Information Flow Control for Stream Processing in Clouds

Xing Xie
Computer Science Department
Colorado State University
Fort Collins, CO 80523
xing@cs.colostate.edu

Indrakshi Ray
Computer Science Department
Colorado State University
Fort Collins, CO 80523
iray@cs.colostate.edu

Raman Adaikkalavan
Computer and Information Sciences
Indiana University South Bend
South Bend, IN 46634
raman@cs.iusb.edu

Rose Gamble
Tandy School of Computer Science
The University of Tulsa
Tulsa, OK 74104
gamble@utulsa.edu

ABSTRACT

In the near future, clouds will provide situational monitoring services using streaming data. Examples of such services include health monitoring, stock market monitoring, shopping cart monitoring, and emergency control and threat management. Offering such services require securely processing data streams generated by multiple, possibly competing and/or complementing, organizations. Processing of data streams also should not cause any overt or covert leakage of information across organizations. We propose an information flow control model adapted from the Chinese Wall policy that can be used to protect against sensitive data disclosure. We propose architectures that are suitable for securely and efficiently processing streaming information belonging to different organizations. We discuss how performance can be further improved by sharing the processing of multiple queries. We demonstrate the feasibility of our approach by implementing a prototype of our system and show the overhead incurred due to the information flow constraints.

Keywords

Chinese Wall Policy, DSMS, Continuous Query Processing

1. INTRODUCTION

Data Stream Management Systems (DSMSs) [1, 6, 9, 8, 14, 7, 16] are needed for situation monitoring applications that collect high-speed data, run continuous queries to process them, and compute results on-the-fly to detect events of interest. Consider one potential situation monitoring application – collecting real-time streaming audit data to thwart various types of attacks in a cloud environment. Detecting such precursors to attacks may involve analyzing streaming audit data belonging to various, possibly competing and/or complementing, organizations. Consequently, it is important to protect such data from unauthorized disclosure and modification. Moreover, the processing of continuous queries should not cause leakage or modification of sensitive data. Towards this end, we redesign a DSMS such that it can process information generated in a multi-domain environment, each consisting of competing entities, in a secure manner.

Researchers have worked on secure data and query processing in the context of DSMSs. However, almost all of these works focus on providing access control to streaming data [20, 13, 22, 12, 21, 4, 5]. Controlling access is not enough to prevent security breaches in cloud computing applications where illegal information flow can occur across multiple domains. The existence of covert and overt channels can cause sensitive information to be illegally passed from one domain to another. We need to prevent unauthorized access but also ensure the absence of such illegal information flow. Towards this end, we propose an information flow control model adapted from the Chinese Wall policy [23] that can be used to provide secure processing of streaming data generated from multiple organizations.

A cloud contains a set of companies that offer services. In order to keep the cloud operational, it is important to detect security and performance problems in a timely manner. Thus, auditing live events streaming from the cloud is very essential. Services offered in a cloud can be competing or complementing. To detect attacks and performance issues, the cloud has to be audited as a whole, though the audit events may be generated by competing or complementing companies. Chinese Wall policy aims to protect disclosure of company sensitive information to potentially competing organizations, but does not deal with complementing organizations. In a cloud, companies are organized into various domains based on the types of services they provide. Each of these domains forms a conflict of interest (COI) class. Companies in the same COI class are in direct competition. We must aim to prevent leakage of a company's sensitive information to other organizations belonging to the same COI class. Companies that offer complementing services can be assigned a complementing interest (CI) class. Companies in the same COI class cannot be in the same CI. Companies belonging to the same CI have no such direct competition and do not require trusted entities to manage their information.

Streaming audit data generated by various organizations must be analyzed in real-time to detect the presence of various types of attacks. A company may want to audit its own data to detect malicious insider threats. Sometimes it may be needed to detect a

SACMAT'13, June 12–14, 2013, Amsterdam, The Netherlands.

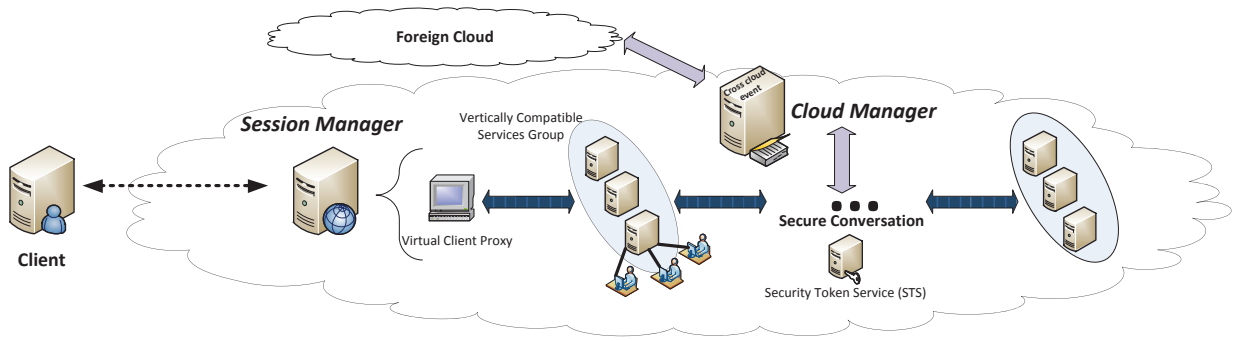

Figure 1: Multi-Tier Architecture of a Cloud

denial-of-service attack for a particular type of service offered by companies in a COI class. On the other hand, detecting the delay between the service request and response may involve analyzing audit streams in a service chain invocation that has multiple companies belonging to some CI class. For each such case, it should be possible to detect the attack without causing a company's sensitive information from being leaked to its competitors.

Our goal in this paper is not to address potential security attacks in the cloud. We focus on how streaming data generated by various organizations can be processed in a secure manner so as to protect against unauthorized disclosure and modification. Our threat model is that processing units may contain unintentional bugs or trojan horses that cause information leakage. We also want protection from honest but curious users who want to know information that they are not authorized to access. We do not address denial-of-service attacks. We assume that the underlying infrastructure is trusted.

In this work, we start with identifying the access requirements and the information flow constraints for processing streaming audit data in a cloud computing environment. We adapt the Chinese Wall policy formulated by Sandhu [23] to formalize the information flow constraints in clouds. We demonstrate how cloud computing queries can be formulated and provide an architecture for executing such queries. We demonstrate how multiple queries can share their computation to improve performance and provide good resource utilization. We also implement a prototype to demonstrate the feasibility of our approach and show how the performance is impacted by the information flow constraints.

The rest of the paper is organized as follows. In Section 2, we present an architecture for processing continuous queries generated from the various tiers in the cloud. In Section 3, we present our information flow control model that formulates the rules for accessing data streams generated by various organizations in the cloud. In Section 4, we present some example queries that are executed in the cloud. In order to accelerate throughput, we show how continuous queries can be shared in Section 5. In Section 6, we discuss our prototype and show some performance results. In Section 7, we mention a few related work. In Section 8, we conclude the paper with pointers to future work.

2. CONTINUOUS QUERY PROCESSING ARCHITECTURE

In this section, we present our example application that motivates the need for secure stream processing in cloud computing environments. We have a service that aims to prevent and detect attacks in real-time in the cloud. Such a service provides warning about various types of attacks, often involving multiple organizations.

Figure 1 shows a multi tier architecture of the cloud adopted from [26]. Various types of auditing may take place in the cloud. The first level is the *company auditing tier*, not explicitly shown in Figure 1, that is represented by the users connected to some service. In this tier, the activities pertaining to an organization are analyzed in isolation. The next level is the *service auditing tier*, identified by shaded ellipses that contain sets of resources and services. Each shaded ellipse depicts vertically compatible services or resources; this implies the services or resources that can be functionally substituted for each other, possibly on demand. The *cloud auditing tier* is shown with connecting dark arrows, which depicts the internal communication within the cloud due to a service invocation chain.

Various types of audit streams must be captured to detect the different types of attacks that may take place in a cloud. The company auditing tier logs the activities of the various users in the organization. If the behavior of an authorized user does not follow his usual pattern, we can perform analysis to determine if the user's authentication information has been compromised. This tier is responsible for analyzing the audit streams of individual companies in isolation. Typically, at this layer, the audit streams generated by a single company are analyzed.

The service auditing tier logs information pertaining to the various companies who provide similar services. Session Manager at this tier can detect whether there is a denial-of-service attack targeted at a specific type of service. Session Manager analyzes audit streams generated from multiple competing organizations, so we need to protect against information leakage and corruption. In short, the Session Manager needs to analyze data from one or more companies belonging to the same COI class.

The cloud auditing tier collects audit information pertaining to a service invocation chain and is able to detect the presence of man-in-the-middle attack. Cloud Provider is responsible for analyzing audit streams from multiple organizations associated with service invocation chains, but the organizations may not have conflict of interest. Thus, at this tier, the audit streams from the companies belonging to one or more CI classes are analyzed.

In order to detect and warn against these attacks, continuous queries must be executed on the streaming data belonging to various organizations. Queries must be processed such that there are no overt or covert leakage of information across competing organizations. We assign security levels to categorize the various classes of

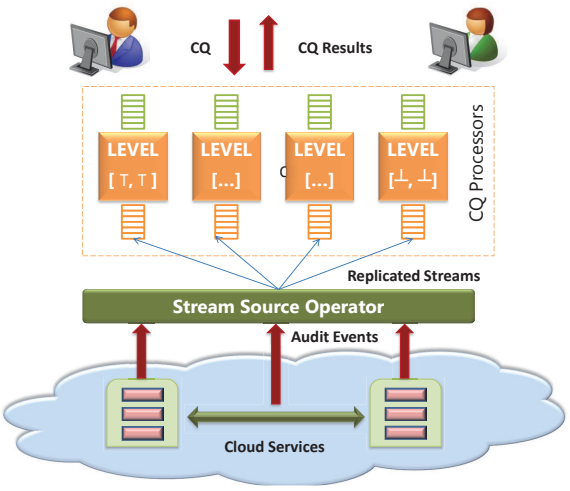

Figure 2: CQ Processing Architecture

data that are being generated and collected at the various tiers. The security level of the data determines who can access and modify it. In the next section, we discuss how security levels are assigned to various classes of data.

We propose the architecture shown in Figure 2 that provides a way to capture events from the cloud, monitor them, and trigger alerts. The architecture uses concepts based on cloud computing [26], data stream processing [16, 8, 6, 17], event processing [15, 3], Chinese wall security [23], replicated and trusted multilevel database management [2] and multilevel secure data stream management system [5].

As shown in Figure 2 there are several services offered in the cloud. Audit data generated by the services are sent to the DSMS. For this paper, we consider a centralized DSMS architecture. Compatible services are grouped and they interact based on client needs. Servers contain event detectors that monitor and detect occurrence of interesting events. The detectors sanitize and propagate the events to the data stream management system, which arrive at the stream source operator. This operator checks for the level of the incoming audit events and propagates them to the appropriate query processor's input queue. The query processor architecture is based on the replicated model, where there is a one-to-one correspondence between query processors and security levels. A query specified by a user at a particular level is executed by the query processor running at that level. Also a query processor at some level can only process data that it is authorized to view. After processing the query results are disseminated to authorized users via the output queues of queries. In addition to the query processors and stream source operator the data stream management system contains various other components (trusted and untrusted) as discussed in the implementation and experimental evaluation section.

3. INFORMATION FLOW MODEL

In the following, we present an information flow model for cloud applications to protect against improper leakage and disclosure. We provide an information flow model that is adapted from the lattice structure for Chinese Wall proposed by Sandhu [23].

We have a set of companies that provide services in the clouds. The companies are partitioned into conflict of interest classes based on the type of services they provide. Companies providing the same type of service are in direct competition with each other. Consequently, it is important to protect against disclosure of sensitive information to competing organizations. We begin by defining how the conflict of interest classes are represented.

DEFINITION 1. **[Conflict of Interest Class Representation:]** *The set of companies providing service to the cloud are partitioned into a set of n conflict of interest classes, which we denote by* COI_1, COI_2, ..., *and* COI_n. *Each conflict of interest class is represented as* COI_i, *where* $1 \leq i \leq n$. *Conflict of interest class* COI_i *consists of a set of* m_i *companies, where* $m_i \geq 1$, *that is* $COI_i = \{1, 2, 3, \ldots, m_i\}$.

On the other hand, a set of companies, who are not in competition with each other, may provide complementing services in the cloud. A single company can provide some service, and sometimes multiple companies may together offer a set of services. In the following, we define the notion of complementing interest (CI) class and show how it is represented.

DEFINITION 2. **[Complementing Interest Class Representation:]** *The set of companies providing complementing services is represented as an n-element vector* $[i_1, i_2, \ldots, i_n]$, *where* $i_k \in COI_k \cup \{\perp\}$ *and* $1 \leq k \leq n$. $i_k = \perp$ *signifies that the CI class does not contain services from any company in the conflict of interest class* COI_k. $i_k \in COI_k$ *indicates that the CI class contains services from the corresponding company in conflict of interest class* COI_k. *Our representation forbids multiple companies that are part of the same COI class from being assigned to the same complementing interest class.*

We next define the security structure of our model. Each data stream, as well as the individual tuples constituting it, is associated with a security level that captures its sensitivity. Security level associated with a data stream dictates which entities can access or modify it. Input data stream generated by an individual organization offering some service has a security level that captures the organizational information. Input streams may be processed by the DSMS to generate *derived streams*. Derived data streams may contain information about multiple companies, some of which are in the same COI class and others may belong to different COI classes. Before describing how to assign security levels to derived data streams, we show how security levels are represented.

DEFINITION 3. **[Security Level Representation:]** *A security level is represented as an n-element vector* $[i_1, i_2, \ldots, i_n]$, *where* $i_j \in COI_j \cup \{\perp\} \cup \{T\}$ *and* $1 \leq j \leq n$. $i_j = \perp$ *signifies that the data stream does not contain information from any company in* COI_j; $i_j = T$ *signifies that the data stream contains information from two or more companies belonging to* COI_j; $i_j \in COI_j$ *denotes that the data stream contains information from the corresponding company in* COI_j.

Consider the case where we have 3 COI classes, namely, COI_1, COI_2, and COI_3. COI_1, COI_2, and COI_3 have 5, 3, and 2 companies, respectively. The audit stream generated by Company 5 in COI_1 has a security level of $[5, \perp, \perp]$. Similarly, the audit stream generated by Company 2 in COI_3 has a security level $[\perp, \perp, 2]$. When audit streams generated from multiple companies are combined, the information contained in this derived stream has a higher security level. For example, audit stream having level $[5, \perp, 2]$ contains information about Company 5 in COI_1 and Company 2 in COI_3. It is also possible for audit streams to have information from multiple companies belonging to the same *COI* class. For example, a security level of $[5, \perp, T]$ indicates that the data stream has information from Company 5 in COI_1, does not contain information

from COI_2, and information about multiple companies in COI_3. We also have a level $[\perp,\perp,\perp]$ which we call *public* and that has no company specific information. The level $[T,T,T]$ correspond to level *trusted* and it contains information pertaining to multiple companies in each COI class and can be only accessed by trusted entities. We next define dominance relation between security levels.

DEFINITION 4. **[Dominance Relation:]** *Let* $\mathbf{L}$ *be the set of security levels,* L_1 *and* L_2 *be two security levels, where* $L_1, L_2 \in \mathbf{L}$. *We say security level* L_1 *is dominated by* L_2*, denoted by* $L_1 \preceq L_2$*, when the following conditions hold:* $(\forall i_k = 1,2,\ldots,n)(L_1[i_k] = L_2[i_k] \vee L_1[i_k] = \perp \vee L_2[i_k] = T)$. *For any two levels,* L_p, $L_q \in \mathbf{L}$*, if neither* $L_p \preceq L_q$*, nor* $L_q \preceq L_p$*, we say that* L_p *and* L_q *are incomparable.*

The dominance relation is reflexive, antisymmetric, and transitive. The level *public*, denoted by $[\perp,\perp,\perp]$, is dominated by all the other levels. Similarly, the level *trusted*, denoted by $[T,T,T]$, dominates all the other levels. Note that the dominance relation defines a lattice structure, where level *public* appears at the bottom and the level *trusted* appears at the top. Incomparable levels are not connected in this lattice structure. In our earlier example, level $[5,\perp,\perp]$ is dominated by $[5,\perp,2]$ and $[5,\perp,T]$. $[5,\perp,2]$ is dominated by $[5,\perp,T]$. That is, $[5,\perp,\perp] \preceq [5,\perp,2]$ and $[5,\perp,2] \preceq [5,\perp,T]$. $[5,\perp,\perp]$ and $[\perp,\perp,2]$ are incomparable.

Each data stream is associated with a security level. Each of the tuples constituting the data stream also has a security level assigned to it. Thus, the schema of the data stream has an attribute called level that captures the security level of tuples. The level attribute is generated automatically by the system and cannot be modified by the users. Note that, the security level of an individual tuple in a data stream is dominated by the level of the data stream. When a DSMS operation is executed on multiple input tuples, each having its own security level, an output tuple is produced. The security level of the output tuple is the least upper bound (LUB) of the security levels of the input tuples.

In our work, various types of queries are executed to detect security and performance problems. Each continuous query Q_i, submitted by a process, inherits the security level of the process. We require a query Q_i to obey the simple security property and the restricted $\star$-property of the Bell-Lapadula model [10].

1. Query Q_i with $L(Q_i) = C$ can read a data stream x only if $L(x) \preceq C$.
2. Query Q_i with $L(OP_i) = C$ can write a data stream x only if $L(x) = C$.

Note that, for our example, a process executing at level $[5,\perp,T]$ can execute streams belonging to Company 5 in COI_1 and all companies in COI_3 and also streams derived from them. Thus, the process is trusted w.r.t. COI_3, but not w.r.t. the other COI classes. Our information flow model thus provides a finer granularity of trust than provided by the earlier models. Our goal is to allow information flow only from the dominated levels to the dominating ones. All other information flow, either overtly or covertly, should be disallowed by our architecture.

4. QUERY PROCESSING IN CLOUD DSMS

In this section, first we discuss the different types of queries that can be executed on cloud audit data at the different tiers.

4.1 Cloud CQL Queries

Consider a simple application that tries to detect example denial-of-service attacks in the cloud. We have two conflict of interest classes denoted by COI_1 and COI_2. The constituent companies in each COI class is given by, $COI_1 = \{1,2\}$ and $COI_2 = \{A,B,C\}$. Examples of security levels in our configuration are $[\perp,\perp]$ (public knowledge), $[T,T]$ (completely trusted), $[1,\perp]$ (data from 1), $[\perp,T]$ (trusted w.r.t. COI_2), $[1,B]$ (data from 1 and B), $[1,T]$ (data from 1 in COI_1 and trusted w.r.t. COI_2). Continuous queries are executed at various tiers to detect performance delays and possibly denial-of-service (DoS) attacks. In any given tier, different types of DoS attacks may occur – some involving the data belonging to single organizations, others involving data belonging to multiple organizations. Thus, a tier can have query processors at different levels, each of which executes queries on data that it is authorized to view and modify.

We consider a single data stream, called `MessageLog`, that contains the audit stream data associated with message events, such as `send` and `receive`. `MessageLog` is obtained from `SystemLog` by filtering the events related to sending and receiving the messages. Note that, `MessageLog` in reality may contain many other fields, but we only deal with those that are pertinent to this example. The various attributes in `MessageLog` are `serviceId`, `msgType`, `sender`, `receiver`, `timestamp`, `outcome`. `serviceId` is a unique identifier associated with each service; `msgType` gives the type of message which is either `send` or `receive`; `sender` (`receiver`) gives the id of the organization sending (receiving) the message; `timestamp` is the time when the event (`send` or `receive`) occurred; `outcome` denotes `success` or `failure` of the event. In addition to these attributes, we have an attribute referred to as `level` that represents the security level of the tuple. The `level` attribute is assigned by the system and it cannot be modified by the user.

```
MessageLog(serviceId, msgType, sender, receiver,
                  timestamp, outcome)
```

The queries are expressed using the CQL language [7]. We describe the various types of queries that can be executed at the various tiers.

4.1.1 Company Auditing Tier

In the company auditing tier, companies have access only to their own audit records.

In this section we give some sample queries that are executed by *Company1* to detect performance delays and DoS attacks. All the queries are executed at level $[1,\perp]$.

Query 1 (Q1)

Company1 requests service from *CompanyB*. It is trying to check the times when such message could be successfully delivered.

```
SELECT timestamp FROM MessageLog
   WHERE msgType = "send" AND outcome = "success"
      AND receiver = "CompanyB"
```

Query 2 (Q2)

Company1 requests service from *CompanyB*. It is trying to check the times when such message could not be successfully delivered.

```
SELECT timestamp FROM MessageLog
   WHERE msgType = "send" AND outcome = "failure"
      AND receiver = "CompanyB"
```

4.1.2 Service Auditing Tier

Service auditing tier receives log records from all the companies making use of some service. However, as the queries below demonstrate, not all the queries need to access all the data from the same COI class.

Query 3 (Q3): Level $[\perp, B]$
Log records received at the service auditing tier can be analyzed by the Session Manager to find out whether *CompanyB* is not available for some service.

```
SELECT timestamp FROM MessageLog
   WHERE msgType = "send" AND outcome = "failure"
      AND receiver = "CompanyB"
```

Query 4 (Q4): Level $[\perp, T]$
Session Manager may wish to find out whether all companies in COI_2 are target of some DoS attacks.

```
SELECT timestamp FROM MessageLog
   WHERE msgType = "send" AND outcome = "failure"
      AND receiver = "CompanyB" OR
      receiver = "CompanyA" OR
      receiver = "CompanyC"
```

4.1.3 Cloud Auditing Tier

Cloud auditing tier gets log records pertaining to all the services. However, the various queries will have different types of security requirements.
Query 5 (Q5): Level $[1, B]$
Cloud Provider may want to look at all records pertaining to `serviceId 5` and measure the delays in order to detect possible man-in-the-middle attack. `serviceId 5` involves *Company1* and *CompanyB*.

```
SELECT  MIN(timestamp), MAX(timestamp)
  FROM MessageLog [ROWS 100]
    WHERE outcome = "success" AND serviceId = "5"
```

Query 6 (Q6): Level $[1, B]$

Cloud Provider wants to find the delay encountered by *Company1* between sending the request and receiving the service from *CompanyB* for the last 100 tuples.

```
SELECT R.timestamp - S.timestamp  AS  delay
   FROM MessageLog R[Rows 100], MessageLog S[Rows 100]
   WHERE S.msgType = "send" AND S.outcome = "success"
   AND R.msgType = "receive" AND R.outcome = "success"
   AND R.receiver = "Company1" AND R.sender = "CompanyB"
   AND S.receiver = "CompanyB" AND S.sender = "Company1"
   AND S.serviceId = R.serviceId
```

Query 7 (Q7): Level $[T, T]$
Cloud Provider may want to find out the delay incurred in the different service invocation chains.

```
SELECT MIN(timestamp), MAX(timestamp)
   FROM MessageLog[ROWS 100]
      WHERE outcome = "success"
         GROUP BY serviceId
```

4.2 Execution of Cloud Queries

For each tier, we may have one or multiple query processors. In the Company Auditing Tier, we have a single query processor for analyzing each company data. Thus, *Company1* has a single query processor at level $[1, \perp]$. In the Service Auditing Tier, we may have one or more query processors running at different levels. In our examples, we can have a query processor at level $[\perp, B]$ and another one at $[\perp, T]$. Alternatively, we can use $[\perp, T]$ to process both the queries. Using $[\perp, T]$ to process the query at level $[\perp, B]$ comes at a cost: the query submitted at $[\perp, B]$ must be rewritten such that it can access only those tuples that it is authorized to view. Similarly, for the Cloud Auditing Tier, we may have a single query processor at level $[T, T]$ or two query processors: one at level $[1, B]$ and the other at $[T, T]$.

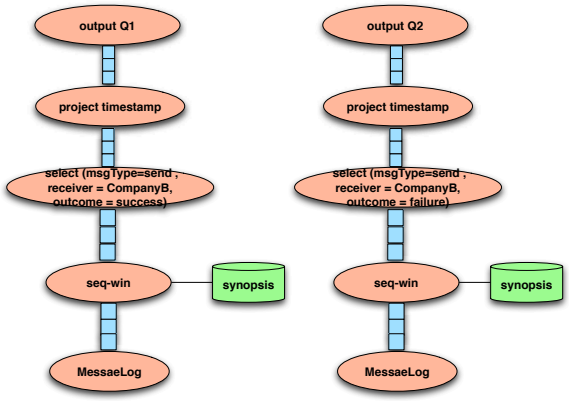

Figure 3: Operator Tree for Q_1 and Q_2

When a query has been submitted by a user, it must be rewritten to ensure that no unauthorized tuples are returned to the user. Our query rewriting algorithm modifies the algorithm in the following ways. Let Q_x be the original query submitted at level $L(Q_x)$. Let $selectCond(Q_x)$ and $window(Q_x)$ be the selection and window condition associated with the query. The query rewriting algorithm restricts the window to filter those tuples that the query is authorized to view; this is denoted by $|window(Q_x)|_{L_{Q_x}}$. Note that, all queries may not have a window operator. In such cases, the query rewriting algorithm adds a new security conjunct to the existing selection condition. This conjunct ensures that the tuples satisfying the query is dominated by the level of the query. The query rewriting algorithm is given below.

Algorithm 1: Secure Query Rewriting

INPUT: (Q_x)
OUTPUT: $OPT(Q'_x)$ representing the rewritten query
if $window(Q_x) \neq \{\}$ **then**
  $window(Q_x) = |window(Q_x)|_{L(Q_x)}$
end
else
  $selectCond(Q_x) = selectCond(Q_x) \cup (level \preceq L(Q_x))$
end

Let us consider *Q5* once again that is submitted at Level $[1, B]$.

```
SELECT  MIN(timestamp), MAX(timestamp)
  FROM MessageLog [ROWS 100]
    WHERE outcome = "success" AND serviceId = "5"
```

If this query is executed by the query processor at Level $[1, B]$, no rewriting is needed. However, if the query is executed at Level $[T, T]$, the query must be rewritten to ensure that it can view only authorized information. In such a case, the query is rewritten as follows:

```
SELECT  MIN(timestamp), MAX(timestamp) FROM
 MessageLog [ROWS 100 WHERE Level DOMINATED BY [1,B]]
    WHERE outcome = "success" AND serviceId = "5"
```

Next, the secure queries are represented in the form of an operator tree. The formal definition of an operator tree appears below.

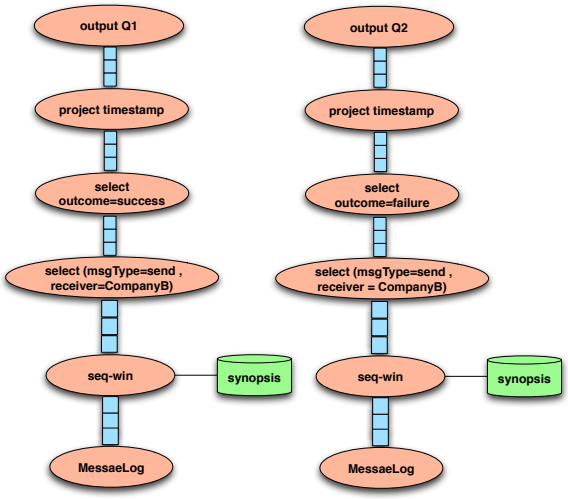

Figure 4: Overlapping Nodes Decomposed in Q_1 and Q_2

DEFINITION 5. **[Operator Tree:]** *An* operator tree *for a query Q_x, represented in the form of* OPT(Q_x), *consists of a set of nodes N_{Q_x} and a set of edges E_{Q_x}. Each internal node N_i corresponds to some operator in the query Q_x. Each edge (i, j) in this tree connecting node N_i to node N_j signifies that the output of node N_i is the input to node N_j. Each node N_i has a label that provides details about processing the corresponding operator. The name component of the label, denoted by $N_i.op$, specifies the type of the operator, such as,* select, project, join *and* average. *The parameter component of the label, represented by $N_i.parm$, denotes the set of conjuncts for the* select *operator or the set of attributes for the* project *operator. The synopsis component, denoted by $N_i.syn$, is needed for operators specified with windows that require a set of tuples to accumulate before processing can start, such as,* join, count, *and* sum. *The synopsis component has two attributes, namely, type and size, that specify the type of window (tuple-based, time-based) and its size. The input queue component of the label, denoted by $N_i.inputQueue$, lists the input streams needed by the operator. The output queue component, denoted by $N_i.outputQueue$, indicates the output streams produced by the operator that can be used by the vertices or sent as response to the user. The leaf nodes of the operator tree represent the source nodes for the data streams and the root nodes are the sink nodes receiving output.*

The operator trees for $Q1$ and $Q2$ are shown in Figure 3. Note that, an operator tree has all the information needed for processing the query.

5. QUERY SHARING

Typically, in a DSMS there can be several queries that are being executed concurrently. Query sharing will increase the efficiency of these queries and save resources such as CPU cycles and memory usage. Query sharing obviates the need for evaluating the same operator(s) multiple times if different queries need it. In such a case, the operator trees of different queries can be merged.

We next formalize basic operations that are used for comparing the nodes belonging to different operator trees. Such operations are needed to evaluate whether sharing is possible or not between queries. We begin with the equivalence operator. If a pair of nodes belonging to different operator trees is equivalent, then only one

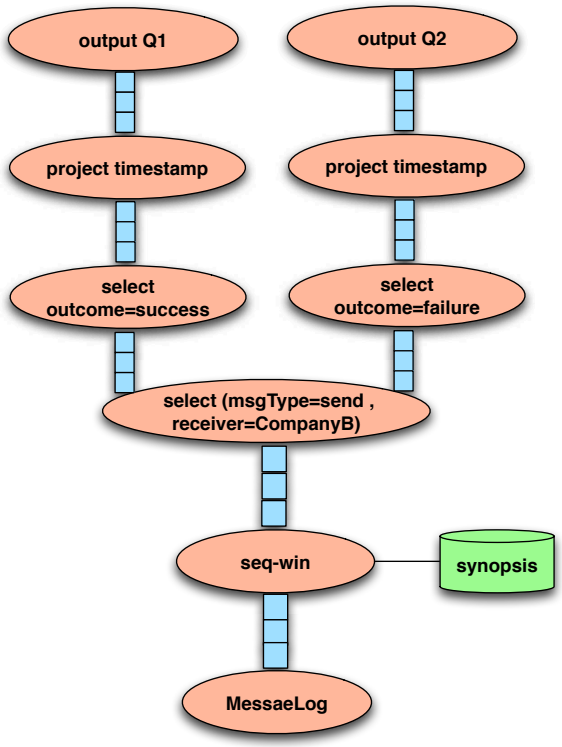

Figure 5: Merged Operator Trees of Q_1 and Q_2

node needs to be computed for evaluating the queries corresponding to these different operator trees.

DEFINITION 6. **[Equivalent Nodes:]** *Node $N_i \in N_{Q_x}$ is said to be* equivalent *to node $N_j \in N_{Q_y}$, denoted by $N_i \equiv N_j$, where N_i, N_j are in the operator trees $OPT(Q_x)$, $OPT(Q_y)$ respectively, if the following condition holds: $N_i.op = N_j.op \wedge N_i.parm = N_j.parm \wedge N_i.syn = N_j.syn \wedge N_i.inputQueue = N_j.inputQueue$*

In Figure 3, the *seq-win* operator nodes are equivalent in the two given operator trees. Two nodes may not be equivalent, but they may have common subexpressions; we may wish to process such subexpressions only once. For example, the *select* nodes in the two trees are not equivalent, but they have common subexpressions. This leads to the definition of *overlapping nodes*.

DEFINITION 7. **[Overlapping Nodes:]** *Node $N_i \in N_{Q_x}$ is said to be* overlapping *with node $N_j \in N_{Q_y}$, denoted by $N_i \equiv N_j$, where N_i, N_j are in the operator trees $OPT(Q_x)$, $OPT(Q_y)$ respectively, if the following condition holds: $N_i.op = N_j.op \wedge Ni.op$ is* non blocking operator, *$\wedge N_i.parm \cap N_j.parm \neq \{\} \wedge N_i.inputQueue = N_j.inputQueue$*

Each of the overlapping nodes is decomposed into two nodes as follows.

DEFINITION 8. **[Decomposition of Overlapping Nodes:]** *Node N_i that overlaps with node N_j is decomposed into node N_k and N_i' as follows.*

1. *$N_k.op = N_i.op$ and $N_i'.op = N_i.op$*
2. *$N_k.inputQueue = N_i.inputQueue$ and $N_i'.inputQueue = N_k.outputQueue$*

3. $N_k.parm = N_i.parm \cap N_j.parm$ *and*
   $N_i'.parm = N_i.parm - N_k.parm$, *if* $N_i.op = select$
4. $N_k.parm = N_i.parm \cup N_j.parm$ *and*
   $N_i'.parm = N_i.parm$, *if* $N_i.op = project$

The *select* nodes in $Q1$ and $Q2$ are overlapping, so they are each decomposed into two nodes as shown in Figure 4.

If two operator trees have equivalent nodes, they are said to *overlap*. The formal definition appears below.

DEFINITION 9. **[Overlap of Operator Trees:]** *Two operator trees* $OPT(Q_x)$ *and* $OPT(Q_y)$ *are said to* overlap *if* $OPT(Q_x) \not\equiv OPT(Q_y)$ *and there exists a pair of nodes* N_i *and* N_j *where* $N_i \in N_{Q_x}$ *and* $N_j \in N_{Q_y}$ *such that* $N_i \equiv N_j$.

Algorithm 2: Merge Operator Trees

```
INPUT: OPT(Q_x) and OPT(Q_y)
OUTPUT: OPT(Q_xy) representing the merged operator tree
Initialize N_Qxy = {}
Initialize E_Qxy = {}
foreach node N_i ∈ N_Qx do
  | N_Qxy = N_Qxy ∪ N_i
end
foreach edge (i, j) ∈ E_Qx do
  | E_Qxy = E_Qxy ∪ edge (i, j)
end
foreach node N_i ∈ N_Qy do
  | if ∄N_j ∈ N_Qxy such that N_i ≡ N_j then
  |   | N_Qxy = N_Qxy ∪ N_i
  | end
end
foreach edge (i, j) ∈ E_Qy do
  | if edge (i, j) ∉ E_Qxy then
  |   | E_Qxy = E_Qxy ∪ edge (i, j)
  | end
end
```

When operator trees corresponding to two queries overlap, we can generate the merged operator tree using Algorithm 2. The merged operator tree signifies the processing of the partially shared queries. The two operator trees shown in Figure 4 can now be merged. The merged tree appears in Figure 5.

6. PROTOTYPE IMPLEMENTATION

We have developed the replicated CW-DSMS shown in Figure 6. This system is a modified version of the Stanford STREAM DSMS [6]. The CW-DSMS supports: (i) multi-user server with user authentication, (ii) replicated query processors executing at different security levels, (iii) a global trusted scheduler that schedules operators across all query processors, (iv) a global trusted interpretation unit that supports centralized query plan generation for all query processors, (v) trusted stream shepherd operator that takes trusted streams and outputs streams based on the security level of the query processor, (vi) security level aware windows, (vii) security level aware query operators i.e., modification to blocking operators (e.g., *join*, *average*) to create output tuples with appropriate level identification, and (viii) single security level input streams and tuples.

The *trusted command unit* shown in Figure 6 is responsible for handling client communication, authentication, and query processor instantiation. It accepts queries at different security levels. The *authentication module* performs user authentication and security level verification and assignment. The *query processor identifier (QPI) module* gets user's client ID, security level and query specifications from the authentication module. The QPI maintains the list of currently running query processors. The QPI first checks whether the user queries can be executed in one of the query processors by matching the incoming and existing security levels. If *Yes*, user's client ID as well as all the input queries are bound to that processor. If *No*, a new query processor is created at that level. This approach allows starting query processors on demand. The maximum number of query processors is bounded by the number of security levels supported. In addition, the trusted command unit still controls the query operations like commit and abort with the help of the QPI module.

The *trusted interpretation unit* is responsible for generating query plans and setting up the operators in the scheduler. This unit receives user and query information from the trusted command unit. It creates the physical and logical query plans. It sends the physical plans to the appropriate query processor based on the user's security level. The list of operators (also the physical plan) is sent to the trusted scheduler. The modified built-in interpretation components (*parser, semantic interpreter,* and *logical plan generator*) and the execution unit of the query processor handle CW security extensions.

As shown in Figure 6 the server has multiple *query processors* running at different security levels, but, only one trusted interpretation unit and trusted scheduler. Each query processor contains an input unit, execution unit, and output unit. The *input unit* can accept input streams from outside sources through the trusted stream shepherd unit. It can also accept the output streams produced from other queries processed by the same query processor. The *execution unit* is used by the server to execute physical plans continuously. This unit contains the physical operators and their corresponding algorithms. We have modified the window processing so that it can support conditions based on security levels. We have modified the aggregate and join operator algorithms to compute the least upper bound (LUB) of the input tuples and use that as the security level of the output tuples. For example, if an aggregate operation computes the maximum timestamp of three input tuples in levels $[\perp, A], [\perp, B]$ and $[\perp, C]$, the output tuple is in level $[\perp, T]$. On the other hand, all the operators are untrusted. The execution unit accepts the commands from the trusted scheduler and executes the corresponding operators. There is only one operator running at any point of time, since we have only one scheduler. The *output unit* sends the results back to users continuously.

The *trusted stream shepherd unit* handles all input streams. It contains the modified input operator (trusted stream shepherd operator) that handles the trusted streams in different security levels and sends it to the appropriate query processors.

In the vanilla STREAM prototype, the scheduler uses round robin algorithm to schedule operators. We have modified the scheduler so that it can handle scheduling of queries in more than one query processor. The scheduler maintains all executing query plan information shared by the trusted interpretation unit. When a query plan is received by the scheduler, operators in the plan are scheduled for execution from bottom to top order. The scheduler sends out commands (including plan id and operator id) to the appropriate query processor to start executing an operator. The operators execute at least once per scheduling round. When a new plan arrives at the scheduler, operators of that plan will be scheduled in the next execution round. Such mechanism prevents starvation of late coming queries, as each operator is scheduled every round. In every round, each operator processes a maximum of 170 data tu-

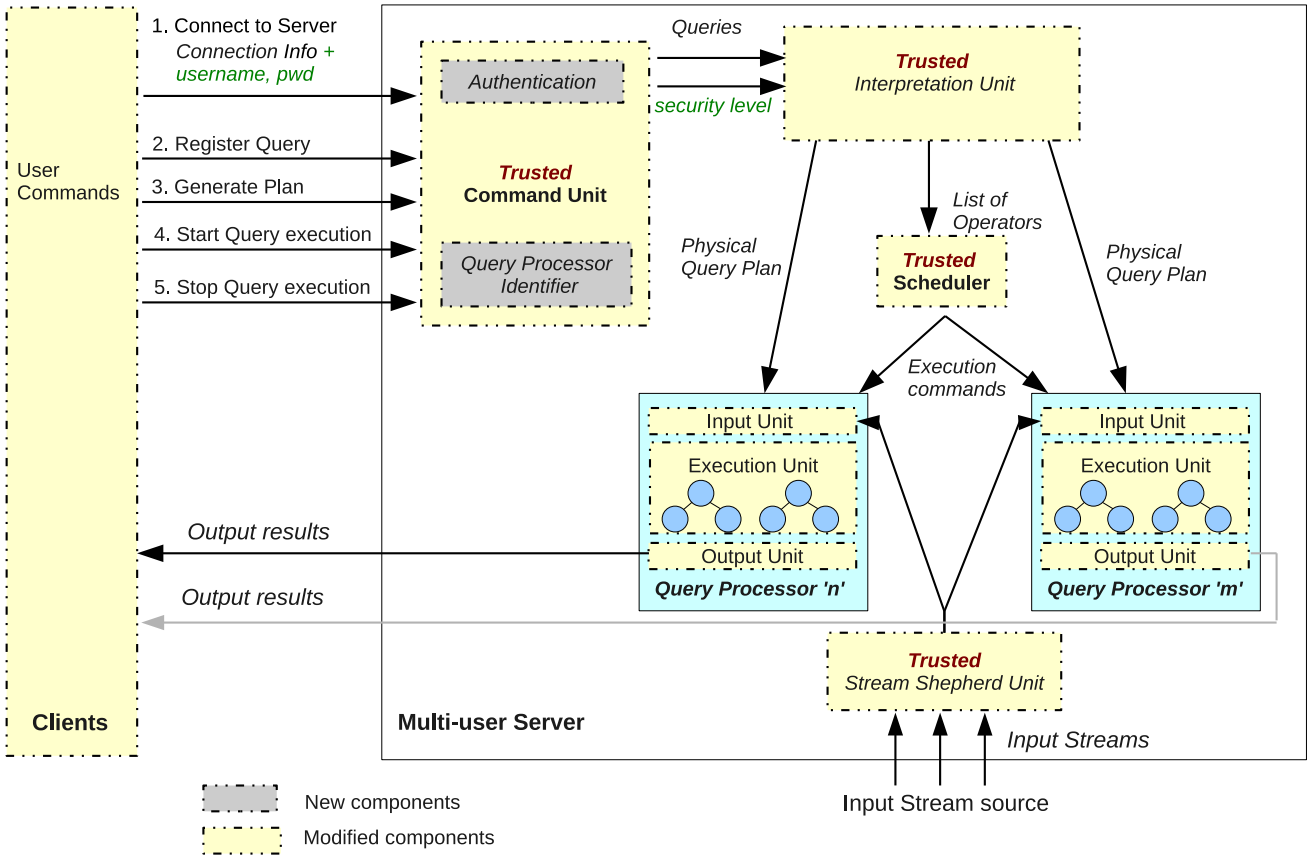

Figure 6: Prototype Architecture

ples before switching to other operators. The DSMS can process a maximum input 100,000 tuples per second. "First come first serve" strategy is used for executing the query plans. We adapt the round-robin method in our trusted scheduler which is able to schedule operators across all query processors.

6.1 Experiments

We conducted experiments in the prototype to compare the performance between the old vanilla DSMS and the redesigned CW-DSMS. The experiments were conducted in a system with Intel i7 Q820 1.73GHZ quad core processor, 6GB RAM, and Ubuntu 11.10 64 bit operating system. Except experiment 6 (join operation), for all other experiments, we used three different data sets with 2, 5, and 10 million tuples with a data input rate of 50,000 tuples per second. Each tuple is associated with a COI class in terms of [x,0] or [0,y], where x refers to company 1 or 2 and y can be one of the companies A, B, or C. 0 means the public knowledge $\perp$. For all experiments, the round robin method is used for operator scheduling. We ran each experiment five times and discarded the first two runs. The average execution time from the last three runs are shown in Table 1. We measured the query execution time (the time taken from first tuple entering the first operator of the query plan and last tuple exiting the query) for the following experiments.

1. **Experiment 1: Company auditing query 1 in level [1, $\perp$]** In order to maximize the difference in execution time, we used 100% selectivity (all tuples are in level [1, $\perp$]) on both the systems, so that no tuples are filtered by the select operator. As shown in Table 1 under Exp 1, the performance overhead due to security modification to the vanilla DSMS is negligible for all the data sets used, and it is between 0.003% and 0.025%.

2. **Experiment 2: Company auditing query 2 in level [1, $\perp$]** In query 2, we used 50% selectivity. The performance overhead is again negligible, and is between 0.002% and 0.017%.

3. **Experiment 3: Service auditing query 3 in level [$\perp$, B]** In the service auditing experiments 3 and 4, the input stream has tuples at 5 different levels: [1, $\perp$], [2, $\perp$], [$\perp$, A], [$\perp$, B] and [$\perp$, C]. Tuples in each level occupied 20% of the input stream. Since query 3 runs in level [$\perp$, B], only 20% tuples from inputs should be processed by query 3. So in the vanilla system we must include the condition based on security level in the query:

```
Vanilla:
 SELECT timestamp FROM MessageLog
```

Table 1: Performance Overhead of Chinese Wall Processing

	Data Size (tuples)	Average Execution Time (ms)		Overhead Due to CW-DSMS (in %)	Standard Deviation (ms)	
		Vanilla	CW-DSMS		Vanilla	CW-DSMS
Exp 1	2M	40031	40041	0.025%	2.52	2.52
	5M	100046	100055	0.009%	2.08	3.21
	10M	200057	200063	0.003%	3.21	2.52
Exp 2	2M	40032	40039	0.017%	2.52	1.53
	5M	100042	100049	0.007%	2.08	2.65
	10M	200052	200056	0.002%	3.51	3.51
Exp 3	2M	40030	40041	0.027%	2.89	1.73
	5M	100043	100057	0.014%	3.06	3.51
	10M	200054	200065	0.005%	2.08	3.06
Exp 4	2M	40044	40053	0.023%	3.51	2.65
	5M	100056	100065	0.009%	2.89	4.93
	10M	200062	200069	0.003%	1.53	2.52
Exp 5	2M	40047	40059	0.030%	2.52	3.51
	5M	100082	100099	0.018%	4.93	6.81
	10M	200094	200116	0.011%	4.58	6.24
Exp 6	100K	40532	40621	0.218%	11.36	30.35
	250K	100593	100726	0.132%	9.85	30.09
	500K	200628	200789	0.081%	14.47	33.86

```
 WHERE msgType = "send" AND outcome = "failure"
 AND receiver = "CompanyB" AND level = [0,B]

CW-DSMS [0,B]:
 SELECT timestamp FROM MessageLog
 WHERE msgType = "send" AND outcome = "failure"
 AND receiver = "CompanyB"
```

In CW-DSMS, unqualified tuples i.e., tuples not in level [⊥, B], are filtered by the trusted stream shepherd operator due to the replicated architecture. The performance overhead is between 0.005% and 0.027% for all data sets used, which is again negligible.

4. **Experiment 4: Service auditing query 4 in level [⊥, T]** Using the same input from experiment 3, the selectivity becomes 60% because level [⊥, T] is authorized to access inputs with levels [⊥, A], [⊥, B] and [⊥, C].

```
Vanilla:
 SELECT timestamp FROM MessageLog
 WHERE msgType = "send" AND outcome = "failure"
 AND (receiver = "CompanyB" OR receiver = "CompanyA"
 OR receiver = "CompanyC")
 AND (level = [0,A] OR level = [0,B] OR level = [0,C])

CW-DSMS [0,T]:
 SELECT timestamp FROM MessageLog
 WHERE msgType = "send" AND outcome = "failure"
 AND (receiver = "CompanyB" OR receiver = "CompanyA"
 OR receiver = "CompanyC")
```

The query language of CW-DSMS uses simplified form because of the replicated architecture. As shown in Table 1, the performance overhead is between 0.003% and 0.023%.

5. **Experiment 5: Cloud auditing query 5 in level [1, B]**: We studied the overhead caused by the extra least upper bound computations in CW-DSMS. The level of output tuple always reflects the highest possible level (COI class) of all the input tuples involved in the computation. Input tuples were either at [1,⊥] or [⊥,B]. To maximize the difference, we used 100% selectivity. The performance difference due to the LUB computation is between 0.011% and 0.030%.

6. **Experiment 6: Cloud auditing query 6 in level [1, B]** In the join query, input stream R and S refer to the same input stream source MessageLog. We set up 50% selectivity for R and S respectively. To activate LUB computation on join, input tuples were kept at either [1, ⊥] or [⊥, B] and streamed in a random fashion.

   Since join is an expensive operation, the input rate of 50,000 tuples used in the previous experiments caused an overload situation in both the systems. Thus, we reduced the data input rate to 2,500 tuples per second. Accordingly, the data sizes of the three input tuple sets were reduced to 100K, 250K, and 500K tuples, respectively. The performance overhead is between 0.081% and 0.218%, which is higher than the other experiments. On the other hand, the overhead is still considered negligible as it is within 0.218%.

Experiments Summary: As shown in Table 1 and discussed above, maximum overhead due to information flow constraints is 0.218% This demonstrates that the overhead due to addition of security constraints is negligble.

THEOREM 1. *The proposed architecture ensures information flow constraints.*

PROOF. Let Q be a query submitted by a process at level l that operates on the relations and streams in the DSMS. For each stream accessed by the query, the query rewriting operator takes into account the security level of the query and only provides the projection of the respective stream that the process is authorized to view. The query is then forwarded to the processor that executes in the same level as the query.

The query processor at level l can view only those input tuples whose levels are dominated by l and produce output streams at level l. Thus, during query processing overt information flow can only occur from levels dominated by level l to level l. In the proposed architecture, level l receives tuples from dominated levels and stores them at its own level. Thus, within the scope of our DSMS architecture there is no common storage that is shared across security levels. Thus, a dominating level cannot manipulate the common storage in our architecture to pass information to the dominated level. This ensures that there are no covert storage channels. The query processor at level l executes queries only in its allotted time slot as decided by the trusted scheduler which ensures that there are no timing channels. □

The above claim holds only when we consider the architecture in isolation. However, in real world this is never the case and it is possible for the underlying framework to have covert channels. For example, if the query processors at different levels are executing on the same server it is possible to have storage and/or timing channels.

7. RELATED WORK

In this section, we will discuss works from closely related areas: DSMS, DSMS security, and Chinese Wall policy.

Data Stream Management Systems (DSMSs): Most of the works carried out in DSMSs address various problems ranging from theoretical results to implementing comprehensive prototypes on how to handle data streams and produce near real-time response without affecting the quality of service. Some of the research prototypes include: Stanford STREAM Data Manager [8, 6], Aurora [9], Borealis [1, 17], and MavStream [19].

DSMS Sharing: In general DSMSs like STREAM [8, 6], Aurora [9], and Borealis [1, 17], queries issued by the same user at the same time can share the Seq-window operators and synopses. In STREAM system, base Seq-window operators are reused by queries on the identical streams. Instead building up sharing parts between plans, Aurora research focus on providing better execution scheduling of large number queries, by batching operators as atomic execution unit. In Borealis project, the information on input data criteria from executing queries can be shared and modified by new incoming queries. Here the execution of operators will be the same but the input data criteria can be revised. Even though many approaches target on better QoS in terms of scheduling and revising, sharing execution and computation among queries submitted at different times by the same user or at the same time between different users are not supported in general DSMS. Besides common source Seq-windows like regular DSMSs, sharing intermediate computation results is a better way to make big performance achievement.

DSMS Security: There has been several recent works on securing DSMSs [20, 13, 22, 12, 21, 4] by providing role-based access control. Though these systems support secure processing they do not prevent illegal information flows. Punctuation-based enforcement of RBAC over data streams is proposed in [22]. Access control policies are transmitted every time using one or more security punctuations before the actual data tuple is transmitted. 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. Secure shared continuous query processing is proposed in [4]. The authors present a three-stage framework to enforce access control without introducing special operators, rewriting query plans, or affecting QoS delivery mechanisms. Supporting role-based access control via query rewriting techniques is proposed in [13, 12]. 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 architecture proposed in [20] uses a post-query filter to enforce stream level access control policies. The filter applies security policies after query processing but before a user receives the results from the DSMS. Designing DSMS taking into account multilevel security constraints has been addressed by researchers [5].

Chinese Wall Policy: Brewer and Nash [11] first demonstrated how the Chinese Wall policy can be used to prevent consultants from accessing information belonging to multiple companies in the same conflict of interest class. However, the authors did not distinguish between human users and subjects that are processes running on behalf of users. Consequently, the model proposed is very restrictive as it allows a consultant to work for one company only. Sandhu [23] improves upon this model by making a clear distinction between users, principals, and subjects, defines a lattice-based security structure, and shows how the Chinese Wall policy complies with the Bell-Lapadula model [10].

Chinese Wall and Cloud Computing: Wu et al. [25] show how the Chinese Wall policy can be used for information flow control in cloud computing. The authors enforce the policies at the Infrastructure-as-a-Service layer. The authors developed a prototype to demonstrate the feasibility of their approach. In our current work, we have adapted the Chinese Wall policy and demonstrated how stream data generated from the various organizations can be processed in a secure manner. Our work is addressed at the Software-as-a-Service level. Tsai et al. [24] discusses how the Chinese Wall policy can be used to prevent competing organizations virtual machines to be placed on the same physical machine. Graph coloring is used for allocating virtual machines to physical machines such that the Chinese Wall policies are satisfied and better utilization of cloud resources is achieved. Jaeger et al. [18] argue that covert channels are inevitable and propose the notion of risk information flows that captures both overt and covert flows across two security levels. Capturing both covert flows and overt flows in a unified framework allows one to reason about the risks associated with information leakage.

8. CONCLUSIONS AND FUTURE WORK

Data streams generated by various organizations in a cloud may need to be analyzed in real-time for detecting critical events of interest. Processing of such data streams should be done in a careful and controlled manner such that company sensitive information is not disclosed to competing organizations. We propose a secure information flow control model, adapted from the Chinese Wall policy, to be used for protecting sensitive company information. We also provide architectures for executing continuous queries, such

that the information flow constraints are satisfied. We implemented a prototype to demonstrate the feasibility of our ideas.

A lot of work remains to be done. We have assumed that certain components are trusted. We have made similar assumptions about the underlying infrastructure. However, we have not explicitly stated our trust assumptions. We need to formally state and analyze these assumptions in view of real-world constraints in order to evaluate the security of our DSMS.

We plan to propose alternative architectures and do a comparative study to find out which approach is the most suitable for processing cloud streaming queries. We also plan to implement our query sharing ideas. Thus, when a new query is submitted, we need to check how plans for existing queries can be reused to improve the performance. Note that, such verification must be carried out dynamically. Towards this end, we plan to see how existing constraint solvers can be used to check for query equivalences. We also plan to evaluate the performance impact of dynamic plan generation and equivalence evaluation. We also plan to investigate more on how scheduling and load shedding can be done with information flow constraints.

Acknowledgement

This material is based on research sponsored in part by the Air Force Office of Scientific Research (AFOSR), under agreement number FA-9550-09-1-0409.

9. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A.Rasin, E.Ryvkina, N.Tatbul, Y.Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *Proc. of the CIDR*, pages 277–289, 2005.

[2] M. D. Abrams, S. G. Jajodia, and H. J. Podell, editors. *Information Security: An Integrated Collection of Essays*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1995.

[3] R. Adaikkalavan and S. Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. *DKE*, 59(1):139–165, 2006.

[4] R. Adaikkalavan and T. Perez. Secure Shared Continuous Query Processing. In *Proc. of the ACM SAC (Data Streams Track)*, pages 1005–1011, Taiwan, Mar. 2011.

[5] R. Adaikkalavan, I. Ray, and X. Xie. Multilevel Secure Data Stream Processing. In *Proc. of the DBSec*, pages 122–137, July 2011.

[6] 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. Technical Report 2004-20, Stanford InfoLab, 2004.

[7] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB Journal*, 15(2):121–142, 2006.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the PODS*, pages 1–16, June 2002.

[9] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. B. Zdonik. Retrospective on aurora. *VLDB Journal: Special Issue on Data Stream Processing*, 13(4):370–383, 2004.

[10] D. E. Bell and L. J. LaPadula. Secure Computer System: Unified Exposition and MULTICS Interpretation. Technical Report MTR-2997 Rev. 1 and ESD-TR-75-306, rev. 1, The MITRE Corporation, Bedford, MA 01730, Mar. 1976.

[11] D. F. C. Brewer and M. J. Nash. The Chinese Wall Security Policy. In *Proc. of the IEEE S & P*, pages 206–214, May 1989.

[12] J. Cao, B. Carminati, E. Ferrari, and K. Tan. Acstream: Enforcing access control over data streams. In *Proc. of the ICDE*, pages 1495–1498, 2009.

[13] B. Carminati, E. Ferrari, and K. L. Tan. Enforcing access control over data streams. In *Proc. of the ACM SACMAT*, pages 21–30, 2007.

[14] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *Proc. of the VLDB*, pages 215–226, August 2002.

[15] S. Chakravarthy and R. Adaikkalavan. Event and Streams: Harnessing and Unleashing Their Synergy. In *Proc. of the DEBS*, pages 1–12, July 2008.

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

[17] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *Proc. of the CIDR*, 2003.

[18] T. Jaeger, R. Sailer, and Y. Sreenivasan. Managing the Risk of Covert Information Flows in Virtual Machine Systems. In *Proc. of the ACM SACMAT*, pages 81–90, 2007.

[19] Q. Jiang and S. Chakravarthy. Anatomy of a Data Stream Management System. In *ADBIS Research Communications*, 2006.

[20] W. Lindner and J. Meier. Securing the borealis data stream engine. In *IDEAS*, pages 137–147, 2006.

[21] R. V. Nehme, H. Lim, E. Bertino, and E. A. Rundensteiner. StreamShield: A Stream-Centric Approach towards Security and Privacy in Data Stream Environments. In *Proc. of the ACM SIGMOD*, pages 1027–1030, 2009.

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

[23] R. Sandhu. Lattice-Based Enforcement of Chinese Walls. *Computers & Security*, 11(8):753–763, 1992.

[24] T. Tsai, Y. Chen, H. Huang, P. Huang, and K. Chou. A Practical Chinese Wall Security Model in Cloud Computing. In *Proc. of the APNOMS*, pages 1–4, September 2011.

[25] R. Wu, G. Ahn, H. Hu, and M. Singhal. Information flow control in cloud computing. In *Proc. of the CollaborateCom*, pages 1–7, October 2010.

[26] R. Xie and R. Gamble. A Tiered Strategy for Auditing in the Cloud. In *IEEE International Conference on Cloud Computing*, June 2012.