in the R2 partition of the right synopses. Since these two tuples cannot match, the Join operator does not combine them. When $\mathtt{t}_{\mathtt{b}}^{2}$ arrives, it is propagated to σ2,4, then to $\bowtie$1,2, and is placed in the R1 partition. At this point, there is one tuple in the left side synopsis and two in the right side. Since there are two tuples that can match, $\mathtt{t}_{\mathtt{h}}^{1}$ and $\mathtt{t}_{\mathtt{b}}^{2}$ are joined. If an output tuple is produced it has R1 as its permissions, and is enqueued to its output queue. This approach allows sharing of queries with different permissions using partitioned windows and matching same roles.

The *cumulative approach* joins tuples regardless of the roles in the synopses. The output tuple created by the Join operator will contain the cumulative roles as its permission set. We create cumulative permissions using the Redundancy Law of Boolean Algebra. In the above example, when all the join conditions are met, cumulative approach will create two tuples ($\mathtt{t}_{\mathtt{h}}^{1}$ and $\mathtt{t}_{\mathtt{b}}^{1}$ with a cumulative permission Roles R1 AND R2) & ($\mathtt{t}_{\mathtt{h}}^{1}$ and $\mathtt{t}_{\mathtt{b}}^{2}$ with Role R1), as opposed to the exact match approach that creates only one tuple ($\mathtt{t}_{\mathtt{h}}^{1}$ and $\mathtt{t}_{\mathtt{b}}^{2}$ with Role R1). The tuple created with the cumulative permission can only be accessed by users who are authorized to all the included roles. This approach can also combine tuples without partitioned windows. See Figure 8 for more examples.

Both these approaches allow role-level sharing of queries and at the same time join tuples without leaking or demoting any tuple permissions. When the size of the sliding window is assumed to be ∞, the tuples produced by the cumulative approach subsumes the tuples produced by the exact match approach. The permission set created by cumulative approach is exactly same as the other approach or more restrictive. This is because, the cumulative approach joins tuples with matching and non-matching permissions, whereas the exact match joins tuples with matching permissions only.

5.3 Stage III: Post-Processing

The post-processing stage discussed in Section 4.5 is modified to handle role-level sharing. In place of checking the users that are associated with queries, we first check the roles associated with queries and then the users. Thus, a tuple exiting the root operator is processed for each role that is part of the tuple permission set.

6 Prototype Implementation

We have modified the MavStream [3] DSMS to support RBAC. MavStream is developed using Java programming language. In MavStream, the Data Source Manager contains a Feeder that generates tuples from a file. After the tuple is generated, it is enqueued to the queue that is associated to a leaf operator of a query plan. The final operator, in the query plan, has an output queue that is then used by the post-processing stage. We have created catalogs to store and maintain security related data. We have modified the input processor to handle security specifications, storing/updating the catalogs, and to support user-level and role-level sharing. We have modified the data source manager so that it will only enqueue privileged tuples to the input queue of the leaf operators (see Algorithm 1 for user-level sharing). The Join operator algorithms have been modified to handle sharing (see Algorithm 2 for cumulative approach role-level sharing). Since the MavStream system does not support partitioned sliding windows, we have implemented the cumulative approach discussed in Section 5. Finally, we have modified the CQ output manager to route the tuples to the authorized users.

6.1 Correctness

User-Level Sharing: Each tuple is filtered by the preprocessing stage and is enqueued to only authorized CQs. Each tuple contains exactly the role that authorizes the CQs. Since each CQ is only shared between users and not roles, it can process only tuples with a particular role. The post-processing stage sends the resulting tuples to users with roles specified in the tuple. The access control is enforced before a tuple enters query processing. Thus, no tuple contains more than one role and no tuple can be processed by underprivileged queries in our framework, guaranteeing correctness.

Role-Level Sharing: Tuples can contain more than one role if the authorized CQs are shared between roles. This does not affect other operators except Join and Aggregate. Both the approaches used by the modified Join operator creates tuples with exact matching roles or cumulative roles. Though the permission set is modified when non-matching roles are combined in

Algorithm 1: User-Level Sharing - Input Routing in Stage I

```
Require: A data tuple $t_i$ with a role set R

PROCEDURE ROUTE_TUPLE($t_i$)
queries = new Set()
for each $r \in R$ do
    queries.AddAll(GetPrivilegedQueriesSet(r))
end
for each $q \in$ queries do
    b = GetAssociatedBuffer(q)
    if b.HasNotAlreadyBeenEnqueued(t) then
        q.Enqueue(t)
    end
end
```

Algorithm 2: Role-Level Sharing - Simple Join using Cumulative Approach

```
/* $t_i$ can be recognized as coming from the left or right branch of the
   operator tree                                                          */
Require: A data tuple $t_i$ with a role set R

PROCEDURE Join($t_i$)
\\ dequeue from input queues
if $t_i$ is the left tuple then
    \\ add $t_i$ to appropriate partitions in Synopsis.Left
    for each $t \in$ Synopsis.Right do
        \\ Check Join Conditions
        \\ If True, Create a new combined tuple with cumulative permissions
    end
end
if $t_i$ is the right tuple then
    \\ add $t_i$ to appropriate partitions in Synopsis.Right
    for each $t \in$ Synopsis.Left do
        \\ Check Join Conditions
        \\ If True, Create a new combined tuple with cumulative permissions
    end
end
```

the cumulative approach, it only further restricts the tuple usage, and does not leak information. In other words, the cumulative approach subsumes the exact match approach. Thus, a tuple that exits the preprocessing stage has the same or restricted set of permissions when it is output from the root operator, guaranteeing correctness.

7 Experimental Evaluations

In this section, we discuss the experimental setup and evaluations.

7.1 Setup

For experimental evaluations, we ran the MavStream system on a machine with the Linux Fedora 10 64-bit Operating System, Intel Core2 Duo 2.0GHz processor, and 4GB of RAM. The datasets were obtained from the MavHome project [24]. Each test was executed three times for the evaluations. Standard deviation for all the tests was less than a second. The experiments used two input streams (each stream with 500K to 1 Million Tuples), a query with Join ($\bowtie$) and Project (Π) operators, Round Robin Priority scheduling strategy, and no load shedding.

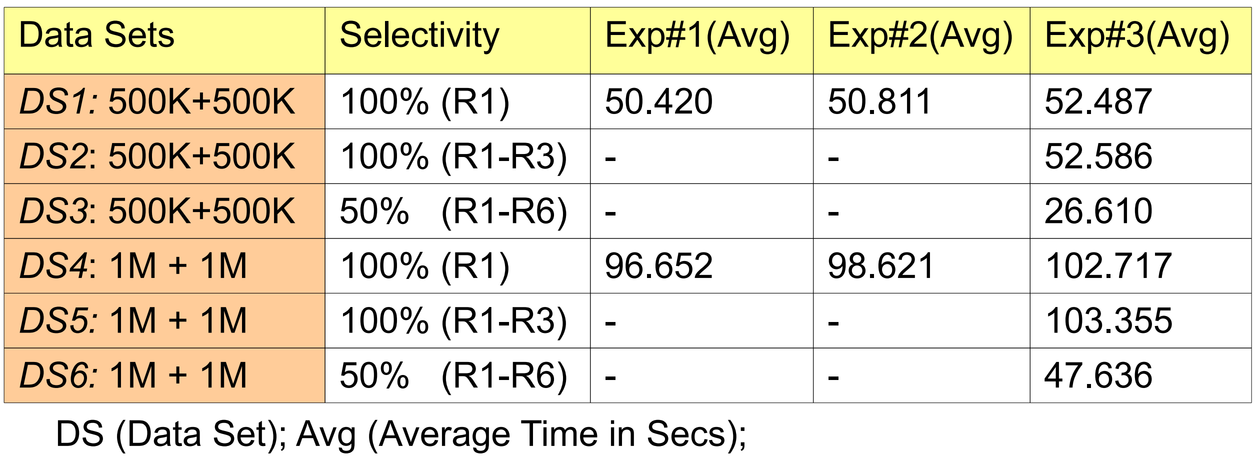
- Datasets `DS1` and `DS4` had tuples with only role `R1` in each stream. The selectivity of the input routing was 100%.
- Datasets `DS2` and `DS5` had tuples with roles `R1`, `R2`, and `R3`. The selectivity of the input routing was at 100%. Five users were active: three in role `R1`, two in `R2`, and two in `R3`, and all of share the same query.
- Datasets `DS3` and `DS6` had a uniform random distribution of six roles: `R1`, `R2`, `R3`, `R4`, `R5` and `R6`. The selectivity of the input routing was at 50%, as the same five users and three roles were used (i.e., tuples with roles `R4`, `R5` and `R6` were dropped at the data source manager).

Results can be viewed in Figure 9. Each experiment builds on the previous to show costs.

- Experiment 1 (*Exp#1*): We captured the current system runtime as a control, without using any access control using data sets DS1 and DS4. Other data sets were not used as they involve access control.
- Experiment 2 (*Exp#2*): We ran the DSMS with user-level sharing enabled using data sets DS1 and DS4. This included all the three stages.
- Experiment 3 (*Exp#3*): We ran the DSMS with role-level sharing enabled. This included all the three stages with the modified Join operator algorithm based on the cumulative approach.

7.2 Analysis

No Access Control: Exp#1 was conducted using datasets DS1 (1 Million Tuples) and DS4 (2 Million Tuples). The system took 50.420 seconds to process DS1 and 96.652 seconds to process DS4.



Data Sets	Selectivity	Exp#1(Avg)	Exp#2(Avg)	Exp#3(Avg)
DS1: 500K+500K	100% (R1)	50.420	50.811	52.487
DS2: 500K+500K	100% (R1-R3)	-	-	52.586
DS3: 500K+500K	50% (R1-R6)	-	-	26.610
DS4: 1M + 1M	100% (R1)	96.652	98.621	102.717
DS5: 1M + 1M	100% (R1-R3)	-	-	103.355
DS6: 1M + 1M	50% (R1-R6)	-	-	47.636

DS (Data Set); Avg (Average Time in Secs);

Figure 9: Experimental Results

User-Level Sharing : Exp#2 evaluated user-level sharing. Three users were active in role R1, and issued the same query. The system took 50.811 seconds to process DS1 and 96.652 seconds to process DS4. As shown in Figure 9, overhead due to our access control enforcement (Exp#2) when compared to no access control (Exp#1) is 0.7% for dataset DS1 and 2% for dataset DS4. If there was no user-level sharing, the system would have executed 3 instances of the query and the total time should have been 3 times the current time, as the input routing selectivity was at 100%. With 'n' users, it will be 'n' times the current runtime.

Role-Level Sharing : We evaluated the overhead caused by role-level sharing using Exp#3. With 100% selectivity, Exp#3 took 52.586 seconds for dataset DS2 and 103.355 seconds for DS5. When comparing Exp#3 on DS2 and Exp#1 on DS1 (no access control), it is an overhead of approximately 4%. When comparing Exp#3 on DS5 and Exp#1 on DS4, the overhead is approximately 6.9%. If there were no sharing, then there should have been seven instances of the same query running, with a total runtime that is seven times rather than 6.9%.

Number of Tuples Processed: The number of tuples processed by the system is also reduced based on the selectivity of the input routing. The total number of tuples processed by the CQs with DS3 and DS6 are reduced by 50% (approximately) since the selectivity of the input routing operator was set at 50%. This is in contrast to the existing approaches where tuples are not filtered before the query processing.

8 Related Work

Currently, there are three security architectures presented in the literature. In this section, we will highlight some of the problems with those architectures.

Punctuation-based enforcement of RBAC over data streams is proposed in [22, 13]. A punctuation is a meta tuple, and can be either a security or query punctuation. Access control policies are transmitted every time using one or more security punctuations before the actual data tuple is transmitted. For example, when a user has an emergency situation, the device first sends the policies that allow emergency room physicians to access the tuple and then sends the data tuple. If the data tuple arrives without policies it will be removed from the system. Query punctuations define the privileges for a CQ. Both punctuations are processed by a special filter operator (stream shield) that is part of the query plan. This operator stores the privileges of the CQ to which it is attached, and checks it against the security punctuations of the data tuple. If the access check is successful, the data tuples that follow the punctuations are allowed to pass, otherwise, the data tuples are dropped. Major limitations of this approach are:

1. A set of these filtering operators are placed throughout the query plan. Thus, a data tuple and its corresponding punctuations entering the system are routed to all queries (authorized and unauthorized) and are dropped if the access check fails. This is not efficient, as punctuations have to be checked and dropped, including tuples, in all the CQs even if they do not have the required permissions. This does not follow the principle of least privilege as under privileged CQs receive over privileged tuples and are eventually dropped by the special filter operators.

2. This approach also modifies the query plan affecting the scheduling strategies and load shedding provided by the underlying system. The addition of special operators affect the QoS requirements significantly due to the redundant and unnecessary computations.

3. Security punctuations create a large overhead when many data providers submit data concurrently. If there is one or more punctuations per data tuple, which is usually the case with DSMS applications (e.g., location-based consumer services, health-care monitoring), and many concurrent data submitters, then there is a lot of overhead. This approach of streaming policies and data may cause a major bottleneck in the system and slow down the system substantially, as the system has to handle both the access control policy streams and data streams. and

4. This approach does not support sharing of queries between different roles.

The second architecture focuses on supporting RBAC via query rewriting techniques [23, 14] and is built on top of Stream Base [25]. This work introduces two operators for providing access control. To enforce access control policies, query plans are rewritten and policies are mapped to a set of map and filter operations. When a query is activated, the privileges of the query submitter are used to produce the resultant query plan. The major limitations of this approach are the modification of query plans and embedding access control within the query plan, affecting QoS optimizations and preventing query sharing. This approach also does not follow principle of least privilege as it processes underprivileged tuples. This places a significant burden on the QoS of the DSMS.

The final architecture [15] uses a post-query filter to enforce access control policies. The filter applies security policies only after query processing but before a user receives the results from the Borealis DSMS [16]. The major limitations of this model are: 1) Access control is only applied at the stream level i.e., all tuples and attributes of a tuple are assumed to be visible if the stream is visible to the user submitting the query. 2) All tuples have to be processed by all queries and finally filtered before the result is shown. This increases the load on the DSMS and affecting the QoS significantly.

9 Conclusions and Future Work

In this paper, we discussed various issues that need to be addressed when enforcing access control, with user-level and role-level sharing, in DSMSs. We discussed how access control affects tuple input, storage, processing and output. We then discussed our approach to enforce access control without introducing special operators, rewriting query plans, modifying query plans, or affecting QoS improvements. Our approach allows CQ sharing and prevents underprivileged CQs from processing/filtering tuples. We have shown how the enforcement can be moved outside of the query processing, while supporting sharing, reducing the cost of access control enforcement. We discussed two different approaches to join tuples for role-level sharing. We have shown the feasibility and demonstrated the low overhead of our approach. It enforces access control with a overhead of less than 2% with user-level sharing and 6% with role-level sharing over 2 Million tuples, and also reduced resource usage by preventing under-privileged tuples from entering the query processor.

As part of the future work, we are investigating the support of attribute-level access control, system-level sharing, optimize Join processing, and the implementation of exact match approach. We are also investigating the use of Stanford STREAM system to demonstrate the wider applicability of our approach.

10 Acknowledgments

We would like to thank Prof. Sharma Chakravarthy for providing the MavStream DSMS.

References

[1] S. Babu and J. Widom, "Continuous queries over data streams," in *Proceedings, International Conference on Management of Data*, September 2001, pp. 109–120.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and Issues in Data Stream Systems," in *Proceedings of ACM SIGMOD International Conference on Principles of Database Systems*, June 2002, pp. 1–16.

[3] A. Gilani, S. Sonune, B. Kendai, and S. Chakravarthy, "The Anatomy of a Stream Processing System," in *BNCOD*, 2006, pp. 232–239.

[4] D. Carney, U. Cetintemel, *et al.*, "Monitoring streams - a new class of data management applications," in *Proceedings, International Conference on Very Large Data Bases*, September 2002.

[5] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy, "MavEStream: Synergistic Integration of Stream and Event Processing." in *IEEE International Workshop on Data Stream Processing, ICDT*, Jul. 2007.

[6] S. Chakravarthy and Q. Jiang, *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*, ser. Advances in Database Systems , Vol. 36. Springer, 2009.

[7] S. Babu, L. Subramanian, and J. Widom, "A data stream management system for network traffic management," in *Proceedings of the Workshop on Network-Related Data Management (NRDM 2001)*, May 2001, pp. 685–686.

[8] C.-M. Chen, H. Agrawal, M. Cochinwala, and D. Rosenbluth, "Stream query processing for healthcare bio-sensor applications," in *Proceedings, International Conference on Data Engineering.* Washington, DC, USA: IEEE Computer Society, 2004, p. 791.

[9] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy, "NFM^i: An Inter-domain Network Fault Management System," in *Proceedings, International Conference on Data Engineering*, Tokyo, Japan, Apr. 2005, pp. 1036–1047.

[10] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma, "Query processing, resource management, and approximation," in *Proceedings, Conference on Innovative Data Systems Research*, 2003, pp. 245–256.

[11] M. Bishop, *Computer Security: Art and Science.* Addison-Wesley Professional, Dec. 2002.

[12] *RBAC Standard, ANSI INCITS 359-2004*, ANSI INCITS 359-2004, InterNational Committee for Information Technology Standards, 2004.

[13] R. V. Nehme, E. A. Rundensteiner, and E. Bertino, "A security punctuation framework for enforcing access control on streaming data," in *Proceedings, International Conference on Data Engineering*, 2008, pp. 406–415.

[14] B. Carminati, E. Ferrari, and K.-L. Tan, "Enforcing access control over data streams," in *Proceedings, ACM Symposium on Access Control Models and Technologies*, 2007, pp. 21–30.

[15] W. Lindner and J. Meier, "Securing the borealis data stream engine," in *IDEAS*, 2006, pp. 137–147.

[16] D. J. Abadi, Y. Ahmad, *et al.*, "The Design of the Borealis Stream Processing Engine," in *Proceedings, Conference on Innovative Data Systems Research*, 2005, pp. 277–289.

[17] B. Babcock *et al.*, "Models and issues in data stream systems." in *Proceedings, International Conference on Principles of Database Systems*, June 2002, pp. 1–16.

[18] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," Stanford InfoLab, Technical Report 2004-20, 2004. [Online]. Available: http://ilpubs.stanford.edu:8090/641/

[19] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.

[20] B. Babcock, S. Babu, R. Motwani, and M. Datar, "Chain: operator scheduling for memory minimization in data stream systems," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, June 2003, pp. 253–264.

[21] *The Economic Impact of Role-Based Access Control*, RTI Project Number: 07007.012, National Institute of Standards and Technology (NIST), 2002. [Online]. Available: http://www.nist.gov/director/prog-ofc/report02-1.pdf

[22] R. V. Nehme, H.-S. Lim, E. Bertino, and E. A. Rundensteiner, "StreamShield: A stream-centric approach towards security and privacy in data stream environments," in *Proceedings, International Conference on Management of Data*, 2009, pp. 1027–1030.

[23] J. Cao, B. Carminati, E. Ferrari, and K.-L. Tan, "Acstream: Enforcing access control over data streams," in *Proceedings, International Conference on Data Engineering*, 2009, pp. 1495–1498.

[24] Q. Jiang and S. Chakravarthy, "Data stream management system for MavHome," in *Proceedings, Annual ACM SIG Symposium On Applied Computing*, 2004, pp. 654–655.

[25] StreamBase, http://www.streambase.com.