Multilevel Secure Data Stream Processing

Raman Adaikkalavan, Indrakshi Ray, Xing Xie

Multilevel Secure Data Stream Processing

Raman Adaikkalavan[1] ⋆, Indrakshi Ray[2], and Xing Xie[2]

[1] Computer and Information Science, Indiana University South Bend
raman@cs.iusb.edu
[2] Computer Science, Colorado State University
{iray, xing}@cs.colostate.edu

Abstract. With sensors and mobile devices becoming ubiquitous, situation monitoring applications are becoming a reality. Data Stream Management Systems (DSMSs) have been proposed to address the data processing needs of such applications that require collection of high-speed data, computing results on-the-fly, and taking actions in real-time. Although a lot of work appears in the area of DSMS, not much has been done in multilevel secure (MLS) DSMS making the technology unsuitable for highly sensitive applications such as battlefield monitoring. An MLS DSMS should ensure the absence of illegal information flow in a DSMS and more importantly provide the performance needed to handle continuous queries. We investigate the issues important in an MLS DSMS and propose an architecture that best meets the goals of MLS DSMS. We discuss how continuous queries can be executed in such a system and sharing across queries accomplished for maximum performance benefits.

Keywords: Multilevel Security, DSMS, Continuous Query Processing

1 Introduction

With the advancement of smart technologies and ubiquitous availability of sensor and mobile devices, situation monitoring applications are becoming a reality. Such applications require collecting high-speed data, processing them, computing results on-the-fly, and taking actions in real-time. Data Stream Management Systems (DSMSs) [7, 14, 4, 9, 1, 5, 16] have been proposed for such applications that allow processing of streaming data and execution of continuous queries. One potential use of this technology is for military applications where DSMS receives information from various devices and sensors, not all of which belong to the same security level. In such applications, users and information are classified into the various security levels and mandatory rules govern the information flow across security levels. DSMSs need to execute queries based on live streaming data classified at various levels in response to request from users at different security levels without causing illegal information flow. Our work attempts to extend an existing DSMS to support such capabilities.

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 [15, 11] to streaming data [21, 13, 22, 12, 3]. However, controlling access is not enough to

⋆ This work was supported, in part, by IU South Bend Research Grant.

prevent security breaches in the above mentioned applications where illegal information flow can occur across security levels. For instance, the existence of covert and overt channels can cause information to be passed from a more sensitive level to a lesser one. Multilevel security (MLS) not only prevents unauthorized access but also ensures the absence of such illegal information flow.

Designing an MLS DSMS requires us to address several research issues. We need to provide a continuous query language for expressing real-world MLS DSMS queries. The formalization of such a language will allow us to determine query equivalence and facilitate query optimization. Note that, traditional notions of query equivalence will not work because the same query issued by users at different security levels will return different results. Moreover, query processing should be efficient to meet the QoS requirements of a DSMS. This necessitates sharing query plans of multiple queries to reduce query execution time without causing illegal information flow. In order to process MLS continuous queries in a secure manner, it is therefore necessary to completely redesign or make major modifications to the components of a DSMS.

In this work, we propose a suitable architecture for processing MLS continuous queries. We also formalize MLS continuous query processing and discuss how such queries can be executed in our proposed architecture. We discuss how query plans can reuse plans from existing queries. We augment the approaches proposed by the Stanford STREAM [4], Aurora [9], and Borealis [1] projects and allow sharing of query plans submitted by different users not all of which have been submitted at the same time. This not only allows good resource utilization but also helps achieve the quality-of-service (QoS) critical to stream processing applications.

The rest of the paper is organized as follows. In Section 2, we define a MLS formalization model for stream data applications where data sources, data streams, queries, and other components in DSMS are assigned with security levels with proper accessing rules. In Section 3, we propose a replicated architecture to address MLS stream applications. In order to accelerate processing rates, we explore different sharing approaches between continuous queries in Section 4. We discuss related work in section 5. In Section 6, conclusions and future work are discussed.

2 Multilevel Security Formalization Model

We begin by presenting our *model* for multilevel secure (MLS) DSMS system. An MLS DSMS is associated with a security structure that is a partial order, $(\mathbf{L}, <)$. $\mathbf{L}$ is a set of security levels, and $<$ is the dominance relation between levels. If $L_1 < L_2$, then L_2 is said to strictly dominate L_1 and L_1 is said to be strictly dominated by L_2. If $L_1 = L_2$, then the two levels are said to be equal. $L_1 < L_2$ or $L_1 = L_2$ is denoted by $L_1 \leq L_2$. If $L_1 \leq L_2$, then L_2 is said to dominate L_1 and L_1 is said to be dominated by L_2. Two levels L_1 and L_2 are said to be incomparable if neither $L_1 \leq L_2$ nor $L_2 \leq L_1$. We assume the existence of a level U, that corresponds to the level unclassified or public knowledge. The level U is the greatest lower bound of all the levels in $\mathbf{L}$. Any data object classified at level U is accessible to all the users of the MLS DSMS. Each MLS DSMS object $x \in \mathbf{D}$ is associated with exactly one security level which we denote as $L(x)$ where $L(x) \in \mathbf{L}$.

(The function L maps entities to security levels.) We assume that the security level of an object remains fixed for the entire lifetime of the object.

The users of the system are cleared to different security levels. We denote the *security clearance* of user U_i by $L(U_i)$. Consider a setting consisting of two security levels: High (H) and Low (L), where $L < H$. The user Jane Doe has the security clearance of High. That is, $L(JaneDoe) = H$. Each user has one or more associated *principals*. The number of principals associated with the user depends on their security clearance; it equals the number of levels dominated by the user's security clearance. In our example Jane Doe has two principals: *JaneDoe.H* and *JaneDoe.L*. During each session, the user logs in as one of the principals. All processes that the user initiates in that session inherit security level of the corresponding principal.

Each continuous query Q_i is associated with exactly one security level. The level of the query remains fixed for the entire execution. The security level of the query is the level of the principal who has submitted the query. For example, if Jane Doe logs in as *JaneDoe.L*, all queries initiated by Jane Doe during that session will have the level Low (L). A continuous query Q_i consists of one or more operators OP_i, where the operators inherit the level of the query. We require a query Q_i to obey the simple security property and the restricted $\star$-property of the Bell-LaPadula model [10].

1. An operator OP_i with $L(OP_i) = C$ can read an object x only if $L(x) \leq C$.
2. An operator OP_i with $L(OP_i) = C$ can write an object x only if $L(x) = C$.

In general, multilevel security can be supported at three *granularities*: attribute, tuple, or stream. Though stream level enforcement (i.e., single level streams within the DSMS) may be the easiest way of supporting multilevel security, it does not work for many MLS applications. We have analyzed stream applications from various domains (e.g., battlefield monitoring, infrastructure security). In such applications, streams containing tuples having different levels are often input to the DSMS. Thus, providing stream level security would not be beneficial to such applications. In this research work, we do security enforcement at tuple level (i.e., we assign level to each tuple). Thus, we do not consider the security level of the attributes individually, in this paper.

We do not present a separate attack model in this paper. Like all MLS systems, 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.

3 Multilevel Stream Processing Architecture

In this section, we begin by discussing a general DSMS architecture and describe how it can be adapted to process MLS continuous queries.

3.1 General DSMS Architecture

A typical DSMS [7, 14, 16] architecture (based on the STREAM system [4]) is shown in Figure 1. A Continuous Query (CQ) can be defined using specification languages [5], or as query plans [14]. 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, Project, Join, Aggregate). Each operator is associated with one or more input *queues*[3] and an output queue. One or more *synopses*[4] [5] 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 to the ready state so that they can be executed. Based on a scheduling strategy (e.g., round robin) [16, 6], the scheduler picks a query, an operator, or a path, and starts the execution. The run-time optimizer monitors the system, and initiates load shedding [16, 25, 8] as and when required. Both these QoS delivery mechanisms minimize resource usage (e.g., queue size) and maximize performance and throughput. Each stream has a stream shepherd operator in the DSMS which handles all the tuples arriving in that stream. Seq window operator reads the tuples from the shepherd operator and propagates to leaf nodes of queries. This operator is shared by all the queries that use that stream. In the directed graph of operators, the 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).

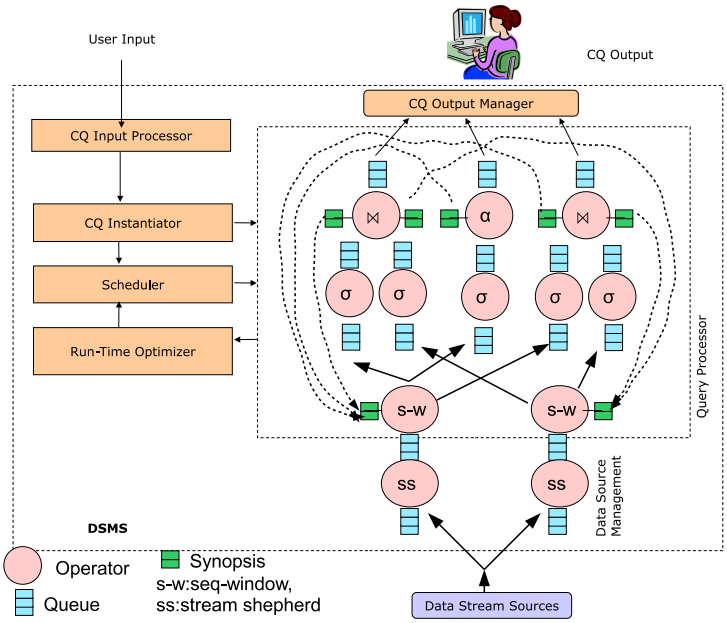

Fig. 1. Data Stream Management System (DSMS)

[3] Queues are used by the operators to propagate tuples.

[4] Synopses are temporary storage structures used by the operators (e.g., Join) that need to maintain a state. In this paper, we use synopses and *windows*, alternatively.

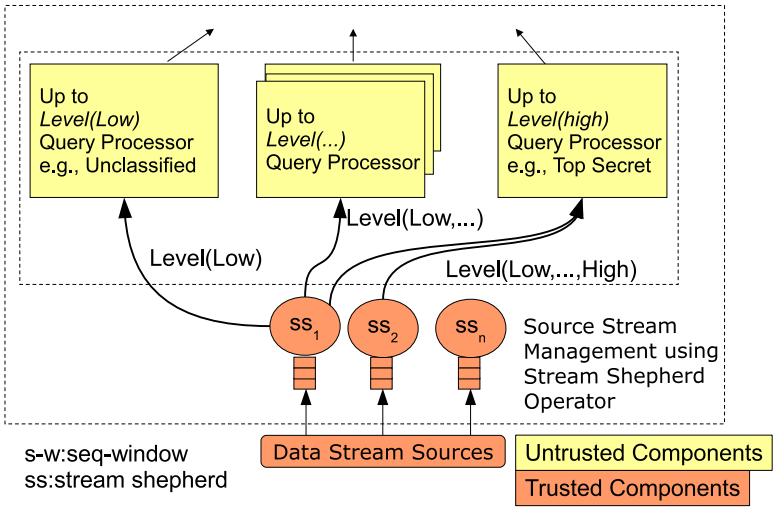

Fig. 2. Replicated MLS DSMS Architecture

3.2 MLS DSMS Architecture

In this section, we discuss how we can adapt the general DSMS architecture to process MLS continuous queries. We focus our attention to the query processor component of the architecture presented in Figure 1. The query processor of an MLS DSMS can have various types of architecture depending on how logical isolation is achieved across the different security levels. We borrow our ideas in this regard from the various architectures (trusted, kernelized, and replicated) that have been proposed in MLS DBMS literature [15, 18, 2]. We choose the replicated architecture as the first step and plan to propose other alternatives as part of our future work.

Our architecture is based on the replicated model where each level L stores not only the tuples with classification L but also those whose classification is dominated by L. We present one example of a replicated query processor in Figure 2, although many variations are possible.

The query processors are untrusted and replicated at various security levels. Each query processor runs at a security level (L) and is responsible for executing queries submitted by the users who have logged on at the same level. The response to a query may involve data belonging to one or multiple security levels; however, the level of all the tuples returned in the response must be dominated by the query level.

The stream shepherd operator must be redefined to ensure that only tuples at the dominated level are passed on to the dominating level. All the other operators are untrusted and are replicated at various levels. The input queues carrying data at dominated levels are replicated at the dominating levels as well. Sequential-Window operators and synopses used for processing blocking operators such as join and aggregation are created as needed for the query processors at that level. In the next section, we discuss query processing in more details.

4 Shared Query Processing in Replicated DSMS

In this section, first we discuss MLS CQL queries informally, and then discuss shared query processing.

4.1 MLS CQL Queries

Consider the following data streams (Vitals and Position) and continuous query Q written using the CQL language [5]. Query Q joins tuples from two streams. The sliding windows maintain the last 100 tuples for computations.

```
Vitals (soldier id (sid), blood pressure (bp), pulse rate (pr));
Position (soldier id (sid), latitude (lat), longitude (lon));

Q: SELECT AVG(bp), AVG(pr) FROM Vitals[ROWS 100], Position[ROWS 100]
   WHERE Vitals.sid = Position.sid
```

To support MLS, stream and query definitions have to be modified to include security levels. Below, we discuss MLS CQL briefly as a complete discussion is outside the scope of this paper. An MLS CQL query can include the LEVEL attribute in the WHERE clause, SELECT clause, and window specification. Let us consider the following examples.

```
SELECT AVG(bp) WHERE LEVEL = "S" FROM Vitals [ROWS 100]
SELECT AVG(bp) FROM Vitals [ROWS 100 LEVEL = "S"]
SELECT AVG(bp) FROM Vitals [ROWS 100] WHERE LEVEL = "S"
```

In the first query the WHERE clause conditions are applied before a tuple enters a window. In the second query, the window keeps only tuples based on the condition specified. In the third query, the window maintains 100 tuples, but the WHERE clause is applied during AVG calculation. The first and second queries are equivalent. Note that, for these queries, we have simple selections and we do not have any join conditions. If the WHERE clause specifies a join condition, this condition can only be checked in the join operator which is processed after the window selection. Our algorithms, presented in this paper, address all three types of queries. However, due to space constraints, our examples are based on the first type of query which processes the WHERE conditions except the join condition before window selection.

We consider only tuple-based (e.g., query Q) and partitioned by windows [5]. In the query shown below, the partitioned window maintains two different partitions (as it gets only tuples with level S or TS), and the average is calculated for each partition.

```
SELECT AVG(bp) WHERE LEVEL = "S" OR "TS"
FROM Vitals [PARTITIONED BY LEVEL ROWS 100]
```

Processing each MLS query involves several steps. First, the selection condition of the query is written in conjunctive normal form. Second, the query must be rewritten to add a where clause that says the level of tuples returned must be dominated by the level of the user. Subsequently, we generate the query plan. In this work, we represent a query plan in the form of a tree which we refer to as an *operator tree*. Note that, many operator

Table 1. Continuous Queries

Query	User	Login Level	Query Specification
Q_1/Q_1'	Ann/Bob	*H*	SELECT AVG(bp) FROM Vitals [PARTITIONED BY LEVEL ROWS 20]
Q_2	Carl	*H*	SELECT AVG(bp) WHERE LEVEL = "L" FROM Vitals [ROWS 20]
Q_3	Dan	*H*	SELECT AVG(bp) WHERE bp > 50 FROM Vitals [PARTITIONED BY LEVEL ROWS 5]
Q_4	Dan	*H*	SELECT AVG(pr) WHERE V.sid = P.sid AND bp > 120 AND lon = "4E" FROM Vitals [ROWS 10] V, Position [ROWS 10] P
Q_5	Ellen	*H*	SELECT V.sid, pr WHERE V.sid = P.sid AND bp > 120 AND lon ="4E" FROM Vitals [ROWS 10] V, Position [ROWS 10] P
Q_6	Frank	*H*	SELECT sid, bp WHERE bp > 120 FROM Vitals
Q_7	Gail	*H*	SELECT sid, bp, pr WHERE LEVEL = "L" AND bp > 120 FROM Vitals
Q_8	John	*H*	SELECT sid WHERE pr > 100 FROM Vitals

trees may be associated with a query corresponding to the different plans. However, we show just one such tree for each query. The formal definition of an operator tree appears below.

Definition 1. *[Operator Tree] An* operator tree *for a query Q_x is 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 node N_i corresponds to some operator in the query Q_x. Each edge (i, j) in this tree connecting node N_i with node N_j signifies that the output of node N_i is the input to node N_j. Each node N_i is labeled with the name of the operator $N_i.op$, its parameters $N_i.parm$, the synopses $N_i.syn$ (for blocking operators), and input queues $N_i.inputQueue$ which are used for its computation. The label of node N_i also includes the output produced by the node, denoted by $N_i.outputQueue$, that can be used by other nodes or sent as response to the users.*

Operator trees for queries Q_6 and Q_7 defined in Table 1 appear in Figures 3(a) and 3(b), respectively. An operator tree has all the information needed for processing the query. Specifically, the labels on the node indicate how the computation is to be done for evaluating that operator, where an operator is the basic unit of data processing in a DSMS. The name component specifies the type of the operator, such as, *SELECT*, *PROJECT*, *AVG*, etc. The parameter is denoted as a set. For the *SELECT* operator, parameter is the set of conjuncts in the selection condition. For the *PROJECT* operator it is the set of attributes. The synopsis is needed for the blocking operators, such as, join and aggregate operations and has type (e.g., tuple-based, partitioned by) and size as its attributes. The input queues are derived from the streams (or relations) needed by the operator.

We use the streams (Vitals and Position) and continuous queries shown in Table 1 to discuss query processing. We also assume the tuples sent by soldiers involved in a highly classified mission to be classified as high (H) and other missions to be classified as low (L). Medics or users can login in at different levels and submit queries. Also note that in Table 1 all queries are issued in high (H) level. The main reason to choose one level is that all queries issued by a user logged in at that level is processed by a query processor running at that level. Hence we use examples from H level to introduce and discuss various sharing methods. All these queries are executed by one query processor at level high, shown in Figure 2.

Queries Q_1 and Q_1', issued by Ann and Bob respectively, compute the average blood pressure of the last 20 tuples at each level in Vitals stream. Query Q_2 computes the average blood pressure of the last 20 tuples having level L. Query Q_3 computes the average blood pressure for the last 5 tuples at each level where the pressure is greater than 50. In queries Q_4 and Q_5, the last 10 tuples that satisfy the selection conditions are maintained in the synopses and are joined. Average and projection are computed over the results from the join. In queries Q_6 to Q_8, there are only selection conditions and projection (duplicate preserving) operations. Query Q_7 selects level L tuples that have $bp > 120$ and projects three attributes.

4.2 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. 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. Figure 3(c) shows the merging of operator trees of queries Q_6 and Q_7 shown in Figures 3(a) and 3(b), respectively. In the Figure 4, we show how the operator trees of Q_4 and Q_5 can be merged. Later we will formalize how such sharing can be done.

In our replicated MLS DSMS query processing architecture, we focus on sharing queries to save resources such as CPU cycles and memory usage. In our architecture, we share queries that are submitted by users with the same principal security level as all these queries run in the same query processor. Since queries shared have the same security level, our replicated MLS DSMS query processor avoids security violations like covert channel during sharing.

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 nodes belonging to different operator trees are equivalent, then only one node needs to be computing for evaluating the queries corresponding to these different operator trees.

Definition 2. *[Equivalence of 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 some cases, for evaluating node N_i belonging to operator tree $OPT(Q_x)$, we may be able to reuse the results of evaluating node N_j belonging to operator tree $OPT(Q_y)$.

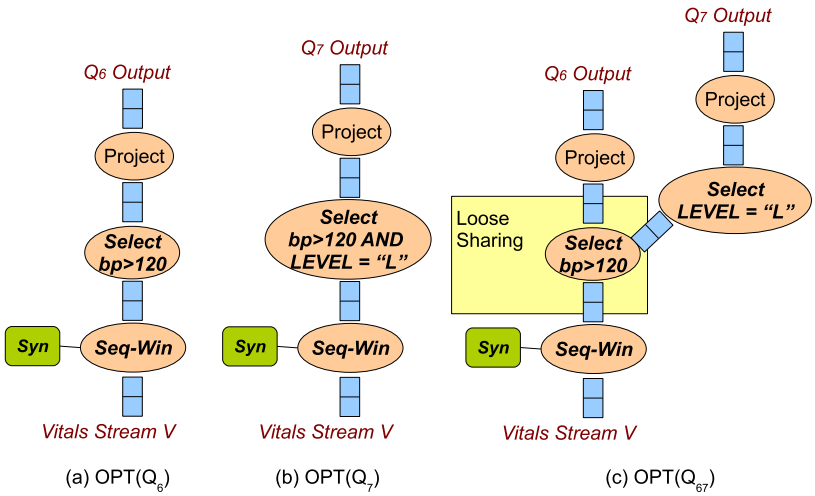

Fig. 3. Operator Tree for Q_6, Q_7, and Loose Partial Sharing of Q_6 and Q_7

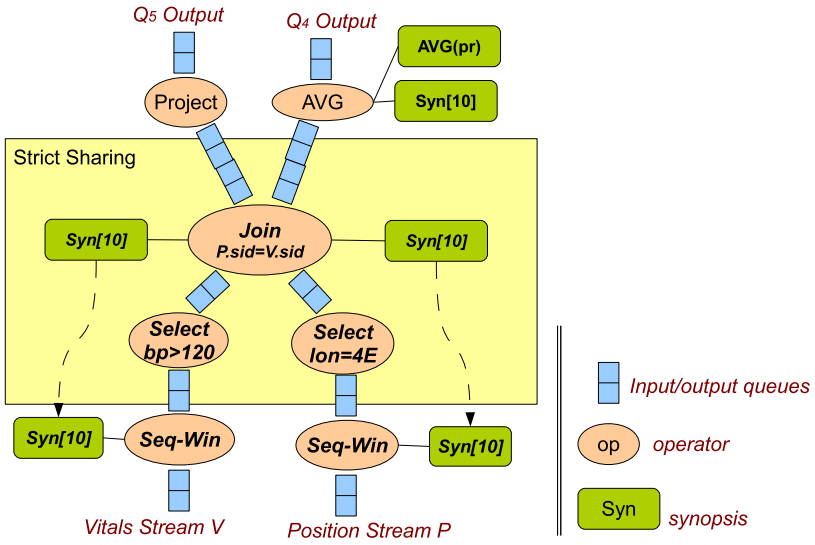

Fig. 4. Strict Partial Sharing Operator Tree for Q_4 and Q_5

This is possible if the nodes are related by the subsumes relationship defined below. Such relationship is possible when the operators match and are non-blocking and the operator parameters are related by a subset relation.

Definition 3. *[Subsume Relation of Nodes] Node $N_i \in N_{Q_x}$ is said to be* subsumed *by node $N_j \in N_{Q_y}$, denoted by $N_i \subseteq N_j$, where N_i, N_j are in the operator trees $OPT(Q_x)$, $OPT(Q_y)$ and are referred to as* subsumed node, subsuming node *respectively, if the following conditions hold:*

1. *Condition 1:*
   - *Case 1 [$N_i.op = PROJECT$]:*
     $N_i.op = N_j.op \wedge N_i.parm \subseteq N_j.parm \wedge N_i.inputQueue = N_j.inputQueue$.
   - *Case 2 [$N_i.op = SELECT$]:*
     $N_i.op = N_j.op \wedge N_j.parm \subseteq N_i.parm \wedge N_i.inputQueue = N_j.inputQueue$.
2. *Condition 2: $N_i.op$ is a non-blocking operator.*

Consider the SELECT nodes of the operator trees of queries Q_6 and Q_7 shown in Figure 3, where the SELECT node of Q_7 is subsumed by the SELECT node of Q_6. We have different forms of sharing that are possible in our architecture which we now discuss.

Complete Sharing

The best form of sharing is complete sharing where no additional work is needed for processing a new query. However, in order to have complete sharing, the two queries must have equivalent operator trees. The notion of equivalence of operator trees is given below.

Definition 4. *[Equivalence of Operator Trees] Two operator trees $OPT(Q_x)$ and $OPT(Q_y)$ are said to be equivalent, denoted by $OPT(Q_x) \equiv OPT(Q_y)$ if the following conditions hold.*

1. *for each node $N_i \in N_{Q_x}$, there exists a node $N_j \in N_{Q_y}$, such that $N_i \equiv N_j$.*
2. *for each node $N_p \in N_{Q_y}$, there exists a node $N_r \in N_{Q_x}$, such that $N_p \equiv N_r$.*

The formal definition of complete sharing appears below.

Definition 5. *[Complete Sharing] Query Q_x can be* completely shared *with an ongoing query Q_y submitted by a user at the same security level only if $OPT(Q_i) \equiv OPT(Q_j)$.*

Complete sharing is possible only when the queries are equivalent. For example, queries Q_1 and Q_1' have identical operator trees and can be completely shared. In such cases, we do not need to do anything else for processing the new query. However, this may not happen often in practice.

Partial Sharing

We next define partial sharing which allows multiple queries to share the processing of one or more nodes, if they are related by the equivalence or subsume relation.

Definition 6. *[Partial Sharing] Query Q_x can be* partially shared *with an ongoing query Q_y submitted at the same security level only if the following conditions hold*

1. $OPT(Q_x) \not\equiv OPT(Q_y)$
2. *there exists $N_i \in N_{Q_x}$ and $N_j \in N_{Q_y}$, such that one of the following holds: $N_i \equiv N_j$, $N_i \subseteq N_j$ or $N_j \subseteq N_i$.*

We have two forms of partial sharing which we describe below. The main motivation is the sharing of blocking operators have to be handled differently from non-blocking operators. The sharing of blocking operators is more restrictive in which the conditions for join operator, for example, must exactly match the other query's join operator. On the other hand, with non-blocking operators they can be subsumed. The formal definition of these two forms of sharing appears below.

Definition 7. *[Strict Partial Sharing] Query Q_x can be* strict partially shared *with an ongoing query Q_y submitted at the same security level only if the following conditions hold*

1. $OPT(Q_x) \not\equiv OPT(Q_y)$
2. *there exists $N_i \in N_{Q_x}$ and $N_j \in N_{Q_y}$, such that $N_i \equiv N_j$*
3. *there does not exist $N_i \in N_{Q_x}$ and $N_j \in N_{Q_y}$, such that $N_i \subseteq N_j$ or $N_j \subseteq N_i$.*

Definition 8. *[Loose Partial Sharing] Query Q_x can be* loose partially shared *with an ongoing query Q_y submitted at the same security level only if the following conditions hold*

1. $OPT(Q_x) \not\equiv OPT(Q_y)$
2. *there exists $N_i \in N_{Q_x}$ and $N_j \in N_{Q_y}$, such that $N_i \subseteq N_j$.*

In the loose partial sharing, we will have a node on the ongoing query that subsumes a node of an incoming query. When nodes are related by subsume relation, then it is possible to decompose the subsumed nodes. The decomposition tries to make use of operator evaluation of the subsuming node in order to evaluate the subsumed node. The decomposition is formalized below.

Definition 9. *[Decomposition of Subsumed Nodes] Let $N_i \subseteq N_j$ where $N_i \in OPT(Q_x)$ and $N_j \in OPT(Q_y)$. Node N_i can be decomposed into two nodes N_i' and N_i'' in the following manner.*

Node N_i'

1. $N_i'.op = N_j.op$
2. $N_i'.inputQueue = N_j.inputQueue$
3. $N_i'.parm = N_j.parm$

Node N_i''

1. $N_i''.op = N_i.op$
2. $N_i''.inputQueue = N_i'.outputQueue$
3. $N_i''.parm = N_i.parm - N_i'.parm(if\ N_i.op = SELECT)$
   $N_i''.parm = N_i'.parm - N_i.parm(if\ N_i.op = PROJECT)$

Consider the SELECT nodes of the operator trees of query Q_6 and Q_7 shown in Figure 3. In this case, the SELECT node of Q_7 is subsumed by the SELECT node of Q_6. Select node of Q_7 which is the subsumed by the select node of Q_6 can be decomposed into two select nodes. One of these new nodes mirror Q_6 and the other is also a select node that checks for the additional select condition. Partial sharing is possible because of the overlap of operator trees.

Definition 10. *[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 1: Merge Operator Trees

INPUT: $OPT(Q_x)$ and $OPT(Q_y)$
OUTPUT: $OPT(Q_{xy})$ representing the merged operator tree
Initialize $N_{Q_{xy}} = \{\}$
Initialize $E_{Q_{xy}} = \{\}$
foreach *node* $N_i \in N_{Q_x}$ **do**
  $N_{Q_{xy}} = N_{Q_{xy}} \cup N_i$
end
foreach *edge* $(i, j) \in E_{Q_x}$ **do**
  $E_{Q_{xy}} = E_{Q_{xy}} \cup edge\ (i, j)$
end
foreach *node* $N_i \in N_{Q_y}$ **do**
  if $\nexists N_j \in N_{Q_x}$ *such that* $N_i \equiv N_j$ **then**
    $N_{Q_{xy}} = N_{Q_{xy}} \cup N_i$
  end
end
foreach *edge* $(i, j) \in E_{Q_y}$ **do**
  if *edge* $(i, j) \notin E_{Q_{xy}}$ **then**
    $E_{Q_{xy}} = E_{Q_{xy}} \cup edge\ (i, j)$
  end
end

When operator trees corresponding to two queries overlap, we can generate the merged operator tree using Algorithm 1. The merged operator tree signifies the processing of the partially shared queries.

Figure 4 illustrates the strict sharing of $OPT(Q_4)$ and $OPT(Q_5)$. As shown, we share select and join operators. The result of the join is processed by duplicate preserving project and aggregation operators. On the other hand, seq-window operator is common to all queries using a stream. Figures 3 (a) and (b) show the $OPT(Q_6)$ and $OPT(Q_7)$, respectively. Figure 3 (c) illustrates the $OPT(Q_{67})$ which shares both the query operations using the loose partial sharing approach. In this case, the query Q_7 is subsumed by Q_6 according to subsume relation definition. Based on Definition 9 (decomposition of subsumed nodes), we split Q_7 select condition into two ($bp > 120$ and level = "L") nodes and then share the $bp > 120$ node with Q_6.

5 Related Work

Though there has been a lot of research on multilevel security, to the best of our knowledge, ours is the first work in multilevel secure data stream processing systems. In this section, we will discuss works from closely related areas: DSMS, DSMS security, and MLS in real-time systems.

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. There have been lots of works on developing QoS delivery mechanisms such as scheduling strategies [16, 6] and load shedding techniques [16, 25, 8]. Some of the research prototypes include: Stanford STREAM Data Manager [7, 4], Aurora [9], Borealis [1, 17], and MavStream [20].

DSMS Sharing: In general DSMSs like STREAM [7, 4], 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 the STREAM system, Seq-window operators are reused by queries. Instead of sharing plans, Aurora research focus on providing better scheduling of large number queries, by batching operators as atomic execution unit. In the Borealis project, 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 sharing common source Seq-window operators, sharing intermediate computations will result in big performance gains.

DSMS Security: There has been several recent works on securing DSMSs [21, 13, 22, 12, 3] by providing role-based access control. Though these systems support secure processing they do not prevent illegal information flows. In addition, in MLS systems we need to classify each component of the DSMS as opposed to access control support. 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 [3]. 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 [21] 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.

MLS in Real Time Systems: In MLS real-time database system, research focuses on designing a DBMS where transactions having timing constraint deadlines executes

in serialization order without data conflicts and security violations. Issues like security breach and task scheduling are similar to our MLS DSMS. Covert channel issues must be addressed due to sharing data among transactions from different levels in real-time DBMS. Many concurrent control protocols, like 2PL high priority, OPT-Sacrifice, and OPT-WAIT [19], deal with the high level transactions by suspending or restarting them if they conflict with low level transactions. However, the starvation on high level transactions becomes serious if there are too many conflicts in the system. S2PL [24] provides a better way on balancing the security and performance among conflicting transactions: high level transactions should wait for the commit of conflicting low level transactions only once then executed. Real-time DBMSs also need proper scheduling strategy in order to satisfy the various transaction deadlines. There are many priority selection algorithms like arrival timestamp, early-deadline-first, least-slack-time-first, etc [23], which impact the scheduling strategies in DSMS research. Although a large number of theories have been proposed on real-time system design, we cannot use them directly into MLS DSMS because of the differences between real-time and data stream systems. For the execution unit in the system, real-time DBMS uses transient transactions while DSMS handles continuous queries. In order to cause a security breach, transactions might set up inference or covert channel via accessing the same data item while continuous queries try to manipulate the response time. Scheduling strategy in MLS real-time transaction processing must address security, serialization and transaction deadlines, whereas scheduling in CQ must address security and query response time and throughput.

6 Conclusions and Future work

Data Stream Management Systems (DSMSs) have been developed to address the data processing needs of situation monitoring applications. However, many situation monitoring applications, such as battlefield monitoring, emergency threat and resource management, involve data that are classified at various security levels. Existing DSMSs must be redesigned to ensure that illegal information flow do not occur in such applications. Towards this end, we developed an architecture for MLS DSMS and showed how MLS continuous queries can be executed in such systems. We have also shown how query plans can be shared across queries submitted by possibly different users to maximize resource utilization and improve performance. Our approach does not have security violations and can be used to process MLS data streams.

We plan to implement a prototype and study the overhead that is being caused due to MLS processing. We plan to investigate MLS DSMS query processing for kernelized and trusted architectures as well and develop prototypes. In the trusted architecture, it may be possible to share query plans across security levels and the performance improved. We plan to do a comparative study of the different architectures to find out which approach is the most suitable for processing MLS DSMS queries.

Currently, we have used simple extensions to CQL to express MLS continuous queries. In future, we plan to extend CQL completely so that we can express more complex MLS continuous queries. In our work, when a user submits a query, we check whether the plans for the 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 building other components such as scheduling and load shedding for MLS DSMS.

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 T. Perez. Secure Shared Continuous Query Processing. In *Proc. of the ACM SAC (Data Streams Track)*, pages 1005–1011, Taiwan, Mar. 2011.
4. 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.
5. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB Journal*, 15(2):121–142, 2006.
6. B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *VLDB Journal*, 13(4):333–353, 2004.
7. 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.
8. B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proc. of the ICDE*, pages 350–361, March 2004.
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. M. Bishop. *Computer Security: Art and Science*. Addison-Wesley, December 2002.
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. Castano, M. G. Fugini, G. Martella, and P. Samarati. *Database Security (ACM Press Book)*. Addison-Wesley, 1994.
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. Committee on Multilevel Data Management Security, Air Force Studies Board, Commission on Engineering and Technical Systems, National Research Council, National Academy Press, Washington D.C. *Multilevel data management security*, March 1983.
19. B. George and J. R. Haritsa. Secure Concurrency Control in Firm Real-Time Databases. *Distributed and Parallel Databases*, 5:275–320, 1997.
20. Q. Jiang and S. Chakravarthy. Anatomy of a Data Stream Management System. In *ADBIS Research Communications*, 2006.
21. W. Lindner and J. Meier. Securing the borealis data stream engine. In *IDEAS*, pages 137–147, 2006.
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. G. Ozsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. *Knowledge and Data Engineering, IEEE*, 7(4):513–532, Aug. 1995.
24. S. H. Son and R. David. Design and analysis of a secure two-phase locking protocol. In *Proc. of the CSAC*, pages 374–379, Nov. 1994.
25. N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *Proc. of the VLDB*, pages 309–320, September 2003.

A Query Sharing

Table 2 shows the ways in which queries Q1 to Q8 defined in Table 1 can be shared. For example, when Q5 is executing and Q4 is the newly issued query then they both can be strict shared.

Table 2. Query Sharing

Incoming / Executing	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Q1	***Complete***	-	-	-	-	-	-	-
Q2	-	***Complete***	-	-	-	-	***Loose*** Select (LEVEL)	-
Q3	-	-	***Complete***	-	-	-	-	-
Q4	-	-	-	***Complete***	***Strict*** Select(bp), Select(lon), Join	***Loose*** Select(bp)	***Loose*** Select(bp)	-
Q5	-	-	-	***Strict*** Select(bp), Select(lon), Join	***Complete***	***Loose*** Select(bp)	***Loose*** Select(bp)	-
Q6	-	-	-	***Loose*** Select(bp)	***Loose*** Select(bp)	***Complete***	***Loose*** Select(bp)	-
Q7	-	-	-	-	-	-	***Complete***	-
Q8	-	-	-	-	-	-	-	***Complete***