Journal of Computer Security 20 (2012) 547–581
DOI 10.3233/JCS-2012-0451
IOS Press

Multilevel secure data stream processing: Architecture and implementation

Raman Adaikkalavan [a,*], Xing Xie [b] and Indrakshi Ray [b]

[a] *Computer and Information Science, Indiana University South Bend, South Bend, IN, USA*
E-mail: raman@cs.iusb.edu
[b] *Computer Science, Colorado State University, Fort Collins, CO, USA*
E-mails: {xing, iray}@cs.colostate.edu

The proliferation of sensors and mobile devices and their connectedness to the network have given rise to numerous types of situation monitoring applications. 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 illustrate why the traditional DSMSs cannot be used for processing multilevel secure continuous queries and discuss various DSMS architectures for processing such queries. We implement one such architecture and demonstrate how it processes continuous queries. In order to provide better quality of service and memory usage in a DSMS, we show how continuous queries submitted by various users can be shared. We provide experimental evaluations to demonstrate the performance benefits achieved through query sharing.

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. Examples include border security monitoring, battlefield monitoring (reconnaissance and soldier health), emergency control and threat monitoring. Data Stream Management Systems (DSMSs) [1,4,5,9,12,17, 20] have been proposed for such applications that allow processing of streaming data

[*]Corresponding author: Raman Adaikkalavan, Computer and Information Science, Indiana University South Bend, South Bend, IN 46634, USA. Tel.: +1 574 520 4295; Fax: +1 574 520 5589; E-mail: raman@cs.iusb.edu.

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 [14, 19] to streaming data [2,15,16,32–34]. Controlling access is often not enough to prevent security breaches for complex applications that process and generate information classified at various security levels. Erroneous omission of an access control check may reveal confidential data. Integration of third party off-the-shelf software may cause policy checks to be bypassed altogether. Moreover, the existence of covert and overt channels may cause the leakage of sensitive data.

Our objective is to address the following threats. We want protection from honest but curious users who want to know information that they are not authorized to access. We also want to provide information security in cases where the processing units may contain Trojan horses and data may be leaked or influenced by unauthorized users and processes. We propose addressing the above threats by imposing information flow constraints on the processing and dissemination of data. Multilevel security (MLS) not only prevents unauthorized access but also ensures that only legal information flow occurs.

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 investigate various architectures for processing MLS continuous queries. In the simplest case, we have a trusted architecture where all components of a query processor are trusted and queries at different security levels can be processed within the same framework. Developing such an architecture is expensive because of the amount of trusted code needed. For the other cases, we consider hybrid architectures which consists of a mix of trusted and untrusted components. The trusted components deal with data at different security levels and they respect the laws of secure information flow. The untrusted components are replicated at each security level and they are allowed to access the data which is authorized at that level. The manner in which data at different security levels are processed and stored in these architectures give rise to various possibilities, one of which we discuss and elaborate in this paper.

We propose a hybrid architecture which has a trusted scheduler and query processors that are replicated across the various security levels. The trusted scheduler is responsible for scheduling the operators of queries belonging to different security levels. The query processor at each level is responsible for executing the queries at that level. The data needed for processing queries must often be replicated in such an architecture. In this architecture, queries are forwarded to the query processor at the appropriate security level. Since the query processor at some security level can only access the data authorized at that level, query rewriting is not needed for this architecture.

We discuss how plans for MLS continuous queries are generated and executed in the proposed architecture. We discuss how a new query can reuse plans from existing queries. We augment the approaches proposed by the Stanford STREAM [4], Aurora [12], and Borealis [1] projects and allow sharing of query plans submitted by different users. We demonstrate how queries submitted at the same level by different users can be shared. Query sharing not only allows good resource utilization but also helps achieve the Quality-of-Service (QoS) critical to stream processing applications.

We extend the Stanford STREAM system [38] and build a prototype based on our architecture that can process multilevel secure continuous queries. We discuss the implementation details and show how query sharing is achieved in such an architecture. We also provide experimental evaluations demonstrating the performance benefits achieved through query sharing.

The rest of the paper is organized as follows. Section 2 touches upon some preliminaries in multilevel secure databases and continuous query processing. Section 3 discusses why existing DSMSs cannot be used for processing MLS continuous queries. Section 4 presents an architecture for processing MLS continuous queries. Section 5 elaborates on how queries are shared to improve throughput and provide better resource utilization. Section 6 describes our prototype implementation. Section 7 gives experimental results on the speedup achieved through query sharing. Section 8 briefly touches upon some related work. Section 9 concludes the paper with some pointers to future directions.

2. Background

2.1. Multilevel secure databases

In this section, we briefly introduce the concepts in MLS databases on which our work is based. An MLS database 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 $L_1 \leqslant L_2$. If $L_1 \leqslant L_2$, 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 $L_1 \leqslant L_2$ nor $L_2 \leqslant L_1$. 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 **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, $L(\mathit{JaneDoe}) = H$. 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 the security level of the corresponding principal. Each process must obey the simple security and the restricted $\star$-property of the Bell–LaPadula model [13] given below.

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

Various architectures for MLS databases have been proposed [6]. In the trusted subject architecture, the DBMS is trusted and access to the individual records is mediated by the trusted DBMS. Such an architecture provides good security, but a large amount of trusted code is needed to implement this architecture. In the kernelized architecture, the data is partitioned on the basis of security level and physically stored as single-level databases. Multiple DBMSs are needed – one for each level – managing data stored at that security level or below. The trusted backend ensures that each DBMS accesses the data without violating the Bell–LaPadula policy. In the replicated architecture, the data is stored at multiple levels. However, each level stores/receives data at its level as well as data corresponding to all levels that it dominates. Here again, multiple DBMSs are needed for managing the data at various levels.

2.2. *Continuous Query Processing*

In this section, we discuss Continuous Query Processing based on the Continuous Query Language (CQL) [4,5]. The input to a DSMS are data coming from various stream sources, such as those transmitted by sensors. The stream shepherd system operator (i.e., stream-to-stream or S2S operator) receives such data, cleans them, and converts them into a stream with a well-defined schema that can be acted upon by the DSMS [5]. A stream can be thought of as a collection of timestamped tuples which adheres to the stream schema. The individual timestamped tuples in a stream are referred to as elements. For processing streaming data, CQL has three types of operators, in addition to the shepherd operator: (i) stream-to-relation (S2R), (ii) relation-to-relation (R2R) and (iii) relation-to-stream (R2S). The S2R operators (*time-based*, *tuple-based*, *partitioned by sequential-window*) convert input streams to relations. The R2R operators (*select*, *project*, *join*, *aggregate*) process tuples in the relations produced by the S2R operators and output relations. The R2S operators (*istream*, *dstream*, *rstream*) convert the relations produced by R2R operators back to streams.

A continuous query specified using the CQL is converted to a three stage query plan using the above operators as discussed below. The stream shepherd system operator cleans and reconstructs the input streams and propagates the tuples to the first stage.

- In the first stage, input data streams are converted to relations using S2R operators. For instance, when a user specifies a sliding window size of 100 tuples, the sequential-window operator that handles time-based, tuple-based and partitioned by window converts the stream to 100 tuple time varying relations. There is a one-to-one relationship between a input data stream and the sequential-window operator. This allows multiple queries to share the same sequential-window operator. This reduces memory usage and processing time involved in copying and maintaining tuples.
- In the second stage, the R2R operators process the incoming tuples/relations from the first stage. The non blocking operators can process tuples as and when they arrive. For example, the non blocking operator *select*, reads the incoming tuples, processes them and produces the output. The blocking operators must wait for all the input tuples to arrive before they can compute the results. The synopses stores the tuples before they can be evaluated. For example, the blocking operator *join* maintains a set of tuples using synopses based on the user's window specification as they need all those tuples for the computation. The output of this stage is a set of time varying relations.
- The last stage converts the relations back to streams using the R2S operators. For example, the *istream* operator streams every new tuple that is inserted into its input relation.

We now present the details of a typical DSMS [9,17,20] architecture (based on the STREAM system [4]) shown in Fig. 1. A Continuous Query (CQ) can be defined using specification languages [5], or as query plans [17]. 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 (*select*, *project*, *join*,

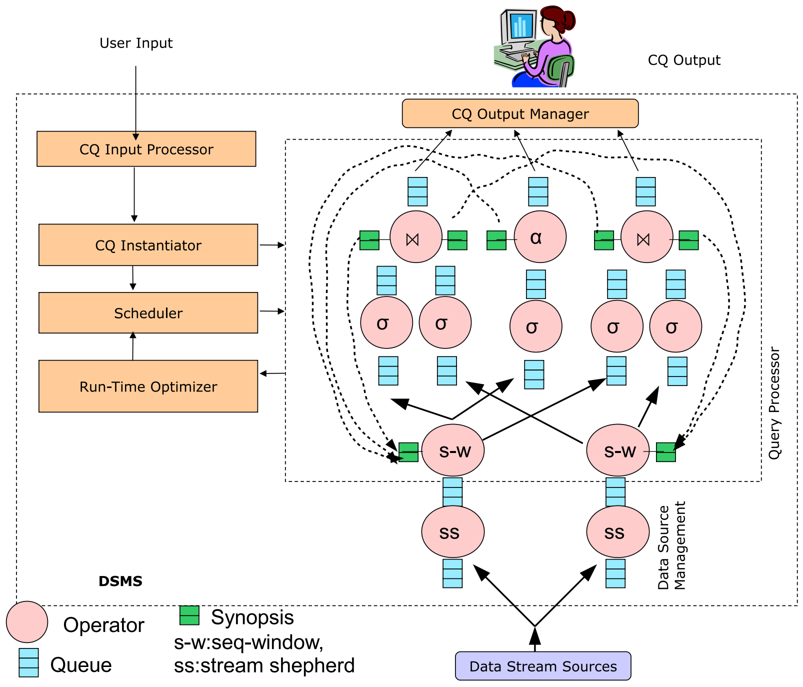

Fig. 1. Data Stream Management System (DSMS). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/JCS-2012-0451.)

aggregate). Each operator is associated with one or more input *queues*[1] and an output queue. One or more *synopses*[2] [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 [7,8,10,18,20,21,30,41], 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 [11,20,25,31,39,40] 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

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

[2]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.

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).

3. Problem statement

In this section, we briefly describe our model and motivate the need for a multilevel secure continuous query processing system. Each multilevel secure continuous query is submitted by a user logged in as a principal at some security level. The continuous query inherits the security level of the principal. In accordance with the simple security and restricted-$\star$ property of the BLP model, each MLS continuous query at level l can read only those objects whose levels are dominated by level l and can produce as output objects that are at level l.

In a relational MLS–DSMS, the data objects can be the relations, streams, tuples of relations, elements of stream, or the fields in a tuple or stream element. We can associate security level with each of these entities. For example, a single level can be associated with a stream or a relation. Although this approach is easy to implement, it is inadequate for many MLS applications, such as battlefield monitoring or infrastructure security. In such applications, streams containing elements having different security levels are often input to the DSMS. In this paper, we associate each element of a stream or each tuple of a relation with a security level. Thus, we do not consider the security level of the attributes individually in this work.

Definition 1. A *multilevel stream* S is a collection of timestamped tuples each of which is associated with a security level. The element $\langle s, l, t\rangle$ signifies that tuple s arrives on stream S at time t and has a security level l.

We next present an example that describes why current stream processing system cannot be used for processing multilevel secure queries. In our example, we have a security structure that consists of the following security levels: *Unclassified (U)*, *Confidential (C)*, *Secret (S)* and *Top Secret (TS)* and the dominance relation among the levels is given by $U < C < S <$ *TS*. We have two streams `VITALS` and `POSITION` that are receiving health and position information from soldiers deployed in various areas. These information are classified into various security levels depending on the sensitivity of the information. Thus, each tuple in these streams is associated with one security level which may be U, C, S or *TS*. The schema for these secure streams are given below.

```
VITALS (soldier id (sid), weight (wt),
       blood pressure (bp), pulse rate (pr), level);
POSITION (soldier id (sid), latitude (lat),
         longitude (lon), level);
```

Note that, the attribute *level* maintains the security level of the tuple. It is similar to the other attributes in that it can be queried or used as part of selection condition. The value of this attribute is generated by the system and it *cannot* be modified by the user. The schemas of all relations and streams in a MLS–DSMS must have this attribute.

The rules imposed by the Bell–LaPadula model [13] poses new challenges for processing MLS continuous queries. The queries inherit the security level of the process initiating it. Consider the following queries that look identical but are issued by users at different levels.

- $Q1$(*TS*): SELECT SUM(wt) FROM vitals [ROWS 100].
- $Q2(C)$: SELECT SUM(wt) FROM vitals [ROWS 100].

The response to $Q1$, which is a *Top Secret* query, will provide the total weight of the last 100 soldiers in the stream. Note that, these soldiers can be at any security level. The response to $Q2$, which is at level *Confidential*, cannot include information about *Secret* or *Top Secret* soldiers. Thus, the query will compute the total weight of the 100 soldiers who are at *Unclassified* or *Secret* levels. In other words, the computation of the queries (although they look identical) will be different in the two cases. Thus, MLS query processing is dependent on the level at which the query has been issued. For instance, query $Q2$ requires to find the total weight for the last 100 soldier tuples with level C or U. Thus, although the two queries look identical, the results returned will be different for the two users.

The manner in which queries are processed must also be modified if we take MLS constraints into account. Let us see this in the context of the STREAM system. In order to process the query, the R2R operator (*aggregate*) should receive up to 100 tuples that has either *Level* $= C$ or *Level* $= U$. The S2R operators in the first stage should create time varying relations with 100 tuples that are either C or U level from the input stream and propagate it to the R2R operator in the second stage. This cannot be done with existing S2R operators, as they do not support filtering conditions. On the other hand, this cannot be done just by using the R2R operators (e.g., *select*) as they execute in the second stage and get inputs from the S2R operators.

The second issue is the result of the R2R operators. For instance, the aggregation operator SUM used in the above query $Q2$ finds the total weight from 100 tuples with level C or U. Thus, the level of each tuple output from the aggregation operator must be the least upper bound of the tuples used in the computation. On the other hand, the relation (output from the R2R operator) itself will need to have its *Level* $= C$, as the level of the query is C.

The third issue is the sharing of the sequential-window (S2R) operators by the STREAM system. Typically, the S2R operators have a one-to-one correspondence with the input stream to reduce resource usage. With MLS–DSMS this may not be possible as each input stream can contain tuples with different security levels and queries running at different levels can access the same input stream.

Execution of MLS continuous queries should not cause illegal information flow through overt or covert channels. Let us illustrate how overt illegal information flow can occur through an example. Suppose a process at some dominated security level, say *TS*, reads a data item at the same level and writes a data item at the dominated level, say S. In this case, we say that information flows from level *TS* to S. Since existing DSMSs do not assign security levels to processes and cannot control the execution of read and write operations on the basis of security levels, it is possible to have such overt illegal information flow. Covert illegal information flow may also occur if queries at the different levels share resources, such as, storage and CPU time. These are referred to as storage channels and timing channels, respectively. If the query processors are not trusted, they may have Trojan horses embedded in them. A Trojan horse at the dominating level can manipulate the usage of the resource and convey information to the Trojan horse at the dominated level. MLS DSMSs must be able to prevent such illegal information flow.

4. MLS data stream 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 Fig. 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 first considered a trusted architecture where all the components of the query processor are trusted. The data is not physically isolated across security levels; the understanding is that the trusted components of the query processor will not cause illegal information flow. Thus, in the trusted architecture we deal with multilevel trusted streams and relations which consist of tuples that may belong to different security levels. The processes operating on these multilevel trusted streams and relations are trusted as well. Note that, developing such an architecture is expensive because of the amount of trusted code involved. Moreover, in such an architecture, when a query is submitted by a user at some login level, the query must first be transformed such that it operates on authorized data. For example, consider the following two queries described in Section 3 submitted by users at level *TS* and C, respectively.

- $Q1$(*TS*): SELECT SUM(wt) FROM vitals [ROWS 100].
- $Q2(C)$: SELECT SUM(wt) FROM vitals [ROWS 100].

The two queries must first be transformed before being submitted for execution.

- $Q1'$(*TS*): SELECT SUM(wt) FROM vitals [ROWS 100 WHERE level $\leqslant$ *TS*].
- $Q2'(C)$: SELECT SUM(wt) FROM vitals [ROWS 100 WHERE level $\leqslant C$].

Thus, query transformation is essential in the trusted architecture and we need to define correct transformational rules. We do not discuss this architecture any further in this paper and it is part of our future work.

We next consider hybrid architectures which have a mixture of trusted and untrusted components. In these architectures, we have a query processor at each security level which can access data at dominated levels. The query submitted by a user logged in at a given security level is redirected to the query processor at the appropriate level. The architectures differ with respect to how the streaming data is stored and processed. The streaming data can be physically partitioned on the basis of security levels giving rise to the *partitioned architecture*. Alternatively, the streaming data at dominated levels can be replicated across the dominating levels giving rise to the *replicated architecture*. We rejected the partitioned architecture for two reasons. The processing of a query at level l may involve data belonging to multiple levels each of which is dominated by l. Moreover, if such a query involves a blocking operator, the data may need to be temporarily stored at the dominated level giving rise to a storage channel. Note that, the data collected from the dominated levels must also be aggregated in order to respond to the query at the dominating level – this introduces delays which often interfere with the QoS requirement.

We next considered a *replicated architecture* where we have a DSMS at each level. The inputs to the DSMS at level l are multilevel streams consisting of elements that are dominated by level l. Thus, all the data needed for processing a query at level l is already available at level l and no incoming/stored data at the dominated level need to be accessed. Moreover, the query processor at a given level can access only authorized data. Consequently, the query transformation step in trusted architecture is not needed in this case. In the following section, we present the replicated architecture in more details.

4.1. MLS–DSMS replicated architecture

In the replicated architecture, we deal with two types of streams: *multilevel trusted stream* and *single level streams*. A multilevel trusted stream consists of inputs from various sources and is not associated with any specific security level, but consists of tuples each of which is associated with a security level. The assumption is that in a multilevel trusted stream the individual tuples are protected and information is not passed from one level to another. In contrast, we may have single level streams which are associated with a specific security level. A single level stream contains tuples, each of whose security level is dominated by the level of the stream. For example, a single level stream at *Secret* level can contain tuples at *Unclassified*, *Confidential*, or *Secret* level. The formal definitions of these two types of streams appear below.

Definition 2 (Multilevel trusted streams). A *multilevel trusted stream SS* is not associated with any security level and consists of a possibly infinite bag of elements $\langle s, l, t\rangle$, where $\langle s, l\rangle$ is a tuple belonging to the schema of S, $t \in \mathbf{T}$ is a timestamp and $l \in \mathbf{L}$ is the security level of the tuple. Note that, $\mathbf{L}$ is the set of security levels and $\mathbf{T}$ is the set of timestamps.

Definition 3 (Single level stream). A *single level stream* S is associated with a single security level L_S and consists of a possibly infinite bag of elements $\langle s, l, t \rangle$, where $\langle s, l \rangle$ is a tuple belonging to the schema of S, $t \in \mathbf{T}$ is a timestamp and l is the security level of the tuple such that $l \leqslant L_S$ and $L_S \in \mathbf{L}$.

We have only one type of relation, which are single level relations (or just relations) as they are associated with exactly one security level. The formal definition appears below.

Definition 4 (Single level relation). A *single level relation* R is a mapping from $\mathbf{T} \times \mathbf{L}$ to a finite but unbounded bag of tuples belonging to the schema of R each of which is associated with a security level. A relation R is associated with a security level L_R which is the input to the mapping function. The security level of each tuple in R is dominated by $L_R \in \mathbf{L}$.

We present one example of a replicated query processor in Fig. 2. The data stream sources send multilevel trusted streams to the DSMS. We have a *trusted* stream shepherd system operator that takes as input a multilevel trusted stream and creates single level streams for each security level. This is the only stream-to-stream operator (S2S) that we consider in this paper. The single level streams that are output by the stream shepherd operator are forwarded to the various query processors.

In this architecture, the query processors are replicated at each security level. A query processor at some security level L is responsible for executing queries submitted by users who have logged on at level L. The query processor at each level requires the implementation of the various stream-to-relation (S2R), relation-to-relation (R2R) and relation-to-stream (R2S) operators. These operators are untrusted and their code may contain malicious code and/or Trojan horses. The input

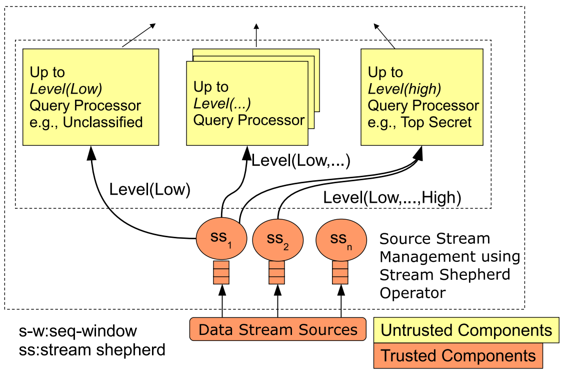

Fig. 2. Replicated MLS–DSMS architecture. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/JCS-2012-0451.)

to these operators are streams or relations consisting of tuples that are dominated by the level of the operator. The output stream or relation produced have a security level same as that of the operator. The input to these operators are stored in queues and synopses. The level of the queues and synopses are the same as the level of the operator.

In accordance with the Bell–LaPadula rules, the query processor at level L can read data at levels that are dominated by L. The output and intermediate results produced by the query processor at level L has the same security classification. However, the response to a query at level L 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 L.

4.1.1. Proof of security

Theorem 1. *Our proposed replicated query processor architecture does not cause unauthorized access.*

Proof. Let us consider a query Q submitted at level L. The query is forwarded to the query processor at level L. The trusted stream shepherd operator ensures that this query processor receives only those streams that are dominated by level L. In addition, the operators in this query processor can access relations that are dominated by level L. The response produced by the query is computed from tuples that are dominated by level L. Thus, the user cannot get any information that belongs to level that dominates L. □

Theorem 2. *Our proposed replicated query processor architecture does not cause illegal information flow.*

Proof. Let us consider a query Q submitted at level L. The query is forwarded to the query processor at level L. We first show that there are no overt channels causing illegal information flow. The trusted stream shepherd operator ensures that this query processor receives only those streams that are dominated by level L. In addition, the operators involved in executing this query can access relations that are dominated by level L. Thus, the operators can read only those tuples that are dominated by level L. The output produced by the operators are at level L. This ensures that the Bell–LaPadula rules are followed in executing this query and there are no overt illegal information flows.

Since the query processors in our architecture share no common storage, we do not have any covert information flow through storage channels. Timing channels may occur if queries at different levels share CPU time. Thus, the presence/absence of timing channels will be determined by the architecture of the scheduler which is responsible for executing the queries. We discuss in Section 6 the architecture of our trusted scheduler which ensures the absence of timing channels. □

5. Shared query processing in replicated DSMS

In this section, we give examples of MLS CQL queries and discuss how the processing of such queries can be shared.

5.1. MLS CQL queries

We have an additional attribute called *level* in each schema of a stream or relation. We can query this attribute, or submit queries based on this attribute.

Consider the following data streams (Vitals and Position)

```
VITALS (soldier id (sid), weight (wt),
        blood pressure (bp), pulse rate (pr), level);
POSITION (soldier id (sid), latitude (lat),
          longitude (lon), level);
```

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.

An MLS CQL query may not reference the security level attribute at all. The query below demonstrates this – it joins tuples from two streams. The sliding windows maintain the last 100 tuples for computations.

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

We consider only two types of windows: tuple-based (Q_2, Q_4, Q_5) and partitioned by windows (Q_1 and Q_3) [5]. Another example of partitioned by window is shown below. In this query, 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 in our architecture involves several steps. First, the selection condition of the query is written in conjunctive normal form. 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 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 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 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 synopsis N_i*.syn*, 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 user.

Operator trees for queries Q_6 and Q_7 defined in Table 1 appear in Fig. 3(a) and (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*, *join* and *average*. The parameter indicates the set of conjuncts for the *select* operator, or the set of attributes for the *project* operator. 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 a set of attributes. The synopsis is needed for the blocking operators, such as, *join* and *aggregate*, and has type (tuple-based or partitioned by) and size as its attributes. The input queues are derived from the streams and 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 at 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 Fig. 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

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`

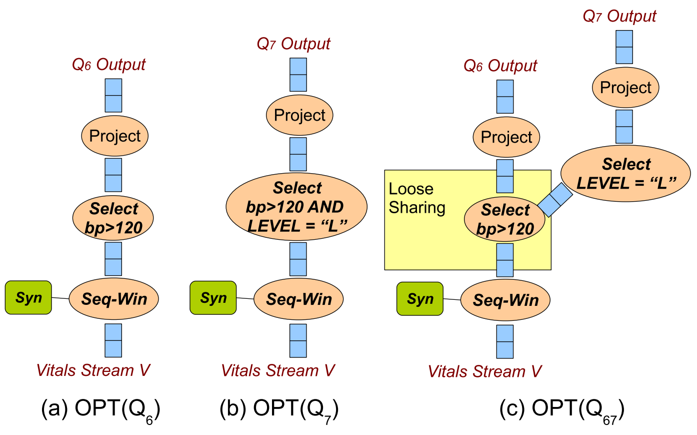

Fig. 3. Operator Tree for Q_6 and Q_7. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/JCS-2012-0451.)

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–Q_8, there are only selection conditions and projection (duplicate preserving) operations.

5.2. 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. In the Fig. 4, we show how the operator trees of Q_4 and Q_5 can be merged. Both use the same *seq-win* operator, as there is one seq-win operator per stream. 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

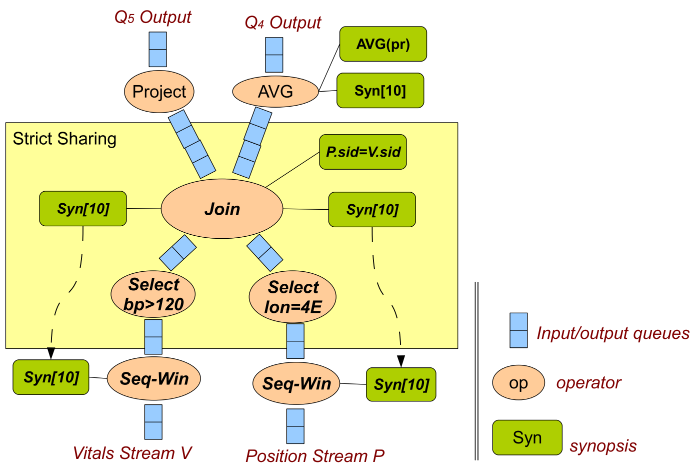

Fig. 4. Strict partial sharing operator tree for Q_4 and Q_5. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/JCS-2012-0451.)

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

Definition 6 (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). 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 7 (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 query Q_6 and Q_7 shown in Fig. 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.

5.2.1. 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 8 (Equivalence of operator trees). Two operator trees *OPT*(Q_x), *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 9 (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.

5.2.2. 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 10 (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 is more restrictive in which the conditions for *join*, for example, must exactly match the other query operator. On the other hand, with non-blocking operator they can be subsumed. The formal definition of these two forms of sharing appears below.

Definition 11 (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 12 (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 13 (Decomposition of subsumed nodes). Let $N_i \subseteq N_j$ where $N_i \in \mathit{OPT}(Q_x)$ and $N_j \in \mathit{OPT}(Q_y)$. 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) $N_i''.parm = N_i.parm - N_i'.parm(if\ N_i.op = select)$,
$N_i''.parm = N_i.parm(if\ N_i.op = project)$.

Consider the *select* nodes of the operator trees of query Q_6 and Q_7 shown in Fig. 3. In this case, the *select* node of Q_7 is subsumed by the *select* node of Q_6. *select* node of Q_7 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 14 (Overlap of operator trees). Two operator trees $\mathit{OPT}(Q_x)$, $\mathit{OPT}(Q_y)$ are said to *overlap* if $\mathit{OPT}(Q_x) \not\equiv \mathit{OPT}(Q_y)$ 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 $N_i \equiv N_j$.

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 $\mathit{OPT}(Q_4)$ and $\mathit{OPT}(Q_5)$. 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. Figure 3(a) and (b) show the $\mathit{OPT}(Q_6)$ and $\mathit{OPT}(Q_7)$, respectively. Figure 3(c) illustrates the $\mathit{OPT}(Q_{67})$ 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 the decomposition of subsumed nodes definition, we split Q_7 select condition into two ($bp > 120$ and level = "L") nodes and then share the $bp > 120$ node with Q_6.

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
```

6. Prototype implementation

We have extended the Stanford Stream Data Manager (STREAM) [38] prototype to support multilevel security (MLS–DSMS). The STREAM DSMS, henceforth referred to as vanilla DSMS, follows the client-server architecture and is implemented in C++. It does not support users, authentication, security levels, query sharing, or MLS processing such as finding Least Upper Bound in R2R operators (e.g., *join*, *average*).

6.1. Vanilla DSMS

We first discuss each component of the *vanilla DSMS* shown in Fig. 5 and its limitations. The *server* operates in two phases. In the first phase it registers queries, streams and relations from the *client* via the command unit. In the second phase, it executes the registered queries and propagates the outputs. Once the second phase starts no new queries, streams, or relations can be registered. The client communicates with the DSMS server via a set of predefined messages in multiple steps. The first two user commands corresponds to the first phase and the next three corresponds to the second phase:

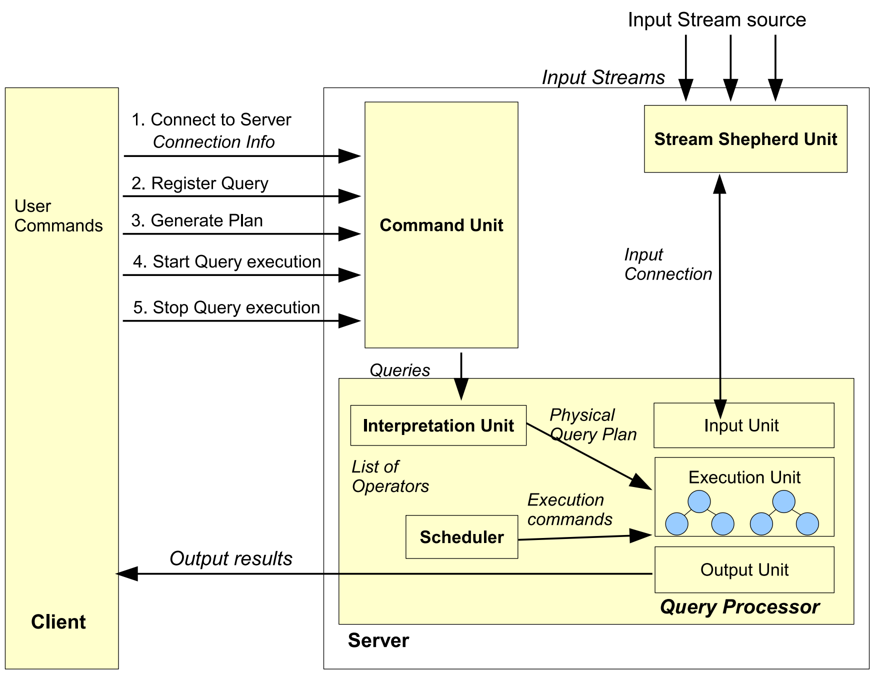

Fig. 5. DSMS system architecture. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/JCS-2012-0451.)

(1) *Connect to server*. The client establishes command communication with the server. The server creates a new server instance specific for that client. All the following command messages are sent to this instance. This does not allow sharing of input streams or queries among different clients. There is no notion of users, authentication, or security levels.

(2) *Register query*. The client registers input stream schemas, relations and queries. At this stage, a query registration message is sent to the interpretation unit which translates the interpreted query to a logical query plan (a link of operators). The naïve physical plan is also generated.
The input streams are connected to the query processor input unit by the stream shepherd unit. Users are required to bind input data sources with the stream schemas explicitly. This is not suitable in the replicated MLS architecture as the users can only access authorized tuples and/or streams following the Bell–LaPadula model. Thus, we have to modify the registration process. In the output unit, output connection between DSMS and client is established after the queries are registered.

(3) *Generate plan*. Once the DSMS receives command from the client indicating that all queries have been registered and binding of input streams have been completed, it optimizes the naïve physical query plans created in the previous

step. Also, graphs of physical plans are generated for user view. The generated physical plans are instantiated in the execution unit of the query processor and the list of operators are sent to the scheduler.

In the replicated MLS–DSMS, the query plans have to be generated in appropriate query processor and should be linked to appropriate single level input streams so that there is no illegal information flow.

(4) *Start and stop query execution*. Once the start query execution command is issued, the scheduler instructs the execution unit to start running the specified operators. Input, output, and execution units process stream tuples continuously, and the computation results are sent to the client until user issues the stop query execution command or there are no more input tuples.

6.2. MLS–DSMS

We have created the MLS–DSMS shown in Fig. 6, based on the architecture discussed in Section 4. In this architecture, the DSMS server has multiple query processors, each corresponding to a security level (e.g., Secret, Top Secret). Specifically, the extended MLS–DSMS system shown in Fig. 6 supports:

- multi-user server with user authentication;
- replicated query processors executing at different security levels;

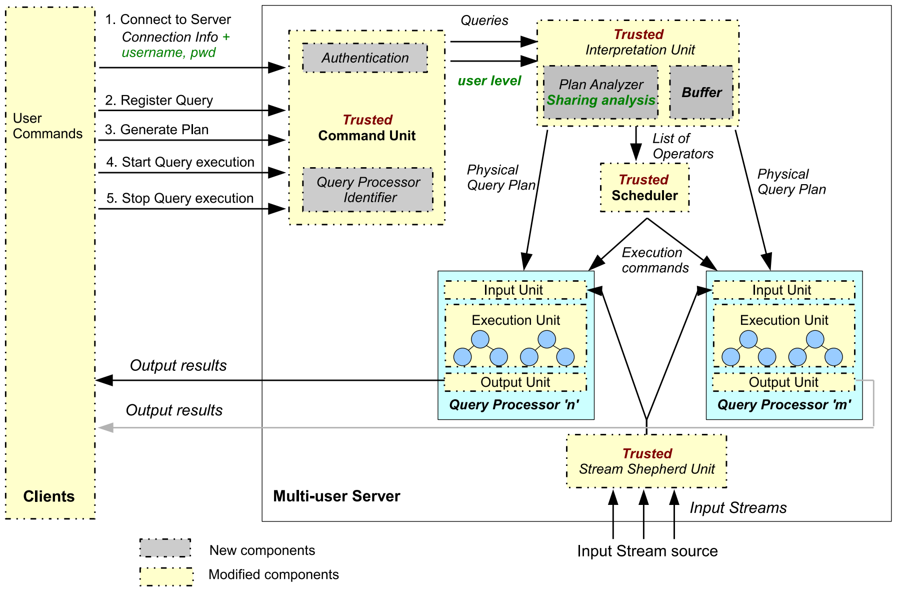

Fig. 6. Shared MLS–DSMS system architecture. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/JCS-2012-0451.)

- a global trusted scheduler that schedules operators across all query processors;
- a global trusted interpretation unit that supports centralized query plan generation for all query processors;
- complete sharing of continuous queries;
- partial sharing of continuous queries;
- trusted stream shepherd operator that takes multilevel trusted streams and outputs streams based on the level of the query processor;
- security level aware windows;
- security level aware query operators i.e., modification to R2R operators (e.g., *join*, *average*) to create output tuples with appropriate security levels;
- multilevel streams and tuples.

We next explain our modifications to the vanilla DSMS. All the modified and new components are shown in Fig. 6.

6.2.1. Trusted command unit

This unit is responsible for handling client communication, authentication and query processor instantiation.

The *command unit* is trusted and it accepts queries from users with different security levels. Each user query needs to be associated with a security level that corresponds to the level at which the query was submitted.

The *authentication module* is built in the trusted command unit to perform user authentication and security level verification. User authenticates by providing user name and password when connecting to the DSMS host. The authentication module then uses this information to retrieve the security level for the particular user.

The *query processor identifier (QPI) module* built in the trusted command unit gets user's client ID and 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. 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. We chose this approach to avoid starting query processors if there are no users. The maximum number of query processors is same as the maximum number of security levels supported (4 in our case).

The trusted command unit sends the user's level and registration information to the trusted interpretation unit. In addition, command unit still controls the query operations like commit and abort with the help of the QPI unit.

6.2.2. Trusted interpretation unit

This unit is responsible for generating query plans, complete and partial sharing of the queries, and for setting up the operators in the scheduler.

The *interpretation unit* receives user and query information from the command unit. The *plan analyzer* checks whether it is possible to share the new query with any of the currently executing queries using the buffer. The plan analyzer checks the queries based on the security level of the user and query processor and by comparing the semi-product plan of the new and existing plans. The semi-product plan is the semantic analysis result of the query string. User-defined keywords like size of window, aggregation ops, number of output attributes are saved/classified into the semi-product plan. The *buffer* maintains the semi-product, logical and physical plans for all existing queries according to security levels.

Specifically, the plan analyzer performs sharing analysis to check the possibilities for *complete* and *partial* sharing starting from the bottom operators (input streams operator) to the top (project operator) one by one. If the new query can be completely shared with an existing query, the buffer will reuse an existing physical plan and make a copy of the query. If there is partial sharing, it will reuse existing part of the logical and physical plans and generate the rest of the plan. The added plan will be buffered and also sent to the corresponding query processor. The list of operators (also the physical plan) is sent to the trusted scheduler.

We have also modified the three built-in interpretation components (*parser, semantic interpreter* and *logical plan generator*) and the execution unit of the query processor to filter tuples within a window based on security levels.

6.2.3. Query processor

As shown in Fig. 6, the server has multiple query processors. Each query processor is untrusted, executes at a security level, and has its own input, execution and output units.

The *input unit* can accept input streams from outside sources through trusted stream shepherd unit. It can also accept the output streams produced from other queries processed by the same query processor; this happens when queries are shared.

The *execution unit* is used by the server to execute the physical plans continuously. This unit contains the physical operators and their corresponding algorithms. In order to support MLS we have modified S2R and R2R operators. We have modified the window processing so that it can support filtering conditions based on security levels. We have modified the average and join operator algorithms to compute the least upper bound of the input tuples and use that as the security level of the output tuples. 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.

6.2.4. Trusted scheduler

There is only one scheduler to schedule operators across all query processors. Currently, the scheduler uses the round robin algorithm to schedule operators. The scheduler maintains all executing plan information (list of operators along with the security level, etc.) shared by the trusted interpretation unit. When a plan is received by the scheduler, the operators are scheduled for execution by the order from bottom to top. The scheduler sends out commands (including plan id and operator id) to the appropriate query processor to start executing an operator.

The registered operators are scheduled to run at least one time. Each operator handles a maximum of 100,000 tuples per second. If there is less number of tuples, the scheduler will switch to other operators after all tuples are handled. It schedules operators plan by plan. Since we do not have priority among processors, we use "first come first serve" strategy for plans. The plan registered first will be executed earliest in each round. When a new plan arrives to the scheduler, the operators will be scheduled in the next execution round.

The problem with the existing round robin strategy is that it can be exploited to cause illegal information flow. The dominating level can manipulate the time taken to execute the queries and convey information to the dominated levels. In order to overcome this problem, we modified the code of the trusted scheduler as follows. Each level is allocated a fixed time-slot during which queries at that level can be executed using the round-robin scheduling algorithm.

6.2.5. Trusted stream shepherd unit

Input streams are handled in this unit. In a real-life DSMS, input tuples from different sources can be collected into one multilevel trusted input stream. This unit contains the modified S2S operator (trusted stream shepherd operator) that converts the multilevel trusted streams to single level streams and sends it to the appropriate query processors.

7. Experimental evaluations

In this section, we discuss the experimental evaluations conducted to study the performance gain due to the addition of query sharing to a non-shared MLS–DSMS. We have measured the differences in time taken to execute CQL queries between Shared MLS–DSMS and Non-Shared MLS–DSMS. The experiments were conducted in a system with Intel i7 Q820 1.73 GHz Quad core Processor, 6 GB RAM, and Ubuntu 11.10 64 bit OS.

Three input data sets with 500 thousand, 1 million and 2 million tuples were used at an input rate of 20,000–40,000 tuples per second. Tuples contained a security level ($TS > S > U > C$). In each set, the number of tuples in each level was 1/4 of total tuples. We used the default round robin scheduling strategy with no load shedding. Each experiment was executed five times, the first two were discarded, and time take by the last three were averaged.

7.1. Complete sharing

We ran the following four queries to study the performance gained due to complete sharing. We did two studies (1) measuring the query execution time (the time taken from first tuple entering the S2R operator and last tuple exiting the query) and (2) measuring the plan generation time. We do not discuss CPU usage and memory as they stayed the same for all the corresponding experiments between shared and non-shared MLS–DSMS.

We executed all the queries at the TS security level. This was mainly due to the replicated architecture, as queries issued by different users at the same level only can be shared.

(1) Experiment 1 (SELECT Queries): We ran 9 identical queries.

```
SELECT sid, weight
FROM   Vitals[ROWS 100 LEVEL = "TS" OR LEVEL = "S"
              OR LEVEL = "C" OR LEVEL = "U"]
WHERE  weight > 100 AND weight < 200;
```

Input rate: 40,000 tuple/s.
Note that we used higher input rate to create the bursty input. As shown in Table 2 under Exp 1, the performance gain due to complete sharing was between 6.55% and 9.16% for the SELECT queries.

(2) Experiment 2 (AVG computation): We ran 9 identical queries.

```
SELECT   AVG(weight)
FROM     Vitals[ROWS 100 LEVEL = "TS" OR LEVEL = "S"
                OR LEVEL = "C" OR LEVEL = "U"]
WHERE    weight > 100 AND weight < 200
GROUP BY level;
```

Table 2
Complete sharing execution – performance gain

	Input rate (tuples/s)	Data size	Average execution time for 3 runs (ms)		Improvement due to sharing (%)	Standard deviation between 3 runs (ms)	
			Complete sharing	No sharing		Complete sharing	No sharing
Expl	40,000	500K	17,795.0	19,228.0	8.053	542.9	467.7
(Execution)		1M	34,189.3	37,321.0	9.160	1173.5	1038.2
		2M	68,207.7	72,673.3	6.547	995.7	577.9
Exp2	20,000	500K	45,443.7	50,023.7	10.078	323.7	790.4
(Execution)		1M	87,061.3	95,074.7	9.204	191.8	498.9
		2M	175,681.7	193,476.0	10.129%	445.4	1802.6
Exp3	20,000	500K	30,386.3	33,563.0	10.454	367.4	503.4
(Execution)		1M	54,479.0	65,988.0	21.126	543.5	1113.2
		2M	105,859.7	131,491.3	24.213%	1156.7	2496.3
Exp4	20,000	500K	34,697.0	36,125.3	4.117	151.1	282.6
(Execution)		1M	67,581.3	70,112.0	3.745	330.1	324.5
		2M	135,166.3	139,601.3	3.281	1033.0	788.8

Input rate: 20,000 tuple/s.
As shown in Table 2 under Exp 2, the performance gain for the Average queries was between 9.20% and 10.12%.

(3) Experiment 3 (JOIN Computation): We ran 5 identical queries.

```
SELECT Vitals.sid, weight, location,
       Vitals.level, Positions.level
FROM   Positions[Rows 100 LEVEL = "TS"
                 OR LEVEL = "S"
                 OR LEVEL = "C"
                 OR LEVEL = "U"],
       Vitals[Rows 50 LEVEL = "TS" OR LEVEL = "S"
              OR LEVEL = "C" OR LEVEL = "U"]
WHERE  Vitals.sid = Positions.sid AND weight > 100;
```

Input rate: 20,000 tuple/s.
As shown in Table 2 under Exp 3, the performance gain due to complete sharing for 5 Join queries was between 10.45% and 24.21%. As shown in the table, standard deviation for Exp 3 with 2M tuples was high compared to other data sizes.

(4) Experiment 4 (Mixed Queries): We ran 9 queries (3 SELECT queries from Experiment 1, 3 AVG queries from Experiment 2, and 3 Join queries from Experiment 3)
Input rate: 20,000 tuple/sec. We wanted to measure the performance difference in the heavy load case.
As shown in Table 2 under Exp 4, the performance gain for running a variety of queries was between 3.28% and 4.11%.

As shown in Table 2, the execution time (in milliseconds) performance gain due to complete sharing for Experiments 1–4 was between 3.28% and 24.21%. This variation is mainly due to the processing time taken by operators and due to the change in input rate. As we did not enable load shedding, we fine tuned the input rates to avoid inconsistent results. For instance, if the input rate is increased to 100,000 tuples per second, the DSMS was producing inconsistent results over the 5 runs of the same experiment.

As shown in Table 3 the performance gain due to not creating already existing plans was between 1.15% and 1.61%. But the gain is negligible when the standard deviation over the runs is taken into account. There is not a lot of performance gain as sharing analysis consumes resources.

7.2. Partial sharing

Currently, we have implemented partial sharing of the *project*, *aggregate* and *join* operators. We support both loose sharing and strict sharing.

(1) Experiment 1 (JOIN Computation): We ran 5 *Join* queries. All the queries are variations of the query defined below. The FROM and WHERE clauses were

Table 3
Complete sharing plan generation – performance gain

	Average execution time for 3 runs (ms)		Improvement due to sharing (%)	Standard deviation between 3 runs (ms)	
	Complete sharing	No sharing		Complete sharing	No sharing
Exp 1 (Plan generation)	762.0	774.0	1.575	2.7	6.0
Exp 2 (Plan generation)	767.6	776.4	1.158	3.5	8.8
Exp 3 (Plan generation)	374.2	378.7	1.188	3.7	7.0
Exp 4 (Plan generation)	771.6	784.0	1.613	3.2	5.4

identical in all the queries and the SELECT clause was different.

```
SELECT Vitals.sid, weight, location,
       Vitals.level, Positions.level
FROM   Positions[Rows 100 LEVEL = "TS"
                 OR LEVEL = "S"
                 OR LEVEL = "C"
                 OR LEVEL = "U"],
       Vitals[Rows 50 LEVEL = "TS" OR LEVEL = "S"
              OR LEVEL = "C" OR LEVEL = "U"]
WHERE  Vitals.sid = Positions.sid
       AND weight > 100 AND weight < 200;
```

Input rate: 30,000 tuples/s. The reason we picked 30,000 was that the lower input rate like 10,000 was not able to cause the heavy load case, and higher rate creates overload case where the system did not provide stable response time.
As shown in Table 4 under Exp 1, performance gain due to partial sharing was between 3.81% and 6.39%. This cannot be compared with the complete sharing case as the selectivity is different.

(2) Experiment 2 (AVG Computation): We ran 7 aggregate queries. All the queries were variations of the query defined below and they were partially shared similar to Experiment 1.

```
SELECT AVG(weight), MAX(weight), MIN(weight)
FROM   Vitals[Rows 100 LEVEL = "TS"
              OR LEVEL = "S"
              OR LEVEL = "C"
              OR LEVEL = "U"]
WHERE  weight > 100 AND weight < 200;
```

Input rate: 20,000 tuples/s. This input rate creates a small heavy load situation. For partial sharing, queries reuse all the aggregation results by adding a

Table 4

Partial sharing execution – performance gain

	Input rate (tuples/s)	Data size	Average execution time for 3 runs (ms)		Improvement due to sharing (%)	Standard deviation between 3 runs (ms)	
			Complete sharing	No sharing		Complete sharing	No sharing
Expl	30,000	500K	18,770.3	19,970.3	6.393	428.1	148.2
(Execution)		1M	35,383.0	36,733.7	3.817	436.0	328.9
		2M	69,903.0	72,841.7	4.204	527.6	722.8
Exp2	20,000	500K	33,588.3	35,839.0	6.701	437.5	333.2
(Execution)		1M	67,405.0	72,322.0	7.295	653.9	944.0
		2M	132,964.7	140,305.0	5.521	1007.9	1415.5

Table 5

Partial sharing plan generation – performance gain

	Average execution time for 3 runs (ms)		Improvement due to sharing (%)	Standard deviation between 3 runs (ms)	
	Partial sharing	No sharing		Partial sharing	No sharing
Exp 1 (Plan generation)	375.0	377.9	0.770	3.3	4.7
Exp 2 (Plan generation)	572.4	578.7	1.087	3.1	2.3

project operator on top of the Q1 output. As shown in Table 4 under Exp 2, performance gain due to partial sharing was between 5.52% and 7.29%.

Based on the above experiments, the performance gain due to partial sharing was between 3.81% and 7.29%. On the other hand, the performance gain due to plan generation was 1% or less as shown in Table 5. This shows that analyzing the existing plans for partial sharing does not cause much overhead in the system.

8. Related work

In this section, we will discuss works from closely related areas: MLS Stream Processing, DSMS, DSMS security and MLS in real-time systems.

Multilevel continuous query processing. The only work that we are aware of in multilevel secure continuous query processing is our earlier work [3]. In this work, we present the architecture of the query processor of a MLS–DSMS and we discuss how queries can be shared to maximize resource utilization. Our current work extends the earlier work in several ways. First, we demonstrate why existing DSMSs cannot be used for processing MLS continuous queries. Second, we briefly describe alternative architectures that are possible for executing MLS continuous queries.

Third, we present the details of the prototype that we developed based on the architecture. Finally, we present experimental results to demonstrate the performance benefit obtained through query sharing.

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 [7,8,10,18,20,21,30,41] and load shedding techniques [11,20,25,31,39,40]. Some of the research prototypes include: Stanford STREAM Data Manager [4,9], Aurora [12], Borealis [1,23] and MavStream [28].

DSMS security. There has been several recent works on securing DSMSs [2,15,16,32–34] 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 [34]. 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 [2]. 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 [15,16]. 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 [32] 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.

DSMS sharing. In general DSMSs like STREAM [4,9], Aurora [12] and Borealis [1,23], 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 identical streams. Instead of sharing parts between plans, Aurora research focuses on providing better execution scheduling of large number of 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. Jin and Carbonell [24] look into the problem of using predicate indexing for query optimizing in streams where not all the continuous queries are submitted at the same time. In this approach, a relation schema stores existing query plan information which is queried and updated when a new query arrives.

DBMS query plan sharing. In the context of database systems, researchers have also investigated how queries can benefit by sharing their computation costs. Finkelstein [26] demonstrated how query graphs can be used for detecting common subexpressions across multiple queries. Sellis [36] investigates the problem of multi-query optimization where the goal is to obtain a good plan for multiple queries. Chen and Dunham [22] have also looked into the problem of efficiently identifying common subexpressions for processing multiple queries. Goldstein and Larson [29] focus on how queries can be optimized by using results from materialized views. These traditional approaches cannot be used for several reasons. Most approaches focus on optimizing join queries. Since the join operations are implemented differently in data streams and database systems, we cannot use many of these optimization techniques. Moreover, the queries in traditional database systems are not continuous and some of the proposed approaches apply to one-time queries only. Also, strategies that optimize multiple queries at any given point of time to find the best possible plan may not work in data stream systems as the queries arrive asynchronously.

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 [27], 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 [37] 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. [35], 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.

9. Conclusions and future work

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. We have also implemented our prototype architecture and provided experimental results to demonstrate performance improvements due to query sharing. Our approach does not have security violations and can be used to process MLS data streams.

We plan to investigate MLS–DSMS query processing for other architectures as well. Specifically, we plan to look in details at trusted architecture. In the trusted architecture, it may be possible to share query plans across security levels and the performance improved. However, the trusted architecture incurs an additional overhead due to query rewriting. We plan to implement the trusted architecture and do a comparative study between the hybrid and trusted architectures on the basis of performance and storage requirements.

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 would like to investigate how existing queries can migrate to the new plan without causing a breach of security.

In this work, we developed a very simple algorithm for our scheduler. In future, we would like to come up with more sophisticated algorithms for the scheduler and observe their impact on performance and resource utilization. During bursty traffic, not all tuples are processed due to the QoS requirements. Load shedding strategies decide which tuples to drop such that the query processing results are not affected. We plan to investigate what impact information flow constraints have on load shedding strategies and develop new ones for our MLS DSMS.

10. Appendix: Query sharing

Table 6 shows the ways in which queries Q_1–Q_8 defined in Table 1 can be shared. For example, when Q_5 is executing and Q_4 is the newly issued query then they both can be strict shared.

Table 6

Query sharing

Executing	Incoming							
	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8
Q_1	*Complete*	–	–	–	–	–	–	–
Q_2	–	*Complete*	–	–	–	–	*Loose* Select (LEVEL)	–
Q_3	–	–	*Complete*	–	–	–	–	–
Q_4	–	–	–	*Complete*	*Strict* Select(bp), Select(Ion), Join	*Loose* Select(bp)	*Loose* Select(bp)	–
Q_5	–	–	–	*Strict* Select(bp), Select(Ion), Join	*Complete*	*Loose* Select(bp)	*Loose* Select(bp)	–
Q_6	–	–	–	*Loose* Select(bp)	*Loose* Select(bp)	*Complete*	*Loose* Select(bp)	–
Q_7	–	–	–	–	–	–	*Complete*	-
Q_8	–	–	–	–	–	–	–	*Complete*

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*, 2005, pp. 277–289.

[2] R. Adaikkalavan and T. Perez, Secure shared continuous query processing, in: *Proc. of the ACM SAC (Data Streams Track)*, Taiwan, 2011, pp. 1005–1011.

[3] R. Adaikkalavan, I. Ray and X. Xie, Multilevel secure data stream processing, in: *Proc. of the DBSec*, 2011, pp. 122–137.

[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 J.* **15**(2) (2006), 121–142.

[6] V. Atluri, S. Jajodia, T.F. Keefe, C.D. McCollum and R. Mukkamala, Multilevel secure transaction processing: status and prospects, in: *Proc. of the DBSec*, 1996, pp. 79–98.

[7] R. Avnur and J. Hellerstein, Eddies: continuously adaptive query processing, in: *Proc. of the ACM SIGMOD*, 2000, pp. 261–272.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani and D. Thomas, Operator scheduling in data stream systems, *VLDB J.* **13**(4) (2004), 333–353.

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

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

[11] B. Babcock, M. Datar and R. Motwani, Load shedding for aggregation queries over data streams, in: *Proc. of the ICDE*, 2004, pp. 350–361.

[12] 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 J.* **13**(4) (2004), 370–383.

[13] D.E. Bell and L.J. LaPadula, Secure computer system: unified exposition and MULTICS interpretation, Technical Report MTR-2997, Revision 1 and ESD-TR-75-306, Revision 1, The MITRE Corporation, Bedford, MA, March 1976.

[14] M. Bishop, *Computer Security: Art and Science*, Addison-Wesley, Reading, MA, 2002.

[15] J. Cao, B. Carminati, E. Ferrari and K. Tan, ACStream: enforcing access control over data streams, in: *Proc. of the ICDE*, 2009, pp. 1495–1498.

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

[17] 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*, 2002, pp. 215–226.

[18] D. Carney, U. Çetintemel, A. Rasin, S.B. Zdonik, M. Cherniack and M. Stonebraker, Operator scheduling in a data stream manager, in: *Proc. of the VLDB*, 2003, pp. 838–849.

[19] S. Castano, M.G. Fugini, G. Martella and P. Samarati, *Database Security*, Addison-Wesley, Reading, MA, 1994.

[20] 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, New York, 2009.

[21] S. Chakravarthy and V. Pajjuri, Scheduling strategies and their evaluation in a data stream management system, in: *Proc. of the BNCOD*, 2006, pp. 220–231.

[22] F.-C.F. Chen and M.H. Dunham, Common subexpression processing in multiple-query processing, *TKDE J.* **10**(3) (1998), 493–499.

[23] 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, pp. 257–268.

[24] J. Chin and J. Carbonell, Predicate indexing for incremental multi-query optimization, in: *Proc. of the International Symposium on ISMIS*, pp. 339–350, 2008.

[25] A. Das, J. Gehrke and M. Riedewald, Approximate join processing over data streams, in: *Proc. of the ACM SIGMOD*, 2003, pp. 40–51.

[26] S.J. Finkelstein, Common subexpression analysis in database applications, in: *Proc. of the ACM SIGMOD*, 1982, pp. 235–245.

[27] B. George and J.R. Haritsa, Secure concurrency control in firm real-time databases, *Distrib. Parallel Databases J.* **8**(1) (2000), 41–83.

[28] A. Gilani, S. Sonune, B. Kendai and S. Chakravarthy, The anatomy of a stream processing system, in: *Proc. of the BNCOD*, 2006, pp. 232–239.

[29] J. Goldstein and P.A. Larson, Common subexpression analysis in database applications, in: *Proc. of the ACM SIGMOD*, 2001, pp. 331–342.

[30] Q. Jiang and S. Chakravarthy, Scheduling strategies for processing continuous queries over streams, in: *Proc. of the BNCOD*, 2004, pp. 16–30.

[31] B. Kendai and S. Chakravarthy, Load shedding in mavstream: analysis, implementation, and evaluation, in: *Proc. of the BNCOD*, 2008, pp. 100–112.

[32] W. Lindner and J. Meier, Securing the Borealis data stream engine, in: *Proc. of the IDEAS*, 2006, pp. 137–147.

[33] 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*, 2009, pp. 1027–1030.

[34] 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*, 2008, pp. 406–415.

[35] G. Ozsoyoglu and R.T. Snodgrass, Temporal and real-time databases: a survey, *IEEE J. Knowl. Data Eng.* **7**(4) (1995), 513–532.

[36] T.K. Sellis, Multi-query optimization, *ACM TODS J.* **13**(1) (1988), 23–52.

[37] S.H. Son and R. David, Design and analysis of a secure two-phase locking protocol, in: *Proc. of the CSAC*, 1994, pp. 374–379.

[38] Stanford Stream Data Manager (STREAM), http://infolab.stanford.edu/stream/code.

[39] N. Tatbul, U. Çetintemel, S.B. Zdonik, M. Cherniack and M. Stonebraker, Load shedding in a data stream manager, in: *Proc. of the VLDB*, 2003, pp. 309–320.

[40] N. Tatbul and S.B. Zdonik, Window-aware load shedding for aggregation queries over data streams, in: *Proc. of the VLDB*, 2006, pp. 799–810.

[41] S.D. Viglas and J.F. Naughton, Rate-based query optimization for streaming information sources, in: *Proc. of the ACM SIGMOD*, 2002, pp. 37–48.