T H E   N A T I O N A L   U N I V E R S I T Y
o f   S I N G A P O R E

S c h o o l of C o m p u t i n g
Computing 1, 13 Computing Drive, Singapore 117417

TRA2/13

ASSIST: Access Controlled Ship Identification Streams

Jianneng Cao, Thomas Kister, Shili Xiang, Baljeet Malhotra, Wee-Juan Tan, Kian-Lee Tan, and Stephane Bressan

February 2013

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

ASSIST: Access Controlled Ship Identification Streams

Jianneng Cao[1], Thomas Kister[2,⋆], Shili Xiang[3], Baljeet Malhotra[4], Wee-Juan Tan[5], Kian-Lee Tan[2], and Stéphane Bressan[2]

[1] Cyber center, Purdue University, West Lafayette, IN, USA
caojn@purdue.edu
[2] School of Computing, National University of Singapore, Singapore
{kister,steph,tankl}@comp.nus.edu.sg
[3] Data Analytics Department, Institute for Infocomm Research, Singapore
sxiang@i2r.a-star.edu.sg
[4] SAP Research, Singapore
baljeet.malhotra@sap.com
[5] Centre for Remote Imaging, Sensing and Processing, National University of Singapore, Singapore
crstwj@nus.edu.sg

Abstract. The International Maritime Organization (IMO) requires a majority of cargo and passenger ships to use the Automatic Identification System (AIS) for navigation safety and traffic control. Distributing live AIS data on the Internet can offer a global view for both operational and analytical purposes to port authorities, shipping and insurance companies, cargo owners and ship captains and other stakeholders. Yet, uncontrolled, this distribution can seriously undermine navigation safety and security and the privacy of the various stakeholders. In this paper we present ASSIST, an application system based on our recently proposed access control framework, to protect streaming data from unauthorized access. Furthermore, we have implemented ASSIST on top of StreamInsight, a commercial stream engine. The extensive experimental results show that our solution is more effective and efficient than existing approaches.

Keywords: Data stream, Access control, Query rewriting, AIS, StreamInsight

1 Introduction

The International Maritime Organization (IMO) requires a majority of cargo and passenger ships to use the Automatic Identification System (AIS) [1]. It allows ships and stations to broadcast messages that contain data, such as navigational status, position course and speed among many other, for the primary purpose of navigation safety and traffic control. AIS messages can be received by other ships

⋆ Contact author

and by vessel traffic services stations in the vicinity (typically and approximately 40 nautical miles). As a matter of fact, AIS messages can be received by anyone in range using an appropriate receiver and decoding hardware and software.

With a network of AIS receivers transmitting data on the Internet, one can combine multiple AIS data streams to offer various services based on a global view of maritime ships for both operational and analytical purposes. Such services could be useful to port authorities, shipping and insurance companies, cargo owners and other stakeholders. A shipping company may track its vessels around the world and analyze their routes. A ship officer, responsible for navigation watch, can be informed of the positions and movements of ships that are relevant to safe navigation. A port authority can complement its traffic control by monitoring ships calling at the port and navigating the port waters. Unfortunately, the free and uncontrolled distribution of AIS data on the Internet can be exploited to seriously undermine navigation safety and security. For instance, it can be used by pirates to plan and launch long range attacks. It can also violate the privacy of the various stakeholders. For instance, it can reveal information about the cargo and the cargo owners. This problem has been highlighted by IMO's Maritime Safety Committee, which declared that "*..the publication on the world-wide web or elsewhere of AIS data transmitted by ships could be detrimental to the safety and security of ships and port facilities and was undermining the efforts of the Organization and its Member States to enhance the safety of navigation and security in the international maritime transport sector*" [2].

The collection of AIS, however, cannot be banned, otherwise, the primary purpose of navigation safety would be impaired. In this context, AIS data streams could be described as Sensitive But Unclassified data streams following the United States government's nomenclature of scientific and technical information [3]. To that end, we argue that the access and accessibility to data streams such as the ones generated by AIS need to be controlled. Although the processing of AIS data has been explored in the past, e.g., to detect anomalies through trajectory data mining [4–6], non-significant attention has been paid to securing AIS data streams with access control policies.

In this paper we present ASSIST, an application system based on our recently proposed access control framework [7], to protect streaming data from unauthorized access. Instead of controlling accesses by security operators as defined in other schemes [8–10], ASSIST adopts query rewriting. Whenever a user submits a query to the stream system, it checks the authorization catalog to verify whether the access can be totally or partially granted, or should be denied. In the case of a partial grant, the query will be rewritten according to the related policies, so that only authorized tuples/attributes will be delivered to the user. Query rewriting makes the stream system intact. The available query optimizer can reorganize rewritten queries for the purpose of query optimization without any constraint, thus maximizing the reusage of existing query processor infrastructure. Furthermore, we have implemented our scheme on top of StreamInsight [11], a commercial stream engine. The extensive experimental results show that our scheme is more effective and efficient than existing approaches.

The remaining of our work is organized as follows. We introduce the background knowledge about data streams, and review the related work of controlling access to data streams in the next section. Section 3 describes the system architecture of ASSIST. We propose the access control model in Sect. 4. Following that, Sect. 5 presents our query rewriting technique to enforce the access control model, and Sect. 6 discusses how our solution is implemented on top of StreamInsight. We report the experimental evaluation in Sect. 7, and conclude our work in Sect. 8.

2 Background and Related Work

In this section, we will first present some background knowledge about data streams, including their applications and stream engines to efficiently manage them. After that, we will review the related work on access control over data streams.

2.1 Data Streams

Some companies such as online stores, telecommunication companies, and social networks, generate data at a rate of millions of records each day. Such data typically appear as a sequence (stream) of append-only tuples, continuously grow over time at high speed, and thus they are typically referred to as data streams. Data streams are often unbounded, and there is no control over their arriving order.

Data streams have a wide range of applications. Examples include but not limited to network traffic monitoring (e.g., click streams and network security), transaction log mining, and financial fraud detection. Data streams have special processing requirements, due to their unique characteristics compared with traditional databases.

- **Sliding window**. Queries on data streams are typically *window-based*. A window can be regarded as a segment of tuples. It can be either time-based (e.g., tuples in the last one hour) or tuple-based (e.g., the last 100 tuples). Such window-based queries return answers only about the data falling in the windows. Thus, the concept of window allows users to retrieve statistical information about most updated data (see Sect. 2.2 for details). As data continuously arrive, the windows slide forward (i.e, data in the windows are updated), and at the same time the corresponding query answers are recalculated.
- **Approximation**. The infinite length of streams also suggests summary structures, such as synopses [12]. Consequently, queries over the structures may return approximate answers.
- **One pass**. Because of the limitations on storage and performance, backtracking over streaming data is not allowed, and online algorithms are restricted to making only one pass over streaming data.

- **Adaptivity**. The highly dynamic data rate and distribution of some source streams require the query processing to be adaptive.
- **Scalability**. The possibility of numerous queries registered at the stream engine needs a scalable processing solution.

Till now, a large amount of works have investigated these newly raised research issues. Some of them are related to models and languages (see [13] for a survey), some focus on continuous query processing problems, e.g., load shedding, join problems and efficient window-based operators [14], and many concentrate on data stream mining [15–17], and so on.

Online processing of streaming data brings unique commercial opportunities to the data owners (i.e., companies), thus it is becoming an indispensable part of business operations. To efficiently manage data streams, quite a few data stream management systems (DSMSs) are designed. Borealis [18] is a distributed stream processing system, which is based on Aurora [19] and Medusa [20]. STREAM [12] is a "general-purpose" data stream management system (DSMS). TelegraphCQ [21] is specially designed to process adaptive data flow with an extension to support shared continuous queries. Other examples are Alert [22], Tribeca [23], OpenCQ [24], NiagaraCQ [25], CAPE [26], and so on.

In the past few years, some DSMSs including Esper [27] and StreamInsight [11] model streaming tuples as events. They consider the incoming events from the source streams as *low-level* events. They filter and combine these low-level events to deduce complex *high-level* events (i.e., composite events), which are then notified to the interested users. Accordingly, these DSMSs are usually referred to as complex event processing engines (CEPs). StreamInsight [11], the test bed of our solution (see Sect. 7), is an event processing system based on an interval-based model, called the CEDR [28] paradigm. Every event is defined by a starting timestamp t_s, an ending timestamp t_e, and a payload. The event is said to be valid during the time interval $[t_s, t_e]$. The event time interval is decoupled from the system time; the timestamps of events can be acquired or modified if needed, when the events are pushed to the engine. A compulsory condition for the join of two events is that their validity time intervals overlap. Queries in StreamInsight are written in the LINQ language [29], which is a Microsoft .NET framework component, integrating SQL query operators (e.g., *select*, *join*, and *where*) with programming languages (e.g, C# and Visual Basic). A further discussion about StreamInsight and LINQ can be found in [30, 31].

2.2 Access Control over Data Streams

Recently, access control over data streams has been investigated. The solutions developed so far can be classified into two main categories: *security operator based* and *query rewriting based*. The former defines security operators and integrates them into DSMSs to filter unauthorized tuples and/or attributes. Approaches belonging to this category include Lindner and Meier [8], and Nehme et al. [9, 10]. The latter rewrites registered continuous queries according to predefined access control policies in such a way that the query outputs contain

only tuples/attributes that are accessible to users. Methods such as Carminati et al. [32, 7] fall into this category.

Lindner and Meier [8] put forward an owner-extended RBAC (OxRBAC) model to protect data streams. The basic idea is to apply a specially designed operator, `SecFilter`, to the end of the query plan, so that only the tuples complying with the access control policies are delivered to users. This *post-processing* approach is not efficient: 1) it does not prune unauthorized tuples as early as possible, 2) it is possible for a user to remain "connected" to an output stream, though s/he does not receive any tuple (e.g., her/his access rights have been revoked), and 3) it is not an intrusive solution, thus incapable of handling access control policies on views defined on multiple streams.

Nehme et al. [9, 10] have studied the problem of access control from a different point of view. They consider a scenario, where the access control polices and the streaming data are interleaved. The policies are not stored in DSMSs, instead they are encoded in *security punctuations* and pushed to DSMSs together with streaming data. A new operator, named *security shield*, is defined, which acts like *select* operator, pre-filtering tuples based on the polices extracted from the punctuations. Each policy has a time stamp. Once a policy with a more recent time stamp arrives, the old one will be replaced. This solution well supports the policy update. However, it is based on an assumption that the policies and the data arrive in order. In particular, it requires that an access control policy (or the corresponding set of security punctuations) always proceed the streaming tuples, on which it is applied. Once the ordering constraint is violated, the protection on the data will be different from the original intention.[6] In real scenario, the arriving order of both streaming tuples and security punctuations are unpredictable. Consequently, this solution is limited in the real applications.

Carminati et al. [32, 7] adopt query rewriting instead of security operators to control accesses to streaming data. Both approaches are based on the expressive role-based access model presented in [33]. Whenever a user submits a query to the stream system, they check the authorization catalog to verify whether the access can be totally or partially granted, or should be denied. In the case of a partial grant, the query will be rewritten according to the related policies, so that only authorized tuples/attributes will be delivered to the user. Query rewriting makes the stream system intact. The available query optimizer can reorganize a rewritten query for the purpose of query optimization without any constraint, thus maximizing the reusage of existing query processor infrastructure.

3 The ASSIST System

Ships, buoys, light houses and other vessel traffic services stations equipped with an AIS transponder send messages at regular intervals. There are 27 different types of AIS messages [1] that are used to identify the sender and indicate

[6] For example, if a segment of tuples arrive earlier than their policy, then the access control over these tuples would be based on some old policy, different from the desired one.

her course and position as well as various other navigational situations such as emergency situations. A typical AIS message, in its original form and with some decoded fields, is shown in Table 1.

Table 1: A sample AIS message.

Encoded: `!AIVDM,1,1,,B,4h5GdQ1ueHeBN7Jj5d0ga6W00<=l,0*42`

Field	**Value**	**Comment**
MMSI	5631108	Transponder identifier, used as Ship ID
Message Type	4	
Latitude	103.717423	} Location of the transponder
Longitude	1.300523	
Stations	884	Number of other transponders detected around
...	...	...

AIS messages are broadcast by transponders on ships and stations by Very High Frequency radio (VHF). AIS messages are received by VHF receivers on other ships or stations in range. In the application that we discuss here, the receivers create streams of AIS messages that are processed by the ASSIST system. Users of ASSIST request and monitor information via a user interface. This scenario and the underlying infrastructure are illustrated in Fig. 1.

Different users of the ASSIST system assume different predefined roles. The roles are associated with access control policies that have been defined by the security administrator. The information displayed on the user interface is controlled by the access control policies and therefore depends on the user's role.

The architecture of the ASSIST system is illustrated in Fig. 2. Its *User Interface* as shown in Fig. 3 is a Web client that allows the user to connect through the Internet to the ASSIST server and to authenticate, submit queries and monitor results. Queries are written as plain text in an SQL-like syntax. The query results can be texts, or visual views on a navigational chart, showing the views of the situation in the port and neighboring waters, as illustrated in Fig. 3.

First, the query is translated into a Directed Acyclic Graph (DAG). This DAG is sent to the `Rooflight` module which is in charge of the query rewriting. If the access of a query should be completely or partially granted, the query is rewritten according to the access control policies stored in the *System Authorization Catalog* (also known as `SysAuth`). The rewritten query is then translated to the target DSMS' language[7], and sent to the *Stream Engine* for execution. Once the engine accepts the query, it starts to process the streaming events continuously for the query.

[7] The query rewriting of DAG is independent of any target stream engine. Given a specific engine and its query language (e.g., StreamInsight and LINQ), the *Translator* model will recode the rewritten DAG according to the given query language.

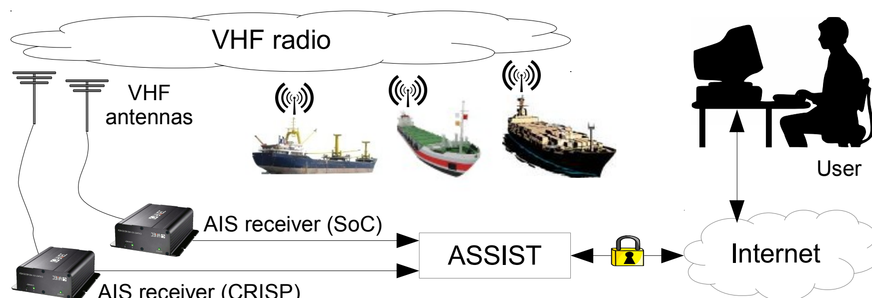


Fig. 1: The scenario and the infrastructure

The *AIS Input Adapter* is in charge of transforming the raw streaming data into events complying with the format requirements by the stream engine. The *RDBMS Input Adapter*, besides the data formatting, typically needs also to render the retrieved data from local databases into streams of *non-expiring* tuples. However, these non-expiring tuples are not persistent, and are removable once none of registered queries needs to access them. Any data source can be input into the system. These operations are completely transparent to the user for whom all sources are data streams.

The query results are continuously delivered to the users via output adapters; in our case we make use of an output adapter encoding each tuple into an XML message for a straightforward integration in the user's AJAX-enabled web interface.

4 Access Control Modelling

The access control model [7] underlying ASSIST is role-based. It can grant the access on a whole data stream, as well as on selected tuples and attributes of a single stream or a joint one. Access at various granularity levels is supported on the basis of *protection object*, which we define formally as follows.

Definition 1 (protection object) *A protection object is a triple* `(STRs, ATTs, EXPs)`*, where* `STRs` *is a set of streams,* `ATTs` *denotes a set of attributes contained in the streams of* `STRs`*, and* `EXPs` *represents the predicates defined on the attributes in* `ATTs`*.*

The concept of protection object adopts an idea similar to that of specifying an *authorization view* in traditional DBMSs. That is, define a view complying with certain access control constraints, and grant the access on the view instead of the underlying relation(s). Basically, the triple `(STRs, ATTs, EXPs)` in the protection object corresponds to `SELECT`, `FROM`, and `WHERE` clauses in a CREATE VIEW SQL statement. In particular, `STRs` denote the streams over which the authorization view is defined. `ATTs` is the set of attributes with granted access,

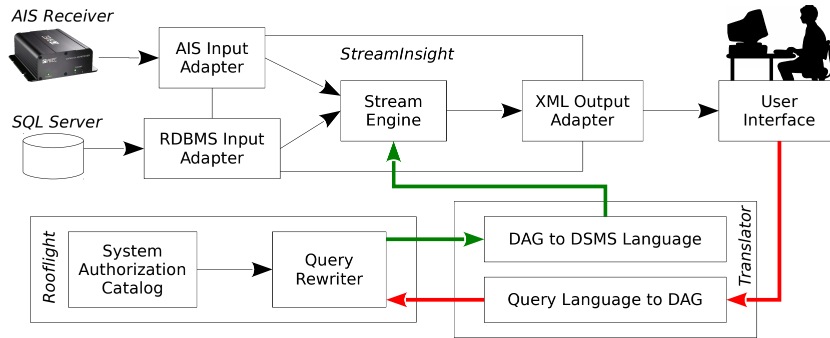


Fig. 2: The architecture of ASSIST

and `EXPs` includes the conditions, specified in terms of predicates on attributes, to be satisfied by the accessible tuples in the view.

ASSIST supports two categories of privileges: *read* privilege allows the selection and projection on attributes in the streaming data, while *aggregate* privilege grants the operations including Count, Avg, Sum, Max, and Min. Therefore, it is possible for a user to check the aggregate information of a set of objects, while hiding the private specific information of each individual one. For example, the port authority can count the number of ships approaching the port, but the details of ships outside the port remain secret.

Many real-world applications are only interested in the information about a recent segment of tuples, instead of the whole unbounded stream. The concept of *window* is proposed especially towards this requirement. A window W is defined by two parameters: window size $|W|$ and step size S. Window W slides forward at every time interval of S. As the sliding continues, a sequence of windows $W_1, W_2, W_3, \ldots,$ are generated, each with the size of $|W|$. Consider Fig. 4 and a data stream DS. Suppose that the origin time of DS is $00:00$, window size is 1 hour and step size is 15 minutes. Window W_1 contains the streaming tuples (in DS) that arrive within the time span of $[00:00, 01:00]$ (i.e., the first hour). After 15 minutes, W_1 slides forward to window W_2, which contains the tuples that arrive between $00:15$ and $01:15$. After another 15 minutes, W_2 advances to W_3, consisting of tuples in the time interval of $[00:30, 01:30]$, and so on. Obviously, a window can be seen as a segment of tuples. It allows a registered continuous query to inquire on the data falling in the window only; the query result is updated each time the window advances. To seamlessly integrate ASSIST with the concept of window, our policy allows the specification of window parameters. In the port authority example, we can define a tumbling window of size 10 minutes, so that the authority is permitted to count ships only in non-overlapping windows, each with size of 10 minutes. Furthermore, ASSIST supports *temporal constraint*, which indicates when a policy is applicable.

All the defined access control policies are stored in a system catalog, i.e., `SysAuth`. Table 2 illustrates an example of SysAuth with 6 policies for 3 roles:

Fig. 3: The visualization of ASSIST

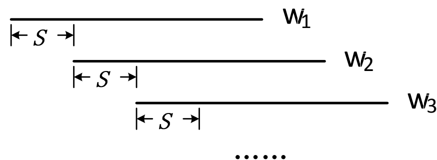


Fig. 4: The illustration of sliding window

port authority, ship company, and ship captain. The policies refer to two data streams: `AIS` and `Route`. The first stream `AIS(SID`[8]`, Name, Type, Long, Lat, T)` contains the information about ship's locations. Attributes `SID`, `Name`, and `Type` record the ship's ID, name, and type, respectively. Attributes `Long` and `Lat` mark the longitude and latitude of the current location of the ship, while timestamp `T` records the time when the tuple arrives. The second stream `Route(SID, next_leg, free_load, ETA)` includes data of logistics. In general, a ship's voyage (or route) is divided into segments (or legs). This stream allows authorized users to check the remaining capacity of a ship and its estimated time of arrival (ETA), before the ship is heading for the next leg. In particular, attribute `SID` is the ship's identify. `next_leg` describes the next port that ship is sailing to, `free_load` records the remaining capacity the ship has, and `ETA` estimates when it will arrive in the next port.

The first policy in Table 2 authorizes port authority to access the information of all the ships within the port waters, where `Location(SID)`, given a ship's ID, outputs the ship's location, and `PortBound(self)` returns the boundary of an addressed port. The second policy permits port authority to count the number of ships, each within a distance of 50 miles from the port waters (`Distance` calculates the distance between two objects). In addition, the count is restricted in windows of minimum 10 minutes with advance size of at least 10 minutes. The third policy is specified on two streams `AIS` and `Route`. Some trading companies may need to know what ships can carry their cargoes from one port to another. The port authority can provide such a service, since the third policy permits her to access logistic data of ships that are within her port waters.

A ship company representative has data access restricted to the company's ships. Such access right is specified in the fourth policy. Function `ShipList` retrieves the set of identifications of the ships in the company. The last two policies are defined for ship captain. The second to last one says that a captain can read the identification and position of any ship within a radius of 200 miles from his/her ship. Finally, the last policy allows a captain to see any information of any ship involved in an emergency event `E` (e.g., collision, grounding, and pirate

[8] A ship's SID is actually its transponder identifier, i.e., MMSI.

Table 2: Example access control policies for AIS data stream.

subject	protection object			privi-lege	time constraint	window	
	STRs	ATTs	EXPs			size	step
Port authority	AIS	*	`Location(SID)` $\in$ `PortBound(self)`	Read			
Port authority	AIS	SID	`Distance(SID, self)` $\leq$ `50 miles`	count		10 min	10 min
Port authority	AIS, Route		`AIS.SID = Route.SID` $\wedge$ `Location(AIS.SID)` $\in$ `PortBound(self)`	Join			
Ship company	AIS	*	`SID` $\in$ `ShipList(self)`	Read			
Ship captain	AIS	SID, Long, Lat	`Distance(SID, self)` $\leq$ `200 miles`	Read			
Ship captain	AIS	*	`InEmergency(E, SID) = true`	Read	[`start(E)`, `end(E)`]		

attack) for the duration of the the emergency (Function `InEmergency` decides whether a ship is involved in a critical situation).

5 Access Control Enforcement

In this section we will enforce the access control model specified in Sect. 4 on the registered continuous queries. We will first define a set of operators in Sect. 5.1 to parse the queries into graphs. Each node in a graph is an operator, and the edge connecting two nodes represents the data flow. Then, in Sect. 5.2 we will introduce three *secure operators* to prune unauthorized data flowing along the edges in the graphs. Based on the above, we present the query rewriting algorithms in Sect. 5.3.

5.1 Query Operators

Before the query rewriting, a query is parsed into a directed acyclic graph (DAG) composed of operators, which include *select*, *project*, *join*, and so on. In the following, we adopt Aurora query algebra [34] to formally define these operators.

Source. Source represents a source stream $\mathcal{SS}(TS,\ A_1,\ A_2,\ \ldots,\ A_n)$, which includes timestamp (i.e., TS) and attributes A_i $(i = 1, 2, \ldots, n)$ as those in a traditional table. Source has no input. It only feeds data to other operators.

Sink. Like source, it is also a stream. But instead of as data source, it is the result generated by the query evaluation. Sink has no output, and only gathers data from other operators.

Select. $\sigma(\mathcal{F})(\mathcal{IS})$ is a select operator, which has an input stream $\mathcal{IS}$ and generates an output stream $\mathcal{OS}$ containing only tuples $t \in \mathcal{IS}$ satisfying predicate $\mathcal{F}$.

Project. $\pi(\mathcal{M})(\mathcal{IS})$ acts like projection. Given an input stream $\mathcal{IS}(TS, a_1, a_2, \ldots, a_m)$, it generates an output stream $\mathcal{OS}(TS, \mathcal{M}(a_1, b_1), \mathcal{M}(a_2, b_2), \ldots, \mathcal{M}(a_m, b_m))$. b_i $(i = 1, 2, \ldots, m)$ is a binary value, and $\mathcal{M}(a_i, b_i)$ keeps attribute a_i in $\mathcal{OS}$ if $b_i = 1$, otherwise, a_i is pruned from $\mathcal{OS}$. Note that the timestamp of input stream is preserved in the output.

Join. $\bowtie(\mathcal{J}, |W_1|, S_1, |W_2|, S_2)(\mathcal{IS}_1, \mathcal{IS}_2)$ is an operator defined on the sliding windows of two input streams. Let W_1 (W_2) be a sliding window of stream $\mathcal{IS}_1$ ($\mathcal{IS}_2$) with step size of S_1 (S_2). Then, join operation is applied between the streaming tuples in W_1 and those in W_2 with respect to the SQL-like predicate $\mathcal{J}$. As windows W_1 and W_2 slide forward continuously at the time interval of S_1 and S_2, respectively, a resultant stream of joined tuples is generated as the output of the join operator.

Aggregate. $\sum(\mathcal{A}, |W|, S)$ $(\mathcal{IS})$ applies "aggregate function", such as `max`, `min`, `count`, `avg`, and `sum`, on the tuples in a sliding window. Let W be a window defined on stream $\mathcal{IS}$ with step size S. Aggregate function $\mathcal{A}$ is evaluated on the tuples within W. We give an example to better clarify this operator. Suppose that we want to learn the number of ships for each `Type` within each hour in `AIS` stream. Then, the following operator can be defined: $\sum$([count(*), group-by `Type`], one hour, one hour)(`AIS`). It specifies a tumbling window on the `AIS` stream with the size of one hour, divides the tuples in the window into groups, each with a distinct `Type` value, and calculates the number of ships within each group.

Union. $\cup(S_1, S_2, \ldots, S_\ell)$ merges two or more input streams into a single output one. The input streams $S_1, S_2, \ldots, S_\ell$ have a common schema.

A query is composed of at least one source operator, any number of intermediary operators (including none) and one and only one sink operator. Each operator's single output stream can be used as input stream for an arbitrary number of operators, and even several times for operators accepting multiple input streams (typically to make a self-join).

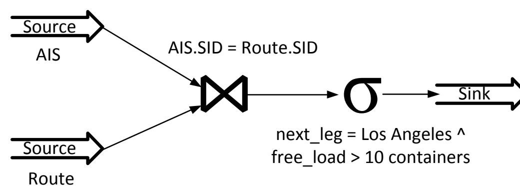


Fig. 5: An example of query graph

Example 1 *Figure 5 shows an example of query graph. The query is to learn the locations and logistical information of ships, which are sailing for Los Angeles and still have a free capacity of at least* 10 *containers. To obtain such information, a join operator is put on the two source streams* `AIS` *and* `Route`*. Then, the joined data is pruned by a selection operator, so that only the data complying*

with the requirements are passed to the sink. The complete representation of the query graph in XML format is available in appendix A.

5.2 Secure Operators

An edge in a query graph Q represents the data flow between two nodes. The data stream flowing along the edge is the output from the upstream node, and the input to the downstream node. In addition, it can also be seen as the output of a subgraph $G_{sub}(Q)$, which contains all the operators (together with the edges connecting them) to generate the stream. A subgraph $G_{sub}(Q)$ is essentially a view, and its output stream can thus be modeled as a protection object ($STRs$, $ATTs$, $EXPs$), where $STRs$ are all the source streams in $G_{sub}(Q)$, $ATTs$ are all the accessed attributes, and $EXPs$ are all involved predicates. Consider Fig. 5. The output stream of the join operator can be modeled as a protection object ({AIS, Route}, {AIS.SID, Route.SID}, AIS.SID = Route.SID), representing a view containing the data by a join between the `AIS` stream and the `Route` stream on the attribute SID.

The data flowing along the edges can also be seen as *internal* streams. They may contain tuples/attributes that are not accessible by the requesting user according to access control policies. ASSIST prunes these unauthorized data by three *secure operators* (defined below): `Secure Read`, `Secure Aggregate`, and `Secure Join`. In brief, the secure operators take these internal streams as the input; they remove unauthorized data by comparing the protection objects in access control policies against those of the internal streams.

Let P be an access control policy in the system catalog, i.e., `SysAuth`. We denote its subject, protection object, privilege, time constraint, and window parameters by $P.R$, $P.PO$, $P.PVG$, $P.TC$, and $P.W$, respectively. Given a user u, we use $\mathsf{Role}(u)$ to denote the set of roles assigned to u. Given an input stream[9] $\mathcal{IS}$, we say a policy P is *read-applicable* on it, if 1) $P.PO.STRs = \mathcal{IS}.STRs$, 2) $\mathcal{IS}.ATTs \subset P.PO.ATTs$, and 3) $P.PVG =$ read. In the following, we use function $\mathsf{read_applicable}(\mathcal{IS}, P)$ to represent the applicability of policy P on stream $\mathcal{IS}$. If it is applicable, then the evaluation of the function is true, otherwise, it is false.

Definition 2 (Secure Read) *Let $\mathcal{IS}$ be an input stream, and u a user. Suppose that $ACP_R(\mathcal{IS}, u) = \{P| \ P.R \in \mathsf{Role}(u) \wedge \mathsf{read_applicable}\ (\mathcal{IS}, P) = true\ \}$ is the set of policies, which are assigned to user u and read-applicable on $\mathcal{IS}$. Then, the data that u is authorized to read from $\mathcal{IS}$ is:*

$$\mathsf{secure_read}(\mathcal{IS}, u) = \bigcup_{P \in ACP_R(\mathcal{IS}, u)} \sigma(\mathcal{F})(\mathcal{IS}),$$

where $\mathcal{F} = P.PO.EXPs \wedge P.TC$.

[9] The input stream can be a source stream or the output stream of an operator.

`Secure Read` receives as input a stream $\mathcal{IS}$ and an access control policy P, and returns the view of $\mathcal{IS}$ containing all and only those tuples/attributes which are accessible according to the constraints, i.e., $P.PO.EXPs$ and $P.TC$, defined in P. If there are multiples policies applicable to $\mathcal{IS}$, the authorized result will be the union of all the returned views derived according to these policies. Assume that ship captain would like to see the ID and locations of ships in `AIS` stream. Then, the last two policies in Table 2 are *read-applicable*, and the data accessible to the captain by a `Secure Read` is:

$$\mathsf{secure_read}(\text{AIS}, \text{ship captain}) = \sigma(\mathcal{F})(\text{AIS})$$

where $\mathcal{F}$=(`Distance(SID, self)` $\leq$ `200 miles`) $\vee$ (AIS.T $\in$ [start(E), end(E)] $\wedge$ `InEmergency(E, SID) = true`).

Given two input streams $\mathcal{IS}_1$ and $\mathcal{IS}_2$, and a join predicate jp defined on them, we say a policy P is *join-applicable* on $\mathcal{IS}_1$ and $\mathcal{IS}_2$ (i.e., $\mathsf{join_applicable}$ ($\mathcal{IS}_1$, $\mathcal{IS}_2$, jp, P) = true), if 1) $P.PO.STRs = \mathcal{IS}_1.STRs \cup \mathcal{IS}_2.STRs$, 2) $(\mathcal{IS}_1.ATTs \cup \mathcal{IS}_2.ATTs \cup jp.ATTs) \subset P.PO.ATTs$, where $jp.ATTs$ are the attributes in jp, and 3) $P.PVG$ = read.

Definition 3 (Secure Join) *Let $\mathcal{IS}_1$ ($\mathcal{IS}_2$) be an input stream, and W_1 (W_2) be a sliding window defined on it with window size $|W_1|$ ($|W_2|$) and step size S_1 (S_2). Suppose user u defines a join predicate jp on W_1 and W_2. Furthermore, suppose that ACP_J ($\mathcal{IS}_1$, $\mathcal{IS}_2$, u) = $\{P|\ P.R \in \mathsf{Role}(u) \wedge \mathsf{join_applicable}\ (\mathcal{IS}_1, \mathcal{IS}_2, jp, P) = true\ \}$ is the set of policies, which are assigned to user u and join-applicable on $\mathcal{IS}_1$ and $\mathcal{IS}_2$. Then, the data that u is authorized to read from a join between W_1 and W_2 with respect to jp is:*

$$\mathsf{secure_join}(\mathcal{IS}_1, \mathcal{IS}_2, jp, u) = \bigcup_{P \in ACP_J(\mathcal{IS}_1, \mathcal{IS}_2, u)} \bowtie (\mathcal{J}, |W_1|', S_1', |W_2|', S_2')(\mathcal{IS}_1, \mathcal{IS}_2),$$

where $\mathcal{J} = jp \wedge P.PO.EXPs \wedge P.TC$, $|W_i|' = \max\{|W_i|, P.W.size\}$, $S_i' = \max\{S_i, P.W.step\}$, and $i = 1, 2$.

The authorization of `Secure Join` is on two incoming streams to ensure that the resultant stream by the join only contains tuples accessible to the target role. It also takes into account the constraints of the window parameters specified in the policy P. In particular, it requires that the permitted size $|W_i|'$ of the sliding window on input stream $\mathcal{IS}_i$ ($i = 1, 2$) is at least as big as $P.W.size$, and that the window step size S_i ($i = 1, 2$) is not smaller than $P.W.step$. As an example, assume that port authority puts a join predicate (i.e., $jp(AIS.SID = Route.SID)$) on the streams `AIS` and `Route` to learn the logistic information of ships in the last 1 hour. Furthermore, she requires that the query result is updated every half hour. The third policy in Table 2 is *join-applicable*, and the resultant data accessible to the port authority by a `Secure Join` is:

$$\mathsf{secure_join}(AIS, Route, jp, \text{port authority}) = \bowtie (\mathcal{J}, 1\ \text{hour}, 0.5\ \text{hour}, 1\ \text{hour}, 0.5\ \text{hour})(AIS, Route),$$

where jp is `AIS.SID = Route.SID`, and we define another predicate lp to be `Location(AIS.SID)` $\in$ `PortBound(self)`, such that $\mathcal{J} = jp \wedge lp$. The window parameters in the submitted query are kept unchanged, since the related policy does not have any specification on them.

Given an input stream $\mathcal{IS}$ and an aggregate function $\mathcal{A}$, we say a policy P is $\mathcal{A}$-*applicable* on $\mathcal{IS}$, if 1) $P.PO.STRs = \mathcal{IS}.STRs$, 2) $\mathcal{IS}.ATTs \subset P.PO.ATTs$, and 3) $P.PVG = \mathcal{A}$.

Definition 4 (Secure Aggregate) *Let $\mathcal{IS}$ be an input stream, and W be a sliding window defined on it with window size $|W|$ and step size S. Suppose user u puts an aggregate function $\mathcal{A}$ on W. Furthermore, suppose that ACP_A ($\mathcal{IS}$, $\mathcal{A}$, u)=$\{P|\ P.R \in \mathsf{Role}(u) \wedge \mathsf{aggregate_applicable}$ ($\mathcal{IS}$, $\mathcal{A}$, P) = true$\}$ is the set of policies, which are assigned to user u and are $\mathcal{A}$-applicable on $\mathcal{IS}$. Then, the aggregated data that u is authorized to access from $\mathcal{IS}$ is:*

$$\mathsf{secure_aggregate}(\mathcal{IS}, \mathcal{A}, u) = \bigcup_{P \in ACP_A(\mathcal{IS}, \mathcal{A}, u)} \sum(\mathcal{A}, |W|', S')(\sigma(\mathcal{F})(\mathcal{IS}))$$

where $|W|' = \max\{|W|, P.W.size\}$, $S' = \max\{S, P.W.step\}$, and $\mathcal{F} = P.PO.EXPs \wedge P.TC$.

Given an aggregate operation $\mathcal{A}$ (e.g., count, sum, and max) over a stream $\mathcal{IS}$, `Secure Aggregate` considers the policies $\mathcal{A}$-*applicable* to $\mathcal{IS}$ for the target aggregate operation. It applies a select operator (i.e., $\sigma(\mathcal{F})(\mathcal{IS})$) on the input stream $\mathcal{IS}$ to create a "view" authorized by these policies, and then returns the aggregate result only over the view. Like `Secure Join`, `Secure Aggregate` also considers the constraints of window parameters. Suppose that port authority needs to count the number of ships around its port waters in the last 30 minutes, and she prefers a query update every 5 minutes. The second policy in Table 2 is *count-applicable*. The aggregate result accessible to the authority by `Secure Aggregate` is:

$$\mathsf{secure_aggregate}(AIS, count, \text{port authority}) = \sum(count, 30 \text{ min}, 10 \text{ min})(\sigma(\mathcal{F})(AIS)),$$

where $\mathcal{F}$ = `Distance(SID, self)` $\leq$ `50 miles`. The window size 30 minutes is kept unchanged, since it is bigger than that defined in the policy. However, according to Definition 4, the original step size (5 minutes) needs to be substituted by the step size (i.e., 10 minutes) in the policy.

5.3 The Algorithm

Our query rewriting approach (Algorithm 1) consists of two passes. In the first pass (Algorithm 2), we apply the secure operators on the query graph to prune unauthorized tuples. In the second pass (Algorithm 3), we combine the authorized graphs generated from the first pass into a single one by uniting the secure operators. In the following, we will present the algorithms.

Data: G: query graph, P: policies set
Result: the authorized graph if rewriting is successful, otherwise, NULL

```
1  RewriteQuery(G, P)
2      op^sink ← Sink operator of G;
3      FirstPass(op^sink, P);
4      if ∃ source operator op^source ∈ G, such that op^source.authorized ≠ true then
5          return NULL;
6      SecondPass(op^sink, false);
7      return authorized graph;
```

Algorithm 1: RewriteQuery

Data: op: a query operator, P: policies set
Result: apply secure operators

```
1  FirstPass(op, P)
2      switch typeof op do
3          case Source
4              if ApplySecureRead(op, P) == success then
5                  Mark source operator op.authorized = true;
6          case Join
7              if ApplySecureJoin(op, P) == success then
8                  Let subGraph(op, G) be the subgraph to generate the output
                   stream of op;
9                  foreach source operator op^source ∈ subGraph(op, G) do
10                     Mark op^source.authorized = true;
11             FirstPass(op.upstream_operator_1(), P);
12             FirstPass(op.upstream_operator_2(), P);
13         case Aggregation
14             if ApplySecureAggregate(op, P) == success then
15                 Let subGraph(op, G) be the subgraph to generate the output
                   stream of op;
16                 foreach source operator op^source ∈ subGraph(op, G) do
17                     Mark op^source.authorized = true;
18             FirstPass(op.upstream_operator(), P);
19         case Union
20             foreach op^up = op.next_upstream_operator() do
21                 FirstPass(op^up, P);
22         case Project
23             FirstPass(op.upstream_operator(), P);
```

Algorithm 2: FirstPass

The first pass (Algorithm 2) traverses the operators of a query graph in a reverse order, that is, from the sink operator to the source ones. When visiting the operators, if some policies are applicable, corresponding secure operators are called to enforce the access control. The enforcement is case by case.

In particular, lines 3-5 are for *Source* operator. We try to apply `Secure Read` on the output stream of the source operator (i.e., the source stream itself). If it is applicable (see Definition 2), then this source operator is marked to be *authorized*, that is, access to it is authorized. Lines 6-12 are for *Join* operator. `Secure Join` is applied on the output stream of the join operator, if possible.

Data: *op*: a graph operator, *needPushUp*: a boolean value
Result: Combine authorized graphs into a single one

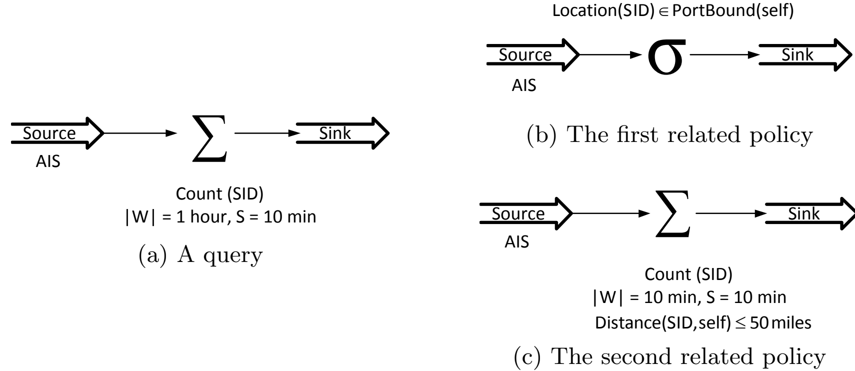
```
1  SecondPass(op, needPushUp)
2    if a secure operator SO is applied on the output stream of op then
3      switch typeof SO do
4        case Secure Read
5          if needPushUp == true then
6            return SO;
7          else
8            Apply SO;
9            return NULL;
10       case Secure Join
11         SO_1^up = SecondPass(op.upstream_operator_1(), true);
12         SO_2^up = SecondPass(op.upstream_operator_2(), true);
13         if SO_1^up ≠ NULL ∧ SO_2^up ≠ NULL then
14           SO = (SO_1^up ∧ SO_2^up) ∨ SO;
15         if needPushUp == true then
16           return SO;
17         else
18           Apply SO;
19           return NULL;
20       case Secure Aggregate
21         SO^up = SecondPass(op.upstream_operator(), true);
22         if SO^up ≠ NULL then
23           SO = SO ∨ SO^up;
24         Apply SO;
25         return NULL;
26   else
27     foreach op^up = op.next_upstream_operator() do
28       return SecondPass(op^up, needPushUp);
```

Algorithm 3: SecondPass

Let *op* be the Join operator, G the query graph, and `subGraph`(op, G) a subgraph, which contains all and only the operators (together with the edges to connect them) in G to generate the output stream of *op*. If `Secure Join` is applicable (see Definition 3), all source operators in `subGraph`(op, G) are marked to be *authorized*, that is, access to them is permitted. Join operator has two input streams. Algorithm `FirstPass` is recursively applied on each input stream. Lines 13-18 are for *Aggregate* operator. If possible, `Secure Aggregate` would be applied and source operators, which are in the subgraph to generate the output stream of the Aggregate operator, are marked. Lines 19-23 are for *Union* and *Project*. The algorithm simply continues to process the upstream operators.

After the first pass, if there exists a source operator in the query graph, such that it is unmarked, then it says that the access to the source stream of the source operator is denied. In such a case, the query is denied (lines 4-5 of Algorithm 1).

Multiple policies can be applicable on a same query. Thus, it is possible that more than one authorized query graph is generated (see Fig. 6 for an example). If this is the case, we try to combine the authorized graphs into a single one by merging secure operators. Algorithm 3 gives the details. Just like Algorithm 2, it is a recursive method, and again it traverses the query operators from the sink operator to source ones (i.e., in a reverse order).



(a) A query

(b) The first related policy

(c) The second related policy

Fig. 6: A query and its related policies (`Secure Aggregate`)

Let *SO* be the secure operator applied on the output stream of the query operator *op*, which is being visited. Lines 4-9 are for the case of *SO* being `Secure Read`. If *SO* can be merged with a downstream secure operator (i.e., $needPushUp ==$ `true`), it is returned, otherwise, *SO* would be applied on the output stream of *op* directly. Lines 10-19 are called, when *SO* is `Secure Join`. Algorithm `SecondPass` is first recursively applied on the two input streams of *op* (here *op* is a Join operator). Let SO_1^{up} and SO_2^{up} be the secure operators pushed

up along the first and second input streams, respectively. If neither SO_1^{up} nor SO_2^{up} is NULL, then they can be combined as $SO_1^{up} \wedge SO_2^{up}$ to prune unauthorized data from the output stream of *op*. In addition, `Secure Aggregate` SO is also applicable, so the three secure operators can be merged together to $(SO_1^{up} \wedge SO_2^{up}) \vee SO$. If the resultant operator does not need to be pushed up, it will be applied on the output stream of *op*. Lines 20-25 are for `Secure Aggregate`. If there are upstream secure operators, they are merged with SO. `Secure Aggregate` puts a select operator (see Fig. 8 for an example) before an aggregate operator to prune unauthorized data. Therefore, we cannot further push up `Secure Aggregate`, otherwise, the output stream of the aggregate operator would be the aggregate results of both authorized data and non-authorized data. Lines 27-28 simply apply the recursion on the upstream operators.

In the following we illustrate our query rewriting process through examples. Mainly, we will show how to apply secure operators to control the accesses to only authorized data. In the first example, two access control policies, one applied by `Secure Read` and the other by `Secure Aggregate`, are involved in the query rewriting. Consequently, Procedure `FirstPass` generates two authorized graphs, each for one policy. Then, Procedure `SecondPass` integrates them into a single one.

Suppose that the port authority would like to count the number of ships in the last 1 hour, and wants the query result to be updated at the time interval of 10 minutes. Then, such a query can be characterized by the graph in Fig. 6 (a). The policies that can be applied to rewrite this given query are the first policy (Fig. 6 (b)) and the second one (Fig. 6 (c)) in Table 2.

The first policy is applied on the source stream `AIS` by `Secure Read`, and creates an authorized view, related only with the ships within the port boundary. Then the aggregate function (i.e., count) is put on the view and returns the number of ships. Figure 7 (a) is the rewritten query.

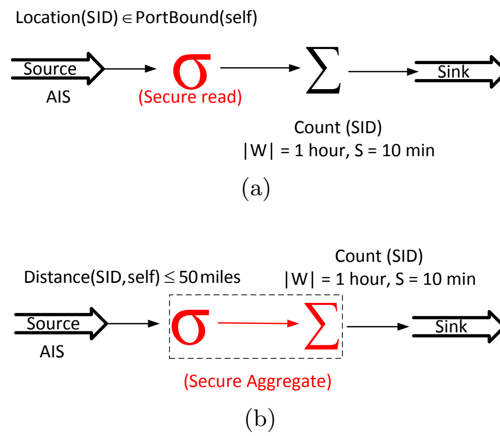


Fig. 7: The two authorized graphs (`Secure Aggregate`)

The second related policy is also applied on source stream `AIS`, but by `Secure Aggregate` instead. The policy restricts the access to those ships with a distance less than 50 miles to the port only. Such a pruning restriction inserts a select operator with predicate `Distance (SID, self)` $\leq$ `50 miles` after the `AIS` stream. The output stream (i.e., a view) of the select operator is the input to the aggregate operator, whose window size has been replace by 1 hour according to the definition of `Secure Aggregate` (Definition 4). The rewritten query is shown in Fig. 7(b).

The two authorized queries by Procedure `FirstPass` are shown in Fig. 7. After that, Procedure `SecondPass` unites the predicates (i.e., those by `Secure Read` and those relevant to `Secure Aggregate`) together, and outputs the final rewritten query in Fig. 8.

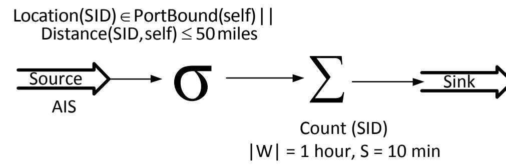


Fig. 8: The united rewritten query (`Secure Aggregate`)

Our second example of query rewriting is an application of `Secure Join`. Suppose that the port authority submits a query as shown in Fig. 5 to access the locations and logistic information of ships, which are sailing for Los Angeles and have a free capacity of at least 10 containers. The *join-applicable* policy for this query is the third one in Table 2. Fig. 9 displays its graph representation.

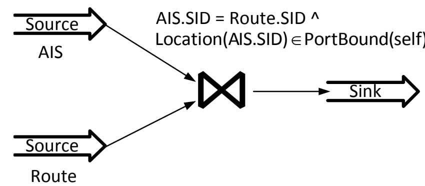


Fig. 9: The graph standing for the third policy in Table 2

According to the specification of the relevant policy, the port authority is permitted to access logistic data of ships only within the boundary of her port. Hence, the extra predicate `Location(AIS.SID)`$\in$`PortBound(self)` is attached to the join operator, and the rewritten query is Fig. 10.

Join operator defined in Sect. 5.1 is put on sliding windows. In the above query rewriting, if the port authority is interested in ship's information within the last 1 hour, then she can specify the sizes of sliding windows on `AIS` and `Route`

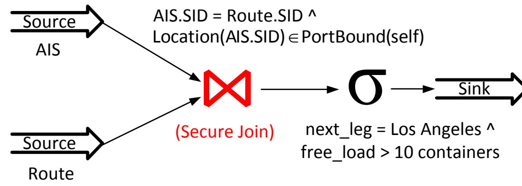


Fig. 10: The rewritten query (`Secure Join`)

to be 1 hour. In addition, she can also configure the step size as half an hour, if she prefers an update every 30 minutes. The relevant access control policy does not have constraints on window parameters, thus the specified window values by the authority will keep unchanged after the query rewriting.

6 Implementation

Data stream engines respond to the need of analyzing large amounts of continuously updating data in a near real-time fashion. They are typically used to power Complex Event Processing frameworks, in which the main goal is to correlate simple events from different sources, infer complex events from them and provide the ability to act upon their occurrence. Many data stream engines are available, Aurora [19], Esper [27], StreamInsight [11], Niagara [35], however no standard has emerged yet.

6.1 StreamInsight

In this work we use StreamInsight [11], which provides an interval-based model derived from the CEDR [28] paradigm; whereas most of the competitors rely on point-based models. In StreamInsight, a tuple is called an event and is defined by its starting time t_s, its ending time t_e, and its payload. The event is said to be valid during the interval $[t_s, t_e[$. The smallest unit of time δ_t is called a *tick*, and an event must respect the property $t_e \geq t_s + \delta_t$. Since the time interval is open on t_e, punctual events are defined by $t_e = t_s + \delta_t$. The stream time is decoupled from the system time, therefore event timestamps can be arbitrarily manipulated in the queries.

The data is inserted through an input adapter which is in charge of creating the events. The result of a query is made available via an output adapter. These adapters must be written in C# to be handled correctly by StreamInsight.

The stream engine provides a mechanism to modify a payload or revoke an event altogether from the query. In order to validate the events that have been emitted, each input adapter must sooner or later insert a Current Time Increment (CTI) event. This special event is composed of only one timestamp t_v and indicates that no more event with $t_s \leq t_v$ will be inserted, modified or revoked. Some operators (e.g. the sink operator) will block and buffer the events until a CTI event ensures that they can release them downstream.

The time model used by StreamInsight modifies the behaviour of the join operator. It adds a necessary condition for joining two events which is that their validity intervals must overlap. In addition to some dedicated operators that allow arbitrary modifications of a stream's events' intervals, some windows can also modify events upon insertion.

StreamInsight also accepts user-defined functions and aggregates that must be written in C# (just like the adapters). This allows us to provide runtime data (user ID, location, etc.) to predicates in a generic query.

6.2 XML Encoding

The query rewriting module of ASSIST is imposed on graphs. Internally, we use XML to encode a query graph. Each query operator (see Sect. 5.1) in the graph is represented as a node in an XML document. Figure 11 shows the join operator of the query graph in Fig. 5. Lines 1 and 20 mark the beginning and ending of the join operator. Lines 2-3 are the internal identifiers of the two input streams (i.e., `AIS` and `Route`) of the join operator. The output schema of the operator is specified in Lines 4-13. Lines 14-19 are the definition of the join predicate. A complete XML encoding of the query graph in Fig. 5 can be found in Appendix A.

```
 1.<join name="Join.0">
 2.  <inputstream name="Source.0.out"/>
 3.  <inputstream name="Source.1.out"/>
 4.  <outputstream name="Join.0.out"/>
 5.  <attribute name="MMSI">
 6.    <input name="MMSI" inputstream="Source.0.out"/>
 7.  </attribute>
 8.  <attribute name="Free_load">
 9.    <input name="Free_load" inputstream="Source.1.out"/>
10.  </attribute>
11.  <attribute name="Next_leg">
12.    <input name="Next_leg" inputstream="Source.1.out"/>
13.  </attribute>
14.  <predicate>
15.    <equals>
16.      <attribute name="MMSI" inputstream="Source.0.out"/>
17.      <attribute name="MMSI" inputstream="Source.1.out"/>
18.    </equals>
19.  </predicate>
20.</join>
```

Fig. 11: The encoding of the join operator in Fig. 5

Besides the query graph, we encode the access control policies also by XML. Appendix B shows the policies defined in Table 2.

6.3 LINQ

The queries are typically written with the help of LINQ [29]. This special library can be used to extend .NET languages with query-like capacities. This allows us to write queries directly in C# for instance and have them automatically translated into the form that StreamInsight needs internally for importing queries.

More importantly, we can export and import queries as XML documents. Therefore once the query is written, it can be exported as an XML document containing a directed acyclic graph. We can then translate it easily to and from our own DAG language used for query rewriting.

The choice of StreamInsight and LINQ is in reference to the existing works [30, 31], that address the challenges posed to event processing systems when dealing with data streams containing spatial information, as it is the case in the application at hand. These works leverage the flexibility provided by StreamInsight and LINQ to integrate SQL Server Spatial Libraries [36] to efficiently process spatial data.

6.4 Rooflight

Our query rewriting module, called *Rooflight*, is written in C# for the sake of keeping a uniform framework. Its code is however independent and contained in a standalone library.

System Authorization Catalog We choose to model the contents of our System Authorization Catalog after the Role-Based Access Control canonical one, but in a simplified form. The result can be seen in Fig. 12. A given role is composed of at least one policy, and at any given time a user is authenticated as one and only one role.

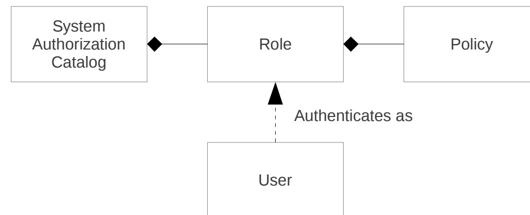


Fig. 12: ASSIST's simplified RBAC model

It is a positive security model or a "whitelist." By default, roles are denied access to all resources. Policies must be written to grant access to these resources

and there is no mechanism to deny access afterwards. Because of that we provide a fine granularity on predicates.

The System Authorization Catalog is stored in an XML document, whose structure is a straightforward implementation of the model as seen in Fig. 12. An example is provided in appendix B.

Query Rewriter The Query Rewriter is based on algorithm 1. Its interface consists of a single method, which takes as input two parameters: a query graph (stored in an XML document), and the name of the role assumed by the user. Upon the condition that the query rewriting is successful, it returns the modified graph, encoded in another XML document. Otherwise, it returns an error code (parsing error, query rejected, unknown role, and etc.).

Internally, the query is represented by a graph, composed of operators defined in Sect. 5.1. We prepare the graph by a initialization pass going from the source operators to the sink operator, executing the following steps for each operator:

1. Because the graph is not necessarily an arborescence, the operators aren't connected at creation time. We reconstruct the streams between the operators thanks to their unique given names.
2. We identify the source attribute of each attribute used by the operator, when it is possible. This allows us to keep track of the use of each source attribute and apply policies on operators that do not deal directly with them.
3. Once we know which source attributes are potentially used in the operator's input, we select policies that match at least one of these. In the case of aggregates, we only select policies with compatible values.

Once the initialisation pass is complete, the graph is ready for rewriting.

7 Evaluation

We conduct our experiments using StreamInsight [11] as stream engine. All experiments were conducted on an Intel i5-650 @3.2GHz with 4 GB of RAM running Windows 7. Since .NET uses a JIT compiler, we make a first run of experiments without benchmarking it so the compilation time of the code doesn't influence the measures. To minimize the effects of the garbage collector, we automatically trigger it before each call to the query rewriter.

7.1 Experiment 1

The first experiment compares the performance between query rewriting and post-filtering techniques. To this end, we measure the number of tuples output by a rewritten query against the number of tuples output by the original query that would then need to be analyzed by the post-filter. The query consists of a join between the AIS stream giving the boats position and a synthetic company feed that provides information about the next destination and available space of

each boat. It has been described at the last example of Sect. 5.3. The original query corresponds to Fig. 9 and the securely rewritten query corresponds to Fig. 10.

The synthetic feed is generated from the AIS feed. We keep 90% of the boats from the AIS feed and create 10 times more fake ones. Each boat has a destination chosen among 10 ports and an amount of free cargo (always greater than zero). Boats are attributed to a given company according to a normal distribution (with mean $\mu = 30$). Every 5 minutes, a company sends messages about its boats destination and free cargo space, with a limit of 10 messages per second. We only decrease the free cargo space of some ships during that update (eventually reaching the threshold defined by the policy).

Results are shown in Table 3. We note that the original query outputs about ten times as many tuples as the rewritten query, which is to be expected since only 10% of the boats have the correct destination and they all start with a fair amount of free cargo space. The benefits of query rewriting are clear when joins are involved since the filtering helps reduce drastically the number of tuples flowing along the streams.

Table 3: Query rewriting vs post-filtering

	Tuples	
	Average number	Std deviation
Query Rewriting	1376.2	561.3
Post-Filtering	15864.1	53.3

7.2 Experiment 2

We now measure the performance of the query rewriter for queries with different sizes. We generated queries with a big number of operators, namely 10, 20, 30 and 60 operators. The different types of operators are listed in Table 4. These queries have the same input streams.

We then generate random policies applicable on all 18 input streams of Q_{60} (and the smaller ones). This set of random policies consists of 45 `Secure Reads`, 10 `Secure Joins` and 5 `Secure Aggregates`. In order to provide consistent results the first 18 `Secure Reads` correspond to `Secure Reads` on all attributes of each stream; therefore the query rewriting will always succeed. We run a hundred times the same query with the same set of policies, and report the average of their results.

The results are available in Fig. 13a. Even though the processing time seems to grow slightly faster than linearly as a function of the number of operators, the execution time for a very large query remains acceptable in the context of running queries that are expected to run for long period of times.

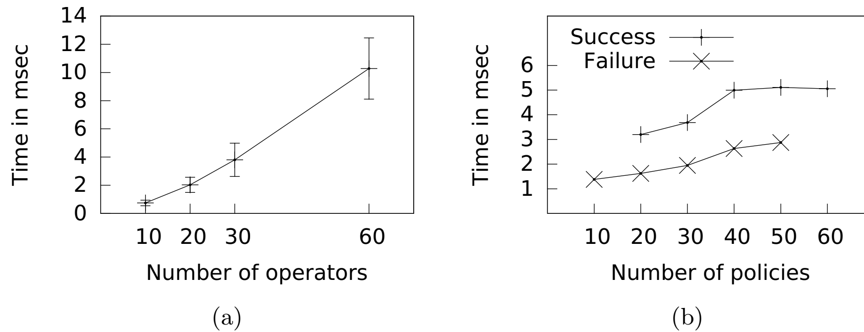
Table 4: The operators of the queries with various sizes

Query	Streams	Operators
Q_{10}	3 IN	4 π, 2 σ, 1 no-op, 1 $\cup$, 1 $\bowtie$, 1 Σ
Q_{20}	6 IN	7 π, 5 σ, 2 no-op, 1 $\cup$, 4 $\bowtie$, 1 Σ
Q_{30}	9 IN	11 π, 7 σ, 2 no-op, 1 $\cup$, 7 $\bowtie$, 2 Σ
Q_{60}	18 IN	22 π, 14 σ, 4 no-op, 2 $\cup$, 15 $\bowtie$, 3 Σ

7.3 Experiment 3

We also investigate the effect of the number of policies on the performance of the query rewriter. To that effect, we evaluate query Q_{30} and we generate random sets of policies for it. As in the previous experiment a sequence of policies were generated, comprised of 36 `Secure Reads`, 10 `Secure Joins` and 5 `Secure Aggregates`. We also add a `Secure Read` for each source stream on all of its attributes, raising the total number of `Secure Reads` to 45. We randomly select policies from the sequence to compose sets of policies, with cardinalities varying from 10 to 60 in steps of 10. We generate 100 different samples for each cardinality lower than 60.

We can see in Fig. 13b that our query rewriting algorithm scales well with respect to the number of policies. When a query is not permitted by the polices, the early termination of the query rewriter introduces a bias in the measurement, this is why we separate the cases of query permission and query refusal. Even so, both cases show a similar behavior in terms of scaling.



(a) (b)

Fig. 13: Performance of the query rewriter

8 Conclusions

The distribution of AIS data over the Internet must be controlled in order to guarantee safety, security, and privacy. In this paper we build on top of StreamIn-

sight (a commercial stream engine) an access control system, ASSIST, which based on our engine/application independent framework [7] allows relevant users to access events conforming to pre-defined policies. We demonstrate the prototype in a scenario with the real AIS streaming events captured from the monitoring of the ship traffic in the port of Singapore. The experiments show that our solution is practical and effective.

Acknowledgements. This research was funded by the A*Star SERC project "Hippocratic Data Stream Cloud for Secure, Privacy-preserving Data Analytics Services" 102 158 0037, NUS Ref: R-702-000-005-305.

References

1. International Telecommunications Union: Technical characteristics for an automatic identification system using time-division multiple access in the vhf maritime mobile band
2. IMO: International maritime organization. `http://www.imo.org/ourwork/safety/navigation/pages/ais.aspx`
3. Knezo, G.J.: Sensitive but unclassified and other federal security controls on scientific and technical information: History and current controversy. USA's Congressional Research Service (2003)
4. Lane, R.O., Nevell, D.A., Hayward, S.D., Beaney, T.W.: Maritime anomaly detection and threat assessment. Proc. of the Int. Conf. on Information Fusion (2010) 1–8
5. Piciarelli, C., Foresti, G.L.: On-line trajectory clustering for anomalous events detection. Pattern Recognition Letters (2006) 1835–1842
6. de Vries, G., van Someren, M.: Clustering vessel trajectories with alignment kernels under trajectory compression. Proc. of the 2010 European Conf. on Machine learning and Knowledge Discovery in Databases (2010) 296–311
7. Carminati, B., Ferrari, E., Cao, J., Tan, K.L.: A framework to enforce access control over data streams. ACM Trans. on Information and System Security **13**(3) (2010)
8. Lindner, W., Meier, J.: Securing the borealis data stream engine. (2006) 137–147
9. Nehme, R.V., Rundensteiner, E.A., Bertino, E.: A security punctuation framework for enforcing access control on streaming data. Proc. of ICDE (2008) 406–415
10. Nehme, R.V., Lim, H.S., Bertino, E.: Fence: Continuous access control enforcement in dynamic data stream environments. Proc. of ICDE (2010) 940–943
11. Microsoft StreamInsight: `http://msdn.microsoft.com/en-us/library/ee362541.aspx`
12. Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., Widom, J.: Stream: The stanford stream data manager. In: Proc. of SIGMOD. (2003)
13. Law, Y.N., Wang, H., Zaniolo, C.: Query languages and data models for database sequences and data streams. In: Proc. of VLDB. (2004) 492–503
14. Babcock, B., Babu, S., Datar, M., Motwani, R., Thomas, D.: Operator scheduling in data stream systems. VLDB Journal **13**(4) (2004) 333–353
15. Domingos, P., Hulten, G.: Mining high-speed data streams. In: Proc. of KDD. (2000) 71–80

16. Zhang, P., Zhu, X., Shi, Y.: Categorizing and mining concept drifting data streams. In: Proc. of KDD. (2008) 812–820
17. Luo, C., Thakkar, H., Wang, H., Zaniolo, C.: A native extension of sql for mining data streams. In: Proc. of SIGMOD. (2005) 873–875
18. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The design of the borealis stream processing engine. In: Proc. of CIDR. (2005) 277–289
19. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Aurora: a new model and architecture for data stream management. VLDB Journal **12**(2) (2003) 120–139
20. Zdonik, S.B., Stonebraker, M., Cherniack, M., Çetintemel, U., Balazinska, M., Balakrishnan, H.: The aurora and medusa projects. IEEE Data Eng. Bull. **26**(1) (2003) 3–10
21. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.A.: Telegraphcq: Continuous dataflow processing for an uncertain world. In: Proc. of CIDR. (2003)
22. Schreier, U., Pirahesh, H., Agrawal, R., Mohan, C.: Alert: An architecture for transforming a passive dbms into an active dbms. In: Proc. of VLDB. (1991) 469–478
23. Sullivan, M.: Tribeca: A stream database manager for network traffic analysis. In: Proc. of VLDB. (1996) 594
24. Liu, L., Pu, C., Tang, W.: Continual queries for internet scale event-driven information delivery. TKDE **11**(4) (1999) 610–628
25. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: Niagaracq: a scalable continuous query system for internet databases. In: Proc. of SIGMOD. (2000) 379–390
26. Zhu, Y., Rundensteiner, E.A., Heineman, G.T.: Dynamic plan migration for continuous queries over data streams. In: Proc. of SIGMOD. (2004) 431–442
27. Esper: `http://esper.codehaus.org/`
28. Barga, R.S., Goldstein, J., Ali, M.H., Hong, M.: Consistent streaming through time: A vision for event stream processing. Biennial Conf. on Innovative Data Systems Research (2007) 363–374
29. LINQ (Language-Integrated Query): `http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx`
30. Ali, M.H., Chandramouli, B., Raman, B.S., Katibah, E.: Real-time spatio-temporal analytics using microsoft streaminsight. Proc. of ACM GIS (2010) 542–543
31. Kazemitabar, S.J., Demiryurek, U., Ali, M.H., Akdogan, A., Shahabi, C.: Geospatial stream query processing using microsoft sql server streaminsight. Proc. of VLDB Endow. **3** (2010) 1537–1540
32. Cao, J., Carminati, B., Ferrari, E., Tan, K.L.: Acstream: Enforcing access control over data streams. In: Proc. of ICDE. (2009) 1495–1498
33. Carminati, B., Ferrari, E., Tan, K.L.: Specifying access control policies on data streams. In: DASFAA. (2007) 410–421
34. Abadi, D.J., Carney, D., çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Aurora: a new model and architecture for data stream management. VLDB Journal **12**(2) (2003) 120–139
35. NiagaraST: `http://datalab.cs.pdx.edu/niagara/`
36. Spatial Data (SQL Server): `http://msdn.microsoft.com/en-us/library/bb933790.aspx`

A XML representation of a Query Graph

```
<root>
  <query>
    <inputstream name="AisGpsStream">
      <outputstream name="Source.0.out" />
      <attribute name="MMSI" />
      <attribute name="Longitude" />
      <attribute name="Latitude" />
    </inputstream>
    <inputstream name="RouteStream">
      <outputstream name="Source.1.out" />
      <attribute name="MMSI" />
      <attribute name="Next_leg" />
      <attribute name="Free_load" />
    </inputstream>
    <join name="Join.0">
      <inputstream name="Source.0.out" />
      <inputstream name="Source.1.out" />
      <outputstream name="Join.0.out" />
      <attribute name="MMSI">
        <input name="MMSI" inputstream="Source.0.out" />
      </attribute>
      <attribute name="Free_load">
        <input name="Free_load" inputstream="Source.1.out" />
      </attribute>
      <attribute name="Next_leg">
        <input name="Next_leg" inputstream="Source.1.out" />
      </attribute>
      <predicate>
        <equals>
          <attribute name="MMSI" inputstream="Source.0.out" />
          <attribute name="MMSI" inputstream="Source.1.out" />
        </equals>
      </predicate>
    </join>
    <filter name="Filter.0">
      <inputstream name="Join.0.out" />
      <outputstream name="Filter.0.out" />
      <predicate>
         <bool_and>
          <equals>
            <attribute name="Next_leg" inputstream="Join.0.out" />
            <litteral type="System.String" value="Los Angeles" />
          </equals>
           <greater_than>
```

```
            <attribute name="Free_load" inputstream="Join.0.out" />
            <litteral type="System.Int64" value="10" />
          </greater_than>
        </bool_and>
      </predicate>
    </filter>
    <outputstream name="OutputAdapter">
      <inputstream name="Filter.0.out" />
    </outputstream>
  </query>
</root>
```

B XML representation of the System Authorization Catalog


```
<?xml version="1.0"?>
<root>
  <role name="Port Authority">
    <policy>
      <inputstream name="AisGpsStream" attributes="*"/>
      <predicate>
        <method name="IsWithinPortBound">
          <attribute inputstream="AisGpsStream" name="Longitude"/>
          <attribute inputstream="AisGpsStream" name="Latitude"/>
        </method>
      </predicate>
      <privilege value="read"/>
      <gtc/>
    </policy>
    <policy>
      <inputstream name="AisGpsStream" attributes="*"/>
      <predicate>
        <lessthan>
          <method name="DistanceFromSelf">
            <attribute inputstream="AisGpsStream" name="Longitude"/>
            <attribute inputstream="AisGpsStream" name="Latitude"/>
          </method>
          <litteral type="double" value="50.0"/>
        </lessthan>
      </predicate>
      <privilege value="aggregate">
        <wtc units="seconds" minsize="600" minhop="600"/>
      </privilege>
      <gtc/>
    </policy>
    <policy>
      <inputstream name="AisGpsStream" attributes="*"/>
      <inputstream name="RouteStream" attributes="*"/>
      <predicate>
        <bool_and>
          <equals>
            <attribute inputstream="AisGpsStream" name="MMSI"/>
            <attribute inputstream="RouteStream" name="MMSI"/>
          </equals>
          <method name="IsWithinPortBound">
            <attribute inputstream="AisGpsStream" name="Longitude"/>
            <attribute inputstream="AisGpsStream" name="Latitude"/>
          </method>
```

```
        </bool_and>
      </predicate>
      <privilege value="join"/>
      <gtc/>
    </policy>
  </role>
  <role name="Ship Company">
    <policy>
      <inputstream name="AisGpsStream" attributes="*"/>
      <predicate>
        <method name="BelongsToCompany">
          <attribute inputstream="AisGpsStream" name="MMSI"/>
        </method>
      </predicate>
      <privilege value="read"/>
      <gtc/>
    </policy>
  </role>
  <role name="Ship Captain">
    <policy>
      <inputstream name="AisGpsStream" attributes="MMSI,Longitude,Latitude"/>
      <predicate>
        <lessthan>
          <method name="DistanceFromSelf">
            <attribute inputstream="AisGpsStream" name="Longitude"/>
            <attribute inputstream="AisGpsStream" name="Latitude"/>
          </method>
          <litteral type="double" value="200.0"/>
        </lessthan>
      </predicate>
      <privilege value="read"/>
      <gtc/>
    </policy>
    <policy>
      <inputstream name="AisGpsStream" attributes="*"/>
      <predicate>
        <method name="HasAnEmergency">
          <attribute inputstream="AisGpsStream" name="Emergency"/>
        </method>
      </predicate>
      <privilege value="read"/>
      <gtc start="Emergency.start" end="Emergency.stop"/>
    </policy>
  </role>
</root>
```