Tagging Stream Data for Rich Real-Time Services

Rimma V. Nehme
Purdue University
West Lafayette, IN 47906
rnehme@cs.purdue.edu

Elke A. Rundensteiner
Worcester Polytechnic Institute
Worcester, MA 01608
rundenst@cs.wpi.edu

Elisa Bertino
Purdue University
West Lafayette, IN 47906
bertino@cs.purdue.edu

ABSTRACT

In recent years, data streams have become ubiquitous as technology is improving and the prices of portable devices are falling, e.g., sensor networks, location-based services. Most data streams transmit only data tuples based on which continuous queries are evaluated. In this paper, we propose to enrich data streams with a new type of metadata called *streaming tags* or short *tick-tags*[1]. The fundamental premise of tagging is that users can label data using uncontrolled vocabulary, and these tags can be exploited in a wide variety of applications, such as data exploration, data search, and to produce "enriched" with additional semantics, thus more informative query results. In this paper we focus primarily on the problem of continuous query processing with streaming tags and tagged objects, and address the *tick-tag* semantic issues as well as efficiency concerns. Our main contributions are as follows. First, we specify a general and flexible *Stream Tag Framework* (or short STF) that supports a *stream-centric* approach to tagging, and where *tick-tags*, attached to streaming objects are treated as first-class citizens. Second, under STF, users can query tags *explicitly* as well as *implicitly* by outputting the tags of the base data together with query results. Finally, we have implemented STF in a prototype Data Stream Management System, and through a set of performance experiments, we show that the cost of stream tagging is small and the approach is scalable to a large percentage of tagged objects.

1. INTRODUCTION

1.1 Tagging in Data Stream Environments

Data streams are common in applications from location-based services and traffic management to environmental and health sensing. Over the past few years, a large amount of research has been dedicated to the design and development of Data Stream Management Systems (DSMSs) [1, 8, 15, 37]. With the exception of a few systems [21, 25, 30, 34], most DSMSs assume that data streams transmit exclusively data tuples (without any additional metadata embedded inside streams), and continuous queries are evaluated on these streaming data tuples.

[1]We chose the name "*tick-tags*" to capture the transient nature of attached labels and distinguish them from traditional "*tags*" (e.g., for web pages, images, files) that tend to be static and persistent.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '09, August 24-28, 2009, Lyon, France
Copyright 2009 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

In this paper, we propose to enrich data stream environments with a special type of metadata called *streaming tags*, or short *tick-tags*[2]. An informal definition of tagging is the process of adding comments or labels to something. The problem of stream tagging is important, because high volume continuous data streams are ubiquitous, and stream processing applications are becoming vital in our every-day life ranging from real-time traffic monitoring to emergency response and health monitoring. Tags on streaming data can enrich existing appications [24, 36] and can enable and inspire novel useful services:

- *Healthcare.* Consider a patient carrying a health monitoring device that measures his heart rate. When a heart rate becomes abnormal, the doctor gets alerted about it. The patient, to prevent unnecessary concerns, attaches a tag to his real-time streaming measurements stating "`Running`". The doctor or a nurse knows (based on the tag) what is causing the change in the health measurements. This helps prevent second-guessing and avert issuing unnecessary alerts.
- *Location-Based Services.* In location-based services, moving objects send their location updates to receive location-aware responses. Consider a traffic scenario depicted in Figure 1. An accident occurs on a highway which causes extensive traffic jams. People stopped far away from the scene of the accident wonder what is causing the stopped traffic: an accident, a road construction, or a "curiosity factor"? A driver close to the accident attaches a tag to his location update describing what he is observing: "`Accident`, `2 cars`, `Near Exit 12`". Using this tag information, other drivers may determine how to procede: take the nearest exit, notify others about their delay, or possibly even help if medical assistance is needed.
- *Scientific Experiments.* Tags (or annotations) play an increasingly crucial role in scientific exploration and discovery, as the amount of experimental data and the level of collaboration among scientists increases [17]. Tags can be attached to real-time experimental measurements that researchers can subsequently exploit in analysis. For instance, a tag "`Heat-Then-Measure`"

[2]The terms "*streaming tags*" and "*tick-tags*" represent the same concept in this paper and are used interchangeably.

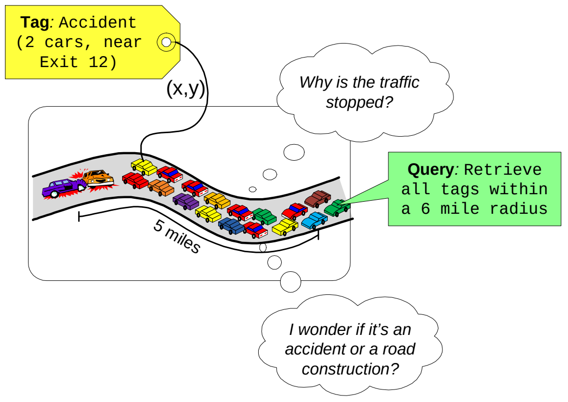

Tag: Accident (2 cars, near Exit 12)
(x,y)
Why is the traffic stopped?
Query: Retrieve all tags within a 6 mile radius
5 miles
I wonder if it's an accident or a road construction?

Figure 1: Traffic example with real-time tags.

may describe how a procedure was performed, "`Magnetic Field`" may specify what other factors were present, and "`Strong Smell`" may characterize what else was observed when a measurement was taken.

The examples above depict a small subset of powerful applications that streaming tags can enable. Other ways of leveraging streaming tags may include: (1) *Stream data tracking*, where tagged objects can be located and tracked unambiguously. (2) *Creation of rich user profiles*, where information about user's interests, mood, observations, and character can be revealed based on the tags employed by her in real time, and used in privacy preservation or tailored services. (3) *Exploration and browsing of data*, which can be achieved by exploiting tags as a navigation mechanism allowing users to find related streaming data based on the tags (see Section 4.1). (4) *Social communication*, where by allowing other people to tag a specific subset of real-time data with their own tags, one can find out what different people think about the same piece of information. For instance, one scientist's opinion (expressed via a tag) on real-time measurements in an experiment might vary significantly from the tags attached by other researchers.

In general, we envision stream tagging being useful in almost any application in which streams are produced or consumed.

1.2 Challenges

Due to the inherent characteristics of streaming environments, tagging of streaming objects is a challenging task. Stream data is typically characterized by large volumes, high input rates, generated by multiple distributed data sources. Processing of continuous queries on streaming data requires near-real response time. Yet contrary to traditional databases, data is not persistently stored on DSMS, but rather streamed through it once and then discarded. A system supporting tagging of streaming data must consider scalability, output rate, latency and resource utilization. To be useful in practice, a tagging mechanism must be able to support a variety of tagging granularities: users should be able to tag streams, tuples, attributes, or specific data values. For instance, if a set of data tuples correspond to a particular physical phenomenon (e.g., a hurricane), then it is useful to tag all those tuples with a single tag. Alternatively, if a particular data value is called into question, users should be able to attach a tag to an individual data value as well. Naturally, the more fine-grained the tagging is, the higher the overhead it may potentially incur. Furthermore, due to the infinite nature of streams and typically long-running queries, frequent changes in data are likely to occur, which translates into possibly very frequent changes in tags' contents and statistics.

1.3 Alternative Tagging Approaches

To motivate our proposed solution, next we discuss alternative approaches for tagging streaming data.

Table Approach. One solution is to build a separate global table in DSMS, where all tags are maintained. For each tag, the "links" to the appropriate streaming data elements in the form of query predicates are stored, as illustrated below:

Tag	**Link To Data**
"Running"	`SELECT measurement FROM HeartRate WHERE time > 9:00AM and time < 9:30AM`

Using this method, the tags are maintained separately from the streaming data. As a result, this method may potentially incur significant overheads. All tags arriving to DSMS (separately from the data) must be processed, and for every tag an entry in the central tag table must be created or updated. To identify the data, to which the tags are applicable to, a separate continuous query must be instantiated (in the worst case, one-per-tag) to find the corresponding streaming data elements. If there are many tags, this may severely impact the performance of the system, as significant amount of resources would be taken away from the regular continuous query processing. Furthermore, after the steaming data passes through the DSMS, their respective tags must be deleted from the global tag table, thus further increasing the tag-related maintenance overhead.

Extended Data Tuples. An alternative approach to tagging is to extend the schema of streaming data tuples by adding an additional attribute, where tag information could be stored. Here, tags are strongly coupled with data tuples. Although attractive, this approach has several limitations. First, by increasing tuples' sizes, more memory and processing resources are consumed. Second, tags may apply to a collection of tuples or data values, but using this method, tags would have to be duplicated, even if several tuples share the same tag. Furthermore and more importantly, when searching for tags or tagged data, every tuple must be looked at to see if the tag content matches the search predicate. This approach suffers from the same problem as the "non-normalized" representation of data in relational databases, which calls for optimization of design [22].

Streaming XML. Another possible solution for tagging is to exploit streaming XML [5, 35]. XML is human-legible and is designed to be self-describing. This enables the capability to define self-describing data elements by users. However, XML technology is complex and XML query processing (using either XQuery or XPath languages) was not intended to be evaluated over bursty streams. Even with the extensions supporting XML data streams such as [31, 42], continuous processing of frequent XML-based tags is likely to be expensive and can seriously limit the performance of DSMS.

Streaming Tags. Our proposed solution is to introduce a special type of streaming metadata called the *streaming tags* or *tick-tags*. *Tick-tags* are embedded inside data streams and uniquely identify the streaming data objects (e.g., tuples, tuple attributes or data values) to which additional semantic labels are attached. The advantage of the *tick-tag* approach is three-fold. First, *tick-tags* can be shared by several streaming objects, thus reducing memory and processing overheads. Second, *tick-tags*, interleaved with streaming data, facilitate a faster search for the objects they are applicable to. Furthermore, *tick-tags* can be just as dynamic as the streaming data and can be exploited in continuous query optimization in a similar fashion as data tuples. The query optimizer can determine the best order of operators by considering both the data statistics as well as the streaming tags' statistics. Finally, if users decide *not* to tag their data, then the data streams are identical to traditional data streams, and the existing query processing solutions for regular streaming environments are applicable as before.

1.4 Our Proposed Solution: STF

In this paper, we present a *Stream Tag Framework* (or short STF) that provides full-fledged support for tagging of streaming data. In our endeavor, we strive to achieve the following goals:

Stream-centric tags. Tags applicable to streaming objects are not transmitted and stored separately from the actual data, but rather interleaved with data tuples inside data streams. Streaming tags have a transient nature – they are not stored permanently on the server, but rather "make a one pass" through the system and then may get discarded.

User-centric tags. Different users may have unique understandings and explanations for the same piece of information, thus it is essential for a tagging framework to support "personalized" tags with respect to data. Users may also want to customize the time setting of their tags – whether they should be attached and be applicable only once or for some time in the near future. We refer to this feature – a *user-centric* tag semantics.

Explicit Querying of Tags. Users or applications should be able to query streaming tags *explicitly*, in an ad-hoc or in a continuous manner. We call this feature – *tag-oriented query processing* (see Figure 2). For example, a location-based application may specify a range query Q: *Continuously retrieve all streaming tags specified by users in the downtown of City X*[3]. Here the results of the query Q are characterized by a continuous stream of *tick-tags* that appear in the specified geographic region.

Enriched Query Results. Regular continuous queries can also produce more superior ("tag-enriched") results [9, 29]. This functionaliy is enabled by *tag-aware query processing* (Figure 2). The goal here is to preserve the tags attached to the original data based on which the query results are computed. For example, if a tag calls into question the veracity of some data value, one would like this information to be available to anyone who sees the results of the query based on this information. The main challenge in this context is to correctly propagate streaming tags through the query plan, while the tags' corresponding data is being filtered, projected out or joined with other data tuples.

Tag Query Language. Finally, a comprehensive tagging system must provide a high-level language to attach tags to streaming data, to query them or to specify that enriched results should be produced for a given continuous query. For this purpose, we introduce a declarative *Tag Query Language* (or short *TAG-QL*), which provides an intuitive interface for users to perform the above-mentioned actions.

1.5 Our Contributions

The contributions of our *Stream Tag Framework* (*STF*) can be summarized as follows:

[3] Here, we assume that *tick-tags* are attached to streaming data with location attribute.

- **Tag Model**. We describe the *tick-tag* metadata model for tagging various streaming objects, e.g., tuples, data values, etc. *Tick-tags* are embedded inside streams and support a wide variety of user-based semantics.
- **Tag Query Language**. We introduce a *Tag Query Language* (or short TAG-QL) that enables declarative specification and querying of streaming tags.
- **Tag-Oriented Query Processing**. Users can attach and *explicitly* query *tick-tags*. We describe the tag-oriented query algebra that enables this functionality.
- **Tag-Aware Query Processing**. We also support *implicit* querying of tags, where query results are enriched with the tags of the base data. We describe the extensions to the continuous query algebra to support the correct propagation of tags in a query pipeline.
- **Implementation and Experiments**. To illustrate the feasibility, STF has been implemented in a prototype DSMS called *CAPE* [15]. Our experimental analysis shows scalability and benefits of the *tick-tag* approach, and the costs associated with tag-awareness.

The rest of the paper is organized as follows: We give an overview of STF, data model and assumptions in Section 2. Section 3 presents the design of streaming *tick-tag*s. In Section 4, we describe the tag-oriented and the tag-aware query processing. Physical implementation of core tag-oriented operators is in Section 5. Our experimental evaluation of STF is presented in Section 6. Section 7 reviews the related work, and Section 8 concludes the paper.

2. STREAM TAG FRAMEWORK OVERVIEW

Figure 2 illustrates a data stream environment with the STF (integrated inside DSMS) and the streaming *tick-tags* embedded inside data streams.

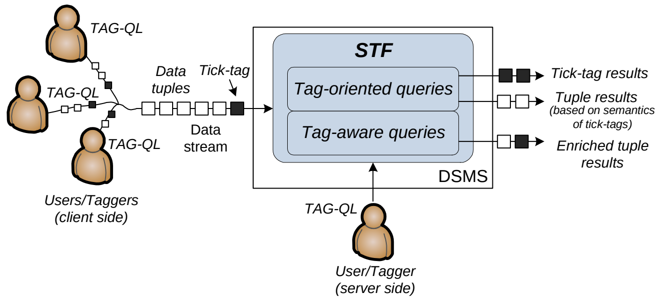


Figure 2: STF overview.

Data Model: We consider a centralized DSMS processing long-running queries on a set of data streams. A continuous data stream s is a potentially unbounded sequence of tuples that arrive over time. Tuples in the stream are of the form $tuple = [sid, tpid, A, ts]$, where sid is the stream identifier, $tpid$ is the tuple identifier, A is a set of attributes in a tuple, and ts is the timestamp of the tuple.

Users $u \in U$ can attach tags $t \in T$ to streaming objects $o \in O$ that can be of any granularity. A *taggable streaming object* o can be a (sub-)stream, a tuple, an attribute of a tuple, or a data value. An object is a piece of data to which additional information (via a tag) can be attached. An object o can have multiple tags at any given time and can be tagged in two ways: by a user providing the streaming data (on the client side) or by a user of the DSMS querying the streaming data (on the server side). Tagging itself can be performed in an ad-hoc manner, or it can also be continuously executed using a special type of continuous query called the *continuous tagging query* (see Section 3.5).

3. STREAMING TAGS (TICK-TAGS)

3.1 What is a Tick-Tag?

Tick-tags are meta-data tuples that attach additional information (a keyword, a label or a desciption) to streaming data objects, e.g., tuples, values or attributes. *Tick-tags* precede the streaming objects they are applicable to, and the tuples in the stream are completely unaware of embedded into stream *tick-tags*. In comparison to traditional keyword metadata, tags are not chosen from a controlled vocabulary defined by a single user, by an organization or by a third party [2]. Instead (as also commonly done on the web), users in their role as taggers can create tags of any content and attach them to streaming objects at any time. As a result, *tick-tags* contribute to a development of a real-time and continuously evolving *folksonomy* [6] – a rich way to characterize and means to discover interesting things about the real-time data based on exploiting the collective knowledge of possibly many users.

Tick-tags have several distinguishing characteristics compared to traditional (static) tags used for tagging web pages, images, files, or relational data. Table 1 gives a brief comparison of dynamic *tick-tags* against traditional static tags.

Property	***Traditional Tags***	***Streaming Tick-Tags***
Persistence	Permanent	Transient
Locality	(Most likely) stored separately from data	Interleaved with data
Access	Random access	Sequential access
Input Rate	Low	High
Size	Finite	Potentially infinite
Tag Processing	One-time	Continuous

Table 1: Traditional tags versus streaming tags.

As one can observe from Table 1, *tick-tags* inherit many characteristics of the dynamic streaming data they are applicable to. Namely, they are infinite, arrive online, stay in DSMS only for a limited time and eventually get discarded by the system.

3.2 Tick-Tag Physical Design

The physical schema of a *tick-tag* is shown in Figure 3[4].
Tagger Identifier (***TID***) depicts the source of the *tick-tag* – the id of a tagger is globally unique and is determined by the system.
Applicability describes the stream objects to which the tag is applicable to, e.g., a data value, an attribute or a collection of tuples. To keep the objects' description compact, *regular expressions* [23], similar to the approaches in [30, 34], are used in this field.
Content is a string datatype and stores the actual tag value. Given that STF supports an uncontrolled vocabulary, this could be anything: a keyword "`Accident`", a description "`Nice Weather`" or an emotional expression "`Happy`", "`Sad`".
Type is used by the framework to *classify* streaming tags. There are a number of taxonomies for tags in the literature, e.g., [26, 38]. Although not the primary focus of this paper, we have added this field in the *tick-tag* schema to support future applications, such as reality mining [13] and tag-based

[4]The fields *not* specified by users are shaded in grey.

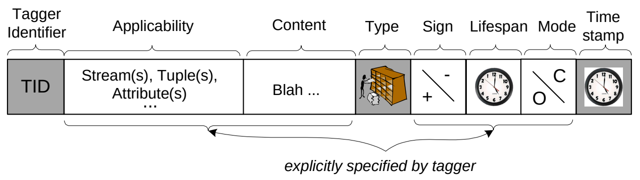


Figure 3: *Tick-tag* schema.

data classification [33]. Our current implementation considers the following five types of tags, while the other types and classification algorithms will be a part of our future work:

- *Objective*: Objective means a description, that does not depend on a particular user. For example, "`Bad Smell`" is not an objective tag (because one needs to know who thought it was bad), whereas "`3 Car Accident`" or "`Thunderstorm`" are objective tags.
- *Subjective*: Subjective tag implies a personal opinion. For example, "`Nice`", "`Awful`", "`Interesting`".
- *Physical*: This type of tag describes something physically, e.g., "`Broken Light`" or "`Icy Road`".
- *Acronym*: This type of tag is an acronym or a lingo that might mean various things. For example, "`ZZZ`" might mean going to sleep, "`GFC`" – going for coffee, and "`911`" – emergency or danger.
- *Junk*: The tag is meaningless or indecipherable. For example, "`J`" or "`FJKDSLAD`".

Sign. Since taggers use diversified vocabulary, often it may be difficult to generate an overall opinion or characterization based on the tags' contents [38]. Therefore, we have added a *sign* field to serve as a qualitative description of a *tick-tag*. *Sign* allows users' tags to rate and express opinions in a more *shareable vocabulary* than conventional tag content. Plus(+) or minus(-) values in the *sign* field easily characterize whether a tag has a positive(+) or a negative(-) context. By counting the numbers of positive and negative tags, a representation of the overall opinion (or a "reputation") or an assessment of the tagged information can be known. Tags without any value in the sign are considered as *neutral* and serve as regular content tags.

For example, consider an online auction system such as *eBay* [14]. This system monitors bids over items available for auction. An *Auction* stream contains items to sell and has a schema: *Auction*(*seller*, *product*, *product_features*, *start_price*, *time*). A collection of tags with different signs applicable to the objects in the stream *Auction* can give various interpretations as shown in Figure 4.

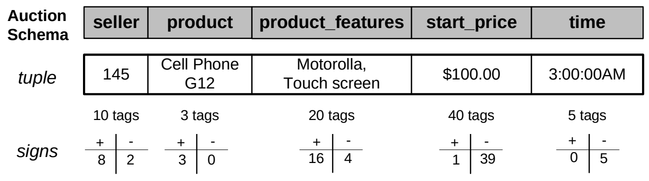


Figure 4: Interpretations based on tag signs.

The system could interpret the collection of positive and negative tags for objects here as:

- "*Most people like the seller.*"[5]
- "*Most people like the features of the product.*"
- "*Most people don't like the start price of the product.*"

[5]This could be based on the services or the quality of the products users might have purchased from the seller before.

The more tags there are, the more diverse interpretations can be made. A *sign* feature, thus, serves as a "bridge" for many diverse tags making them more shareable and enabling richer tag semantics.

Lifespan. The lifespan of a *tick-tag* is the time interval during which the tag is active. A user specifies for how long (in the near future) the tag should be applicable to a streaming object. After the tag's lifespan expires, the tag becomes inactive and the system garbage collects it. If only a single instance's applicability is wanted (i.e., one pass through the system), the keyword "`I`" (meaning "`Instant`") is specified in the field.

Mode. The tag mode indicates a user's preference regarding the combination of the tag with the earlier tags (those tags that are in the system and whose lifespans have not yet expired). "`O`" indicates "`Overwrite`", and "`C`" means "`Combine`"[6] respectively. Taggers can specify the mode with respect to *their* tags only, i.e., a user's tag cannot overwrite the tags generated by other users (taggers) applicable to the same streaming objects. We use *TID* field to track the sources of tags (i.e., the taggers) for this purpose. This field enables users to retract their earlier tags or to add more elaborate descriptions via multiple tags.

Timestamp. Timestamp describes the time when the tag was generated by a user, i.e., the tagger.

3.3 Tag Query Language

To enable attachment and querying of streaming tags, STF is equipped with a declarative language called *Tag Query Language* (or TAG-QL for short). The syntax for attaching a *tick-tag* to a streaming object is shown below[7]:

```
ATTACH TAG <tag_content>
TO <object_description>
(WHERE <condition_description>)
(WITH
   TAG_SIGN = < + | - >
   TAG_LIFESPAN = <lifespan_value>
   TAG_MODE = <mode_value>)
```

The <`object_description`> specifies the applicability of the tag, namely the object(s) to which the tag is being attached to. The `WHERE` <`condition_description`> clause is used to describe the conditions that the tagged data must satisfy. Implicitly, the `WHERE` clause also conveys the "location" in the stream, where the *tick-tag* will be inserted. Since *tick-tags* always come before the data they are applicable to, the `WHERE` clause states which data the tag should precede in the stream. The <`condition_description`> can be a simple condition or a nested sub-query. Other TAG-QL statements are depicted in Table 2. We will describe them in detail in the rest of the paper and give the corresponding query examples.

Syntax	*Meaning*
`ATTACH TAG ...`	Attaches a tag to a streaming object
`SELECT TAGS ...`	Selects tags satisfying a search predicate
`SELECT TAGGED OBJECTS...`	Selects tagged objects
`SELECT ... WITH TAGS`	Returns tag-enriched query results

Table 2: Overview of key TAG-QL statements.

3.4 Tick-Tag Examples

Here, we present several *tick-tag* examples to illustrate the syntax and the semantics of *tick-tags*. Consider a data stream *Patients*(*sid*, *tpid*, *measure*, *loc*, *time*) transmitting real-time health measurements and locations of patients.

[6]Different semantics can be used here to combine *tick-tags*.

[7]The `WHERE...` and the `WITH...` clauses are optional here.

The following *tick-tags* may be generated[8]:

t_1: `■|*,*,{loc.value}|Panic Attack|■|-|1 min|O|■`

represents a tag attached to the current *location* **value** of the user, and indicates that the user is having a panic attack at her current location. The user feels negative about this experience (- sign), which may also explain the changes in the health measurements (e.g., increase in the heart rate of the patient). The lifespan of the tag is 1 min, and it overwrites any other tags associated with this location value previously sent by the user. This is an example of a tag attached to a specific data value[9].

t_2: `■|*,*,{measure}|Running|■|+|30 min|O|■`

is a tag attached to the heart rate measure **attribute** in the stream, and indicates that the user is currently running, which is something the user likes to do (as described by the negative '+' sign). The lifespan of the tag is 30 min (possibly indicating how long the user intends to exercise), and it overwrites any other tags associated with the heart rate measure attribute specified by the user. This is an example of a tag attached to a tuple attribute. Using TAG-QL, the above tags are expressed as follows:

```
t1 ATTACH TAG 'Panic attack'        | t2 ATTACH TAG 'Running'
   TO Patients.loc.value            |    TO Patients.measure
   WITH                             |    WITH
       TAG_SIGN = '-' AND           |        TAG_SIGN = '+' AND
       TAG_LIFESPAN = 1 min AND     |        TAG_LIFESPAN = 30 min AND
       TAG_MODE = OVERWRITE         |        TAG_MODE = OVERWRITE
```

The absence of the `WHERE...` clause in the TAG-QL statements above indicates that there are no constraints regarding which data values the *tick-tags* must precede. Thus, the *tick-tag*s will be inserted into the stream interleaved with whatever the tuples happen to be transmitted at the time.

3.5 Tick-Tag Generation

Users can create *tick-tags manually* (in an ad-hoc manner) as described above. Alternatively, users can perform *continuous tagging* by instantiating a special type of query, called the *Continuous Tagging Query*. A novel operator, called the *Tagger* operator (described in detail in Section 4.1) always exists in such a query. This operator consumes an input data stream, continuously evaluates the tagging condition, and produces the corresponding *tick-tag*s that get inserted into the output stream and represent the tags being attached to the following after them data. We describe the physical implementation of the *Tagger* operator in Section 5. The processing of a tagging query is almost identical to an ordinary continuous query, except that the data tuples in the output stream are now interleaved with generated *tick-tags*. An example of a continuous tagging query expressed in TAG-QL is shown below:

```
ATTACH TAG 'Dangerous'
CONTINUOUSLY
TO Patients.pid.value
WHERE (SELECT pid
       FROM Patients
       WHERE measure > 80)
WITH
  TAG_SIGN = '-'
```

The keyword `CONTINUOUSLY` in the TAG-QL statement above indicates that the tagging is continuous, that is, a tag will be

[8]We only illustrate the fields specified by users. The system fields (that are not exposed to users) are denoted by ■.

[9]To attach a tag to an attribute value "<`attribute_name`>`.value`" syntax is used, where <`attribute_name`> is replaced with an actual attribute name.

attached to every patient id (*pid*) value with the heart rate above the specified threshold. Specifically, a *tick-tag* (with the value "`Dangerous`") will be created and inserted into the stream ahead of every tuple with heart rate *measure* > 80.

4. TAG-BASED QUERY PROCESSING

We distinguish between two types of tag-based query processing in STF, namely the *tag-oriented* query processing and the *tag-aware* query processing.

4.1 Tag-Oriented Query Processing

Expressing Tag-Oriented Queries in TAG-QL

In tag-oriented query processing, users or applications query *tick-tags explicitly.* Explict tag querying is useful for the following two purposes: (1) to locate tags where the tag values themselves are of interest, e.g., *Show me all tags which have a sign = '-'*; and (2) to locate tags where the associated base data values are of interest, e.g., *Show me all data tuples that are tagged with tags that have a value = 'dangerous'.* Such explicit querying gives the ability to see what other streaming objects have been tagged with the same keyword or a sign, as well as browse through the tags related to the same streaming objects. For specifying such queries, TAG-QL provides "`SELECT TAGS`" and "`SELECT TAGGED OBJECTS`" statements. Queries Q_1 and Q_2 shown below are examples of such tag-oriented queries.

Syntax	*Meaning*

Q_1: `SELECT TAGS`
`FROM Patients`
`WHERE OBJECT =`
`Patients.measure AND`
`TAG_SIGN = '-'`
	Finds all negative tags attached to the heart rate measure attribute in the *Patients* stream

Q_2: `SELECT TAGGED OBJECTS`
`FROM Patients`
`WHERE TAG = 'Emergency'`
	Returns tuples that contain objects tagged with word '`Emergency`' in the stream *Patients*

Table 3: Examples of tag-oriented queries.

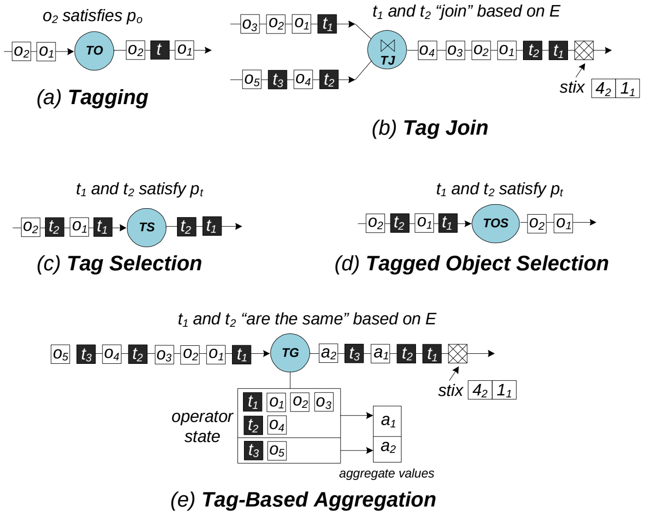
Tag-Oriented Query Algebra

Here, we describe several tag-oriented operators introduced into the continuous query algebra. Let t denote a tag, o – a streaming data object, T – a stream of tags, O – a stream of data objects, and p – a search predicate, which can be either on data objects (denoted as p_o) or tags (denoted as p_t). The following tag-oriented operations are defined in STF[10]:

Tagger Operator $[TO(O,\ p_o,\ t) \rightarrow \overbrace{O}^{T'}$, where $\forall\ t_i \in T'$, $t_i = t]$. Tagger operator is a unary operator that processes tuples on-the-fly, by attaching a tag t to an object $o \in O$, if o satisfies the condition p_o. As a result, the tagger operator inserts a tag t into the output stream preceding the object o. Figure 5(a) shows an example where the object o_2 gets tagged with the tag t.

Tag Selection $[TS(\overbrace{O}^{T},\ p_t) \rightarrow T']$. Tag selection is a unary operator that returns tags T' ($T' \subseteq T$) (without their respective objects) that satisfy the tag search predicate p_t. Figure 5(c) illustrates an example, where tags t_1 and t_2 get returned as results by the tag selection operator based on the search predicate p_t.

[10]We denote an object o tagged with a tag t as $\overbrace{o}^{t}$, and a stream with embedded inside it *tick-tags* as $\overbrace{O}^{T}$.



o_2 satisfies p_o
TO
(a) Tagging
t_1 and t_2 "join" based on E
TJ
stix
(b) Tag Join
t_1 and t_2 satisfy p_t
TS
(c) Tag Selection
t_1 and t_2 satisfy p_t
TOS
(d) Tagged Object Selection
t_1 and t_2 "are the same" based on E
TG
stix
operator state
aggregate values
(e) Tag-Based Aggregation

Figure 5: Tag-oriented algebra (examples).

Tagged Object Selection $[TOS(\overbrace{O}^{T},\ p_t) \rightarrow O']$. Tagged object selection is similar to tag selection operator (TS), except that instead of tags it returns the tagged objects $O' \subseteq O$, whose tags satisfy the selection predicate p_t (for example, objects o_1 and o_2 in Figure 5(d) based on their respective tags t_1 and t_2). A variant of this operator returns the tagged objects together with their respective tags.

Tag Join $[TJ(\overbrace{O_1}^{T_1}, \overbrace{O_2}^{T_2}, E) \rightarrow \overbrace{O'}^{T'}$, where $T' = E(T_1, T_2) \neq \emptyset]$. Tag join is a binary operator that joins two streams of objects, interleaved with *tick-tags*, based on the *tag join condition* E. The join condition E can be a tag equivalence, or a tag similarity, or some other tag join criteria. For example, a *tag equivalence* function $TE(t_1,t_2)$ checks for the content equivalence of two tags, which can be word-based "`Accident`" = "`Accident`", or semantics-based (e.g., when words mean the same thing) using the system's dictionary[11], e.g., "`Accident`" = "`Disaster`", or by co-occurrence, if both tags contain the same objects. One of the simplest co-occurrence methods is using *absolute co-occurrence*, that is counting the number of times two tags are assigned to the same object. The similarity can also be estimated based on *relative co-occurrence*, also called "tag overlapping", and can be measured by *Jaccard coefficient* [41]. If A and B are the collections of stream objects described by two tags, relative co-occurrence is then defined as: $JC(A,B) = \frac{|A \cap B|}{|A \cup B|}$. That is, relative co-occurrence is equal to the division between the number of objects in which tags co-occur, and the number of objects in which appear any one of two tags. For example, by this definition, the tag "`Fireworks`" could be equivalent to the tag "`Cool`", if users had tagged all objects annotated with the term "`Fireworks`" with the tag "`Cool`."

The result of the tag join operator is a single stream of tagged objects whose tags join based on the join function E. Figure 5(b) shows an example of a tag join operator output. Here, tags t_1 and t_2 from the two input streams join based on function E, and are sent to the output stream followed by their respective tagged objects o_1-o_4. Note, that although the tags join, physically they are *not* combined into a single

[11]STF maintains the internal dictionary to support tag classification, tag equivalence and tag similarity function.

tick-tag tuple. The reason for such design is the need to preserve the correct base tags' semantics. Since the tag join is based on only the *tick-tag*'s content field, we maintain the original *tick-tags* with their values (for attributes like lifespan, sign, mode, etc.,) to ensure that the user-specified tag semantics is enforced correctly even after the tags join. After tag join, an additional streaming element is inserted into the stream – the so-called "*streaming tag index*" tuple (or short "*stix*"). The purpose of the *stix* is to store the references of the joined tags (which are placed consecutively together in the output stream) to their respective base data tuples. In the tag join example in Figure 5(b), the *stix* contains the following information: $\boxed{4_2|1_1}$[12]. The subscripts ($_1$ and $_2$) are the indicies to the subsequent after the *stix* *tick-tags*, i.e., "$_1$" refers to the first *tick-tag* and "$_2$" to the second *tick-tag*, respectively. The values in each index refer to the offsets of the data tuples to which that *tick-tag* applies to. Thus, the above *stix* means the following: the first *tick-tag* applies to the tuples 1-3 (o_1-o_3) and the second *tick-tag* applies to the tuple 4 (i.e., o_4).

The main idea of the tag join operator is to combine two streams of tagged data based on the similarity of their embedded tags'. This operation can be useful for applications searching for related (based on the tags) streaming data that may arrive from various sources.

Tag-Based Aggregation $[TG(\overbrace{O}^{T}, E, G_T^{agg}) \rightarrow \underset{G_T^{agg}}{\overbrace{O'}^{T}}]$. This operator groups objects in a stream by their tags (the groups are based on the tag function E) and incrementally updates the value of a given aggregate for each tag-based group (see Figure 5(e)). For every arrived tuple, the operator first adds it to the state buffer, and determines which group it belongs to (based on its tag's content and the tag similarity function E), and then returns an updated result for this group (preceded by the subgroup's corresponding tags), which is understood to replace a previously reported answer for this group. It may happen that a data tuple may belong to several groups based on the attached tag to it. In this case, the operator picks the "closest" (again based on the function E) tag group for the streaming object and updates the answer for that group. Objects without any attached tags can be either completely ignored by the operator or can be placed into a separate "non-tagged" objects group, for which the result is maintained similar to the tag-based groups. In the second case, a dummy *tick-tag* (with an empty content) is inserted prior to sending the answer to preserve correct semantics – to ensure that the earlier outputted tags are not applied to this aggregate answer.

4.2 Tag-Aware Query Processing

Expressing Tag-Aware Queries in TAG-QL

In addition to explicit querying of tags, users and applications may find it useful to receive continuous query results that are "enriched" with the tags attached to the original data, based on which the query results were produced. We call this functionality "*implicit tag querying*" and achieve it by performing tag-aware query processing. To indicate that enriched results should be outputted for a given query, a user simply adds a "`WITH TAGS`" statement when specifying a continuous query to the system as depicted in Table 4.

[12]The *stix* illustration should be read from right to left.

	Syntax	***Meaning***

Q_3:	`SELECT pid, loc, time`
	`FROM Patients`
	`WHERE measure > 80`
	`WITH TAGS`
		Select-project query that will produce results with interleaved tags of the base data.

Q_4:	`SELECT A.pid, B.pid`
	`FROM Patients A [1 min], Animals B [3 min]`
	`WHERE Dist(A.loc, B.loc) < 0.2`
	`WITH TAGS`
		Join query that will produce results with interleaved tags of the base data.

Table 4: Examples of tag-aware queries.

One of the immediate challenges in tag-aware query processing is the support for correct propagation of streaming tags through the continuous query pipeline, as data objects are being filtered, joined, or projected out. If one thinks of a tag as a form of mark-up on a streaming object, the key question here is how should that mark-up be transferred into the results of a query. We enable this functionality by adding tag-awareness to continuous query operators.

Tag-Aware Query Algebra

Projection is an unary operator that processes tuples by discarding unwanted attributes. This operator simply propagates *tick-tags* and thereafter the projected tuples. If a *tick-tag* applies only to the projected attributes, it is discarded by the project operator as well.

Selection is a unary operator that drops tuples that do not satisfy the selection condition. A select operator delays a *tick-tag* propagation until at least one of the tagged tuples that follow it satisfies the selection predicate. If all tagged tuples are filtered, their corresponding *tick-tag* is discarded then as well.

Join is a binary operator that joins the tuples of its input streams. If a tuple joins with another tuple, before being sent to the output stream the tags of the base tuples are physically arranged in a similar fashion as in the tag-oriented join (discussed in Section 4.1), with the following two main differences:

1. Since the data tuples (after the join) are physically combined into a single physical tuple, the *tick-tags* attached to the base data tuples will now refer to this (new) joined tuple. This reference is stored in the *stix* metadata tuple (described in Section 4.1) that gets inserted prior to the *tick-tags*.
2. In addition to maintaining the tuple-level granularity reference in *stix*, we now also store the references to the joined tuple attributes (that correspond to the base tuples' attributes) that the *tick-tags* apply to.

Figure 6 illustrates an example of a tag-aware join output.

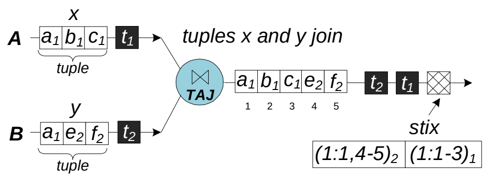


Figure 6: Tag-aware join example.

Here tuples x and y from streams A and B join based on the equality of the first attribute value a_1. Their respective base tuples' tags t_1 and t_2 precede the join tuple, and the *stix* stores the following information $\boxed{(1{:}1,4\text{-}5)_2|(1{:}1\text{-}3)_1}$, where "$(1{:}1\text{-}3)_1$" means the first (after the *stix*) *tick-tag* t_1 applies to the first tuple and to the attributes 1-3 in the join tuple. Similarly, "$(1{:}1,4\text{-}5)_2$" means that the second *tick-tag*

t_2 applies to the first tuple and to the attributes 1, 4 and 5 in the join tuple, respectively.

Aggregation. In a tag-aware aggregation operator, each attribute domain is partitioned into attribute sub-groups, where each sub-group contains tuples with the same attribute value. A result is calculated for each sub-group and then sent to the output stream preceded by the collection of tags that have arrived and have not yet expired from the window and are applicable to *any* object in that sub-group. The motivation for such comprehensive tag propagation is to make all tags associated with the base data (used to compute the outputted aggregate value) available with the aggregate query result.

5. PHYSICAL IMPLEMENTATION

Here, we describe the physical implementation of two key operators in the tag-oriented algebra, namely the tagger operator and the tag join operator. Due to space limitations, we reserve the description of implementation details for other operators to our technical report.

Tagger Operator: This operator is designed to continuously attach tags to streaming objects that satisfy tagging predicate p_o. Conceptually, the tagger operator is similar to a selection operator, except that it doesn't discard the tuples that don't satisfy the predicate p_o, but instead forwards them to the output stream without inserting a *tick-tag* ahead of them. Figure 7 shows the pseudocode for the tagger operator execution.

```
TaggerOperator ($p_o$ tagging predicate, $c$ tag content,
$s$ tag sign, $l$ tag lifetime, $m$ tag mode)
01 for (every new element $e$ received from input stream)
02    if ($e$ is a tuple) // input is a tuple
03       if ($e$ satisfies $p_o$)
04          $ts$ = geTime(now)
05          $tid$ = getCurrentQueryId()
06          $t$ = CreateNewTickTag($tid$,$P$,$c$,$s$,$l$,$m$,$ts$)
07          send $t$ to output
08          send $e$ to output
09       else send $e$ to output
10    else // input is a tick-tag
11       send $e$ to output // propagate tick-tag
```

Figure 7: Tagger operator algorithm.

For every arrived data tuple, the tagger operator evaluates the tagging predicate to determine whether the streaming object should be tagged (Line 3). If yes, then a new *tick-tag* is created with the tag properties specified as parameters to the operator (Line 6). The operator assigns the current *query id* as the tagger identifier (*tid*) in the *tick-tag* (Line 5), and the time the *tick-tag* was created is stored in its timestamp field. The newly created *tick-tag* is then forwarded to the output stream followed by the data tuple (Lines 7-8). If the tagger operator receives a *tick-tag* as its input, it simply propagates it to the output stream (Line 11). One of the optimizations that can be employed by the tagger operator (the pseudocode is not shown), is to cache the last outputted *tick-tag*. If the next tuple satisfies the same tagging condition, and the regular expression in the applicability field of the already outputted *tick-tag* is suitable for the new tuple, then no additional *tick-tag* needs to be created and sent to the output stream. The tuple will simply be forwarded to the output stream. The understanding here is that several tuples share the same *tick-tag* and follow it consecutively in the output data stream.

Tag Join: The *TagJoin* algorithm is shown in Figure 8. We present the *TagJoin* as a sliding window E-based join algorithm, where E is a tag join function (which can be a tag similarity, a tag equivalence function or any other tag join criteria as described in Section 4.1). In our pseudocode we describe the nested-loop version of the *TagJoin*. The optimized version of the operator employing an index on tuples and *tick-tags* in the window is a part of our future work. The *TagJoin* maintains a time-based sliding window. We employ a list structure to link all tuples and their *tick-tags* in a chronological order (most recent at the tail). *Tick-tags* are interleaved with tuples in the window, and, thus, the tuple list is "partitioned" by the *tick-tags* into segments, where the tuples in each segment may be tagged by the preceding them *tick-tags*. A collection of tuples between any two non-adjacent collections of *tick-tags* is called a *tagged segment*. We discuss the processing of tuples and *tick-tags* from the input stream A. The processing for input stream B is similar due to the symmetric execution logic.

```
TagJoin ($A$ stream, $B$ stream)
01 $W_A$ ← join time window for stream $A$
02 $W_B$ ← join time window for stream $B$
03 if (a new element $e_A$ is received from stream $A$)
04    if ($e_A$ is a tick-tag) // input is a tick-tag
05       TagCollection($e_A$, $A$, $W_A$)
06    else if ($e_A$ is a tuple) // input is a tuple
07       Invalidate($e_A$, $B$, $W_B$)
         // retrieve tags that have arrived prior to tuple $e_A$
08       $T_A$ ← GetTags($e_A$)
09       Probe($T_A$, $A$, $W_A$, $B$, $W_B$, $E$)
10 if (a new element $e_B$ is received from stream $B$)
   // Similar to above
```

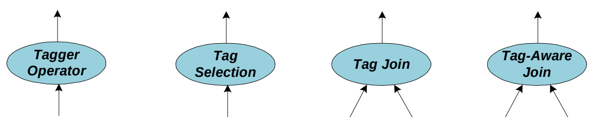
```
Probe ($T_A$ - set of tick-tags from the current stream,
$A$ - current stream, $W_A$ - current stream window,
$B$ - opposite stream, $W_B$ - opposite stream window,
$E$ - join condition)
11 $T_B$ ← GetTags($B$[$W_B$])
12 for (every tick-tag $t_A \in T_A$)
13    for (every tick-tag $t_B \in T_B$)
14       if (Join($t_A$,$t_B$,$E$) // tags join based on $E$
15          $S_A$ ← $A$[$W_A$,$t_A$] // objects tagged by $t_A$
16          $S_B$ ← $B$[$W_B$,$t_B$] // objects tagged by $t_B$
17          $stix$ ← CreateNewStix()
18          send $stix$ to output
19          send $t_A$, $t_B$, $S_A$, $S_B$ tuples to output
```

Figure 8: Tag join operator algorithm.

Tag Collection. As *tick-tags* arrive, they are stored in the sliding window. They represent the labels (annotations) for the upcoming data tuples (Lines 3-4).

Invalidation. When a new data tuple e_A is retrieved from the input stream A, it is used to invalidate the expired tuples from the head of the window of the stream B (Line 7). If all tuples from a tagged segment have been invalidated, their corresponding *tick-tags* are purged from the head of the window as well.

Probe. After the invalidation is done, the *tick-tag*(s) preceding the tuple e_A are used to probe the window of the stream B. For concreteness of discussion, let's consider there is a single tag t_A in the window of stream A that precedes tuple e_A and represents the tag attached to e_A. If t_A joins with *tick-tag* t_B from stream B based on the tag join function E, the *tick-tags* are placed consecutively together followed by their corresponding base tuples (see Section 4.1 for detailed



(a) Query 1 (b) Query 2 (c) Query 3 (d) Query 4

Figure 9: Experimental Queries.

explanation and an example of this step). The *stix* metadata tuple is then created (Line 17) to store the reference to the base tuples and is inserted into the output stream ahead of the "joined" *tick-tags*. The *stix*, the *tick-tags*, and their respective tuples are then forwarded to the output stream (Lines 18-19). If the tag join is empty (i.e., the tags do not join based on E), then neither the *tick-tags* nor their respective tuples are forwarded to the output stream.

6. EXPERIMENTAL ANALYSIS

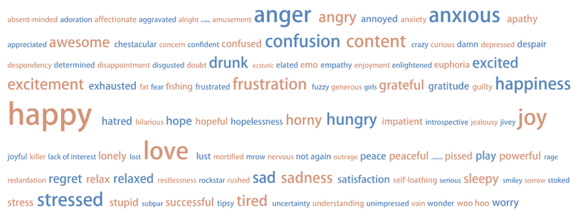
6.1 Experimental Setup

We have implemented our proposed *Stream Tag Framework* in a DSMS prototype called *CAPE* [15]. We execute *CAPE* on Intel Pentium IV CPU 2.4GHz with 2GB RAM running Windows Vista and 1.6.0.0 Java SDK. For data, we use the *Network-based Moving Objects Generator* [4] to generate a moving objects dataset on which we evaluate our experimental queries. The input to the generator is the road map of Worcester county, MA, USA. The output of the generator is a set of objects that move on the given road network and continuously send their location updates. We generate 100K of moving objects, which represent cars, cyclists, and pedestrians. Each tuple in the stream consists of the following fields: update type, object id, report number, object type, timestamp, current location, speed, and the location of the next destination node (see [4] for more details on the generator's output). We break the moving objects stream up into several streams based on the ids of objects. Such setup simulates objects sending updates to different service providers or base stations and allows us to test join queries. Tuples' arrival distribution is modeled using a Poisson distribution with a mean tuple inter-arrival rate equal to 10 milliseconds.

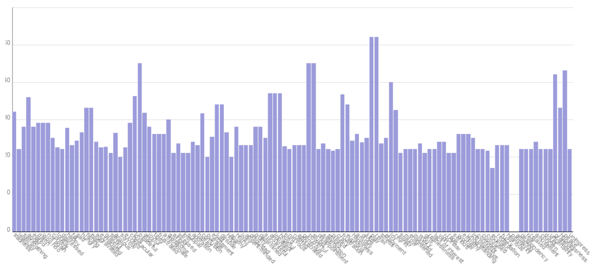
Unless mentioned otherwise, the tagging is done at the tuple granularity and the *tick-tags* arrive to the DSMS already interleaved with the streaming data. We chose the tuple level, because it is likely to be the most common granularity of tagging in mobile environments. For comparison, we have implemented alternative tagging solutions described in Section 1.3, namely the table approach, the extended data tuple approach, and the streaming XML approach. In this section, we refer to them as *TABLE*, *TUPLE*, and *XML* respectively. Our technique is abbreviated as *TICK-TAG*.

For tag content, we employ the "emotion tags" dataset from the *ManyEyes* application [28] supported by IBM. Figure 10(a) shows the "tag cloud" for the emotion tags used in our experiments (with more frequent tags depicted in larger fonts). Figure 10(b) illustrates the overall tag distribution, and Table 5 lists some of the examples of tag values.

Four types of queries are used in our experiments which are depicted in Figure 9. Query 1 attaches tags (with values chosen at random) to streaming data tuples, with the tagging predicate being on the current location of a moving object. We use Query 1 to test the performance of our proposed tagger operator. Query 2 selects the tags satisfying the tag search predicate (the predicate is based on two types of emo-



(a) Tag cloud (emotion tags)



(b) Tag distribution (emotion tags)

Figure 10: Tag properties.

tions: "sad" and "angry" in a specific geographic area[13]) on the incoming stream with already interleaved tags. It is used to test the tag selection operation. Query 3 joins two streams of tagged tuples based on the tag equality function E, which is defined as semantics-based equivalence. Specifically, we have partitioned the emotion tags from the dataset [28] into 5 separate groups based on the type of the emotion, e.g., "happy", "sad", "neutral", "angry", etc., and in the tag-based join, we perform the join based on whether the tags belong to the same emotion group. For example, if tags "`Joyful`" and "`Excited`" belong to the same group "happy emotions", then the two tags join, if they happen to arrive at the same time and are in the windows of the streams being joined. This type of query may be useful to find people who experience similar emotions at the same time, and can possibly help correlate it to their location or a nearby event. Finally, Query 4 performs a tag-aware join on two incoming streams of location updates based on the mutual proximity of the moving objects, e.g., two objects join, if they are within 0.1 miles from each other. Query 4 is used to test the cost of tag-awareness in the continuous join operator.

The real-life application (based on the data, tag values and queries described above) that we consider in our experimental setting is a *geo-social networking application*, e.g., [3, 19]. Here, users may want to tag their location updates with their emotions to update their friends on their well-being, or possibly look for someone to meet and socialize with in a given geographic area.

6.2 Experimental Results

Cost of Tagger Operator

Figure 11 compares the cost of our proposed *Tagger* operator to the cost of a regular *Selection* operator. In the case of selection operator, we process a regular data stream (without any *tick-tags* interleaved). We use Query 1 (from Figure 9) in this experiment. The selection predicate here is equivalent to the tagging predicate. The percentage of tagged objects

[13] A query of the form: "*Continuously monitor all emotional tags (attached by people to their real-time information) that fall into 'angry' and 'sad' categories in downtown*" may be executed by the law enforcement authorities to prevent potential violence or accidents in the city.

Most frequent emotions tags – *ManyEyes* dataset [28]	happy, sad, jealous, self-loathing, angry, elated, content, lonely, depressed, frustrated, aggravated, exhausted, grateful, sleepy, anxious, sorry, excited, anxious, loved, peaceful, joyful, tipsy, affectionate, cool, alright, stressed, lost, confused, outraged, despaired, hopeful, sympathetic, relaxed, unimpressed, ...

Table 5: Emotion tags examples used in the experiments.

is varied from 0% – none of the tuples are tagged to 100% – meaning all tuples are tagged (the same selectivity is for the selection operator). We use the selection operator here as the cost baseline to which we compare the tagger's cost while varying the tagging frequency. The difference between the selection and the tagger is as follows: (1) the selection operator discards the tuples that don't satisfy its predicate, whereas the tagger operator simply propagates them to the output stream (without tagging); (2) if the predicate is satisfied, the tagger inserts a *tick-tag* prior to the data tuple being tagged, whereas the selection simply propagates this tuple to the output stream. The cost for the selection operator increases, as the selectivity of the operator increases, largely due to more work being done by the operator when more tuples have to be propagated up-stream. Similarly, for tagger operator, as the percentage of objects being tagged increases, the tagger operator's cost increases. This is due to a larger number of *tick-tags* being generated (in the case of 100% tagging - twice as many streaming elements are enqueued to the output stream. The cost of the tagger is larger than of the selection (which is expected), on average, by 1.08x for 0% tagging[14] (when no *tick-tags* are inserted) by 1.8x for 100% tagging (where for every data tuple a *tick-tag* is inserted).

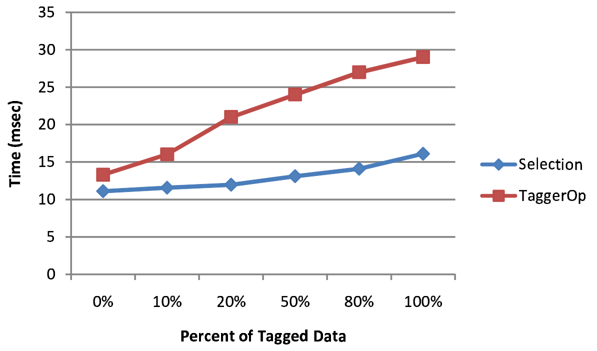


Figure 11: Cost of tagging operator.

Comparison of Tick-Tag Approach Against Alternatives

The goal in this section is to compare the *TICK-TAG* approach against the alternative tagging solutions (described in Section 1.3), namely, the *TABLE*, the *TUPLE*, and the streaming *XML* methods. All these solutions were implemented in *CAPE* and re-use as much of the same code as possible for fair comparison. We use Query 2 (Figure 9) in this experiment, and present the average output rate and memory utilization results when performing *tag selection* using these methods. The query is a square region inside which we continuously monitor moving objects' tags. For the focal point of the tag selection query, we choose a random location on the road network (the same for all four cases) and consider it as the center of the query. The focal point the query is static, hence both the *tick-tags* as well as the moving objects that appear in the query region are the same for all four cases. The space is represented as the unit square, the query size is a square region of side length 2.

[14]There are minor execution overheads in the tagger operator that are not present in the selection – e.g., propagation of (un-tagged) data elements upstream.

For *TABLE* tagging approach, we have a separate stream transmitting *tick-tags* that continuously arrive to the system. For every arrived *tick-tag*, we process it by inserting it into the global tag table and initiating an evaluation on the streaming data tuples to determine if the newly arrived tag is applicable to them. For the *TUPLE* approach, we have added an additional attribute in the stream's schema to store the tag value (in addition to all other tag parameters, e.g., mode, lifetime, to be fair in comparison with other approaches). For *XML* approach, we have embedded "xml tags" inside streaming tuples that are interleaved with regular data tuples, and process them when they arrive to the system[15]. Figures 12 and 13 compare the alternative approaches when varying the percentage of moving objects tagged from 0% to 100% in terms of average output rate and memory usage, respectively.

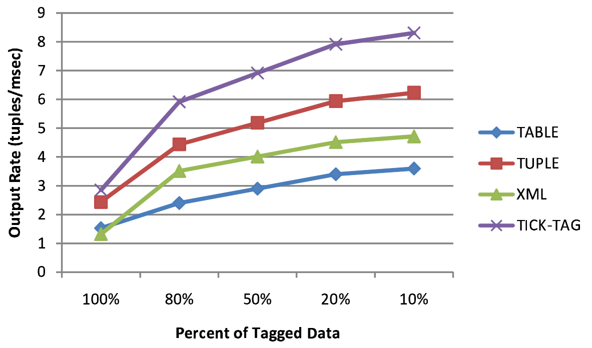


Figure 12: Comparison of alternatives (output rate)

Figure 12 shows the average output rate results for the four different alternatives. In the figure, we measure the average number of result tuples produced per time unit. In Figure 13 we measure the memory usage by these different solutions over time. We can see from both Figures 12 and 13 that the *TICK-TAG* approach results in higher output rate and smaller memory usage compared to the alternatives. The relative performance of *TICK-TAG* over the other tagging approaches increases with the increase of the number of streaming objects that can share their tags. The main reason is that the search cost of *TICK-TAG* is much lower than updating in search costs (to find the tagged objects) in the *TABLE* approach. Although *XML* and *TUPLE* approaches also take a "stream-centric" approach for tag implementation, they do not exploit the commonalities between the different tuples, and thus result in more memory being used and higher processing cost.

Cost of Tag Join Operator

In this section we evaluate the cost of the tag join operator again with varying percentage of tagged data in both streams (from 0% to 100%). We use Query 3 in this experiment (Figure 9). Sliding windows are time-based and state buffers are implemented as linked lists. The tag join condition is described in Section 6.1. Figure 14 shows the average cost (computed after several runs) of the *TagJoin*. As can be expected, the cost of tag join increases as the percentage

[15]In the implementation, one XML tag is represented by two physical tuples – one storing the start xml tag and the other – the end tag.

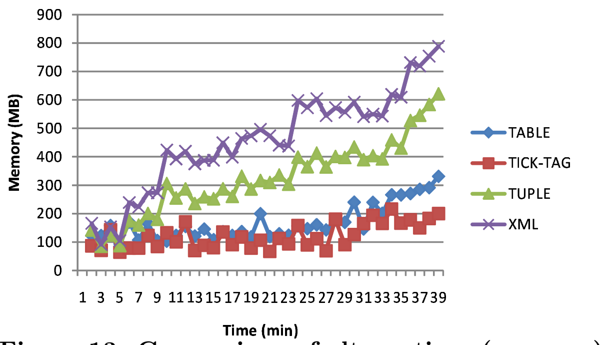


Figure 13: Comparison of alternatives (memory)

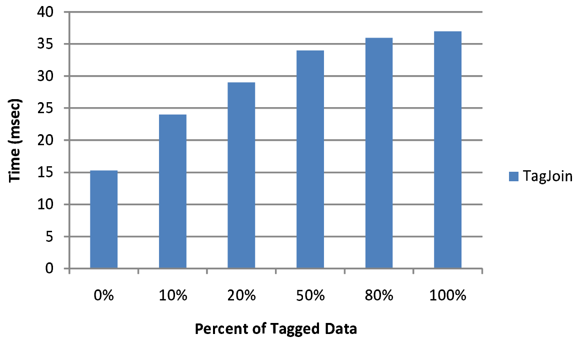


Figure 14: Cost of tag join.

of tagged objects increases by about 65% when 100% of objects are tagged. The more tags are embedded inside data streams, the more overhead is incurred by the tag join. Also, in order to preserve the correct base tag semantics, e.g., tag lifetime, the operator continuously creates *stix* tuples that are used to maintain the references to the original data after the *tick-tags* and their respective data tuples get physically re-arranged in the stream as a result of the tag join. The more tags are interleaved in the streams, the more the join function E has to be invoked, the more *stix* elements will be created and the more elements will need to be enqueued into the output stream.

Cost of Tag-Aware Join Operator

In this section we compare the cost of the *Tag-Aware Join* operator (described in Section 4.2) to a regular join operator using Query 4 from Figure 9. The goal of this experiment is to measure the overhead of tag-awareness in a join operator. Figure 15 shows the cost of the tag-aware join with respect to the regular join, when varying the percentage of tagged objects. We see that, the larger the number of tagged objects, the higher the cost of the tag-awareness. For 0%, the tag-aware join cost is nearly idenitical to the regular join operator cost, since there are no *tick-tags* in the streams, and the operator executes just like a regular join. Whereas for 100% of tagged objects, the tag-awareness incurs a "penalty" of processing a larger quantity of streaming *tick-tags*, incurring about 43% of additional cost for 100% of tagging. However, we believe, the case when 100% of data is tagged is highly unlikely in real-life applications. And the average overhead case, between 14%-33% for 20%-80% of tagged data seems to be a reasonable overhead for the added tag-awareness functionality and correct tag propagation through the query pipeline.

Summary of Experimental Conclusions

1. The *tick-tag* approach is scalable in the number of tagged objects.
2. The *tick-tag* approach outperforms alternative tagging

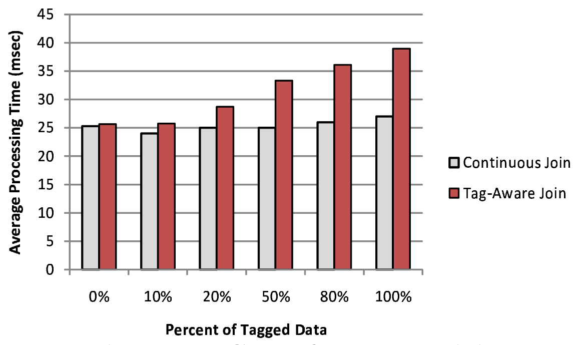


Figure 15: Cost of tag-aware join.

methods in terms of output rate and processing time.
3. The *tick-tag* approach consumes less memory, especially if many objects share the same *tick-tag*.
4. The average overhead of tag-awareness in a continuous join operator is small.

7. RELATED WORK

The works related to our paper span a number of research areas. Here, we highlight a few.

Tagging Systems. There are several ongoing projects that deal with annotation propagation and management for scientific databases, e.g., DBNotes [9], Mondrian [16], bdbms [27], and MMS [10]. Social bookmarking systems, such as *Flickr* [18], *Delicious* [11] and *Technorati* [40] support annotations of web resources and images with free-text keywords. For more examples of tagging systems and their taxonomy, we refer the reader to [6]. To the best of our knowledge, none of these existing works address the problem of tagging in the context of dynamic data stream environments.

Chi et al. [12] looked at the entropy of tagging systems, in an effort to understand how tags grow, and how the groupings of tags change over time and affect browsing. Halpin et al.'s work [20] looks at the nature of tag distributions with information theoretic tools. There has been some work on association rules in tagging systems, including [7] and [32]. [32] primarily focuses on prediction of tags. Oldenburg et al. [39] look at how to integrate tags across tagging systems by using Jaccard measure and discuss different types of tagging systems: social bookmarking, research paper tagging systems, but not Data Stream Management Systems.

Streaming XML. Research on self-describing streaming XML which can be viewed as "data tags" has received a lot of attention in recent years [31, 42]. However, as we have pointed out in Section 1.3, XML processing is typically more expensive compared to traditional stream data processing, and requires a special XML stream management functionality (in addition to the XML-aware optimizer and executor). Our proposed tagging approach is simpler in design and is more light-weight compared to streaming XML, while it still provides support for rich user-based tag semantics.

Annotations in Relational Databases. Relational databases have had an extraordinarily successful history of commercial success and fertile research. It is not surprising, therefore, that database researchers have attempted to understand annotations and "tagging" in the context of relational databases [9].

One of the biggest challenges in relational databases is the correct propagation of annotations through queries' pipelines. This is similar to the problem we've discussed in the context of tag-aware query processing in Section 4.2. In [9], a practical approach is taken to handling annotation in which an extension of SQL is developed, which allows for explicit user control over the propagation of annotations. The idea in [9] is to allow the user to control the flow of annotations by adding propagation instructions to the SQL query language. In our case, the STF performs (by default) the system-driven propagation, when processing tag-aware continuous queries. Adding support for user preferences regarding tag propagation in tag-aware queries is a subject of our future work.

Most of the work on annotations of relational data focuses on annotating individual values in a table. Geerts et al. [16] have taken a more sophisticated approach and provide support for annotating associations between values in a tuple. For example, in a query one might want to annotate fields A and B in the output with information that they came from input table R, and the B and C fields with information that they came from table S. The authors introduce the concept of a "block" – a set of fields in a tuple to which one attaches an annotation and a "colour" which is essentially the content or some property of the annotation. They investigate both the theoretical aspects and the overhead needed to implement the system. Our approach supports various tagging granularity by using regular expressions in the *Applicability* field in the *tick-tags*, and to maintain the tags' "lineage" we employ the streaming *stix* concept.

We are unaware of any work that addresses the problem of real-time data tagging in the context of Data Stream Management Systems and provides support for both explicit and implicit tag querying. Furthermore, our proposed approach is unique in that it is stream-centric, where tags are interleaved with data tuples in data streams, and the processing of these streaming tags is encapsulated inside tag-based query operators that can be combined with regular continuous query operators.

8. CONCLUSION

In this paper we have proposed a technique for tagging streaming data using a special type of metadata called the *tick-tags*. *Tick-tags* can serve a variety of purposes, including labelling or describing some underlying real-time information, and serving as means of disseminating useful knowledge in addition to what is captured by the content of data tuples. Our experimental results showed the scalability and performance benefits of the *tick-tag* approach compared to alternative solutions. We have also evaluated the costs of executing tag-aware and tag-oriented continuous queries.

We intend to pursue two directions as a part of our future work. First, we want to extend the support for real-time mining and data classification using tags. Furthermore, we intend to explore continuous query optimization using the knowledge of *tick-tags*. An interesting problem to investigate is whether the *tick-tag* awareness can also be used in query optimization at compile-time when determining a query execution plan, as well as at runtime (similar to *punctuations* [30]) to adapt the query execution strategy based on the observed streaming *tick-tags*.

9. ACKNOWLEDGEMENTS

The work reported in this paper has been partially supported by the AFOSR grant FA9550-07-0041 "Systematic Control and Management of Data Integrity, Quality and Provenance for Command and Control Applications" and the NSF grant 0712846 "IPS: Security Services for Healthcare Applications".

10. REFERENCES

[1] A. Arasu et.al. STREAM: The stanford stream data manager. In *SIGMOD*, page 665, 2003.
[2] B. Berendt et.al. Tags are not metadata, but "just more content" - to some people. In *ICWSM*, 2007.
[3] Bright Kite. http://brightkite.com/.
[4] T. Brinkhoff. A framework for generating network-based moving objects. *Geoinformatica*, 6(2):153–180, 2002.
[5] C. Koch et.al. FluXQuery: an optimizing XQuery processor for streaming XML data. In *VLDB*, pages 1309–1312, 2004.
[6] C. Marlow et.al. HT06, tagging paper, taxonomy, Flickr, academic article, to read. In *HYPERTEXT*, pages 31–40, 2006.
[7] C.Schmitz et.al. Mining association rules in folksonomies. In *IFCS*, pages 261–270. Springer, July 2006.
[8] D. Abadi et.al. Aurora: A data stream management system. In *SIGMOD*, page 666, 2003.
[9] D. Bhagwat et.al. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.
[10] D. Srivastava et.al. Intensional associations between data and metadata. In *SIGMOD*, pages 401–412, 2007.
[11] Delicious. http://del.icio.us/.
[12] E. Chi et.al. Understanding the efficiency of social tagging systems using information theory. In *HT*, pages 81–88, 2008.
[13] N. Eagle and A. Pentland. Reality mining: sensing complex social systems. *PUC*, 10(4):255–268, 2006.
[14] EBay. http://www.ebay.com/.
[15] E.Rundensteiner et.al. Cape: continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, 2004.
[16] F. Geerts et.al. MONDRIAN: Annotating and querying databases through colors and blocks. In *ICDE*, page 82, 2006.
[17] F. Morchen et.al. Anticipating annotations and emerging trends in biomedical literature. In *KDD*, pages 954–962, 2008.
[18] Flickr. http://flickr.com/, accessed in 2009.
[19] Google Latitude. http://www.google.com/latitude/intro.html.
[20] H.Halpin et.al. The complex dynamics of collaborative tagging. In *WWW*, pages 211–220, 2007.
[21] J. Li et.al. Out-of-order processing: a new architecture for high-performance stream systems. *VLDB*, pages 274–288, 2008.
[22] J. Ullman et.al. *Database Systems: The Complete Book*. Prentice Hall PTR, 2001.
[23] K. Ellul et.al. Regular expressions: new results and open problems. *J. Autom. Lang. Comb.*, 9(2-3):233–256, 2004.
[24] L. Chen et.al. Semantic tagging for large-scale content management. In *Web Intelligence*, pages 478–481, 2007.
[25] L. Ding et.al. Evaluating window joins over punctuated streams. In *CIKM*, pages 98–107, 2004.
[26] M. Ames et.al. Why we tag: motivations for annotation in mobile and online media. In *CHI*, pages 971–980, 2007.
[27] M. Eltabakh et.al. bdbms - a database management system for biological data. In *CIDR*, pages 196–206, 2007.
[28] Many Eyes DS. http://manyeyes.alphaworks.ibm.com/, 2009.
[29] O.Benjelloun et.al. Databases with uncertainty and lineage. *The VLDB Journal*, 17(2):243–264, 2008.
[30] P. Tucker et.al. Exploiting punctuation semantics in continuous data streams. *TKDE*, 15(3):555–568, 2003.
[31] F. Peng and S. S. Chawathe. Xpath queries on streaming data. In *SIGMOD*, pages 431–442, 2003.
[32] P.Heymann et.al. Social tag prediction. In *SIGIR*, 2008.
[33] R. Jaschke, et.al. Tag recommendations in social bookmarking systems. *AI Commun.*, 21(4):231–247, 2008.
[34] R.Nehme et.al. Security punctuation framework for enforcing access control on streaming data. In *ICDE*, 2008.
[35] S. Bose et.al. Data stream management for historical xml data. In *SIGMOD*, pages 239–250, 2004.
[36] S. Braun et.al. Social semantic bookmarking. In *PAKM*, pages 62–73, 2008.
[37] S. Chandrasekaran et.al. TelegraphCQ: Continuous dataflow processing. In *SIGMOD*, page 668, 2003.
[38] S. Golder et.al. The structure of collaborative tagging systems. *Journal of Information Science*, 32(2):198–208, 2006.
[39] S. Oldenburg et.al. Similarity cross-analysis of tag/co-tag spaces in social class. systems. In *SSM*, pages 11–18, 2008.
[40] Technorati. http://www.technorati.com/.
[41] W. Hung et.al. Similarity measures between type-2 fuzzy sets. *IJUFKS*, 12(6):827 – 841, 2004.
[42] X. Li et.al. Efficient evaluation of xquery over streaming data. In *VLDB*, pages 265–276, 2005.