Knowl Inf Syst (2019) 58:341–370
https://doi.org/10.1007/s10115-018-1195-9

REGULAR PAPER

Smart scheme: an efficient query execution scheme for event-driven stream processing

Salman Ahmed Shaikh[1] · **Yousuke Watanabe**[2] · **Yan Wang**[3] · **Hiroyuki Kitagawa**[1]

Received: 5 February 2017 / Revised: 10 January 2018 / Accepted: 18 April 2018 / Published online: 24 April 2018

Abstract With the increase in stream data, a demand for stream processing has become diverse and complicated. To meet this demand, several stream processing engines (SPEs) have been developed which execute continuous queries (CQs) to process continuous data streams. Event-driven stream processing, which is one of the important requirements, continuously gets the incoming stream data and, however, generates query results only on the occurrence of specified events. In the basic query execution scheme, even when no event is raised, input stream tuples are continuously processed by query operators, though they do not generate any query result. This results in increased system load and wastage of system resources. For this problem, we propose a smart event-driven stream processing scheme, which makes use of smart windows to buffer the stream tuples during the absence of an event. When the event is raised, the buffered tuples are flushed and processed by the downstream operators. If the buffered tuples in the smart window expire due to the window size before the occurrence of an event, they are deleted directly from the smart window. Since CQs once registered are executed for several weeks, months or even years, SPEs usually execute several CQs in parallel and merge their query plans whenever possible to save processing cost. Due to the presence of smart window, existing multi-query optimization techniques cannot work for smart event-driven stream processing. Hence, this work proposes a multi-query optimization for the proposed smart scheme to cover the cases where multiple continuous queries are

✉ Salman Ahmed Shaikh
salman@kde.cs.tsukuba.ac.jp

Yousuke Watanabe
watanabe@coi.nagoya-u.ac.jp

Yan Wang
wangyan@kde.cs.tsukuba.ac.jp

Hiroyuki Kitagawa
kitagawa@cs.tsukuba.ac.jp

1 Center for Computational Sciences, University of Tsukuba, Tsukuba, Japan

2 Institute of Innovation for Future Society, Nagoya University, Nagoya, Japan

3 Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Japan

registered. Extensive experiments are performed on real and synthetic data streams to show the effectiveness of the proposed smart scheme and its multi-query optimization.

Keywords Data stream processing · Event-driven processing · Smart query execution · Smart window · Multi-query optimization · Gate operator

1 Introduction

Big data is usually defined by 3Vs: volume, velocity and variety [1]. The focus of this work is velocity, which refers to the speed at which streaming data is generated. For instance, on average 510 comments are posted, 293,000 statuses are updated, and 136,000 photos are uploaded on Facebook per minute,[1] 350,000 tweets are posted on twitter per minute,[2] 52,500 searches per second are made on Google,[3] etc. In order to process the high velocity data, several stream processing engines (SPEs) have been developed [2–8]. SPE executes continuous queries (CQs) to process continuously arriving data streams and generates continuous results. End-users are not always interested in all the results generated by continuous queries, but in the results generated for some specific classes of incoming data and events. This is known as *event-driven stream processing* [9]. An event-driven query is activated by the occurrence of an event and is deactivated after the expiry of that event or after the execution for a fixed time duration. An event can be *simple*, i.e., a traffic accident, a purchase made at a grocery store, a sudden rise/fall of some stock price, etc., or it can be *complex*, i.e., a sequence of correlated simple events. There exist a lot of works on simple and complex event detection and their processing [9–11]; however, the focus of this work is not the event detection but the efficient event-driven processing. Here we assume that the events have already been detected or they are available as separate event streams and the goal is to process the non-event data streams efficiently based on the events available from the event streams. Event-driven stream processing is one of the important research issues among the data stream researchers and has many applications. For instance:

Event-based surveillance Let there is an audio stream and a video stream from a surveillance device mounted to monitor an area. Rather than recording all the video from the surveillance camera, users may be interested in recording the video for some fixed time duration after there is an abnormal sound detected, to save the storage space.

Tsunami detection If there is an earthquake of magnitude greater than some threshold, users may be interested in monitoring the sea/ocean waves for some fixed time duration for the possible tsunami waves. In this case, an earthquake of magnitude greater than threshold is a triggering event.

In order to express the *tsunami detection* application as a CQ, consider two data streams: earthquake notification stream and tsunami detection stream. Whenever there is an earthquake notification of magnitude greater than m from the earthquake notification stream, end-user wants to monitor the tsunami detection stream for the duration of τ time units, for the possible tsunami waves. A CQL style event-driven query for the *tsunami detection* application is shown below.

[1] https://zephoria.com/—accessed 01/21/2017.

[2] http://www.internetlivestats.com/—accessed 01/21/2017.

[3] http://www.internetlivestats.com/—accessed 01/21/2017.

```
Select tsunamiStream.waveMagnitude
From earthquakeStream[Range τ], tsunamiStream[Rows
    n]
Where earthquakeStream.regionID = tsunamiStream.
   regionID
And earthquakeStream.magnitude > m
```

Query. Event-driven query for tsunami detection

Although the above-mentioned event-driven stream processing can be achieved by join query utilizing time-based window for event stream, the use of time-based window generates a lot of useless intermediate tuples. In the basic query execution scheme available in most of the existing stream processing engines including Spark Streaming [12], STREAM [5], Borealis [3], Aurora [2] and Storm [4], an event-driven query continuously processes the incoming data streams, i.e., event-driven query operators keep their synopses updated with the newly arrived tuples, however, generates query results on the occurrence of an event. Although the absence of an event does not generate any query result, SPEs need to process all the incoming tuples from ordinary (non-event) streams continuously, so that correct query results may be generated on the arrival of an event stream tuple. These tuples are also processed by some of the query downstream operators to update their synopses. Continuous processing of these tuples in the absence of event stream tuples unnecessarily consumes system resources resulting in reduced system throughput. To address this problem, a *smart event-driven stream processing scheme* is introduced in this work. We focus on join queries using time-based windows expressing event-driven stream processing. The smart scheme uses smart windows to buffer the tuples arriving from ordinary streams during the absence of event stream tuples. (We say the query is *inactive*.) Hence during the query inactive duration or in the absence of event, ordinary stream tuples are only processed (and buffered) by our proposed smart window operator and are not sent to the other query operators, thus reducing the system load. The tuples in the smart window buffers are flushed and processed by the downstream operators only on the arrival of tuples from the event stream(s). (We say the query is *active*.) In addition, the buffered tuples which get expired due to the window sizes are deleted directly from the window buffers without being processed by the downstream operators. This results in reduced system load and eventually improved system throughput. The proposed scheme is especially useful for the cases where the event stream input rate is lower than the ordinary streams.

An SPE must be capable of executing multiple CQs simultaneously because once a CQ is registered to an SPE it is usually executed for several days, weeks, months or even years. Therefore, at any time an SPE may be executing several CQs. Many existing stream processing engines make use of different multi-query optimization techniques to merge similar partial query plans whenever possible to save computation cost. However, the existing multi-query optimization techniques cannot work in their present form with the proposed smart event-driven stream processing scheme due the introduction of smart windows. Hence, this work proposes a multi-query optimization scheme for the smart scheme, which is an extension of the multi-query optimization scheme to share sub-query plans studied by many researchers [13–15]. The smart windows contain buffering functionality, which must be retained for all the queries sharing smart windows when merging multiple smart windows. For this sake a gate operator is introduced in Sect. 6.2. We have developed a prototype SPE, JsSpinner, which implements the proposed smart scheme and multiple query optimization for the smart

Table 1 Symbols and abbreviations used in the manuscript

Symbol/abbreviation	Description
S	Stream
R	Relation
e	Stream tuple
Γ	Discrete, ordered time domain
t	Timestamp assigned to stream tuples from Γ
$+$	Flag assigned to insertion tuple
$-$	Flag assigned to deletion tuple
$\langle e, t, +/-\rangle$	A JsSpinner element with tuple e and timestamp t
n	Row-based window size
τ	Time-based window size
I_e	Event stream tuples arrival interval (s)
W_e	Event stream window size (s)
R_o	Ordinary stream arrival rate (tuples\s)
W_o	Ordinary stream window size (number of rows)
SPE	Stream processing engine
CQ	Continuous query
CQL	Continuous query language
SQL	Structured query language

scheme. JsSpinner is written in C++, is capable of processing semi-structured JSON data [16,17] and enables users to register CQL queries.

This work is an extended version of our previous work [18]. In [18], a smart query execution scheme for event-driven stream processing is given for which experiments are performed on synthetic data stream. The main contributions of this work can be summarized as follows:

- A multi-query optimization approach for smart event-driven stream processing.
- An extensive experimental evaluation on synthetic and real data streams to prove the effectiveness of the smart event-driven stream processing scheme and its multi-query optimization approach.
- A discussion on smart event-driven stream processing scheme and its multi-query optimization costs and benefits.

Table 1 lists the symbols and abbreviations used in the manuscript. The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 presents preliminaries and assumptions. In Sect. 4, the basic event-driven stream processing is presented. In Sect. 5, smart scheme for event-driven stream processing is introduced. The multi-query optimization for the smart scheme is presented in Sect. 6. In Sect. 7, an extensive experimental study is presented while Sect. 8 presents a discussion on smart scheme and its multi-query optimization costs and benefits. Section 9 concludes this paper and discusses future directions.

2 Related work

Since the focus of this work is the continuous event-driven query execution and multi-query optimization, we divide this section into two parts.

Execution strategies for CQs There are two main execution strategies for CQs: timer-based [19,20] and change-based [2,3,21–23] [4]. Timer-based (periodic) CQs are triggered only at the time specified by the user and are executed periodically for some constant time interval. Each time the timer-based CQ is triggered; it evaluates the newly arrived data. The timer-based CQs are analogous to the event-driven CQs, where the occurrence of an event or arrival of data from event stream triggers the query. On the other hand, change-based CQs are triggered as soon as new data become available.

Since a data stream is an infinite sequence of tuples, window management within limited memory is also an important research issue. The works [24,25] proposed methods for approximate join computation over data streams by using sliding windows. GrubJoin [26] considers sliding-window join with CPU load shedding. Grubjoin uses window-harvesting which picks the most profitable segments of individual windows for the join processing, in an effort to maximize the join output rate. However, the proposed smart scheme tries to improve system throughput by reducing system load. That is, the proposed smart scheme uses smart windows which allow only those tuples to the downstream operators which can contribute to query results.

The queries in this paper are based on the CQL query language [27]. Although the main focus of CQL is change-based CQs, we can use time-based windows to achieve the functionality of event-driven CQs. For this sake, the interval between the arrivals of event stream tuples serves as the interval between consecutive query triggering, and the size of the time-based window serves as the duration for which the query remains active. In this paper, a CQ which employs the time-based windows is called an *event-driven CQ*. This work proposes a smart scheme for efficient processing of event-driven continuous queries.

(Multi-)query optimization for CQs How to efficiently execute CQs is one of the most important research areas among the data stream community. According to [28], reordering operators in query plans contributes to reducing redundant intermediate results. The optimizer in [28] rewrites query plans to push down operators which have strict conditions and short computational time. Such reordering filters out a lot of unnecessary tuples at early stages.

A cost model to estimate the resource utilization and output rate of a query plan is proposed in [29]. To treat unstable data streams (with unstable data statistics and arrival rates), [29] finds an execution plan suitable for the available computational resources. When resources are sufficient, it finds an execution plan that minimizes the resource usage and when resources are insufficient it finds an execution plan that sheds some of the input load. [30] proposes an adaptive reordering method for pipelined filters. It monitors selectivities and correlations of filters to decide the optimal order. In [31], an operator eddy is proposed for adaptive query processing. Eddy controls tuples routing paths among operators, i.e., for each tuple, eddy dynamically determines the next operator.

A multi-query optimization scheme to share sub-query plans for static queries is studied by [13]. In their proposed scheme, the optimizer finds common sub-query plans from multiple queries registered to a system and generates a query plan sharing common sub-query plans. Their optimization scheme is also applicable to the CQs and was later used by CACQ [14] and PSoup [15] to share operators of change-based CQs. NiagaraCQ [32] deals with both timer-based CQs and change-based CQs and can share operators triggered by different events. NiagaraCQ proposes incremental grouping of CQs by identifying and merging sub-query plans to reduce computation cost and primary memory usage. However, their proposed grouping approach cannot work with the smart event-driven stream processing where the queries are activated by the arrival of their respective events. Hence, this work proposes the use of a special gate operator when multiple smart windows related to multiple queries, respectively, are merged. The gate operator enables merging of multiple similar smart event-driven query plans to save processing and memory costs by keeping the advantage of smart processing intact. In NiagaraCQ [32], resource sharing for specific operators is also considered. The problem of resource sharing when a large number of CQs are being processed simultaneously is also studied by [33]. However, their focus is sliding-window aggregates over data streams. An adaptive cache placement and removal technique for multi-way join is studied by [34] to reuse intermediate join results.

The multi-query optimization schemes discussed above are not applicable to the proposed smart scheme due to the special working of smart windows. Hence, this work extends a multi-query optimization to deal with sharing smart windows and introduced a gate operator to retain its functionality.

3 Preliminaries and assumptions

In order to process and query continuously evolving data streams, many SPEs have been developed. STREAM [5], Borealis [3], Aurora [2] and Storm [4] are a few examples of the well-known and commonly used SPEs. When a user registers a CQ on an SPE, it is executed continuously on the incoming stream tuples and generates continuous output. Like SQL (Structured Query Language) [35], which is a standard declarative query language for storing, manipulating and retrieving static data in databases, CQL (Continuous Query Language) [27] is a declarative query language for CQs over data streams. Unlike SQL queries, which are executed once to generate one-time results, CQL queries are executed continuously to generate continuous results. CQL is a state-of-the-art continuous query language, initially developed for STREAM, the prototype Data Stream Management System (also known as STanford stREam datA Manager) [5]. CQL is more general than many other continuous query languages and is therefore adopted by many SPEs. The prototype SPE, JsSpinner, developed for the proposed smart query execution scheme also supports CQL queries. For completeness, we summarize the CQL abstract semantics, its query plan and its incremental computation from [5,27], which are required to understand the proposed smart scheme. For details of CQL, readers are encouraged to refer to [5,27].

3.1 CQL abstract semantics

The CQL abstract semantics is based on two data types, streams and relations. Let Γ be discrete, ordered time domain then a *stream* is an unbounded multiset of pairs $\langle e, t\rangle$, where e is a tuple and $t \in \Gamma$ is the timestamp that denotes the arrival time of tuple e on stream S. Similarly, a *relation* is a time-varying multiset of tuples. The multiset at time $t \in \Gamma$ is denoted by $R(t)$, where $R(t)$ is an instantaneous relation.

The abstract semantics uses three classes of operators over streams and relations. (1) *relation-to-relation* operator takes one or more relations as input and produces a relation as output. (2) *stream-to-relation* operator takes a stream as input and produces a relation as output. (3) *relation-to-stream* operator takes a relation as input and produces a stream as output. A CQ Q is a tree of operators belonging to the three classes. Q uses leaf operators to receive inputs which could be streams and relations, and a root operator to produce output which could be either a stream or a relation. At time t, an operator of Q produces new outputs corresponding to t which depends on its inputs up to t.

CQL is defined by instantiating the operators of the abstract semantics. For the *relation-to-relation* operators, CQL uses existing SQL constructs. The *stream-to-relation* operators in CQL are based on sliding window over a stream, which are specified using window specification language derived from SQL-99. A window at any point of time holds a historical snapshot of a finite portion of the stream. In this work, two classes of sliding-window operators are used. (1) *tuple-based window* operator on a stream S is specified using an integer n. At any time t it returns a relation R of n most recent tuples from stream S. (2) *time-based window* takes a parameter τ and at any time t returns a relation R containing tuples with timestamps between $t - \tau$ and t from stream S. CQL has three *relation-to-stream* operators which are also adopted in our framework: i-stream, d-stream and r-stream. At time t, the i-stream (insert stream) applied to relation R results in a stream element $\langle e, t\rangle$ whenever tuple e is in $R(t) - R(t-1)$.[4] The d-stream (delete stream) returns a stream element $\langle e, t\rangle$ from R whenever tuple e is in $R(t-1) - R(t)$. The r-stream (relation stream) applied to R, results in a stream element $\langle e, t\rangle$ whenever tuple e is in R.

An example of a query written in CQL [27] is shown in Query 1, which performs continuous binary join with respect to common integer attribute A of streams S1 and S2.

```
Select S1.B, S2.C
From S1[Range τ], S2[Rows n]
Where S1.A = S2.A
```

Query 1 A simple CQL query

3.2 CQL query plan

A CQL query is translated into a query plan and is executed continuously. Query plans are composed of operators, queues and synopses. *Operators* perform actual processing on data streams. The data arrive at an operator as a sequence of timestamped tuples, where each tuple is additionally flagged as either an insertion (+) or deletion (−) as explained later. These tuple-timestamp-flag triplets are referred as elements. Each operator reads from one or more input queues, processes the input and writes any output to the output queue. *Queues* buffer elements as they move between operators. *Synopsis* is a buffer which belongs to a specific operator. It stores an operator's state that may be required for future evaluation of that operator.

A query plan for Query 1 is shown in Fig. 1. The query plan in Fig. 1 consists of seven operators: a root, an i-stream, a binary join, two instances of window operators and two instances of leaf operators. Note that the projection is performed as part of the binary join, so no separate projection operator or synopsis is employed. Queues q_1 and q_2 hold the input stream elements read by their respective leaf operators. Queues q_3 and q_4 hold elements representing the relations $S1$[Range τ] and $S2$[Rows n], respectively. Queue $q5$ holds elements for the result of joining relations $S1$[Range τ] and $S2$[Rows n]. Queue q_6 holds the elements coming out of the i-stream operator, which may lead to output or input to other query. The query plan has five synopses, *synopsis1–synopsis5*. Each window operator has a synopsis so that it can hold the current window elements and generate "−" elements when elements expire from the sliding window. The binary-join operator has two synopses, one for each input, to materialize each of its relational inputs. The i-stream operator has a synopsis to convert its relational input to stream output depending upon its semantics.

[4] For simplicity, we assume that a new tuple arrives at every time instant t.

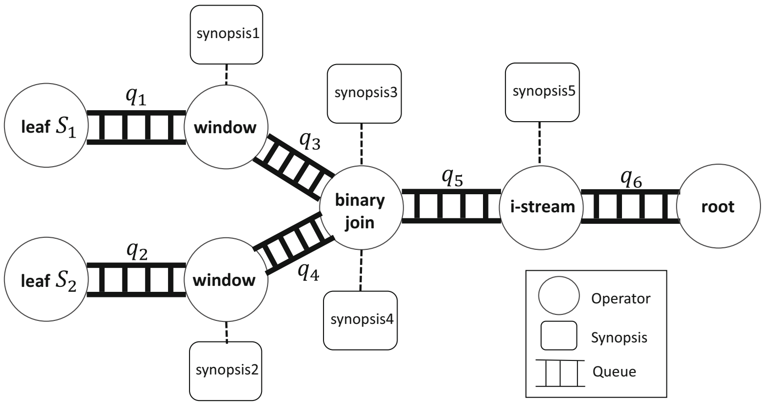


Fig. 1 Query plan for query 1

3.3 Incremental computation

A CQL query logically outputs elements based on $R(t)$ and $R(t-1)$, but computations required for $R(t)$ and $R(t-1)$ often have a lot of overlap. To eliminate redundant computation, incremental computation is used.

Considering the query plan shown in Fig. 1, the window operator on $S1$, on being executed reads element $\langle e, t, +\rangle$. It inserts element e in *synopsis1*, and if an old element e' expires, it removes that element from the synopsis. The window then outputs elements $\langle e, t, +\rangle$ and $\langle e', t, -\rangle$ to q_3 to reflect the addition and deletion of elements e and e', respectively. The other window operator executes in the similar fashion. When the binary-join operator is executed, it reads the newly arrived element from one of its two input queues, i.e., q_3 or q_4. If it reads an element $\langle e, t, +\rangle$ from q_3, then it inserts e into *synopsis3* and joins e with the contents of *synopsis4*, generating output elements $\langle e.f, t, +\rangle$ for each matching element f in *synopsis4*. Similarly, if the binary-join operator reads an element $\langle e', t, -\rangle$ from q_3, it generates $\langle e'.f, t, -\rangle$ for each matching element f in *synopsis4*. The same process is done for the elements read from q_4. The output elements from the binary join are enqueued to q_5. The i-stream operator reads the data from q_5, inserts it into *synopsis5*, converts the relational input to stream output and enqueues it to q_6 which is then output by the root operator.

4 Event-driven stream processing

Event-driven stream processing can be defined as the processing of a CQ activated by the occurrence of an event. The event-driven stream processing can be achieved by the use of time-based window available in most of the existing SPEs; however, their use in the basic form results in unnecessary system load. In this section, we discuss how the existing SPEs can handle the event-driven stream processing and the problem of the basic scheme. Key terms used in the following discussion are summarized below.

- *Event-driven query* A continuous query Q is event-driven if it generates any query result only after arrival of tuples from specified streams.
- *Event stream* The above specified stream is called an event stream. The event stream has its associated time-based window.

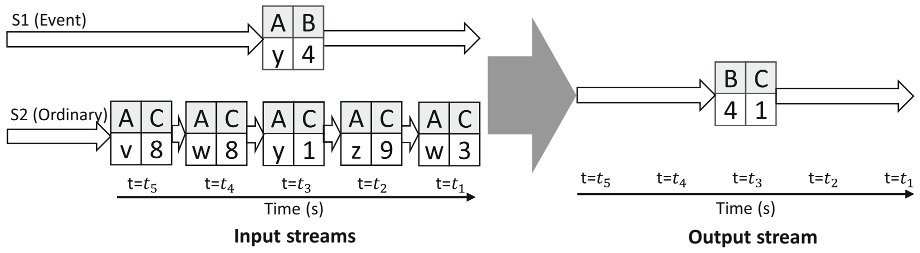


Fig. 2 Input and output streams for Examples 1 and 2

- *Ordinary stream* A non-event stream.
- *Active* An event-driven query Q is active when the time-based window of the event stream is not empty.
- *Active duration* The duration for which Q remains active. The active duration of Q is equivalent to the size of the time-based window.
- *Inactive* An event-driven query Q is inactive when the time-based window of the event stream is empty.

4.1 Basic scheme for event-driven stream processing

Query 1 shows a simple event-driven query written in CQL. In the query, S1 is an event stream whose tuples activate the query, whereas S2 is an ordinary stream. The query is executed continuously and generates output using the tuples arriving during the query active duration and the tuples available in the window when the query is activated. Even during the inactive duration, ordinary stream tuples are processed by some of the query operators; however, they may not contribute to the query results. The basic scheme works quite similar to that of the state-of-the-art SPE, STREAM [5].

Example 1 Figure 2 shows sample tuples from S1 and S2 and the output tuple generated from the query. For this example we assume $n = 2$ for Query 1. At timestamp t_1, a tuple $\langle w, 3\rangle$ arrives from S2, for which a corresponding "+" element $\langle w, 3, t_1, +\rangle$ is generated, stored in *synopsis2* and sent downstream. Since there is no tuple from S1 at t_1, no join output is generated. The states of the five query synopses (*synopsis1–synopsis5*) at timestamps t_1, t_2 and t_3 are shown in Table 2. At t_2, the query is still inactive due to the absence of tuples from S1; however, a corresponding element for the tuple arriving from S2 is generated and sent downstream, resulting in no output. The query becomes active with the arrival of a tuple from S1 at t_3 and "+" elements for the tuples arriving from S1 and S2 are stored in respective window synopsis and sent to the respective downstream operators. Since $n = 2$, arrival of a tuple from S2 at t_3 causes the tuple which arrived at t_1 to expire. For an expired element, corresponding "−" element is generated by its window operator and sent downstream to inform other operators about its expiration. Since there are elements available in *synopsis3* and *synopsis4* at t_3, join is performed and the result is generated and a "+" element corresponding to the join result is sent to the i-stream operator. The query is executed in the similar fashion at t_4, t_5 and at later timestamps.

4.2 The problem of the basic scheme

As shown above, the ordinary stream tuples that arrive during the query inactive duration may not contribute to the query results. However, in the basic scheme, they are added to the window

Table 2 Synopses snapshots of the basic scheme

Synopsis	t_1	t_2	t_3
1			$\langle y, 4, t_3, +\rangle$
2	$\langle w, 3, t_1, +\rangle$	$\langle w, 3, t_1, +\rangle$	$\langle w, 3, t_1, +\rangle$
		$\langle z, 9, t_2, +\rangle$	$\langle z, 9, t_2, +\rangle$
			$\langle w, 3, t_3, -\rangle$
			$\langle y, 1, t_3, +\rangle$
3			$\langle y, 4, t_3, +\rangle$
4	$\langle w, 3, t_1, +\rangle$	$\langle w, 3, t_1, +\rangle$	$\langle w, 3, t_1, +\rangle$
		$\langle z, 9, t_2, +\rangle$	$\langle z, 9, t_2, +\rangle$
			$\langle w, 3, t_3, -\rangle$
			$\langle y, 1, t_3, +\rangle$
5			$\langle 4, 1, t_3, +\rangle$

synopsis and corresponding "+" elements are sent to the downstream operators. Similarly, for each element that expires from the window, a "−" tuple is sent to the downstream operators to cancel the corresponding "+" element as shown in Table 2. The "+" and "−" tuples are also processed by the downstream operators and unnecessarily increase system load.

5 Smart scheme for event-driven stream processing

The basic scheme for event-driven stream processing generates a lot of useless intermediate tuples which may not contribute to the query results. To avoid this, we propose a smart scheme for event-driven stream processing. The key idea is that ordinary stream tuples need to be processed to maintain the current status of row-based windows to guarantee correct query results on the arrival of an event stream tuple. However, corresponding "+" elements do not need to be sent to the downstream operators when the query is inactive. The smart scheme makes use of a smart window to buffer the tuples arriving from the ordinary stream during the absence of event stream tuples. In the smart scheme, when a tuple e arrives from an ordinary stream, the system checks whether the query is active or inactive. If the query is inactive, the smart window operator buffers e inside the smart window and does not output any "+" element. If the query is active, the smart window outputs "+" element corresponding to e. On the query activation, the smart window generates "+" elements for all the buffered elements. During the query inactive duration, the buffered elements which expire due to the window size are deleted directly from the smart window without the need to generate "−" tuples for them.

5.1 Smart window

The synopsis of the smart window operator is divided into two parts: *output* and *suspended*. Both the output and suspended parts keep recent incoming tuples of the ordinary stream. If the query is inactive, a new arriving tuple is first put into the suspended part. When the query is activated due to the arrival of a tuple from the event stream, "+" elements corresponding to the tuples inside the suspended part are output to the downstream operators. In addition, the elements in the suspended part are moved to the output part. When elements in the output part get expired, corresponding "−" elements are sent to the downstream operators. However,

Table 3 Synopses snapshots of the smart scheme

Synopsis	t_1	t_2	t_3
1			$\langle y, 4, t_3, +\rangle$
2	S $\langle w, 3, t_1, +\rangle$	S $\langle w, 3, t_1, +\rangle$	S $\langle w, 3, t_1, +\rangle$
		S $\langle z, 9, t_2, +\rangle$	S $\langle w, 3, t_3, -\rangle$
			O $\langle z, 9, t_2, +\rangle$
			O $\langle y, 1, t_3, +\rangle$
3			$\langle y, 4, t_3, +\rangle$
4			$\langle z, 9, t_2, +\rangle$
			$\langle y, 1, t_3, +\rangle$
5			$\langle 4, 1, t_3, +\rangle$

when the elements in the suspended part get expired, no "+" or "−" elements are sent to the downstream operators. Algorithms 1 and 2 show the working of the smart window when a new tuple arrives and when query becomes active, respectively.

Example 2 Once again consider the input tuples from streams S1 and S2 shown in Fig. 2. Assuming $n = 2$ for Query 1, arrival of a tuple $\langle w, 3\rangle$ from S2 at timestamp t_1 causes the generation of a corresponding "+" element $\langle w, 3, t_1, +\rangle$ which is stored in the suspended part of *synopsis2*. (The characters "S" (suspended) and "O" (output) with the elements in *synopsis2* show whether the elements belong to the suspended or output part of the smart window.) Note that in the smart scheme in contrast to the basic scheme, for the elements arriving during the query inactive duration, no "+" element is sent to the downstream operators. Since there is no tuple from S1 at t_1, no join output is generated. The states of the five query synopses (*synopsis1*–*synopsis5*) at timestamps t_1, t_2 and t_3 are shown in Table 3. At t_2, the query is still inactive due to the absence of tuples from S1, however a corresponding "+" element for the tuple arriving from S2 is generated and stored in the suspended part of *synopsis2*. The query becomes active with the arrival of a tuple from S1 at t_3 and "+" elements for the tuples arriving from S1 and S2 are generated and stored in the respective window synopsis. Since $n = 2$, arrival of a tuple from S2 at t_3 causes the tuple which arrived at t_1 to expire. The expired element is deleted directly from the smart window synopsis without the need to send "−" tuple to the downstream operators, in contrast to the basic scheme. Furthermore, the elements in the suspended part of the smart window are moved to the output part and their corresponding "+" elements are sent to the downstream operators. Since there are elements available in *synopsis3* and *synopsis4* at t_3, join is performed and the result is generated and a "+" element corresponding to the join result is sent to the i-stream operator. The query is executed in the similar fashion at t_4, t_5 and at later timestamps.

Comparing the synopses snapshots of the basic and smart schemes in Tables 2 and 3, respectively, one can observe that the smart scheme generates less intermediate elements. At timestamps t_1 and t_2, in contrast to the basic scheme, *synopsis4* in the smart scheme does not need to hold any element. This is because the elements which arrived during the inactive duration are buffered in the suspended part of the smart window. Even when the query gets active at t_3, *synopsis4* in the smart scheme contains smaller number of tuples than the basic scheme. This results in the reduced system load.

Algorithm 1 Smart Window (W_o): When new tuple arrives

```
1: for each arrival of ordinary stream tuple $e \in Q$ at timestamp $t$ do
2:   if isActive($Q$) then
3:     Insert $e$ in the output part and send $\langle e, t, + \rangle$ downstream
4:   else
5:     Buffer $e$ in the suspended part
6:   end if
7:   if # of elements $\in W_o >$ size of $W_o$ then
8:     Find $e'$; {$e'$: oldest element in $W_o$}
9:     if $e' \in$ suspended part then
10:      delete $e'$
11:    else
12:      delete $e'$ and send $\langle e', t, - \rangle$ downstream
13:    end if
14:  end if
15: end for
```

Algorithm 2 Smart Window (W_o): When query activates

```
1: if query $Q$ becomes active with the arrival of an event stream tuple then
2:   Move elements from the suspended to the output part and send corresponding "+" elements downstream
3: end if
```

5.2 Effectiveness of the smart scheme

The smart scheme is especially useful when one or more event streams have low arrival rate. In the event-driven stream processing, the query is activated only on the arrival of a tuple from the event stream and remains active for the duration of time-based window size.

In order to evaluate the effectiveness of the smart scheme for a simple join query like Query 1, consider I_e, W_e, R_o and W_o defined in Table 1. The main advantage of the smart scheme comes from the ordinary stream tuples being deleted directly from the suspended part of the smart window. The number of ordinary stream tuples arriving during the inactive duration can be given by $(I_\mathrm{e} - W_\mathrm{e}) * R_\mathrm{o}$. Hence, the smart scheme is advantageous if the number of ordinary stream tuples arriving during the inactive duration is greater than W_o:

$$(I_\mathrm{e} - W_\mathrm{e}) * R_\mathrm{o} > W_\mathrm{o} \tag{1}$$

6 Multi-query optimization for smart scheme

A stream processing engine must be capable of executing multiple continuous queries in parallel because once a continuous query is registered to a stream processing engine it is usually executed for several days, weeks, months or even years. Therefore at any time a stream processing engine is usually executing several continuous queries and many times on the same data sources. To avoid wastage of computational and memory resources, most of the existing stream processing engines make use of different multi-query optimization techniques to merge similar partial query plans whenever possible. Most of the existing solutions to merge query operators or sub-query plans work by first identifying common data sources among the registered queries [13] and then identifying and merging downstream query operators. However, the existing multi-query optimization techniques cannot work in their present form with the proposed smart event-driven stream processing scheme due the introduction of smart windows. The smart window provides a buffering mechanism during

Table 4 Operators mergeability rules

Operator	Class[a]	Rule
Selection	r-to-r	Have the same selection condition
Projection		Have the same projection condition
Binary join		Have the same join condition
Aggregation		Have the same aggregate attribute and the list of group by attributes
Window	s-to-r	Have the same input stream source, window type and size
i-stream	r-to-s	If preceding operators can be merged
d-stream		If preceding operators can be merged
r-stream		If preceding operators can be merged

[a] *r* relation, *s* stream

query inactive duration which cannot work if two similar smart windows from two different queries are merged with different event streams.

Example 3 Assume two simple join queries Q_1 and Q_2. Let the Q_1 is a join query between event stream $S1$ with time window size 1 s and ordinary stream $S2$ with row window size 1000 rows, while Q_2 being join query between ordinary stream $S2$ with row window size 1000 rows and event stream $S3$ with time window size 5 s. Further assume that the event stream arrival interval of $S1$ is 5 s and of $S3$ is 30 s. Since Q_1 and Q_2 share ordinary stream $S2$ and its window size, existing multi-query optimization schemes can simply merge the two queries by merging the stream source $S2$ and its window. However in doing so, the smart stream processing functionality of the proposed smart scheme is lost. In other words, the query activation time instance and the activation interval of the two queries are different due to different event stream sources and their window sizes. Therefore, the merged smart window of stream $S2$ must buffer stream tuples for two different durations for two queries Q_1 and Q_2, respectively, which is not possible

To solve the above-mentioned problem, this work proposes a multi-query optimization scheme for the smart scheme, which is an extension of the multi-query optimization scheme to share sub-query plans studied by many researchers [13,14] [15]. To retain the buffering functionality of smart windows while merging them and to enable smart processing of ordinary stream tuples, a gate operator is proposed in this work.

6.1 Merging query plans

When a continuous query Q is registered to our prototype SPE, JsSpinner, it is translated into a query plan. A query plan consists of a tree of operators, where each operator belongs to a class mentioned in Table 4, to execute in turn. The proposed multi-query optimization approach merges the common sub-query plans including smart windows of the registered queries whenever possible. In order to merge query plans of two queries, their operators are compared one by one starting from the leaf (the first operator in any query plan responsible for reading data). If the first operator can be merged, the next operator is checked for mergeability and so on. In the following, the details of operators merging are discussed. We divide operators merging into two groups, i.e., (1) Merging ordinary operators and (2) Merging window operators, due to the special case of smart window operator.

Merging ordinary operators Two operators from two different query plans are mergeable if and only if they are same and follow the rule of mergeability. Table 4 lists the rules of mergeability for different operators. Assuming that the operators O_1 and O_2 of queries Q_1 and Q_2, respectively, are mergeable according to Table 4. To merge O_2 into O_1, the output queue of O_2 is replaced by the output queue of O_1 and O_2 is deleted. The process of merging query plans starts from their leaf operators. If the input sources are same, their leaf operators can be merged. After merging the leaf operators, next operators in the query plans are checked for mergeability. On the other hand, if the leaf operators cannot be merged, none of their query plan operators can be merged.
Merging window operators According to Table 4, two windows can be merged if they have the same window type and size. In this work we consider tuple-based and time-based sliding windows. The smart scheme converts the tuple-based sliding window associated with an ordinary (non-event) stream into a smart window which contains buffering functionality during the query inactive duration. Hence, the merging of window operators can be classified as follows.

– *Time-based windows merging* Two time-based windows can be merged if they have the same stream source and the window size. In smart scheme, time-based windows are associated with event streams. If two time-based windows related to two event streams of two different queries are merged into one, it activates both the queries simultaneously for the same time duration.
– *Tuple-based windows merging* Two tuple-based windows can be merged if they have the same stream source and the window size. The smart scheme converts tuple-based windows into smart windows, and their merging follows the rule of smart windows merging as discussed below.
– *Smart windows merging* Smart windows are associated with the ordinary (non-event) streams. Smart windows buffer tuples during the query inactive duration and either directly delete the buffered tuples on their expiration or forward them to the downstream operators on the corresponding query activation. If two smart windows associated with two different queries are merged, their activation timings may differ since they may be activated by different event streams. If we simply merge smart windows, the resulting smart window will output the buffered tuples to all the queries' downstream operators if any of the queries becomes active. This causes problem for the inactive queries, as the output tuples are also processed by their downstream operators. In order to deal with this problem, a gate operator is introduced which retains the functionality of smart window by providing buffering, forwarding and direct deletion functionality of stream tuples and is discussed in Sect. 6.2.

6.2 The gate operator

The gate operator is proposed in this work to retain the buffering, forwarding and direct deletion functionality of smart windows tuples when they are merged. The gate operators are placed at the end of the shared sub-query plans starting at the smart window, one for each merged query as shown in Fig. 3. When two smart windows are merged into one, it behaves like an ordinary row-based window rather than a smart window. A gate operator decides when to pass the incoming "+" or "−" tuples downstream, and its behavior is very similar to the smart window operator. If one or both the queries sharing the merged sub-query plan are inactive, the corresponding gate operator buffers the incoming "+" tuples. Furthermore, when the gate operator receives a "−" tuple from upstream, it directly deletes the corresponding

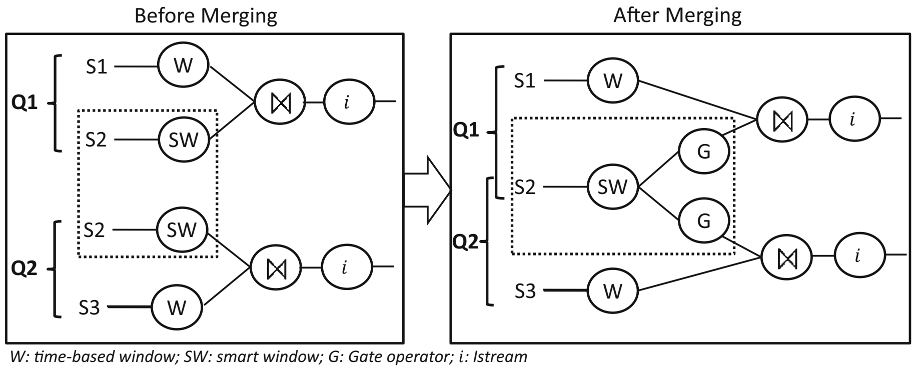

Before Merging
After Merging
Q1
Q2
S1
S2
S3
W
SW
G
i
W: time-based window; SW: smart window; G: Gate operator; i: Istream

Fig. 3 Merging query plans

"+" tuple if it is in the buffer. On the other hand when the query becomes active with the arrival of event stream tuple(s), the buffered "+" tuples are forwarded by the corresponding gate operator to the downstream operators. Figure 3 shows a simple case for explanation, however the proposed scheme is general and is applicable to more complex cases.

Consider two queries $Q1$ and $Q2$ shown in Fig. 3. $Q1$ and $Q2$ share stream source $S2$ and lets assume that the window size of $S2$ is same in both the queries. Hence, according to the rules queries $Q1$ and $Q2$ can be merged by merging the common stream source $S2$. In order to enable both the queries to retain their smart query processing functionality, a gate operator is placed for each query after the merged smart window as shown on the right of Fig. 3. Hence, when the query $Q1$ is active while the query $Q2$ is inactive, the gate operator for query $Q2$ buffers the incoming stream tuples from the shared smart window while the gate operator for query $Q1$ let the stream tuples flow to the downstream operators.

7 Experiments

7.1 Experimental setup

In this section we present a detailed experimental study to evaluate the effectiveness of the smart scheme [18] and the proposed multi-query optimization for smart scheme. For the sake of experiments a prototype SPE, JsSpinner, which enables users to register CQL queries is used. The JsSpinner source program consists of about 13,000 lines of C++ code. JsSpinner supports both the basic and the smart event-driven stream processing schemes. It also supports multiple queries. Experiments are performed on Dell Precision T3400 with Intel Core2 Quad (Q6700 @ 2.66 GHz x 4) CPU and 4 GB RAM running Ubuntu 14.10 OS.
Data streams For the experiments, we used both the synthetic and the real data streams. In total four synthetic data streams are used for experiments, i.e., S0, S1, S2 and S3. The schemas of S0, S1, S2 and S3 are as follows: S0(A, E, G), S1(A, B), S2(A, C) and S3(A, D). Here A is a common string attribute of all the streams, B, C, D and G are the integer attributes, and E is a string attribute. The synthetic data streams are generated at different rates using random strings for the string attributes and random integer values for the integer attributes.

The real data streams include Tokyo metropolitan people flow stream (*people flow* for short) [36] and Tsukuba mobility stream.[5] The people flow is a spatio-temporal data stream of around 580,000 people in Tokyo city of Japan, who report their location coordinates along with some other attributes every minute. The original stream contains 14 attributes; however we extracted some useful attributes for our experiments. The people flow stream schema used in this work is as follows: peopleFlow (PID char, PDate datetime, Longitude double, Latitude double, Sex int, Age int, Work char). The Tsukuba mobility is a spatio-temporal data stream of taxis and buses in Tsukuba city of Japan, collected by National Institute for Land and Infrastructure Management, Japan. The stream contains the location coordinates along with some other attributes reported by the taxis and buses GPS system periodically. The Tsukuba mobility stream schema used in this work is as follows: tsukuMobility (ObservationID char, RecordTimestamp datetime, Latitude double, Longitude double, GeneratedTimestamp datetime, Validity char). Since the arrival rates of the real data streams are not so high, we used a simulator to feed the real data streams to the JsSpinner at a faster rate.

Evaluated queries To prove the effectiveness of the proposed smart scheme and its multi-query optimization, we evaluated the system load and maximum system throughput on the Queries 1, 2 and 3. The *system load* is defined as the total number of tuples processed by all the query operators, whereas the *maximum system throughput* is defined as the total number of input streams (including event and ordinary) tuples processed by the system per unit time. In each query, the stream with the time-based window is an event stream, and arrival of data from it activates the query. The query remains active for the duration of the time-base window size. Queries 1 and 2 are join queries involving one and two event streams, respectively. Query 2 with two event streams is active when both time-based windows are not empty. Query 3 is a join query with an aggregate function and a group-by clause. Each query is executed for 60 s and each experiment is performed 5 times and their average values and standard deviations are used in the graphs. Unless otherwise stated, the following default parameter values are used in the experiments: $I_e = 5$ s, $W_e = 1$ s, $R_o = 300$k tuples/s and $W_o = 10$ rows.

```
Select S0.E, S1.B, S3.D
From S0[Rows n], S1[Range τ1], S3[Range τ2]
Where S0.A=S1.A And S1.A=S3.A
And S0.E = "abc"
```

Query 2 Multiple event stream

```
Select S0.E, avg(S0.G)
From S0[Rows n], S1[Range τ]
Where S0.A = S1.A
Group by S0.E
```

Query 3 Aggregation query

7.2 Experimental evaluation

Since there are two major contributions of this work, i.e., smart scheme and its multi-query optimization, the experimental evaluation is divided into smart scheme and multi-query.

[5] Tsukuba mobility data stream is provided by Tsukuba city, National Institute for Land and Infrastructure Management and University of Tsukuba.

7.2.1 Smart scheme

The objective of the experiments in this section is to show the advantage of the proposed smart scheme. The experiments are performed by varying the parameters I_e, W_e, R_o and W_o. We further sub-divided the experiments in this section into synthetic and real data streams. *Evaluation on synthetic data streams* Here we evaluate the maximum system throughput and the system load of our proposed smart scheme.

Varying I_e In this part of experimental evaluation, we compare the maximum system throughput and the system load of the smart scheme with the basic scheme and two state-of-the-art stream processing engines, STREAM [5] and Spark Streaming [12]. STREAM, also known as stanford stream data manager, is based on incremental stream processing framework. STREAM executes CQL queries to process continuous data stream. In general, our prototype system JsSpinner works quite similar to that of STREAM SPE when executed in the basic mode. Apache Spark Streaming on the other hand processes data in mini batches. Spark streaming queries are usually written in Scala, Python and R scripting languages.

Firstly we perform experiments to compare the maximum system throughput. Since the JsSpinner, STREAM and Spark are different stream processing engines with different architectures and data processing models, we compared the percent increase in the maximum system throughput to keep the comparison fair. To measure the percent increase, we measured the maximum system throughput of the smart scheme and other comparative approaches by varying I_e from 1 to 9 s. To recall, at $I_e = 1$ s with $W_e \geq 1$ s, Eq. 1 does not hold, i.e., the query remains active all the time. If the query is active always, smart scheme behaves like basic scheme or ordinary processing in STREAM and Spark Streaming. We would like to measure the percentage increase in the system throughput when the query is inactive for some duration which can be managed by varying the parameter I_e.

Since at $I_e = 1$ s smart scheme behaves similar to that of ordinary stream processing, percent increase in maximum system throughput is taken as 0 for all the queries as shown in Fig. 4. For $I_e > 1$ s, the percent increase in maximum system throughput is computed by comparing it against the system throughput at $I_e = 1$ s. With $I_e > 1$ s and $W_e = 1$ s Eq. 1 holds, i.e., the query remains inactive for the duration of $I_e - W_e$. Once the query is inactive, ordinary incoming stream tuples are buffered by smart scheme in smart window whereas basic scheme and other SPEs process them, though not generating any output. Thus smart scheme results in sharp increase in the percentage of system throughput compared to basic scheme, STREAM and Spark Streaming. Similar phenomenon can be observed for all the queries, i.e., Queries 1, 2 and 3 in Fig. 4a–c, respectively. Furthermore it can be observed from Fig. 4 that the advantage is higher in case of Queries 2 and 3. This is due to the larger number of operators in Queries 2 and 3 compared to Query 1. Hence, more complex the query is, higher the advantage will be as the smart scheme stops the incoming stream tuples from being delivered to the downstream operators in the absence of event.

Next we compare the system load of the proposed smart scheme with the basic scheme and the STREAM SPE. Here we do not compare our scheme with the Spark Streaming, as the query processing model of it is quite different from our prototype system. From Fig. 5, it is clear that the system load is smaller for the proposed smart scheme than the basic scheme and the STREAM SPE for all the queries. However, the system load of both the schemes (smart and basic) and the STREAM SPE are equal at $I_e = 1$ s because at this value, Eq. 1 does not hold for the default parameter values and the smart scheme works like the basic scheme. For the higher I_e values, the smart window buffers and deletes directly a lot of tuples (as can be observed from Fig. 6a) resulting in reduction in average system load. On

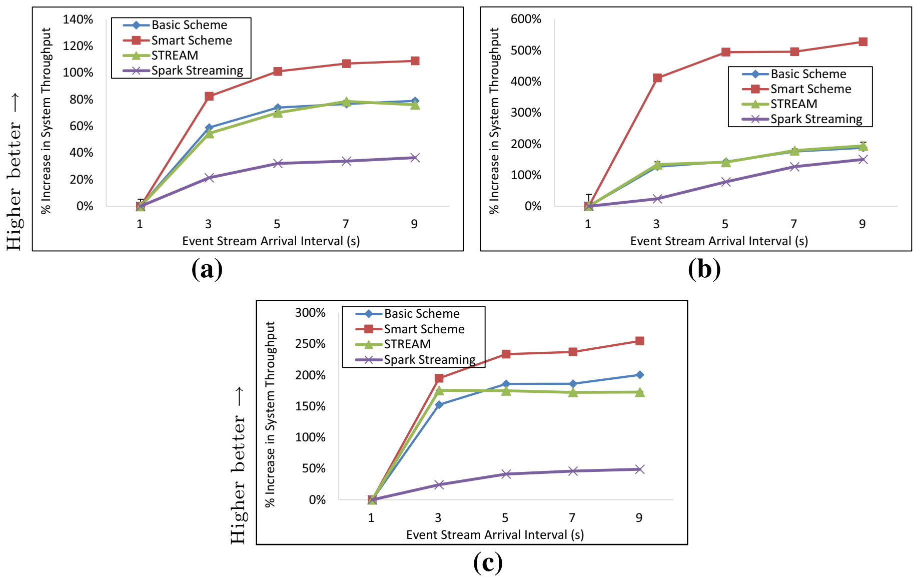


Fig. 4 Percent increase in system throughput evaluation on synthetic data stream ($W_e = 1$ s and $W_o = 10$ rows). **a** Query 1, **b** query 2, **c** query 3

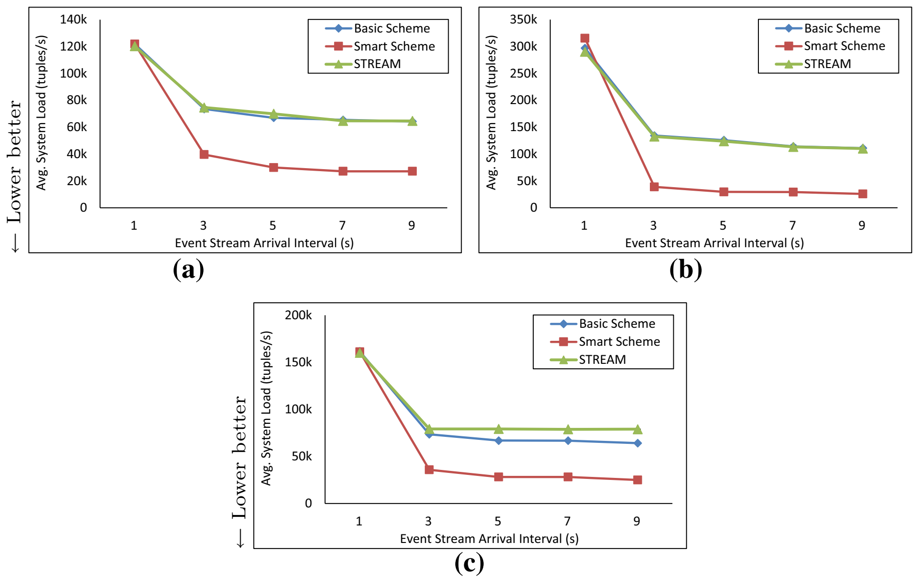


Fig. 5 Average system load evaluation on synthetic data stream ($R_o = 20$ k tuples/s, $W_e = 1$ s and $W_o = 10$ rows). **a** Query 1, **b** query 2, **c** query 3

the other hand, basic scheme and STREAM SPE continue to be overloaded even during the query inactive duration, i.e., at $I_e > 1$ s. This is due to the fact that the basic query execution scheme and other ordinary stream processing engines continue to maintain query operators' synopses during the query inactive duration resulting in higher system load. One can further

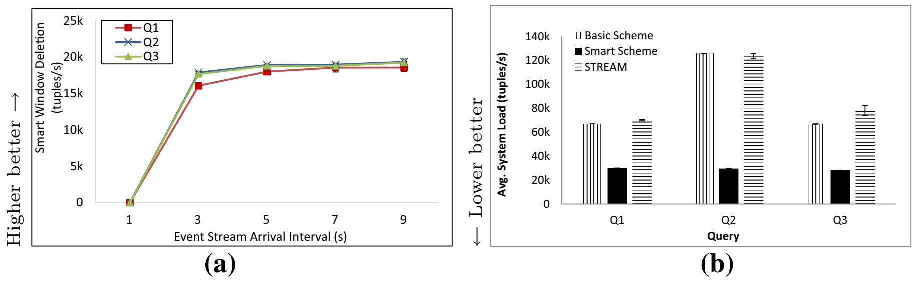


Fig. 6 Directly deleted tuples and average system load evaluation ($I_e = 5$ s, $R_o = 20$ k tuples/s, $W_e = 1$ s and $W_o = 10$ rows). **a** Directly deleted tuples, **b** avg. system load

observe that the system load is higher for Queries 2 and 3 compared to Query 1. This is because the Queries 2 and 3 contain larger number of operators than Query 1, where each operator (except a few operators) maintains a synopsis. Since each synopsis stores a number of tuples to maintain its state, larger number of operators results in higher overall system load.

Figure 6b compares the average system load of Queries 1, 2 and 3. The smart scheme seems to be more efficient for Query 2. This is because of the large number of operators in Query 2 compared to Queries 1 and 3 following the smart window. During the query inactive duration, the basic scheme and the STREAM SPE process the ordinary stream tuples among many operators before they are deleted by their respective "−" tuples. However, direct deletion of tuples from the smart window in the smart scheme during the query inactive duration reduces system load.

Varying R_o, W_e and W_o From Figs. 4 and 5, we found that the basic scheme and the STREAM SPE behave similarly. Furthermore, we found that the percentage increase in the system throughput in case of Spark Streaming is far less than the Smart scheme. Hence in the rest of the experiments, the smart scheme is compared only with the basic scheme while omitting comparison with STREAM SPE and Spark Streaming.

Figures 7 and 8 compare the basic and the proposed smart schemes for the average system load and the system throughput by varying parameters R_o, W_e and W_o on Query 1. Note that here we measure the maximum system throughput and not the percent increase in maximum system throughput, as both the schemes are executed on the same stream processing engine, i.e., the JsSpinner. From Fig. 7a, it can be observed that the proposed scheme behaves better than the basic scheme, i.e., the average system load of the smart scheme is always smaller than the basic scheme as the smart scheme does not send unnecessary tuples to the query downstream operators during the query inactive duration. In Fig. 7b, average system load is measured by varying the time-based window size (W_e) from 1 to 10,000 ms. Here again, the smart scheme system load is smaller than the basic scheme except for the $W_e = 10$ k, because for $W_e = 10$ k Eq. 1 does not hold and the smart scheme behaves like the basic scheme. We also measured the average system load by varying the ordinary stream window size from 1 to 10,000 rows. Figure 7c clearly shows the advantage of the smart scheme, i.e., lower average system load compared to the basic scheme.

In Fig. 8, maximum system throughput is evaluated for Query 1 by varying parameters R_o, W_e and W_o. In Fig. 8a we varied the ordinary stream arrival rate (R_o) and measured the system throughput in tuples/second instead of maximum system throughput. In other words we supplied JsSpinner SPE with increasing number of ordinary stream tuples to find which approach can handle the load smoothly. From the figure one can observe that the smart scheme

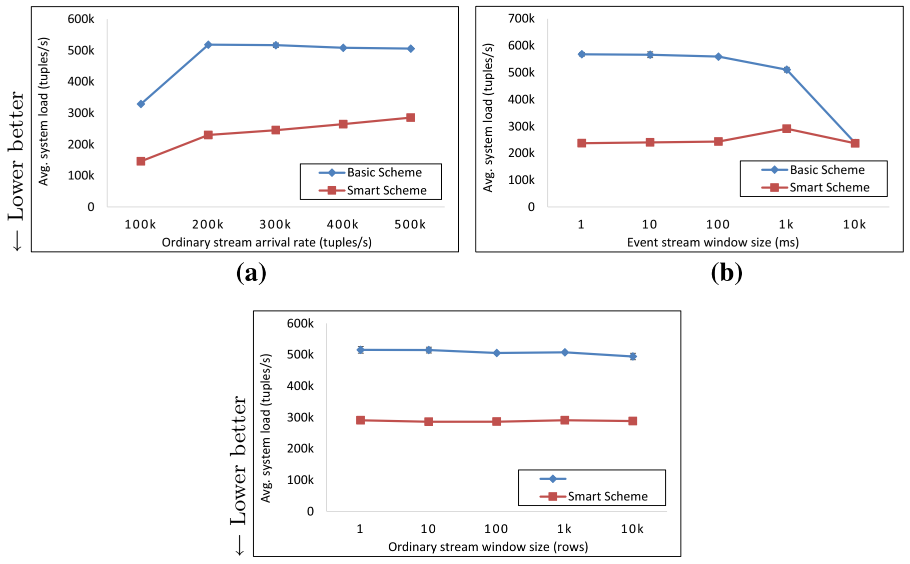


Fig. 7 Average system load evaluation using Query 1 on synthetic data stream. **a** Varying R_o ($I_e = 5\,\mathrm{s}$, $W_e = 1\,\mathrm{s}$ and $W_o = 10\,\mathrm{rows}$), **b** varying W_e ($I_e = 5\,\mathrm{s}$, $R_o = 300\,\mathrm{k}$ tuples/s and $W_o = 10\,\mathrm{rows}$), **c** varying W_o ($I_e = 5\,\mathrm{s}$, $R_o = 300\,\mathrm{k}$ tuples/s and $W_e = 1\,\mathrm{s}$)

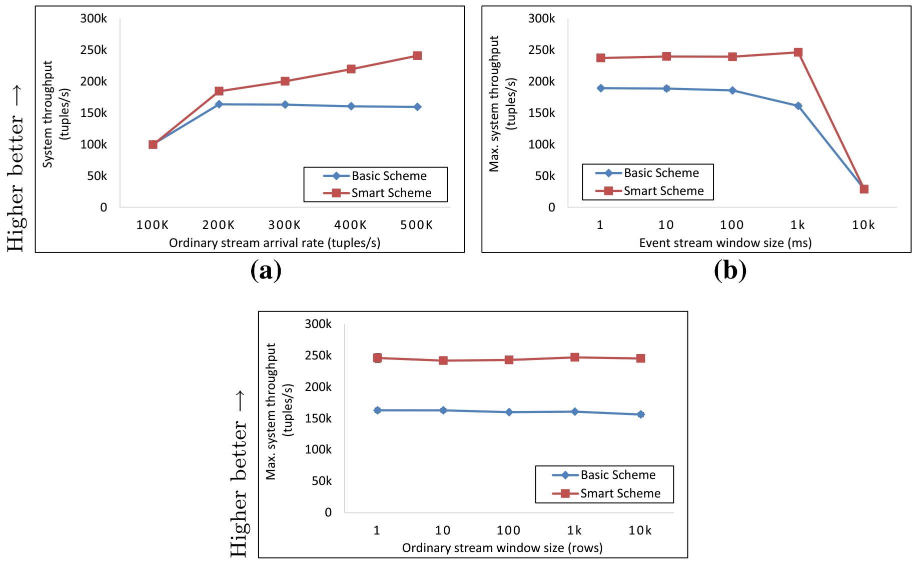


Fig. 8 System throughput evaluation using Query 1 on synthetic data stream. **a** Varying R_o ($I_e = 5\,\mathrm{s}$, $W_e = 1\,\mathrm{s}$ and $W_o = 10\,\mathrm{rows}$), **b** varying W_e ($I_e = 5\,\mathrm{s}$ and $W_o = 10\,\mathrm{rows}$), **c** varying W_o ($I_e = 5\,\mathrm{s}$ and $W_e = 1\,\mathrm{s}$)

can manage increase in R_o quite smoothly as compared to the basic scheme. In other words, smart scheme can better handle the fast arriving data streams. This is due to the lower system load of the smart scheme as shown in Fig. 7a, enabling to handle higher input streams. On the other hand, higher system load causes the basic scheme to handle far less data compared to the smart scheme.

Next we evaluated maximum system throughput by varying W_e and W_o for Query 1. From Fig. 8b it can be observed that the smart scheme results in higher maximum system throughput than the basic scheme except for $W_e = 10\,\text{k}$, for which Eq. 1 does not hold. Similarly, we compared maximum system throughput by varying the parameter W_o and from Fig. 8c we can observe the clear advantage of the smart scheme, i.e., the smart scheme results in higher overall system throughput.

Evaluation on real data streams For the experiments on real data streams we did not use any event stream; however we activate the query periodically for some fixed time duration. We used two real data streams for the experiments, i.e., peopleFlow and tsukuMobility, the query on these streams contains only the projection operation, i.e.,

$$\pi_{PID,Longitude,Latitude,Sex}(peopleFlow)$$

and

$$\pi_{ObservationID,Longitude,Latitude}(tsukuMobility)$$

The experiments on the real streams are performed by varying (i) the time period after which the query activates (I_e), (ii) the query activation duration (W_e), and (iii) the real stream window size (W_o). Since the arrival rate of the real streams is low, we developed a simulator to feed the buffered streams at around 100,000 tuples/s.

Figures 9 and 10 compare the average system load of the smart schemes with the basic scheme for the people flow and Tsukuba mobility data streams, respectively. Both the figures show variation of different parameters. In Fig. 9a, parameter I_e is varied from 1 to 9 s with a step of 2 s. The figure shows clear advantage of smart scheme over basic scheme for the people flow data stream as the later one results in smaller overall system load for all I_e values except for $I_e = 1\,\text{s}$ where Eq. 1 does not hold. Fig. 9b and c presents the evaluation of overall system load against the variation of event stream window size (W_e) and ordinary stream window size (W_o), respectively. Here again the system load for the smart scheme is lower compared to the basic scheme for all the values of W_e and W_o, except for $W_e = 10{,}000\,\text{ms}$, where Eq. 1 does not satisfy and the smart scheme behaves like the basic scheme. Similar graphs could be observed for the Tsukuba mobility data stream in Fig. 10.

Figures 11 and 12 show the comparison of the maximum system throughput for the two real data streams. As can be observed from the figures, smart scheme's throughput is almost twice to that of basic scheme in all cases. The higher system throughput of the smart scheme is the consequence of the lower average system load of the smart scheme discussed for Figs. 9 and 10.

7.2.2 Multi-query

This section evaluates the system throughput when multiple continuous queries are registered to our prototype system, JsSpinner. Since enough comparison between smart and basic schemes have been presented in Sect. 7.2.1, this section focuses on evaluating the advantage of using the proposed gate operator when multiple continuous smart queries are registered

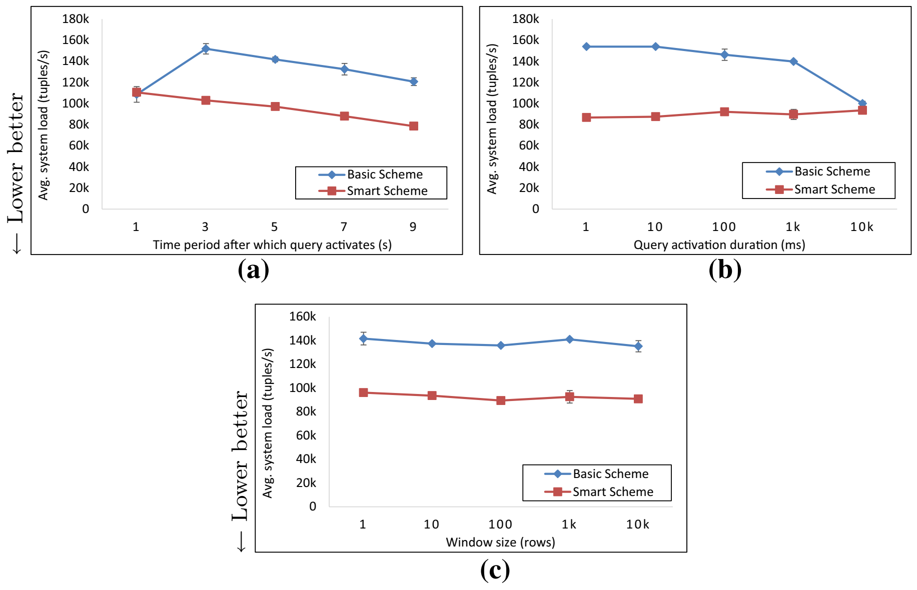


Fig. 9 Average system load evaluation on People Flow data stream. **a** Varying I_e ($W_e = 1$ s and $W_o = 10$ rows), **b** varying W_e ($I_e = 5$ s and $W_o = 10$ rows), **c** varying W_o ($I_e = 5$ s and $W_e = 1$ s)

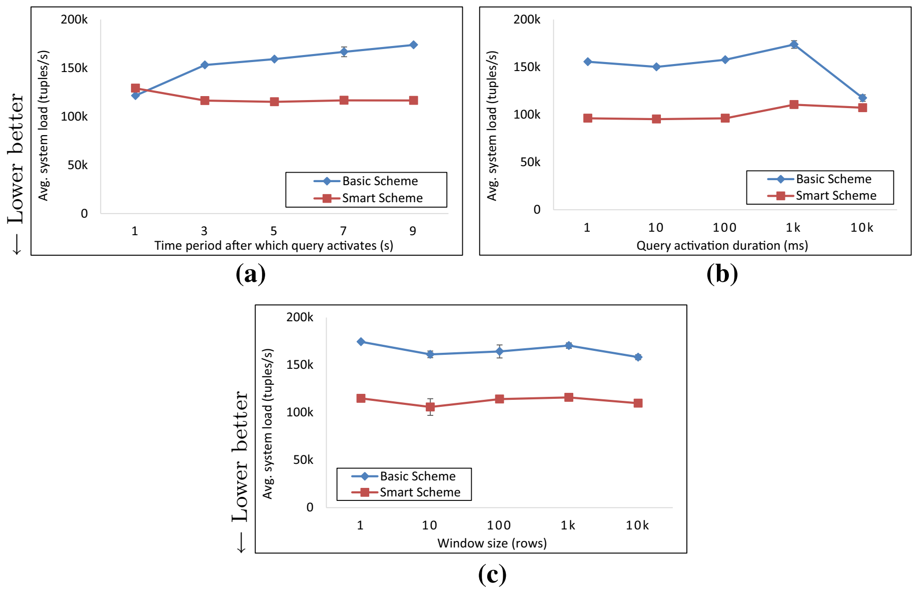


Fig. 10 Average system load evaluation on Tsukuba Mobility data stream. **a** Varying I_e ($W_e = 1$ s and $W_o = 10$ rows), **b** varying W_e ($I_e = 5$ s and $W_o = 10$ rows), **c** varying W_o ($I_e = 5$ s and $W_e = 1$ s)

to our SPE. To recall, when multiple continuous queries are registered to our prototype system, smart windows corresponding to same data sources are merged and gate operators are appended to the merged query plan tree to guarantee smart query processing. We divide the

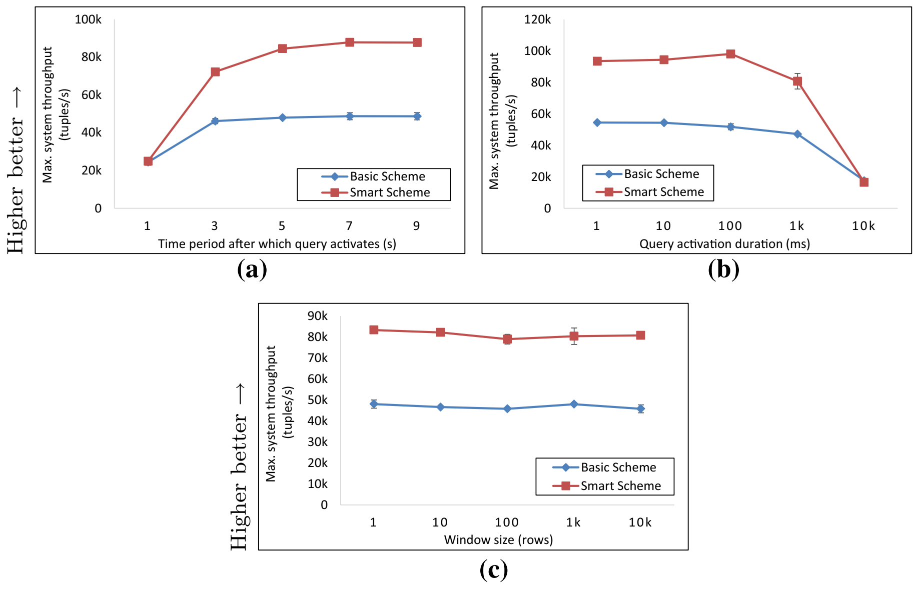


Fig. 11 Maximum system throughput evaluation on People Flow data stream. **a** Varying I_e ($W_e = 1$ s and $W_o = 10$ rows), **b** varying W_e ($I_e = 5$ s and $W_o = 10$ rows), **c** varying W_o ($I_e = 5$ s and $W_e = 1$ s)

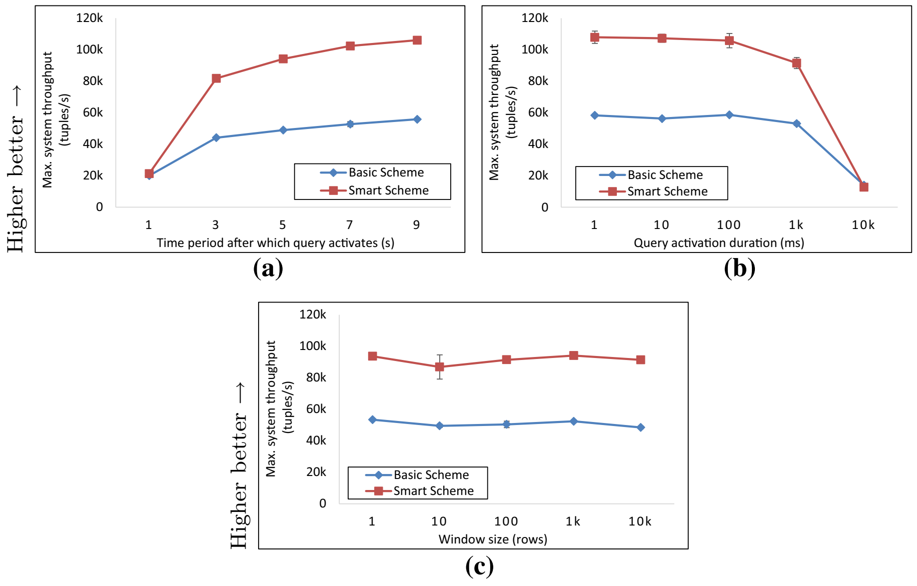


Fig. 12 Maximum system throughput evaluation on Tsukuba Mobility data stream. **a** Varying I_e ($W_e = 1$ s and $W_o = 10$ rows), **b** varying W_e ($I_e = 5$ s $W_o = 10$ rows), **c** varying W_o ($I_e = 5$ s and $W_e = 1$ s)

evaluation in this section into the following: (1) Advantage of the gate operator, and (2) Effectiveness of merging window operators.

```
Select  S1.B,  S2.C
From  S1[Range  τ],  S2[Rows  n]
Where  S1.A=S2.A

Select  S2.C,  S3.D
From  S2[Rows  n],  S3[Range  τ]
Where  S2.A=S3.A
⋮
Select  S(2N-1).X,  S(2N).Y
From  S(2N-1)[Range  τ],  S(2N)[Rows  n]
Where  S(2N-1).A=S(2N).A
```

Query 4 Multiple queries

Advantage of the Gate operator To prove the effectiveness of the gate operator, we merged Query 3 with the one with S1 replaced by S3. The two queries share S0 but have different event streams S1 and S3, respectively. When the two queries are merged, gate operators are appended to the shared query plan, one for each query. For the experiments, the default parameter values are used except for the parameters $I_e(S1)$ and $I_e(S3)$, which are set to 5 and 3 s, respectively. Figure 13 compares the maximum system throughput with and without gate operators for the above query setting. From the figure it is clear that the use of gate operators results in higher system throughput. This is due to the fact that the gate operator buffers the tuples during query inactive duration and directly deletes the expired tuples from its synopsis without sending corresponding "−" tuples downstream just like the smart window resulting in reduction in the processing load of the downstream operators.

Effectiveness of merging window operators To understand the advantage of merging similar window operators, we evaluated the maximum system throughput by varying the parameter I_e for the following four cases: (1) Sharing both time-based and tuple-based windows, (2) Sharing only time-based window, (3) Sharing only tuple-based window, (4) No sharing. Although other operators can also be merged and can further improve the system

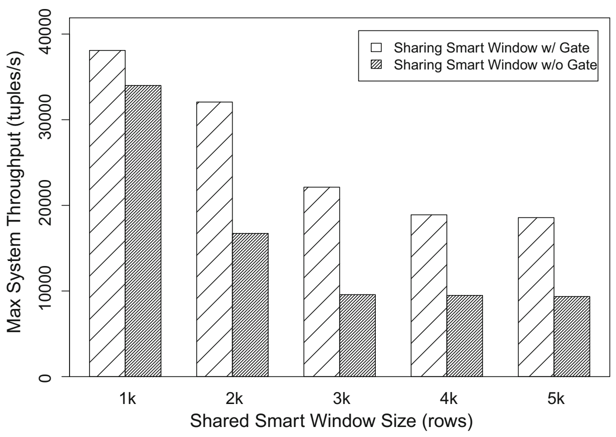


Fig. 13 Effect of gate operator (#Queries: 2, $I_e(S1) = 5$, $I_e(S3) = 3$)

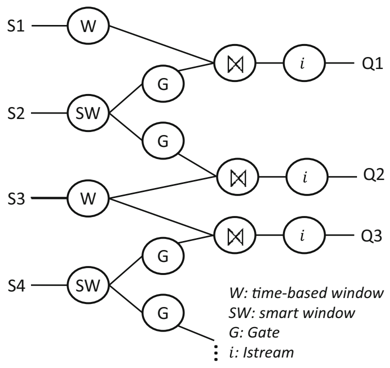


Fig. 14 Multi-query plan

throughput, here we focus only on merging of window operators and the addition of gate operator.

For the experiments, we use a set of queries listed in Query 4, whose query plan is shown in Fig. 14. The first query is similar to Query 1. The first query in multi-query experiments is $Q1$ with stream input sources $S1$ and $S2$, where $S1$ is an event stream and $S2$ is an ordinary stream. The second query $Q2$ is again with two stream input sources $S2$ and $S3$, where $S2$ is an ordinary common input stream and $S3$ is an event stream. Note that $S2$ is shared by Queries $Q1$ and $Q2$. Assuming even number of queries, the set of query used in the experiments is shown in Query 4.

The multi-query experiments are performed by varying I_e for 10, 20, 30, 40 and 50 queries sharing the common input stream sources (leaf and window operators). From Fig. 15a–e it is evident that merging query plans can significantly improve the system throughput. In most of the cases in the figures, improvement in throughput reaches to about 2.5 times when sharing only window operators i.e., tuple-based and time-based (smart) windows. The throughput is expected to increase far more than this for complex queries sharing multiple operators.

8 Discussion: smart scheme and multi-query optimization costs and benefits

Experiments prove that the proposed smart scheme can outperform the basic scheme and the other state-of-the-art SPEs in relative maximum system throughput and is capable of reducing overall system load. Implementation of smart scheme and its multi-query optimization require the following two operators in addition to the existing CQL operators: (1) Smart Window and (2) Gate.

Smart window is just like a CQL row-based window operator, however its synopsis is divided into two parts i.e., suspended and output. It does not require any additional memory space; however a negligible additional processing cost is needed to hash the incoming stream tuples to correct window synopsis part based on the query state, i.e., active or inactive. This division of smart window operator's synopsis on the other hand saves a lot of computation time and memory space by buffering the stream tuples during the query inactive duration. Furthermore, a small processing overhead is incurred once the query changes its state from inactive to active, where all the buffered tuples need to be sent and processed by the down-

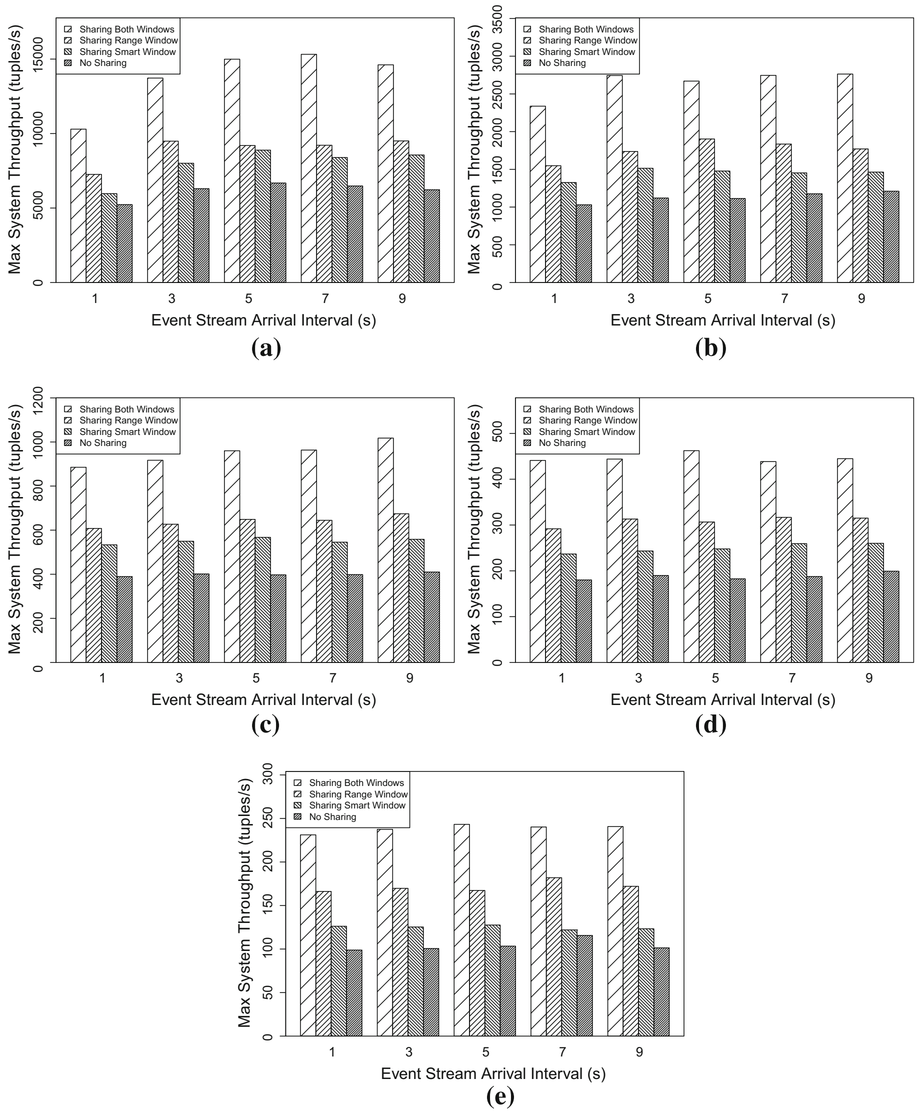


Fig. 15 Merging query plans ($R_o = 100$ k tuples/s, $W_e = 1$ s, $W_o = 10$ rows). **a** #Queries:10, **b** #Queries:20, **c** #Queries:30, **d** #Queries:40, **e** #Queries:50

stream operators. The overhead depends on the smart window size. Smaller the smart window size, smaller the overhead is.

When multiple continuous event-driven queries are registered to our prototype system, their query plan trees are merged, whenever possible, to reduce computation and memory cost. Since smart windows are different from ordinary windows, when two queries each with a smart window are merged, a gate operator is placed one for each smart window. This seems to be a computation and memory overhead in the first place as two operators are replaced by three (i.e., two smart windows are replaced by a smart window and two gate operators as can be observed from Fig. 3), however gate operators buffer incoming stream tuples during the query inactive duration just like smart window operators, reducing the computational and memory overhead from the downstream operators. This results in reduced system load and improved throughput. On the other hand if both the merged queries remain active all the time (the case of continuous event generation), there is a small computation and memory overhead in the merged query plan depending on the window size.

9 Conclusion and future work

In this work, we have proposed a smart scheme for event-driven stream processing and a multi-query optimization for it. The proposed smart scheme reduces system load resulting in increased system throughput by the use of the proposed smart window. Furthermore, multi-query optimization proposed in this work enables execution of multiple smart queries simultaneously by sharing their query plans, whenever possible, to save computational cost. Multi-query optimization makes use of the proposed gate operator to achieve this. In addition, we have developed a prototype SPE, JsSpinner, implementing the proposed smart event-driven stream processing scheme and its multi-query optimization. In order to show the effectiveness of the proposed smart scheme and its multi-query optimization, detailed experiments are performed on real and synthetic data streams. The experiments prove that the proposed smart scheme is capable of reducing system load which results in increased system throughput. In addition, the multi-query experiments prove that the proposed gate operator can effectively merge multiple smart queries by retaining the advantage of smart windows. In the future, we have plans to work on sophisticated query optimization techniques incorporating the proposed smart scheme and complex event-driven distributed data processing.

Acknowledgements This research was partly supported by the program "Research and Development on Real World Big Data Integration and Analysis" of the Ministry of Education, Culture, Sports, Science and Technology (MEXT) and RIKEN, Japan.

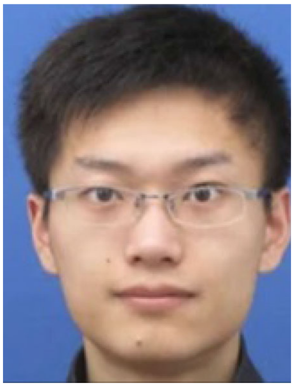
References

1. Gartner IT Glossary (2016). http://www.gartner.com/it-glossary/big-data/. Accessed 17 Sept 2016
2. Abadi DJ, Carney D, Cetintemel U, Cherniack M, Convey C, Lee S, Stonebraker M, Tatbul N, Zdonik S (2003) Aurora: a new model and architecture for data stream management. VLDB J 12(2):120–139
3. Abadi DJ, Ahmad Y, Balazinska M, Cherniack M, Hwang J hyon, Lindner W, Maskey AS, Rasin E, Ryvkina E, Tatbul N, Xing Y, Zdonik S (2005) The design of the borealis stream processing engine. In: Proceedings of CIDR, pp 277–289
4. Apache Storm project (2017). https://storm.apache.org/. Accessed 21 Jan 2017
5. Arasu A, Babcock B, Babu S, Cieslewicz J, Datar M, Ito K, Motwani R, Srivastava U, Widom J (2003) STREAM: The Stanford data stream management system. Tech. Report, Stanford InfoLab, IEEE Data Engg. Bulletin 26(1)
6. Wu Y, Tan K (2015) ChronoStream: elastic stateful stream computation in the cloud. In: Proceedings of the ICDE, pp 723–734
7. Cetintemel U, Du J, Kraska T, Madden S, Maier D, Meehan J, Pavlo A, Stonebraker M, Sutherland E, Tatbul N, Tufte K, Wang H, Zdonik SB (2014) S-store: a streaming NewSQL system for big velocity applications. In: Proceedings of the VLDB, pp 1633–1636
8. Chandramouli B, Goldstein J, Barnett M, DeLine R, Fisher D, Platt JC, Terwilliger JF, Wernsing J (2014) Trill: a high-performance incremental query processor for diverse analytics. In: Proceedings of the VLDB, pp 401–412
9. Wang D, Rundensteiner EA, Ellison RT (2011) Active complex event processing over event streams. Proc VLDB Endow 4(10):634–645

10. Wu E, Diao Y, Rizvi S (2006) High-performance complex event processing over streams. In: Proceedings of the ACM SIGMOD, pp 407–418
11. Brenna L, Demers A, Gehrke J, Hong M, Ossher J, Panda B, Riedewald M, Thatte M, White W (2007) Cayuga: a high-performance event processing engine. In: Proceedings of ACM SIGMOD, pp 1100–1102
12. Apache Spark Streaming (2017). https://spark.apache.org/streaming/. Accessed 21 Jan 2017
13. Roy P, Seshadri S, Sudarshan S, Bhobe S (2000) Efficient and extensible algorithms for multi query optimization. In: Proceedings of the SIGMOD, pp 249–260
14. Madden S, Shah M, Hellerstein JM, Raman V (2002) Continuously adaptive continuous queries over streams. In: Proceedings of the SIGMOD, pp 49–60
15. Chandrasekaran S, Franklin MJ (2003) PSoup: a system for streaming queries over streaming data. VLDB J 12(2):140–156
16. Beyer Kevin S, Ercegovac Vuk, Gemulla Rainer, Eltabakh Mohamed, Balmin Andrey (2011) Jaql: a scripting language for large scale semistructured data analysis. Proc VLDB Endow 4(12):1272–1283
17. The JSON Data Interchange Format (2013) Standard ECMA-404. ECMA International, Geneva
18. Shaikh SA, Watanabe Y, Wang Y, Kitagawa H (2016) Smart query execution for event-driven stream processing. In: Proceedings of 2nd IEEE international conference on multimedia big data, pp 97–104
19. Terry D, Goldberg D, Nichols D, Oki B (1992) Continuous queries over append-only databases. SIGMOD Rec 21(2):321–330
20. Zaharia M, Das T, Li H, Shenker S, Stoica I (2012) Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: Proceedings, HotCloud
21. Motwani R, Widom J, Arasu A, Babcock B, Babu S, Datar M, Manku G, Olston C, Rosenstein J, Varma R (2003) Query processing, resource management, and approximation in a data stream management system. In: Proceedings of CIDR, pp 245–256
22. Chandrasekaran S, Cooper O, Deshpande A, Franklin MJ, Hellerstein JM, Hong W, Krishnamurthy S, Madden SR , Reiss F, Shah MA (2003) Telegraphcq: continuous dataflow processing. In: Proceedings of ACM SIGMOD, pp 668–668
23. Neumeyer L, Robbins B, Nair A, Kesari A (2010) S4: distributed stream computing platform. In: Proceedings of the ICDMW, pp 170–177
24. Jaewoo K, Naughton JF, Viglas SD (2003) Evaluating window joins over unbounded streams. In: Proceedings of ICDE, pp 341–352
25. Srivastava U, Widom J (2004) Memory-limited execution of windowed stream joins. In: Proceedings of very large database (PVLDB)
26. Gedik B, Wu KL, Yu PS, Liu L (2007) GrubJoin: an adaptive, multi-way, windowed stream join with time corr.-aware CPU load shedding. IEEE TKDE 19(10):1363–1380
27. Arasu A, Babu S, Widom J (2006) The cql continuous query language: semantic foundations and query execution. VLDB J 15(2):121–142
28. Viglas SD, Naughton JF (2002) Rate-based query optimization for streaming information sources. In: Proceedings of the SIGMOD, pp 37–48
29. Ayad AM, Naughton JF (2004) Static optimization of conjunctive queries with sliding windows over infinite streams. In: Proceedings of the SIGMOD, pp 419–430
30. Babu S, Motwani R, Munagala K, Nishizawa I, Widom J (2004) Adaptive ordering of pipelined stream filters. In: Proceedings of the SIGMOD
31. Avnur R, Hellerstein JM (2000) Eddies: continuously adaptive query processing. In: Proceedings of the SIGMOD, pp 261–272
32. Chen J, DeWitt DJ, Tian F, Wang Y (2000) NiagaraCQ: a scalable continuous query system for Internet databases. In: Proceedings of the SIGMOD, pp 379–390
33. Arasu A, Widom J (2004) Resource sharing in continuous sliding-window aggregates. In: Proceedings of the VLDB, pp 336–347
34. Babu S, Munagala K, Widom J, Motwani R (2005) Adaptive caching for continuous queries. In: Proceedings, ICDE
35. ANSI/ISO/IEC International Standard (1999) Database language SQL: foundation (SQL/Foundation)
36. Tokyo Metropolitan People Flow Data Stream (2016). https://joras.csis.u-tokyo.ac.jp/. Accessed 15 May 2016

Salman Ahmed Shaikh received the B.E. degree in computer systems, the M.E. degree in communication systems and networks, from the Mehran University of Engineering and Technology, Pakistan, and the Ph.D. degree in computer science, from the University of Tsukuba, Japan, in 2005, 2008 and 2014, respectively. He is currently a post-doc researcher at the Center for Computational Sciences, University of Tsukuba, Japan. His research interests include stream processing, big data manipulation, transaction processing, uncertain data processing and data mining . He is a member of the Database Society of Japan (DBSJ), International Association of Computer Science and Information Technology (IACSIT) and Pakistan Engineering Council (PEC).

Yousuke Watanabe received the B.S, M.E. and Dr. E degrees from University of Tsukuba in 2001, 2003 and 2006. He is currently a designated associate professor at Institute of Innovation for Future Society, Nagoya University. His research issues are Information Integration, Data Stream Processing, Database Systems and Data Mining. He is a member of ACM.



Yan Wang received the B.S degree from Zhejiang University, China in 2012 and the M.S. degree from University of Tsukuba, Japan in 2015. He is currently working as a Software Engineer at Works Applications Co. Ltd., Japan. His interests include big data processing, distributed processing, system design, elegant coding and automatic scripting.

Hiroyuki Kitagawa received the B.Sc. degree in physics and the M.Sc. and Dr.Sc. degrees in computer science, all from the University of Tokyo, in 1978, 1980, and 1987, respectively. He is currently a full professor at Center for Computational Sciences, University of Tsukuba. His research interests include databases, data integration, stream processing, data mining, social media mining, information retrieval, and scientific databases. He is an IEICE Fellow, an IPSJ Fellow, and an Associate Member of the Science Council of Japan. He served as President of the Database Society of Japan from 2014 to 2016, Chairperson of the IEICE Special Interest Group on Data Engineering from 1999 to 2001, Chairperson of ACM SIGMOD Japan Chapter from 2003 to 2007. He is a member of ACM, IEEE, and JSSST.