

# Sharon: Shared Online Event Sequence Aggregation

Technical Report  
January 15, 2018

Olga Poppe\*, Allison Rozet\*, Chuan Lei\*\*, Elke A. Rundensteiner\*, and David Maier\*\*\*

\*Worcester Polytechnic Institute, Worcester, MA 01609

\*\*IBM Research, Almaden, 650 Harry Rd, San Jose, CA 95120

\*\*\*Portland State University, 1825 SW Broadway, Portland, OR 97201

\*opoppe|amrozet|rundenst@wpi.edu, \*\*chuan.lei@ibm.com, \*\*\*maier@pdx.edu

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Sharon Approach Overview</b>	<b>6</b>
2.1	Sharon Data and Query Model . . . . .	6
2.2	Sharon Framework . . . . .	7
<b>3</b>	<b>Sharing Benefit Model</b>	<b>7</b>
3.1	Sharing Candidate . . . . .	8
3.2	Non-Shared Method . . . . .	8
3.3	Shared Method . . . . .	10
3.4	Benefit of a Sharing Candidate . . . . .	11
<b>4</b>	<b>Sharing Conflict Modeling</b>	<b>11</b>
<b>5</b>	<b>Sharing Candidate Pruning</b>	<b>13</b>
<b>6</b>	<b>Sharing Plan Finder</b>	<b>15</b>
<b>7</b>	<b>Extensions of the Sharon Approach</b>	<b>19</b>
7.1	Sharing Conflict Resolution . . . . .	19
7.2	Different Predicates, Grouping, and Windows . . . . .	21
7.3	Multiple Occurrences of an Event Type in a Pattern . . . . .	22
7.4	Dynamic Workloads . . . . .	22
<b>8</b>	<b>Performance Evaluation</b>	<b>22</b>
8.1	Experimental Setup . . . . .	22
8.2	Sharon Executor versus State-of-the-Art Approaches . . . . .	24
8.3	Sharon Optimizer . . . . .	25
<b>9</b>	<b>Related Work</b>	<b>27</b>
<b>10</b>	<b>Conclusions and Future Work</b>	<b>28</b>
<b>A</b>	<b>Sharable Pattern Detection</b>	<b>30</b>
<b>B</b>	<b>GWMIN Algorithm</b>	<b>32</b>

## Abstract

Streaming systems evaluate massive workloads of event sequence aggregation queries. State-of-the-art approaches suffer from long delays caused by not sharing intermediate results of similar queries and by constructing event sequences prior to their aggregation. To overcome these limitations, our Shared Online Event Sequence Aggregation (SHARON) approach shares intermediate aggregates among multiple queries while avoiding the expensive construction of event sequences. Our SHARON optimizer faces two challenges. One, a sharing decision is not always beneficial. Two, a sharing decision may exclude other sharing opportunities. To guide our SHARON optimizer, we compactly encode sharing candidates, their benefits, and conflicts among candidates into the SHARON graph. Based on the graph, we map our problem of finding an optimal sharing plan to the Maximum Weight Independent Set (MWIS) problem. We then use the guaranteed weight of a greedy algorithm for the MWIS problem to prune the search of our sharing plan finder without sacrificing its optimality. The SHARON optimizer is shown to produce sharing plans that achieve up to an 18-fold speed-up compared to state-of-the-art approaches.

## 1 Introduction

Complex Event Processing (CEP) is a prominent technology for supporting time-critical streaming applications ranging from public transport to e-commerce. CEP systems continuously evaluate massive query workloads against high-rate event streams to detect event sequences of interest, such as vehicle trajectories and purchase patterns. Aggregation functions are applied to these sequences to provide summarized insights, such as the number of trips on a certain route to predict traffic jams or the number of items purchased after buying another item for targeted advertisement. CEP applications must react to critical changes of these aggregates in real time to compute best routes or increase profit [4, 28, 29].

**Motivating Examples.** We now describe two time-critical multi-query event sequence aggregation scenarios.

- **Urban transportation services.** With the growing popularity of ridesharing services such as Uber and Lyft, their systems face multiple challenges including real-time analysis of vehicle trajectories, geospatial prediction, and alerting. These systems evaluate query workloads against high-rate streams of drivers’ position reports and riders’ requests to infer the current supply and demand situation on each route. They incorporate traffic conditions to compute the best route for each trip. They instantaneously react to critical changes to prevent waste of time, reduce costs and pollution, and increase riders’ satisfaction and drivers’ profit. With thousands of drivers and over 150 requests per minute in New York City [3, 2], real-time traffic analytics and ride management is a challenging task.

Queries  $q_1$ – $q_7$  in Figure 1 compute the count of trips on a route as a measure of route popularity. They consume a stream of vehicle-position reports. Each report carries a time stamp in seconds, a car identifier and its position. Here, event type corresponds to a vehicle position. For example, a vehicle on Main Street sends a position report of type *MainSt*. Each trip corresponds to a sequence of position reports from the same vehicle (as required by the predicate *[vehicle]*) during a 10-minutes

$q_1$ : RETURN COUNT(*) PATTERN OakSt, MainSt, StateSt WHERE [vehicle] WITHIN 10 min SLIDE 1 min	
$q_2$ : OakSt, MainSt, WestSt	$q_5$ : MainSt, StateSt
$q_3$ : LindenSt, ParkAve, OakSt, MainSt	$q_6$ : EastPark, ElmSt, ParkAve
$q_4$ : ParkAve, OakSt, MainSt, WestSt	$q_7$ : ElmSt, ParkAve, GreenHill

Figure 1: Traffic monitoring workload  $Q$ 

Pattern $p$	Queries $Q_p \subseteq Q$
$p_1 = (OakSt, MainSt)$	$q_1, q_2, q_3, q_4$
$p_2 = (ParkAve, OakSt)$	$q_3, q_4$
$p_3 = (ParkAve, OakSt, MainSt)$	$q_3, q_4$
$p_4 = (MainSt, WestSt)$	$q_2, q_4$
$p_5 = (OakSt, MainSt, WestSt)$	$q_2, q_4$
$p_6 = (MainSt, StateSt)$	$q_1, q_5$
$p_7 = (ElmSt, ParkAve)$	$q_6, q_7$

Table 1: Sharing candidates of the form  $(p, Q_p)$  in the workload  $Q$ 

long time window that slides every minute. The predicates and window parameters of  $q_2$ – $q_7$  are identical to  $q_1$  and thus are not shown for compactness. Table 1 summarizes the common patterns in this workload. For example, pattern  $p_1 = (OakSt, MainSt)$  appears in queries  $q_1$ – $q_4$ . Sharing the aggregation of common patterns among multiple similar queries is vital to speed up system responsiveness.

- *E-commerce* systems monitor customer click patterns to identify the purchase of which item increases the chance of buying another item. Such purchase dependencies between items serve as a foundation for prediction, planning, and targeted ads on an online shopping website.

Queries  $q_8$ – $q_{11}$  consume a stream of item purchases. Each event carries a time stamp in seconds, customer identifier, type of item, e.g., *Laptop*. The choice of a laptop often determines the laptop cases, adapters, keyboard protectors, etc. that may be purchased next ( $q_8$ – $q_{10}$ ). A laptop may even determine a customer’s phone preferences, e.g., Mac users are likely to choose an iPhone over other phones. The model of an iPhone, in turn, determines screen protectors for it ( $q_{11}$ ). Thus, queries  $q_8$ – $q_{11}$  compute the count of item sequences during a 20-minute time window that slides every minute. The pattern  $(Laptop, Case)$  appears in all four queries in this workload. The aggregation of such patterns could be shared among these queries to achieve prompt updates of the recommendation model according to dynamically changing user preferences.

**State-of-the-Art Approaches** can be divided into three groups (Figure 3):

- *Non-shared two-step approaches*, including Flink [1], SASE [29], Cayuga [10], and ZStream [22], evaluate each query independently from other queries in the workload. Furthermore, these approaches do not offer optimization strategies specific to event sequence aggregation queries. Without special optimization techniques, these approaches first construct event sequences and then aggregate them. Since the number of event sequences is polynomial in the number of events [29, 24],

---

```

q8: RETURN COUNT(*)
    PATTERN Laptop, Case, Adapter, Mouse
    WHERE [customer] WITHIN 20 min SLIDE 1 min

q9: Laptop, Case, KeyBoardProtector
q10: Monitor, Laptop, Case, Adapter
q11: Laptop, Case, Phone, ScreenProtector

```

---

Figure 2: Purchase monitoring workload

	Non-Shared	Shared
Two-step	<b>Flink, SASE, Cayuga, ZStream</b>	<b>SPASS, ECube</b>
	1. Event sequence construction 2. Event sequence aggregation	1. Event sequence construction 2. Event sequence aggregation
Online	<b>A-Seq, GRETA</b> Event sequence aggregation	<b>Sharon</b> Event sequence aggregation

Figure 3: Event sequence aggregation approaches

event sequence construction is an expensive step. Our experiments in Section 8 confirm that such a *non-shared two-step* approach implemented by the popular open-source streaming system Flink [1] does not terminate, even for low-rate streams of a few hundred events per second.

- **Shared two-step approaches** such as SPASS [25] and ECube [21] focus on *shared* event sequence construction, not on event sequence aggregation. If these approaches are applied to aggregate event sequences, they would construct all sequences prior to their aggregation. This event sequence construction step degrades system performance. Our experiments in Section 8 confirm that such a *shared two-step* approach implemented by SPASS [29] requires 41 minutes per window, even for low-rate streams of a few hundred events per second. Such long delays are not acceptable for time-critical applications that require a response within a few seconds [6].

- **Non-shared online approaches** such as A-Seq [24] and GRETA [23] compute event sequence aggregation *online*, i.e., without constructing the sequences. A-Seq incrementally maintains a set of aggregates for each pattern and instantaneously discards each event once it updates the aggregates. GRETA extends A-Seq to a broader class of queries and thus has higher computation costs. Neither of these approaches tackles the sharing optimization problem to determine which queries should share the aggregation of which patterns such that the latency of a workload is minimized – which is the focus of this paper. These approaches lack optimizers that can identify shared computations among multiple queries.

**Challenges.** We tackle the following open problems:

- **Online yet shared event sequence aggregation.** These two optimization techniques cannot be simply combined because they impose contradictory constraints on the underlying execution strategy. For example, if query  $q_4$  *shares* the aggregation results of patterns  $p_2$  and  $p_4$  with other queries (Table 1), the aggregates for  $p_2$  and  $p_4$  must be combined to form the final results for  $q_4$ . To ensure correctness, this result combination must be aware of the temporal order between sequences matched by  $p_2$  and  $p_4$  and their expiration. To analyze these temporal relationships, event

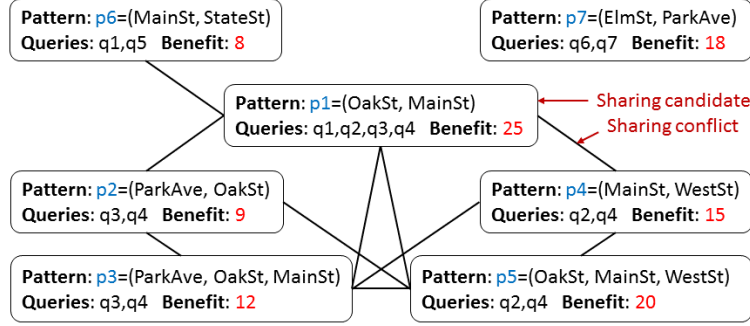


Figure 4: SHARON graph for the traffic use case (Figure 1)

sequences must be constructed. This requirement contradicts the key idea of the *online* approaches that avoid event sequence construction.

- **Benefit of sharing.** Sharing the aggregation computation for a pattern  $p$  by a set of queries  $Q_p$  containing  $p$  is not always beneficial, since this sharing may introduce considerable CPU overhead for combining shared intermediate aggregates to form the final results for each query in  $Q_p$ . Thus, an accurate sharing benefit model is required to assess the quality of a sharing plan.

- **Intractable sharing plan search space.** The search space for a high-quality sharing plan is exponential in the number of sharing candidates (Table 1). Since the event rate may fluctuate, the benefit of sharing a pattern may change over time. To achieve a high sharing benefit, the sharing plan may have to be dynamically adjusted. Hence, an effective yet efficient optimization algorithm for sharing plan selection is required.

**Our SHARON Approach.** We propose the Shared Online Event Sequence Aggregation (SHARON) optimization techniques to tackle these challenges. Since sharing a pattern  $p$  by a set of queries  $Q_p$  is not always beneficial, we develop a *sharing benefit model* to assess the quality of a sharing candidate  $(p, Q_p)$ . The model compares the gain of sharing  $p$  among queries  $Q_p$  to the overhead of combining shared aggregates of  $p$  to form the final results for each query in  $Q_p$ . Non-beneficial candidates are pruned. Since a decision to share a pattern may prevent the sharing of another pattern by the same query, we define the notion of *sharing conflicts* among sharing candidates. We compactly encode sharing candidates as vertices and conflicts among these candidates as edges of the SHARON graph (Figure 4). Each vertex is assigned a weight that corresponds to the benefit of sharing the respective candidate. Based on the graph, we map our Multi-query Event Sequence Aggregation problem to the Maximum Weight Independent Set (MWIS) problem. We then utilize the guaranteed minimal weight of the approximate algorithm GWMIN [26] for MWIS to prune conflict-ridden candidates. Since conflict-free candidates always belong to an optimal sharing plan, they can also be excluded from the search early on. Based on the reduced graph, our sharing plan finder further prunes sharing plans with conflicts and returns an optimal plan (i.e., plan with minimal estimated latency) to guide our executor at runtime. In summary, SHARON seamlessly combines two optimization strategies into one integrated solution. Namely, it *shares* sequence aggregation among multiple queries, while computing sequence aggregation *online*.

**Contributions.** Our key innovations are the following.

1) We design the *sharing benefit model* to assess the quality of a sharing candidate. Non-beneficial candidates are pruned.

2) We identify *sharing conflicts* among candidates and encode candidates, their benefits, and conflicts among them into the SHARON *graph*.

3) We map our Multi-query Event Sequence Aggregation problem to the Maximum Weight Independent Set (MWIS) problem and utilize the guaranteed weight of the approximate algorithm for MWIS to prune conflict-ridden candidates.

4) Based on the reduced SHARON graph, we introduce the *sharing plan finder* that prunes sharing plans with conflicts and returns an optimal sharing plan.

5) Our experiments using real data sets [3, 6] demonstrate that sharing plans produced by the SHARON optimizer achieve an 18-fold speed-up and use two orders of magnitude less memory compared to Flink [1], A-Seq [24], and SPASS [25].

**Outline.** Section 2 provides an overview of our approach. We define the sharing benefit model and sharing conflicts in Sections 3–4. We reduce the search space and design the sharing plan finder in Sections 5–6. We discuss extensions of our core approach in Section 7. Experiments are described in Section 8. Section 9 covers related work, while Section 10 concludes the paper and describes future work.

## 2 Sharon Approach Overview

### 2.1 Sharon Data and Query Model

*Time* is represented by a linearly ordered set of time points  $(\mathbb{T}, \leq)$ , where  $\mathbb{T} \subseteq \mathbb{Z}^{\geq}$  (non-negative integers). An *event* is a message indicating that something of interest to the application happened in the real world. An event  $e$  has a *time stamp*  $e.time \in \mathbb{T}$  assigned by the event source. An event  $e$  belongs to a particular *event type*  $E$ , denoted  $e.type = E$  and described by a *schema* that specifies the set of *event attributes* and the domains of their values. Events are sent by event producers (e.g., vehicles) on an input *event stream*  $I$ . An event consumer (e.g., carpool system) monitors the stream with a workload of queries that detect and aggregate event sequences. We borrow the query syntax and semantics from SASE [4].

**Definition 1. (Event Sequence Pattern)** Given event types  $E_1, \dots, E_l$ , an *event sequence pattern* has the form  $P = (E_1 \dots E_l)$  where  $l \in \mathbb{N}$ ,  $l \geq 1$ , is the length of  $P$ .

Given a stream  $I$ , an *event sequence*  $s = (e_1 \dots e_l)$  is a match of a pattern  $P$  if  $\forall i, j \in \mathbb{N}, 1 \leq i < j \leq l$ , the following conditions hold:  $e_i, e_j \in I$ ,  $e_i.type = E_i$ ,  $e_j.type = E_j$ , and  $e_i.time < e_j.time$ . Event  $e_1$  is called a *START* event,  $e_l$  is an *END* event, and  $\{e_2, \dots, e_{l-1}\}$  are *MID* events of  $s$ .

**Definition 2. (Event Sequence Aggregation Query)** An *event sequence aggregation query*  $q$  consists of five clauses:

- Aggregation result specification (*RETURN* clause),
- Event sequence pattern  $P$  (*PATTERN* clause),
- Predicates  $\theta$  (optional *WHERE* clause),

- Grouping  $G$  (optional *GROUP-BY* clause), and
- Window  $w$  (*WITHIN* and *SLIDE* clause).

A query  $q$  requires that all events in an event sequence  $s$  are matched by the pattern  $P$  (Definition 1), satisfy the predicates  $\theta$ , have the same values of the grouping attributes  $G$ , and are within one window  $w$ . Event sequences matched by  $q$  are grouped by the values of  $G$ . The aggregation function of  $q$  is applied to each group and a result is returned per group and per window. We focus on distributive (such as *COUNT*, *MIN*, *MAX*, *SUM*) and algebraic aggregation functions (such as *AVG*), since they can be computed incrementally [13].

Let  $e$  be an event of type  $E$  and  $attr$  be an attribute of  $e$ .  $COUNT(*)$  returns the number of all sequences per group, while  $COUNT(E)$  computes the number of all events  $e$  in all sequences per group.  $MIN(E.attr)$  ( $MAX(E.attr)$ ) returns the minimal (maximal) value of  $attr$  for all events  $e$  in all sequences per group.  $SUM(E.attr)$  calculates the summation of the value of  $attr$  of all events  $e$  in all sequences per group. Lastly,  $AVG(E.attr)$  is computed as  $SUM(E.attr)$  divided by  $COUNT(E)$  per group.

**Assumptions.** To initially focus the discussion, we assume that: (1) A pattern  $p$  is shared among all queries containing  $p$ . (2) All queries have the same predicates, grouping, and windows. (3) An event type appears at most once in a pattern. (4) The workload is static. We sketch extensions of our approach to relax these assumptions in Section 7.

## 2.2 Sharon Framework

We target *time-critical* applications that require aggregation results within a few seconds (Section 1). Given a workload  $Q$  and a stream  $I$ , the **Multi-query Event Sequence Aggregation (MESA) Problem** is to determine which queries share the aggregation of which patterns (i.e., a sharing plan  $\mathcal{P}$ ) such that the *latency of evaluating the workload  $Q$  according to the plan  $\mathcal{P}$  against the stream  $I$  is minimal*.

To solve this problem, our SHARON framework deploys the following components (Figure 5). Given a workload  $Q$ , our **Static Optimizer** finds an optimal sharing plan at compile time. To this end, it identifies sharing candidates of the form  $(p, Q_p)$  where  $p$  is a pattern that could potentially be shared by a set of queries  $Q_p \subseteq Q$ . It then estimates the benefit of each candidate  $(p, Q_p)$  (Section 3), determines sharing conflicts among these candidates, and compactly encodes all candidates, their benefits and conflicts into a SHARON graph (Section 4). Based on the graph, the optimizer prunes large portions of the search space (Section 5) and returns an optimal sharing plan (Section 6). Based on this plan, our **Runtime Executor** computes the aggregation results for each shared pattern and then combines these shared aggregations to obtain the final results for each query in the workload  $Q$  (Section 3).

## 3 Sharing Benefit Model

Our optimizer first identifies sharing candidates in a workload (Section 3.1). It then decides whether to apply the *Non-Shared* or the *Shared method* to each query (Sections 3.2–3.3). Both methods are borrowed from A-Seq [24]. Lastly, it estimates the benefit of sharing each candidate (Section 3.4).



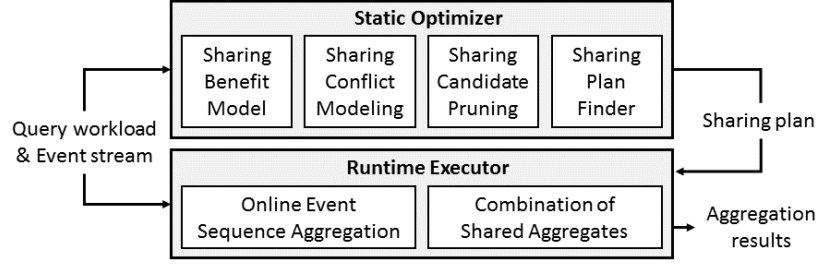


Figure 5: SHARON framework

### 3.1 Sharing Candidate

First, our optimizer identifies those patterns that could potentially be shared by queries in a given workload.

**Definition 3. (Sharable Pattern, Sharing Candidate)** Let  $Q$  be a workload and  $p$  be a pattern that appears in queries  $Q_p \subseteq Q$ . The pattern  $p$  is sharable in  $Q$  if  $p.length > 1$  and  $|Q_p| > 1$ . A sharable pattern  $p$  and queries  $Q_p$  constitute a sharing candidate, denoted as  $(p, Q_p)$ .

Existing pattern mining approaches can detect sharable patterns. Due to space constraints, they are described in Appendix A.

The pattern  $P^i$  of a query  $q_i \in Q_p$  consists of three sub-patterns, namely,  $prefix^i$ ,  $p$ , and  $suffix^i$  defined below.

**Definition 4. (Prefix and Suffix of a Sharable Pattern in a Query)** Let  $P^i = (E_1^i \dots E_n^i)$  be the pattern of a query  $q_i$  and  $p = (E_m^i \dots E_{m+l}^i)$  be a sharable pattern that appears in  $q_i$  where  $m, l, n \in \mathbb{N}$ ,  $1 \leq m$ , and  $m + l \leq n$ . Then  $prefix^i = (E_1^i \dots E_{m-1}^i)$  is called the prefix and  $suffix^i = (E_{m+l+1}^i \dots E_n^i)$  is called the suffix of  $p$  in  $q_i$ .

### 3.2 Non-Shared Method

While A-Seq [24] considers grouping, predicates, negation, and various aggregation functions, we now sketch only its key ideas, namely, online event sequence aggregation and event sequence expiration. We use event sequence count as an example, i.e., COUNT(\*) (Definition 2).

**Online Event Sequence Aggregation** A-Seq computes the count of event sequences online, i.e., without constructing and storing these event sequences. To this end, it maintains a count for each prefix of a pattern. The count of a prefix of length  $j$  is incrementally computed based on its previous value and the new value of the count of the prefix of length  $j - 1$ .

**Example 1.** Let an event be described by its type and time stamp, e.g.,  $a_1$  is an event of type  $A$  with time stamp 1. In Figure 6(a), we count event sequences matched by the pattern  $(A, B)$ , denoted  $count(A, B)$ . A count is maintained for each prefix of the pattern, i.e., for  $(A)$  and  $(A, B)$ . The value of  $count(A, B)$  is updated every time a  $b$  arrives by summing  $count(A)$  and  $count(A, B)$ . For example, when  $b_4$  arrives, it is appended to each previously matched  $a$  to form new sequences  $(a_1, b_4)$  and  $(a_2, b_4)$ . The number of new

Counts	Event stream				
	a1	b2	a3	b4	b5
$count(A)$	1		2		
$count(A, B)$		1		3	5

└── Window w1 ──┘

(a) Online sequence aggregation

Counts	Event stream				
	a1	b2	a3	b4	b5
$count(a1, B)$		1		2	
$count(a2, B)$				1	2
$count(A, B)$		1		3	2

└── Window w1 ──┘  
└── Window w2 ──┘

(b) Event sequence expiration

**Figure 6:** Non-Shared method

sequences is  $count(A) = 2$ . In addition, the previously formed sequence  $(a_1, b_2)$  is kept. The number of previous sequences is  $count(A, B) = 1$ . Thus,  $count(A, B)$  is updated to 3.

**Event Sequence Expiration.** Due to the sliding window semantics of our queries (Definition 2), event sequences expire over time. To avoid the re-computation of all affected aggregates, we observe that a START event of a sequence (Definition 1) expires sooner than any other event in it. Thus, we maintain the aggregates per each matched START event. When a new event arrives, only the counts of not-expired START events are updated. When an END event  $e$  arrives, it updates the final counts for all windows that  $e$  falls into.

**Example 2.** In Figure 6(b), assume a window of length four slides by one. A count is now maintained per each matched  $a$ . When  $b_5$  arrives,  $a_1$  is expired,  $count(a_1, B)$  is ignored,  $count(a_2, B)$  and  $count(A, B)$  are updated for window  $w_2$ .

**Data Structures.** Our SHARON Executor maintains a hash table that maps a pattern to its count. Thus, we can access and update a count in constant time.

**Time Complexity.** The query  $q_i$  processes each event that it matches. The rate of matched events is computed as the sum of rates of all event types in the pattern  $P^i$  of  $q_i$  (Definition 4):

$$Rate(P^i) = \sum_{j=1}^n Rate(E_j^i) \quad (1)$$

Since counts are maintained per START event and an event type appears at most once in a pattern, each matched event updates one count per each not-expired START event. There are  $Rate(E_1^i)$  START events. In summary, the time complexity of processing the query  $q_i$  by the Non-Shared method is:

$$NonShared(p, q_i) = Rate(E_1^i) \times Rate(P^i) \quad (2)$$

For the set of queries  $Q_p$ , the time complexity corresponds to the summation of the time complexity for each query  $q_i$ .

$$NonShared(p, Q_p) = \sum_{q_i \in Q_p} NonShared(p, q_i) \quad (3)$$

Counts	Event stream								
	a1	b2	a3	c3	b4	b5	d5	c7	d8
$count(A, B)$		1			3	5			
$count(c3, D)$							1		2
$count(c7, D)$									1
$count(A, B, C, D)$							1		7

Window w1

Figure 7: Shared method

### 3.3 Shared Method

Let  $(p, Q_p)$  be a sharing candidate (Definition 3). Let  $prefix^i$  and  $suffix^i$  be the prefix and the suffix of  $p$  in a query  $q_i \in Q_p$  (Definition 4). The aggregates for  $prefix^i$ ,  $p$ , and  $suffix^i$  are combined to obtain the aggregate for  $q_i$ . Due to event sequence semantics, the sequences matched by  $prefix^i$  must appear before the sequences matched by  $p$  which in turn must appear before the sequences matched by  $suffix^i$  in the stream. To this end, the executor performs two steps:

(1) **Count computation.** Counts are maintained per each START event of  $prefix^i$ ,  $p$ , and  $suffix^i$  (Section 3.2).

(2) **Count combination.** The count of  $prefix^i$  is multiplied with the count for each START event of  $p$ . The resulting counts are summed to obtain  $count(prefix^i, p)$ . This count is further combined with the count of  $suffix^i$  analogously.

**Example 3.** In Figure 7, we compute the count of  $(A, B, C, D)$  based on the counts of  $(A, B)$  and  $(C, D)$ . Assuming that  $a_1$ – $d_8$  belong to the same window,  $count(A, B)$  is computed as shown in Figure 6(a). In addition, a count for each  $c$  (i.e.,  $c_3$  and  $c_7$ ) is maintained. When  $c_3$  arrives,  $count(A, B) = 1$ . We multiply it with  $count(c_3, D) = 2$  to obtain  $count(A, B, c_3, D) = 2$ . Analogously, when  $c_7$  arrives,  $count(A, B) = 5$ . It is multiplied with  $count(c_7, D) = 1$  to get  $count(A, B, c_7, D) = 5$ . Lastly, we sum these counts to obtain  $count(A, B, C, D) = 7$ .

**Time Complexity.** 1) **Count computation.** Since the shared pattern  $p$  is processed once for all queries in  $Q_p$ , the time complexity of processing each query  $q_i$  by the Shared method corresponds to the sum of the time complexity of processing  $prefix^i$  and  $suffix^i$  of  $q_i$ .

$$Comp(p, q_i) = Rate(E_1^i) \times Rate(prefix^i) + Rate(E_{m+l+1}^i) \times Rate(suffix^i) \quad (4)$$

2) **Count combination.** The time complexity of count multiplication is the product of the number of counts.

$$Comb(p, q_i) = Rate(E_1^i) \times Rate(E_m) \times Rate(E_{m+l+1}^i) \quad (5)$$

The time complexity of processing  $q_i$  by the Shared method is the sum of the time complexity of these two steps.

$$Shared(p, q_i) = Comp(p, q_i) + Comb(p, q_i) \quad (6)$$

For the set of queries  $Q_p$ , the time complexity corresponds to the summation of time complexity for each query  $q_i$ . In contrast to the Non-Shared method (Equation 3), the pattern  $p$  is computed once by the Shared method.

$$\text{Shared}(p, Q_p) = \text{Rate}(E_m) \times \text{Rate}(p) + \sum_{q_i \in Q_p} \text{Shared}(p, q_i) \quad (7)$$

### 3.4 Benefit of a Sharing Candidate

**Definition 5. (Benefit of a Sharing Candidate)** The benefit of sharing a pattern  $p$  by the set of queries  $Q_p$  is computed as the difference between the time complexity of the Non-Shared and Shared methods (Equations 3 and 7):

$$B\text{Value}(p, Q_p) = \text{NonShared}(p, Q_p) - \text{Shared}(p, Q_p) \quad (8)$$

A sharing candidate  $(p, Q_p)$  is called *beneficial* if  $B\text{Value}(p, Q_p) > 0$ . It is called *non-beneficial* otherwise.

**Non-Beneficial Candidate Pruning.** All non-beneficial candidates can be excluded from further analysis.

Based on this cost model, we conclude that *the number of queries, the length of their patterns, and the stream rate* determine the benefit of sharing. We experimentally study the effects of these factors in Section 8.

## 4 Sharing Conflict Modeling

A decision to share a pattern  $p$  by a query  $q \in Q_p$  may prevent sharing another pattern  $p'$  by  $q$  if  $p$  and  $p'$  overlap in  $q$ . Such sharing candidates are said to be in a *sharing conflict*. We now encode sharing candidates, their benefit, and conflicts into the SHARON graph. Based on the graph, we then reduce the search space of our sharing plan finder (Sections 5–6).

**Example 4.** In Table 1, queries  $q_3$  and  $q_4$  contain the overlapping patterns  $p_2 = (\text{ParkAve}, \text{OakSt})$  and  $p_1 = (\text{OakSt}, \text{MainSt})$ . Since the executor computes and stores the aggregates for a pattern as a whole (Section 3),  $q_3$  and  $q_4$  can either share  $p_1$  or  $p_2$ , but not both. Thus, the sharing candidates  $(p_1, \{q_1, q_2, q_3, q_4\})$  and  $(p_2, \{q_3, q_4\})$  give “contradictory instructions” for  $q_3$  and  $q_4$ . These candidates are said to be in a *sharing conflict*. However, if  $p_1$  were to be shared only by  $q_1$  and  $q_2$ , the sharing conflict between these candidates would be resolved. We sketch the techniques for sharing conflict resolution in Section 7.

**Definition 6. (Sharing Conflict)** Let  $p_A = (A_0 \dots A_n)$  and  $p_B = (B_0 \dots B_m)$  be patterns and  $Q_A$  and  $Q_B$  be query sets. The sharing candidates  $(p_A, Q_A)$  and  $(p_B, Q_B)$  are in *sharing conflict* if  $p_A$  overlaps with  $p_B$  in a query  $q \in Q_A \cap Q_B$ , i.e.,  $\exists k \in \mathbb{N}, 0 \leq k \leq n, m \text{ } A_{n-k} \dots A_n = B_0 \dots B_k$  in  $q$ . The query  $q$  causes the conflict between  $(p_A, Q_A)$  and  $(p_B, Q_B)$ .

**Definition 7. (Valid Sharing Plan)** A sharing plan  $\mathcal{P}$  is a set of sharing candidates.  $\mathcal{P}$  is *valid* if it contains no candidates that are in conflict with each other.  $\mathcal{P}$  is *invalid* otherwise.

---

**Algorithm 1** SHARON graph construction algorithm

---

**Input:** A hash table  $H$  mapping each sharable pattern  $p$  to a list of queries  $Q_p$  that contain  $p$

**Output:** SHARON graph  $G = (V, E)$

```

1:  $V \leftarrow \emptyset$ ;  $E \leftarrow \emptyset$ ;  $G \leftarrow (V, E)$ 
2: for each  $p$  in  $H$  do  $Q_p \leftarrow H.get(p)$ 
3:   if  $BValue(p, Q_p) > 0$  and  $Q_p.size > 1$  then
4:      $v \leftarrow (p, Q_p)$ ;  $v.weight \leftarrow BValue(p, Q_p)$ 
5:      $V \leftarrow V \cup v$ 
6:     for each  $u$  in  $V$  do
7:       if  $v$  and  $u$  are in sharing conflict then
8:          $E \leftarrow E \cup (v, u)$ 
9: return  $G$ 

```

---

**Definition 8. (Score of a Sharing Plan)** The score of a sharing plan  $\mathcal{P} = \{(p_1, Q_{p1}), \dots, (p_s, Q_{ps})\}$  is:

$$Score(\mathcal{P}) = \sum_{j=1}^s BValue(p_j, Q_{pj}) \quad (9)$$

**Definition 9. (Optimal Sharing Plan)** Let  $\mathbb{P}_{val}$  be the set of all valid sharing plans.  $\mathcal{P}_{opt} \in \mathbb{P}_{val}$  is an optimal sharing plan if  $\nexists \mathcal{P} \in \mathbb{P}_{val}$  with  $Score(\mathcal{P}) > Score(\mathcal{P}_{opt})$ .

**Example 5.** Given the workload in Figure 1, the plan  $\mathcal{P} = \{(p_2, \{q_3, q_4\}); (p_4, \{q_2, q_4\})\}$  is valid. Its sharing candidates are not in conflict since the patterns  $p_2 = (ParkAve, OakSt)$  and  $p_4 = (MainSt, WestSt)$  do not overlap. However,  $\mathcal{P}$  is not an optimal plan because  $Score(\mathcal{P}) = 24$  is not maximal among all valid plans. Indeed, another valid plan  $\{(p_1, \{q_1, q_2, q_3, q_4\})\}$  has higher score 25.

**Definition 10. (SHARON Graph)** Let  $S$  be the set of sharable patterns in a workload  $Q$ . The SHARON graph  $G = (V, E)$  has a set of weighted vertices  $V$  and a set of undirected edges  $E$ . Each vertex  $v \in V$  represents a sharing candidate  $(p, Q_p)$  where  $p \in S$  is a pattern and  $Q_p \subseteq Q$  is the set of all queries containing  $p$ . Each vertex is assigned a weight  $BValue(p, Q_p) > 0$  that corresponds to the benefit value of  $(p, Q_p)$  (Equation 8). Each edge  $(v, u) \in E$  represents a sharing conflict between the candidates  $v, u \in V$ .

**Example 6.** Figure 4 shows the SHARON graph for the traffic monitoring workload in Figure 1 and Table 1.

**SHARON Graph Construction Algorithm** consumes a hash table  $H$  that maps each sharable pattern  $p$  to the list of queries  $Q$  that contain  $p$ . If a pattern  $p$  is beneficial to be shared by at least two queries, the vertex  $v = (p, Q_p)$  with weight  $BValue(p, Q_p)$  is inserted into the graph (Lines 3–5 in Algorithm 1). Non-beneficial candidates are omitted. The edges representing the sharing conflicts between  $v$  and other vertices in the graph are inserted (Lines 6–8). The graph is returned at the end (Line 9).

**Data Structures.** We implement the SHARON graph as an adjacency list to efficiently retrieve the neighbors of a vertex  $v$ , i.e., identify the sharing conflicts of  $v$ . Each vertex stores a sharable pattern  $p$ , a list of queries  $Q_p$  that contain  $p$ , and the benefit value of the sharing candidate  $(p, Q_p)$ . The position of a query  $q$  in the list  $Q_p$  corresponds to the identifier of  $q$ . Thus, we can conclude whether

two candidates are in conflict in linear time in the maximal number of queries  $|Q_p|$  containing a sharable pattern  $p$ , i.e.,  $O(|Q_p|)$ .

**Complexity Analysis.** The time complexity is determined by the number of sharable patterns  $|H|$ , the number of sharing candidates  $|V|$ , and the maximal number of queries  $|Q_p|$  containing a pattern. In total,  $O(|H||V||Q_p|)$ . The space complexity corresponds to the size of the graph, i.e.,  $\Theta(|V||Q_p| + |E|)$ .

## 5 Sharing Candidate Pruning

Since the search space for an optimal plan is exponential in the number of candidates (Section 6), we prune two classes of candidates from a SHARON graph. One, *conflict-ridden candidates* are guaranteed not to be in the optimal plan because their benefit values are too low to counterbalance the loss of benefit from the sharing opportunities they exclude. Two, *conflict-free candidates* are guaranteed to be in the optimal plan since they do not prevent any other sharing opportunities.

**Conflict-Ridden Candidates.** We now map our MESA problem to the problem of finding a Maximum Weight Independent Set (MWIS) in a graph, which is known to be NP-hard [16]. The greedy algorithm GWMIN [26] for MWIS does not always return a high-quality plan as confirmed by our experiments in Section 8.3. However, its guaranteed minimal weight can be used to prune conflict-ridden candidates.

**Definition 11. (Maximum Weight Independent Set)** Let  $G = (V, E)$  be a graph with a set of weighted vertices  $V$  and a set of edges  $E$ . For a set of vertices  $V' \subseteq V$ , we denote the sum of the weights of the vertices in  $V'$  as  $Weight(V')$ .  $IS \subseteq V$  is an independent set of  $G$  if for any vertices  $v_i, v_j \in IS$ ,  $(v_i, v_j) \notin E$  holds. Let  $S_{IS}$  be the set of all independent sets of  $G$ .  $IS \in S_{IS}$  is a maximum weight independent set of  $G$  if  $\nexists IS' \in S_{IS}$  with  $Weight(IS') > Weight(IS)$ .

**Lemma 1.** Let  $Q$  be a query workload,  $G$  be the SHARON graph for  $Q$ , and  $\mathcal{P}_{opt}$  be an optimal sharing plan for  $Q$ . Then,  $\mathcal{P}_{opt}$  is an MWIS of  $G$ .

*Proof.* By Definitions 7 and 9,  $\mathcal{P}_{opt}$  is valid, i.e., contains no conflicting sharing candidates. By Definition 10, no vertices in  $\mathcal{P}_{opt}$  are connected by an edge in  $G$ . By Definition 11,  $\mathcal{P}_{opt}$  is an independent set of  $G$ . By Definition 9,  $\mathcal{P}_{opt}$  has the maximum score among all valid plans. By Definition 10,  $\mathcal{P}_{opt}$  has the maximum weight among all independent sets of  $G$ . By Definition 11,  $\mathcal{P}_{opt}$  is an MWIS of  $G$ .  $\square$

GWMIN returns an independent set  $IS$  with weight:

$$Weight(IS) \geq \sum_{u \in V} \frac{weight(u)}{degree(u) + 1} \quad (10)$$

To safely prune a conflict-ridden candidate  $v$ , we define the maximal score of a plan containing  $v$ , denoted  $Score_{max}(v)$ . In best case, a plan containing  $v$  includes all other candidates that are not in conflict with  $v$ . Thus,  $Score_{max}(v)$  corresponds to the summation of benefit values of all sharing candidates that are not in conflict with  $v$ .

**Definition 12. (Maximal Score of a Plan Containing a Sharing Candidate)** Let  $v \in V$  be a sharing candidate in a SHARON graph  $G = (V, E)$  and  $\mathcal{N}(v) \subseteq V$  be the set of candidates that are in conflict with  $v$ . The maximal score of a sharing plan containing  $v$  is defined as follows:

$$Score_{max}(v) = \sum_{u \in V \setminus \mathcal{N}(v)} BValue(u) \quad (11)$$

**Lemma 2.** For a valid sharing plan  $\mathcal{P}$  and a sharing candidate  $v \in \mathcal{P}$ ,  $Score(\mathcal{P}) \leq Score_{max}(v)$  holds.

*Proof.* Let  $G = (V, E)$  be the SHARON graph such that  $\mathcal{P} \subseteq V$  is an independent set of  $G$  and  $v \in \mathcal{P}$ . By Definition 7,  $\mathcal{P}$  contains no conflicting candidates. By Definition 10, all vertices in  $\mathcal{N}(v)$  are in conflict with  $v$  and thus are not in  $\mathcal{P}$ . Since  $\mathcal{P}$  may need to remove additional vertices to avoid other conflicts,  $\mathcal{P} \subseteq V \setminus \mathcal{N}(v)$ . By Definition 12,  $Score_{max}(v)$  is the sum of BValues of all candidates in  $V \setminus \mathcal{N}(v)$ . Since all BValues of vertices in  $V$  are positive (Section 3.4),  $\mathcal{P} \subseteq V \setminus \mathcal{N}(v)$  implies  $Score(\mathcal{P}) \leq Score_{max}(v)$ .  $\square$

**Definition 13. (Conflict-Ridden Sharing Candidate)** Let  $G = (V, E)$  be a SHARON graph. A sharing candidate  $v \in V$  is conflict-ridden if the maximal score of a sharing plan containing  $v$  is lower than the guaranteed weight of GWMIN.

$$Score_{max}(v) < \sum_{u \in V} \frac{BValue(u)}{degree(u) + 1} \quad (12)$$

**Conflict-Ridden Candidate Pruning.** All conflict-ridden candidates are pruned from the SHARON graph without sacrificing the optimality of the resulting sharing plan.

**Example 7.** The guaranteed weight on the graph in Figure 4 is  $\frac{25}{6} + \frac{9}{4} + \frac{12}{5} + \frac{15}{4} + \frac{20}{5} + \frac{8}{2} + \frac{18}{1} \approx 38.57$ . Since  $Score_{max}(p_3, \{q_3, q_4\}) = BValue(p_3, \{q_3, q_4\}) + BValue(p_6, \{q_1, q_5\}) + BValue(p_7, \{q_6, q_7\}) = 38 < 38.57$ , an optimal sharing plan cannot contain  $(p_3, \{q_3, q_4\})$ . Thus, this candidate and its conflicts can be pruned.

**Conflict-Free Candidates** do not exclude any other sharing opportunities and increment the score of a plan by their benefit values. Such candidates can be directly added to an optimal plan and removed from further analysis.

**Definition 14. (Conflict-Free Sharing Candidate)** A sharing candidate  $v \in V$  in a SHARON graph  $G = (V, E)$  is conflict-free if  $\nexists u \in V$  with  $(v, u) \in E$ .

**Example 8.** The conflict-free candidate  $(p_7, \{q_6, q_7\})$  in Figure 4 increments the score of a plan by its benefit 18.

**SHARON Graph Reduction Algorithm** (Algorithm 2) consumes a SHARON graph  $G$  and the guaranteed weight of GWMIN. It removes all conflict-ridden or conflict-free candidates from the graph  $G$ . The algorithm returns the reduced graph and the set of conflict-free candidates.

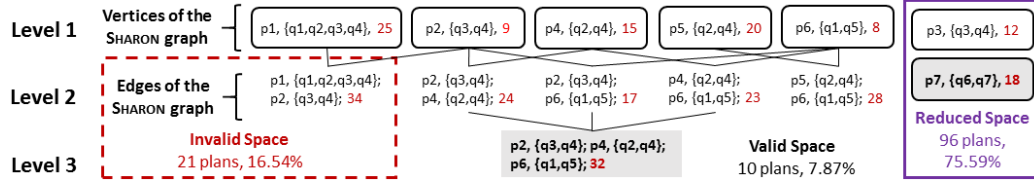


Figure 8: Search space of the sharing plan finder algorithm

**Algorithm 2** SHARON graph reduction algorithm**Input:** SHARON graph  $G$ , guaranteed weight  $min$  of GWMIN**Output:** Reduced graph  $G$ , conflict-free candidates  $F$ 

```

1:  $F \leftarrow \emptyset$ 
2: while  $G$  can be reduced do
3:   for each  $v$  in  $V$  do
4:     if  $\text{degree}(v) = 0$  then
5:        $F \leftarrow F \cup v$ ;  $G.\text{remove}(v)$ 
6:     else if  $\text{Score}_{max}(v) < min$  then
7:        $G.\text{remove}(v)$ 
8: return  $G, F$ 

```

**Complexity Analysis.** The time complexity is determined by the nested loops that iterate  $O(|V|)$  and  $\Theta(|V|)$  times respectively. The time complexity of removing a candidate  $v$  from the graph in Line 7 is  $O(|E|)$  since all conflicts of  $v$  are also deleted. Thus, the time complexity is quadratic  $O(|V|^2|E|)$ . The space complexity is determined by the size of the graph  $G$  and the set  $F$ . Since  $|F| \leq |V|$ , the space costs are linear, i.e.,  $O(|V| + |E|)$ .

**Example 9.** Figure 8 depicts the search space for an optimal plan for our running example. Since the conflict-ridden candidate  $(p_3, \{q_3, q_4\})$  is pruned (Example 7), while the conflict-free candidate  $(p_7, \{q_6, q_7\})$  is added to the optimal plan (Example 8), the search space is reduced by  $2^7 - 2^5 = 96$  plans. This **reduced** space is indicated by a solid frame in Figure 8. It corresponds to 75.59% of the search space.

## 6 Sharing Plan Finder

Based on the reduced SHARON graph, we now propose the optimal sharing plan finder. In addition to the non-beneficial and conflict-ridden candidate pruning principles, we define invalid branch pruning. It cuts off those branches of the search space that contain only invalid plans early on.

**Search Space for an Optimal Sharing Plan.** The parent-child relationships between sharing plans are defined next.

**Definition 15. (Parent-Child Relationship between Sharing Plans)** Let  $\mathcal{P}$  and  $\mathcal{P}'$  be sharing plans. If  $\mathcal{P} \subset \mathcal{P}'$ , then  $\mathcal{P}$  is an ancestor of  $\mathcal{P}'$  ( $\mathcal{P}'$  is a descendant of  $\mathcal{P}$ ). If  $|\mathcal{P}| = |\mathcal{P}'| - 1$ , then  $\mathcal{P}$  is a parent of  $\mathcal{P}'$  ( $\mathcal{P}'$  is a child of  $\mathcal{P}$ ).

The search space has a lattice shape (Figure 8). The plans in Level 1 correspond to the vertices in Figure 4. Level  $s$  contains sharing plans of size  $s$ . The size of the search space is exponential in the



number of candidates, denoted  $|V|$ . It is computed as the sum of the number of plans at each level:

$$\sum_{s=0}^{|V|} \binom{|V|}{s} = 2^{|V|} \quad (13)$$

**Lemma 3.**  $Score(\mathcal{P}') > Score(\mathcal{P})$  if  $\mathcal{P}$  is a parent of  $\mathcal{P}'$ .

*Proof.* By Definition 15,  $\mathcal{P} \subset \mathcal{P}'$  and  $|\mathcal{P}| = |\mathcal{P}'| - 1$ . Let  $\mathcal{P}' \setminus \mathcal{P} = (p, Q_p)$ . By Definition 10, only a candidate  $(p, Q_p)$  with  $BValue(p, Q_p) > 0$  is included into a SHARON graph. Thus,  $(p, Q_p)$  increases the score of  $\mathcal{P}'$  compared to  $\mathcal{P}$ .  $\square$

A naive plan finder considers all combinations of candidates and keeps track of a valid plan with the maximal score seen so far. However, this solution constructs many invalid plans that are subsequently discarded. To avoid such exhaustive search, we prove the following properties of the search space.

**Lemma 4.** *All descendants of an invalid plan are invalid.*

*Proof.* Let  $\mathcal{P}$  be an invalid sharing plan and  $\mathcal{P}_d$  be its descendant. By Definition 15,  $\mathcal{P} \subset \mathcal{P}_d$ . Thus,  $\mathcal{P}_d$  “inherits” all sharing conflicts from  $\mathcal{P}$  which makes  $\mathcal{P}_d$  invalid.  $\square$

**Invalid Branch Pruning.** Invalid plans of size two correspond to edges of a SHARON graph (Figures 4 and 8). Thus, all descendants of invalid plans of size two can be safely pruned. Our plan finder cuts off invalid branches at their roots.

**Example 10.** *In Figure 8, only 7.87% of the search space is **valid**. It consists of 10 plans. This valid space is traversed to find the optimal plan  $\{(p_2, \{q_3, q_4\}); (p_4, \{q_2, q_4\}); (p_6, \{q_1, q_5\}); (p_7, \{q_6, q_7\})\}$  highlighted by a darker background.*

*In Figure 8, 16.54% of the search space is **invalid**. The invalid space consists of 21 plans =  $2^5$  not reduced plans – 10 valid plans – 1 empty plan. The invalid space is indicated by the dashed frame. It is pruned by our plan finder.*

**Valid Search Space Traversal.** A plan of size one is valid by Definition 7. A plan of size two  $\{v_1, v_2\}$  is valid if there is no edge  $(v_1, v_2)$  in the SHARON graph. Validity of a larger plan is determined as described next.

**Lemma 5.** *A sharing plan  $\mathcal{P}$ ,  $|\mathcal{P}| > 2$ , is valid if and only if all its parents are valid.*

*Proof.* “ $\Rightarrow$ ” The proof follows directly from Lemma 4.

“ $\Leftarrow$ ” Let  $\mathcal{P}$  contain a sharing conflict, say, between candidates  $v$  and  $u$ . Then there exists a parent  $\mathcal{P}_p$  of  $\mathcal{P}$  that contains  $v$  and  $u$ . Hence,  $\mathcal{P}_p$  is invalid.  $\square$

By Definition 15, a plan of size  $s$  has  $s$  parents. Instead of accessing all parent plans to generate one new valid plan, we prove that only two parents and one ancestor of size two must be valid to guarantee validity of a sharing plan (similarly to Apriori candidate generation [5]).

---

**Algorithm 3** Level generation algorithm

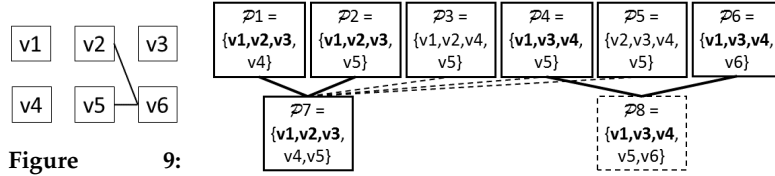
**Input:** SHARON graph  $G = (V, E)$ , set of sharing plans of size  $s$  *Parents* =  $\{P_0, \dots, P_{s-1}\}$

**Output:** Set of sharing plans of size  $s + 1$  *Children*

```

1: getNextLevel( $G, Parents$ ) {
2:    $Children \leftarrow \emptyset$ 
3:   for each ( $i = 0; i < s; i++$ ) do
4:     for each ( $j = i + 1; j < s; j++$ ) do
5:       if  $s = 1$  and  $(P_i.v_1, P_j.v_1)$  not in  $E$  then
6:          $Children.add(P_i \cup P_j)$ 
7:       if  $P_i.v_1 = P_j.v_1, \dots, P_i.v_{s-1} = P_j.v_{s-1}$  and  $(P_i.v_s, P_j.v_s)$  not in  $E$  then
8:          $Children.add(P_i \cup P_j)$ 
9:   return  $Children$  }
```

---



**Figure 9:**  
SHARON graph

**Figure 10:** Generation of a new valid sharing plan

**Lemma 6.** Let  $G = (V, E)$  be a SHARON graph,  $\mathcal{P}_1$  and  $\mathcal{P}_2$  be valid parents of  $\mathcal{P}$ ,  $|\mathcal{P}| > 2$ . For two candidates  $v_1 = \mathcal{P}_1 \setminus \mathcal{P}_2$  and  $v_2 = \mathcal{P}_2 \setminus \mathcal{P}_1$ , if  $(v_1, v_2) \notin E$ , then  $\mathcal{P}$  is valid.

*Proof.* Assume all the above conditions hold but  $\mathcal{P}$  is invalid. Then  $\mathcal{P}$  contains at least one pair of conflicting candidates. By Definition 15,  $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$  and  $\mathcal{P}$  has one additional candidate compared to  $\mathcal{P}_1$  (or  $\mathcal{P}_2$ ). Since  $\mathcal{P}_1$  and  $\mathcal{P}_2$  are valid, there can be only one pair of conflicting candidates  $v_1$  and  $v_2$  in  $\mathcal{P}$  such that  $v_1 = \mathcal{P}_1 \setminus \mathcal{P}_2$  and  $v_2 = \mathcal{P}_2 \setminus \mathcal{P}_1$ . By Definition 10,  $(v_1, v_2) \in E$  which is a contradiction.  $\square$

**Level Generation Algorithm** consumes a SHARON graph  $G$  and a set of sharing plans of size  $s$ , called *Parents*. It returns level  $s + 1$  of the search space, i.e., the set of all sharing plans of size  $s + 1$ , called *Children*. Algorithm 3 iterates through all pairs of parent plans of size  $s$  (Lines 3–4). In the base case, the *Parents* are the vertices of  $G$  and the *Children* are non-adjacent pairs of vertices (Lines 5–6). In the inductive case, to generate a valid plan of size  $s + 1$ , the algorithm identifies two plans of size  $s$ ,  $P_i$  and  $P_j$ , that begin with the same  $s - 1$  decisions. If the plan containing the last decisions of  $P_i$  and  $P_j$  (denoted  $P_i.v_s$  and  $P_j.v_s$ ) is valid, the plan  $P_i \cup P_j$  is also valid (Lines 7–8).

**Complexity Analysis.** The time complexity of Algorithm 3 is determined by the number of plans at one level, namely, the binomial coefficient  $\binom{|V|}{s}$  in Equation 13. Due to two nested loops, the time complexity is  $O(\binom{|V|}{s}^2)$ . The space complexity is also determined by the number of plans at one level, i.e.,  $O(\binom{|V|}{s})$ .

**Example 11.** Figure 10 shows a portion of a search space with valid plans  $\mathcal{P}_1$ – $\mathcal{P}_6$  of size four.  $\mathcal{P}_7$  is the only valid plan of size five. It is generated as follows. (1) We identify two plans of size four that start with the same three candidates, e.g.,  $\mathcal{P}_1$  and  $\mathcal{P}_2$  start with  $\{v_1, v_2, v_3\}$ . (2) We compute their symmetric difference

---

**Algorithm 4** Sharing plan finder algorithm

---

**Input:** SHARON graph  $G = (V, E)$ , set of conflict-free candidates  $F$ 
**Output:** Optimal sharing plan  $opt \cup F$ 

```

1:  $opt \leftarrow \emptyset$ ;  $max \leftarrow 0$ 
2: for each  $v$  in  $V$  do
3:   if  $BValue(v) > max$  then
4:      $opt \leftarrow \{v\}$ ;  $max \leftarrow BValue(v)$ 
5:  $Level \leftarrow getNextLevel(G, V)$ 
6: while  $Level \neq \emptyset$  do
7:   for each  $P$  in  $Level$  do
8:     if  $Score(P) > max$  then
9:        $opt \leftarrow P$ ;  $max \leftarrow Score(P)$ 
10:   $Level \leftarrow getNextLevel(G, Level)$ 
11: return  $opt \cup F$ 

```

---

$\mathcal{P}_1 \Delta \mathcal{P}_2 = \{v_4, v_5\}$ . (3) Since there is no edge  $(v_4, v_5)$  in Figure 9,  $\mathcal{P}_7$  is valid. There is no need to check the other three parents of  $\mathcal{P}_7$ . In contrast,  $\mathcal{P}_8$  is invalid since  $v_5$  and  $v_6$  are in conflict.

**Sharing Plan Finder Algorithm** traverses valid search space using Breadth-First-Search. Algorithm 4 effectively prunes invalid branches at their roots. It constructs only valid plans and returns an optimal plan among them.

**Correctness.** We prove that Algorithm 4 considers all valid sharing plans, i.e., it returns the optimal sharing plan.

**Lemma 7.** *If a sharing plan is valid, then it is considered by the sharing plan finder algorithm.*

*Proof.* We prove Lemma 7 by induction. The base cases are  $s = 1$  and  $s = 2$ . First,  $V$  is the set of all valid sharing plans of size 1. It is considered by Algorithm 4. Second, Algorithm 4 (Line 5) generates all plans of size 2 by considering all non-adjacent vertex pairs in Algorithm 3 (Lines 5–6).

Now, we assume that all valid plans of up to and including size  $s$ , such that  $s \geq 2$ , are considered. We will prove that all valid plans of size  $s + 1$  are also considered. Let  $\mathcal{P} = \{v_1, \dots, v_s, v_{s+1}\}$  be a valid plan of size  $s + 1$ . Then  $\mathcal{P}_1 = \{v_1, \dots, v_{s-1}, v_s\}$ ,  $\mathcal{P}_2 = \{v_1, \dots, v_{s-1}, v_{s+1}\}$ , and  $\mathcal{P}_3 = \{v_s, v_{s+1}\}$  are ancestors of  $\mathcal{P}$ . By Lemma 5,  $\mathcal{P}_1$ ,  $\mathcal{P}_2$ , and  $\mathcal{P}_3$  are valid. By the induction assumption, they were considered by Algorithm 4. Algorithm 3 generates the plan  $\mathcal{P}$  from  $\mathcal{P}_1$ ,  $\mathcal{P}_2$ , and  $\mathcal{P}_3$ . Thus, Algorithm 4 considers  $\mathcal{P}$ .  $\square$

**Data Structures.** For each plan  $\mathcal{P}$ , we store the list of sharing candidates and the score of  $\mathcal{P}$ . These candidates are sorted alphabetically by their patterns within a plan because sequential access of candidates in plans enables efficient generation of new valid plans (Lemma 6, Example 11). Our plan finder stores the best plan found so far and only one level of the search space at a time. It discards a level immediately after generating the next level.

**Complexity Analysis.** Since the entire valid search space is traversed, the algorithm has exponential time and space complexity in the worst case (Equation 13). However, the SHARON optimizer is efficient on average thanks to its effective pruning principles (Section 8.3).

**Effectiveness of Sharing Plan Finder.** Our experiments in Section 8.3 demonstrate that our SHARON optimizer finds an optimal plan in reasonable time due to three effective pruning principles (i.e., non-beneficial, conflict-ridden, and invalid candidates in Sections 3.4, 5, and 6). Only in the following two extreme cases our solution may be ineffective:

1) Since the algorithm traverses the entire valid space, its time complexity is exponential (Equation 13). If the search space is too large, we can constrain the optimization time by  $l$  seconds. If our SHARON optimizer does not return an optimal plan within  $l$  seconds, we instead would run GWMIN [26] with polynomial time complexity to find a sharing plan and start our SHARON executor using this plan. Later, when the optimal plan is produced by our SHARON optimizer, we can replace the greedily found plan by the optimal plan.

2) The valid search space becomes small if many sharing conflicts exist (Figure 8). In this case, only a few patterns can be shared and a fairly low score of a sharing plan would be achieved. In the worst case, no pattern can be shared, i.e., SHARON defaults to the Non-Shared Method (Section 3.2). Our optimizer finds such a trivial plan very quickly.

**Optimal versus Greedily Chosen Plan.** While the greedy algorithm GWMIN is useful to reduce the search space (Section 5), the score of a greedily chosen plan might be considerably lower than the score of an optimal plan.

**Example 12.** *Even in our small example in Figure 4, the greedily chosen plan  $\mathcal{P}_{gre} = \{(p_1, \{q_1, q_2, q_3, q_4\}); (p_7, \{q_6, q_7\})\}$  has score 43, while the optimal plan  $\mathcal{P}_{opt} = \{(p_2, \{q_3, q_4\}); (p_4, \{q_2, q_4\}); (p_6, \{q_1, q_5\}); (p_7, \{q_6, q_7\})\}$  increases  $Score(\mathcal{P}_{gre})$  by more than 16% to 50.*

## 7 Extensions of the Sharon Approach

In this section, we briefly describe the extensions of our approach to relax the simplifying assumptions in Section 2.1.

### 7.1 Sharing Conflict Resolution

Our analysis in Section 4 reveals that promising sharing opportunities might be excluded by sharing conflicts. Generally, the more queries share a pattern the higher the probability of sharing conflicts becomes (Definition 6). We now open up additional sharing opportunities by resolving sharing conflicts as follows.

Given a SHARON graph  $G = (V, E)$ , we expand each candidate  $v = (p, Q_p) \in V$  with conflicts  $E_v \subseteq E$  to a set of options  $\mathcal{O}_p$ . Each option  $v' = (p, Q'_p) \in \mathcal{O}_p$  resolves a different subset of conflicts  $E'_v \subseteq E_v$  of the original candidate  $v$  with other candidates  $u \in V \setminus \mathcal{O}_p$ . In contrast to the original candidate  $v$ , an option  $v'$  considers sharing the pattern  $p$  by a *subset* of queries containing  $p$ , i.e.,  $Q'_p \subseteq Q_p, |Q'_p| > 1$ .

**Example 13.** *In Figure 4, the sharing candidate  $(p_1, \{q_1, q_2, q_3, q_4\})$  can be expanded to a set of options. The option  $(p_1, \{q_1, q_3\})$  is not in sharing conflict with the candidates  $(p_4, \{q_2, q_4\})$  and  $(p_5, \{q_2, q_4\})$ . Thus,*

---

**Algorithm 5** Sharing candidate expansion algorithm
 

---

**Input:** SHARON graph  $G = (V, E)$ ,  $v = (p, Q_p) \in V$ 
**Output:** Set  $O_p$  of sharing candidate options for  $p$ 

```

1:  $getSet(G, v)$  {
2:    $L_c, L_n \leftarrow \text{empty stacks}; L_c.push(v); O_p \leftarrow \{v\}$ 
3:   while  $!L_c.isEmpty()$  do
4:      $v \leftarrow L_c.pop()$ 
5:     for each conflict  $(v, u)$  in  $E$  do
6:        $Q_c \leftarrow \text{queries in } Q_p \text{ that cause } (v, u)$ 
7:       for each combination  $C$  of  $Q_c$  that can resolve  $(v, u)$  do
8:          $Q'_p \leftarrow Q_p \setminus C$ 
9:         if  $|Q'_p| > 1$  and  $Q'_p$  is new then
10:           $v' \leftarrow (p, Q'_p); L_n.push(v')$ 
11:           $O_p \leftarrow O_p \cup \{v'\}$ 
12:       if  $L_c.isEmpty()$  then
13:          $L_c \leftarrow L_n; L_n \leftarrow \text{empty stack}$ 
14: return  $O_p$  }
```

---

they could belong to the same sharing plan which may have a higher score than a plan containing the original candidate  $(p_1, \{q_1, q_2, q_3, q_4\})$ .

**Definition 16. (Resolved Sharing Conflict.)** Let the candidates  $v_1 = (p_1, Q_1)$  and  $v_2 = (p_2, Q_2) \in V$  be in conflict  $(v_1, v_2) \in E$  caused by the queries  $Q = Q_1 \cap Q_2$  such that  $Q = Q'_1 \cup Q'_2$ .<sup>\*</sup> The conflict  $(v_1, v_2)$  is resolved by omitting  $Q'_1$  and  $Q'_2$  from  $Q_1$  and  $Q_2$  respectively.

By Definition 6, the sharing candidates  $v'_1 = (p_1, Q_1 \setminus Q'_1)$  and  $v'_2 = (p_2, Q_2 \setminus Q'_2)$  are not in conflict since  $(Q_1 \setminus Q'_1) \cap (Q_2 \setminus Q'_2) = \emptyset$ . The conflict  $(v_1, v_2)$  is resolved if *any* query sets  $Q'_1$  and  $Q'_2$  that compose  $Q$  are omitted from  $Q_1$  and  $Q_2$  respectively. In the worst case, *all* combinations of queries  $Q$  are included into the sets of options for  $v_1$  and  $v_2$ .

**Sharing Candidate Expansion Algorithm.** For a SHARON graph  $G$  and a candidate  $v = (p, Q_p) \in V$ , Algorithm 5 builds a tree of options  $O_p$  using Breadth First Search. The root of this tree is the original candidate  $v$ . To generate a child of  $v$ , the algorithm skips the queries from  $Q_p$  that cause a conflict of  $v$  with another sharing candidate  $u \in V \setminus O_p$ . We label an edge between  $v$  and its child by the sharing candidate  $u$ . The algorithm terminates when no new option with at least two queries can be generated.

**Complexity Analysis.** The time and space complexity of Algorithm 5 are determined by the maximal size of a set  $|O_p^{max}|$ . Let  $d$  be the maximal degree of a candidate  $v \in V$  and  $k$  be the maximal number of queries that cause a conflict. For each conflict  $(v, u) \in E$ , all combinations of queries causing this conflict are considered (nested for-loops in Lines 5–10 and 7–10). Thus,

$$|O_p^{max}| = \sum_{i=0}^d \binom{d}{i} \sum_{j=0}^{k-1} \binom{k}{j} \quad (14)$$

---

<sup>\*</sup> $\cup$  denotes disjoint set union, meaning that  $Q = Q'_1 \cup Q'_2$  but  $Q'_1 \cap Q'_2 = \emptyset$ .

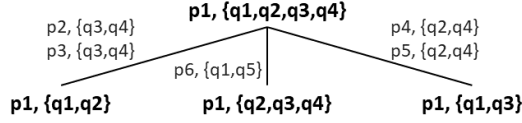


Figure 11: Sharing candidate options for pattern  $p_1$

---

**Algorithm 6** Sharing conflict resolution algorithm

---

**Input:** SHARON graph  $G = (V, E)$

**Output:** Expanded SHARON graph  $G$

- 1:  $V' \leftarrow \emptyset; E' \leftarrow \emptyset$
  - 2: **for each**  $v = (p, Q_p)$  in  $V$  **do**
  - 3:    $O_p \leftarrow \text{getSet}(G, v); V' \leftarrow V' \cup O_p$
  - 4:   **for each**  $v'$  in  $O_p$  **do**
  - 5:     **for each**  $u$  in  $V'$  **do**
  - 6:       **if**  $v'$  and  $u$  are in sharing conflict **then**
  - 7:          $E'.\text{add}(v', u)$
  - 8: **return**  $G \leftarrow (V', E')$
- 

where  $i$  denotes the number of resolved conflicts, while  $j$  corresponds to the number of skipped queries to resolve one conflict.

**Example 14.** Figure 11 illustrates the sharing candidate options for the candidate  $v = (p_1, \{q_1, q_2, q_3, q_4\})$  in Figure 4. To resolve the conflict with  $u_1 = (p_2, \{q_3, q_4\})$  and  $u_2 = (p_3, \{q_3, q_4\})$ , queries  $q_3$  and  $q_4$  are dropped from the set of queries of  $v$ . The edge between  $v$  and its child  $(p_1, \{q_1, q_2\})$  is labeled by  $u_1, u_2$ . Other conflicts of  $v$  are resolved analogously.

**Sharing Conflict Resolution Algorithm.** For a SHARON graph  $G$  and each candidate  $v = (p, Q_p) \in V$ , (Algorithm 6) expands  $v$  to a set of options  $O_p$  to open up additional sharing opportunities. The algorithm updates the conflicts of these options and returns the expended graph.

**Complexity Analysis.** The time complexity is determined by three nested for-loops that are called  $\Theta(|V|)$ ,  $|O_p^{\max}|$  and  $\Theta(|V'|)$  times respectively where  $|O_p^{\max}|$  denotes the maximal size of a set (Equation 14). Since  $|V| \leq |V'|$  and  $|O_p^{\max}| \leq |V'|$ , the time complexity is cubic in the number of candidates in the expended SHARON graph in the worst case, i.e.,  $O(|V'|^3)$ . The space complexity is determined by the size of the expended graph, i.e.,  $\Theta(|V'| + |E'|)$ .

**Example 15.** The SHARON graph in Figure 4 is expanded in Figure 12. The sharing candidate for pattern  $p_1$  is expanded into a set of options and highlighted by a rectangle frame. The sets for other candidates contain only the original candidate. Conflicts within sets are omitted for readability.

The expended graph is then reduced (Section 5) and serves as input to our sharing plan finder (Section 6).

## 7.2 Different Predicates, Grouping, and Windows

Leveraging existing techniques, our SHARON approach can share event sequence aggregation among queries with different grouping, windows, and predicates. Grouping partitions the stream into

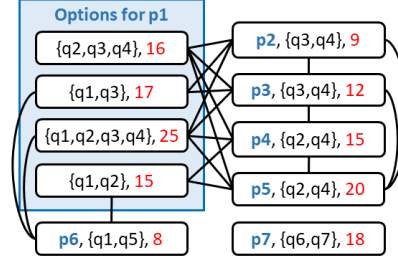


Figure 12: Expanded SHARON graph

sub-streams by the values of grouping attributes [24, 14]. Windows and predicates further partition these sub-streams into disjoint segments and share the intermediate aggregates per segment to compute the final results for each query [14, 17, 7, 20]. These refinement strategies might not always be effective, because of a large number of small segments and the overhead of their computation. However, these are orthogonal problems. Our SHARON approach can be applied within each segment to tackle different query patterns.

### 7.3 Multiple Occurrences of an Event Type in a Pattern

If an event type  $E$  occurs  $k$  times in a pattern, an event of type  $E$  updates the counts of  $k$  prefix patterns that end at  $E$  (Section 3). Then, the time complexity of both the Non-Shared and the Shared methods increases by the multiplicative factor  $k$  (Equations 2, 4, and 7). Our SHARON optimizer is not affected by this extension.

### 7.4 Dynamic Workloads

In dynamic environments, new queries may be added or existing queries may be removed. Even if the queries remain the same, the workload may still vary due to event rate fluctuations. Thus, a chosen plan may become sub-optimal. In this case, our SHARON approach leverages runtime statistics techniques [18] to detect such fluctuations and to trigger the SHARON optimizer to produce a new optimal plan based on the new workload. Dynamic plan migration techniques [17, 33] can be employed to migrate from the old to the new sharing plan and ensure that no results are lost or corrupted for stateful operators such as aggregation.

## 8 Performance Evaluation

### 8.1 Experimental Setup

**Infrastructure.** We have implemented our SHARON approach in Java with JRE 1.7.0\_25 running on Ubuntu 14.04 with 16-core 3.4GHz CPU and 128GB of RAM. We execute each experiment three times and report the average here.

**Data Sets.** We evaluate the performance of our SHARON approach using the following data sets.

- **TX: New York City Taxi and Uber Real Data Set.** We use the real data set [3] (330GB) containing 1.3 billion taxi and Uber trips in New York City in 2014–2015. Each event carries pick-up and drop-off locations and time stamps in seconds, number of passengers, price, and payment method.

- **LR: Linear Road Benchmark Data Set.** We use the traffic simulator of the Linear Road benchmark [6] for streaming systems to generate a stream of position reports from cars for 3 hours. Each position report carries a time stamp in seconds, a car identifier, its location and speed. Event rate gradually increases from few dozens to 4k events per second.

- **EC: E-Commerce Synthetic Data Set.** Our stream generator creates sequences of items bought together for 3 hours. Each event carries a time stamp in seconds, item and customer identifiers. We consider 50 items and 20 users. The values of item and customer identifiers of an event are randomly generated. The stream rate is 3k events per second.

We ran each experiment on the above three data sets. Due to space limitations, similar charts are not shown here.

**Event Queries.** We evaluate a workload similar to  $q_1$ – $q_7$  in Section 1 against the taxi and Linear Road data sets and a workload similar to  $q_8$ – $q_{11}$  against the e-commerce data set. Based on our cost model (Section 3), we vary the major cost factors, namely, number of queries, the length of their patterns, and the number of events per window. Unless stated otherwise, we evaluate 20 queries. The default length of their patterns is 10. The default number of events per window is 200k.

**Methodology.** We run two sets of experiments.

1) *Sharon Executor vs. State-of-the-Art Approaches* (Section 8.2). We demonstrate the effectiveness of our SHARON executor (Section 3) by comparing it to the state-of-the-art techniques A-Seq [24], SPASS [25], and Flink [1] covering the spectrum of approaches to event sequence aggregation (Figure 3). While Section 9 is devoted to a detailed discussion of these approaches, we briefly sketch their main ideas below.

- **A-Seq** [24] avoids sequence construction by incrementally maintaining a count for each prefix of a pattern. However, it has no optimizer to determine which queries should share the aggregation of which patterns. By default, it computes each query independently from other queries and thus suffers from repeated computations (Section 3.2).

- **SPASS** [25] defines shared event sequence construction. Their aggregation is computed afterwards and is not shared. Thus, SPASS is a two-step and only partially shared approach.

- **Flink** [1] is a popular open-source streaming system that supports event pattern matching and aggregation. We express our queries using Flink operators. Flink constructs all event sequences prior their aggregation. It does not share computations among different queries.

To achieve a fair comparison, we have implemented A-Seq and SPASS on top of our platform. We execute Flink on the same hardware as our platform.

2) *Sharon Optimizer* (Section 8.3). We study the efficiency of our SHARON optimizer (Sections 4–7) by comparing it to the greedy algorithm GWMIN [26] and to exhaustive search. We also compare the quality of a greedily chosen plan returned by GWMIN to an optimal plan returned by our SHARON optimizer and the exhaustive search.



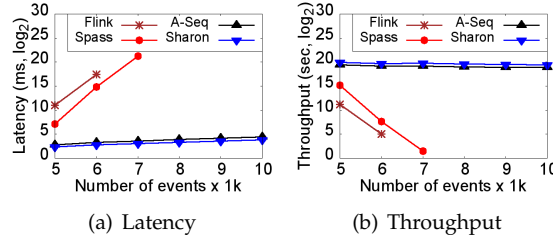


Figure 13: Two-step versus online approaches (Linear Road data set)

**Metrics.** We measure the following metrics common for streaming systems. *Latency* is measured in milliseconds as the average time difference between the time point of aggregate output and the arrival time of the latest event that contributed to this result. *Throughput* corresponds to the average number of events processed by all queries per second. *Peak memory* is measured in bytes. For event sequence aggregation algorithms, it is the maximal memory for storing aggregates, events, and event sequences. For the optimizer algorithms, the peak memory is the maximal memory for storing the SHARON graph and the sharing plans during space traversal.

## 8.2 Sharon Executor versus State-of-the-Art Approaches

**Two-step Approaches.** In Figure 13, we vary the number of events per window and measure latency and throughput of the event sequence aggregation approaches using the Linear Road benchmark data set. Latency of the two-step approaches (SPASS and Flink) increases exponentially, while throughput decreases exponentially in the number of events.

*SPASS* achieves 6-fold speed-up compared to Flink for 6k events per window because SPASS shares event sequence construction. Due to event sequence construction overhead, SPASS does not terminate when the number of events exceeds 7k. These measurements are not shown in Figure 13.

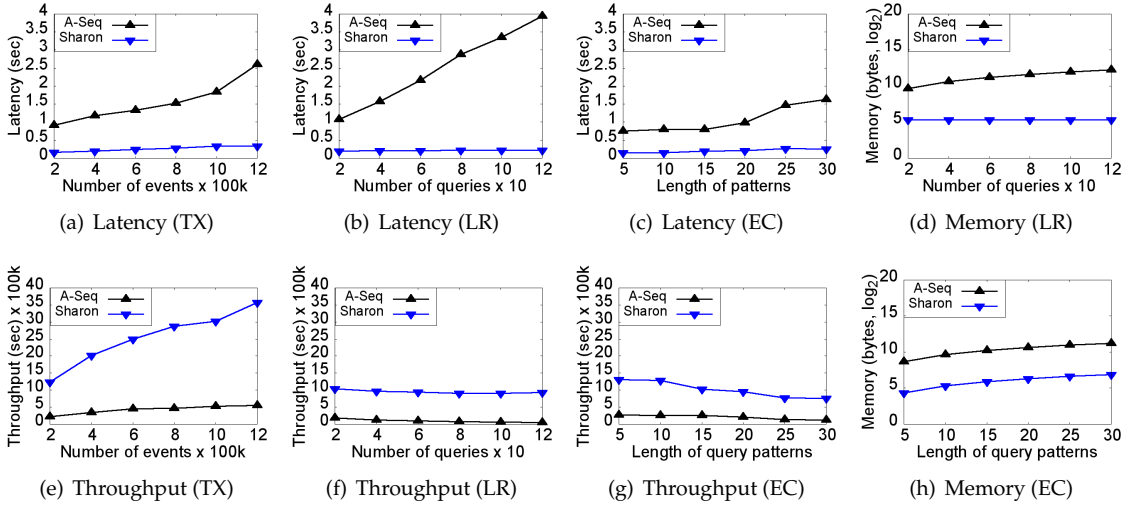
*Flink* not only constructs all event sequences but also computes each query independently from other queries in the workload. Flink fails for more than 6k events per window.

The event sequence construction step has polynomial time complexity in the number of events [29, 24] and may jeopardize real-time responsiveness for high-rate event streams (Figure 13). Thus, these two-step approaches cannot be effective for time-critical processing of high-rate streams.

**Online Approaches.** The online approaches (A-Seq and SHARON) perform similarly for such low-rate streams. They achieve five orders of magnitude speed-up compared to SPASS for 7k events per window because they aggregate event sequences without first constructing these sequences.

Figure 14 evaluates the online approaches against high-rate streams. We vary the number of events per window, the number of queries, and the length of their patterns and measure latency, throughput and memory of the online approaches.

The *Sharon Executor* shares event sequences aggregation among all queries in the workload according to an optimal sharing plan that is computed based on an expanded SHARON graph (Section 7). The latency of SHARON and A-Seq grows linearly in the number of queries. SHARON



**Figure 14:** Online approaches (Taxi (TX), Linear Road (LR), and e-commerce (EC) data sets)

achieves from 5-fold to 18-fold speed-up compared to *A-Seq* when the number of queries increases from 20 to 120. Indeed, the more queries share their aggregation results, the fewer aggregates are maintained and the more events can be processed by the system (Figures 14(b) and 14(f)). SHARON requires up to two orders of magnitude less memory than *A-Seq* for 120 queries (Figure 14(d)). For low parameter values, SHARON defaults to *A-Seq* due to limited sharing opportunities.

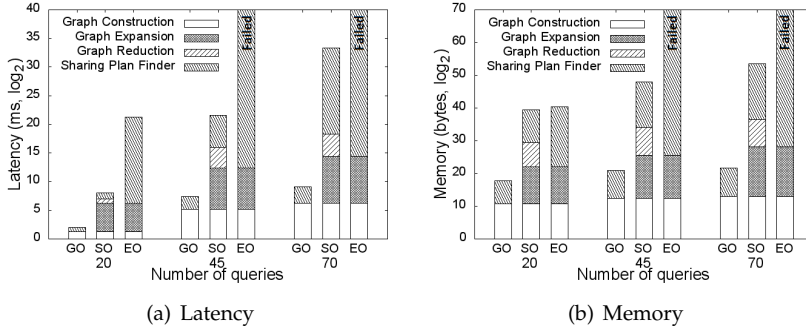
While SHARON processes each event by each shared pattern exactly once, each event can provoke repeated computations in *A-Seq*. Thus, the gain of SHARON grows linearly in the number of events per window. SHARON wins from 5-fold to 7-fold with respect to latency and throughput when the number of events increases from 200k to 1200k (Figures 14(a) and 14(e)). Similarly, the speed-up of SHARON grows linearly from 4-fold to 6-fold with the increasing length of patterns (Figure 14(c)). SHARON requires 20-fold less memory than *A-Seq* if the pattern length is 30 (Figure 14(h)).

Based on the experimental results in Figures 13 and 14, we conclude that the latency, throughput and memory utilization of event sequence aggregation can be considerably reduced by the seamless integration of *shared* and *online* optimization techniques as proposed by our SHARON approach to enable real-time in-memory event sequence aggregation.

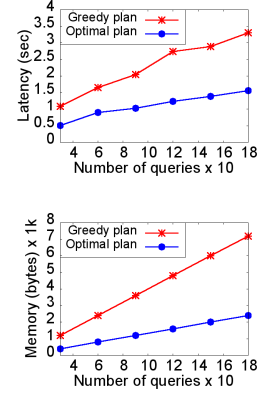
### 8.3 Sharon Optimizer

In Figure 15 we compare three optimizer solutions, while varying the number of queries. Each bar is segmented into phases as described below.

The *Greedy Optimizer* consists of the following two phases: SHARON graph construction (Section 4) and the GWMIN plan finder. In the worst case, both phases have polynomial latency and linear memory. However, our experiments show that on average more time and space is required to construct the SHARON graph than to run GWMIN. For 70 queries, 90% of the time is spent con-



**Figure 15:** SHARON optimizer (SO) versus greedy optimizer (GO) and exhaustive optimizer (EO) (E-commerce query workload)



**Figure 16:** Sharing plan quality (Taxi data set)

structing the graph.

The *Exhaustive Optimizer* consists of three phases, namely, SHARON graph construction, graph expansion (Section 7), and an exhaustive search that traverses the entire search space to find an optimal plan. Thus, its latency and memory costs grow exponentially in the number of queries. The exhaustive optimizer fails to terminate for more than 20 queries. For 20 queries, its latency is 4 orders of magnitude higher than the latency of the greedy optimizer.

The *Sharon Optimizer* consists of four phases, namely, SHARON graph construction, graph expansion, graph reduction, and the sharing plan finder that returns an optimal plan (Sections 4–7). While its time and space complexity is exponential in the worst case (Equation 13), its latency and memory usage are reduced by our pruning principles compared to the exhaustive optimizer. On average, 36% of the sharing candidates are pruned from the expanded SHARON graph, which is 99% of the plan finder search space. For 20 queries, SHARON outperforms the exhaustive optimizer by three orders of magnitude with respect to latency and by two orders of magnitude regarding memory usage.

Our SHARON plan finder traverses the entire valid space to find an optimal plan. In contrast, GWMIN greedily selects one candidate with the highest benefit and eliminates its adjacent candidates from further consideration. For example, for 70 queries, the latency of SHARON is three orders of magnitude higher, while its memory usage is two orders of magnitude larger compared to the greedy optimizer.

**Sharing Plan Quality.** The greedy optimizer tends to return a sub-optimal sharing plan for two reasons. One, it greedily selects a candidate  $v$  with the maximal benefit in each step. By deciding to share  $v$  it excludes all candidates adjacent to  $v$  even though they may be more beneficial to share than  $v$  alone. Two, the greedy optimizer does not resolve sharing conflicts (Section 7). However, the sharing opportunities in the original SHARON graph may be rather limited (Figure 4).

In Figure 16, we vary the number of queries and compare the latency and memory consumption of our SHARON executor when guided by a greedily chosen plan versus an optimal plan. We run these experiments on the Taxi real data set. The latency of the SHARON executor is reduced 2–fold and its memory consumption decreases 3–fold when 180 queries are processed according to

an optimal plan compared to a greedily chosen plan. Thus, an optimal plan ensures real-time, light-weight event sequence aggregation.

## 9 Related Work

**Complex Event Processing (CEP)** approaches such as SASE [4, 29], Cayuga [10], and ZStream [22] support both event aggregation and event sequence detection over streams. SASE and Cayuga employ a Finite State Automaton (FSA)-based query execution paradigm, meaning that each event query is translated into an FSA. Each run of an FSA corresponds to a query match. In contrast, ZStream translates an event query into an operator tree that is optimized based on rewrite rules. However, these approaches evaluate *each query independently from other queries* in the workload – causing both repeated computations and replicated storage in multi-query settings. Furthermore, they do not optimize event sequence aggregation queries – which is the focus of our work. Thus, they require event sequence construction prior to their aggregation. Since the number of event sequences is polynomial in the number of events per window [29, 24], this *two-step approach* introduces long delays for high-rate streams (Section 8).

In contrast, A-Seq [24] defines *online* event sequence aggregation that eliminates event sequence construction. It incrementally maintains an aggregate for each pattern and discards an event once it updated the aggregates. We leverage this idea in our executor (Section 3). However, A-Seq has no optimizer to decide which patterns should be shared by which queries. Thus, A-Seq does not share event sequence aggregation. GRETA [23] extends A-Seq by nested Kleene patterns and expressive predicates at the cost of storing of all matched events. Similarly to A-Seq, GRETA optimizes *single* queries.

**CEP Multi-Query Optimization (MQO)** approaches such as SPASS [25], E-Cube [21], and RUMOR [15] propose event sequence sharing techniques. SPASS exploits event correlation in an event sequence to determine the benefit of shared event sequence construction. E-Cube defines a concept and a pattern hierarchy of event sequence queries and develops both top-down and bottom-up processing of patterns based on the results of other patterns in the hierarchy. RUMOR proposes a rule-based MQO framework for traditional RDBMS and stream processing systems. It defines a set of rules to merge NFAs representing different event queries. However, no optimization techniques for online aggregation of event sequences are proposed by the approaches above. They too construct all event sequences prior to their aggregation. Event sequence construction degrades system performance.

**Data Streaming.** Streaming approaches typically support incremental aggregation [14, 17, 7, 20, 11, 19, 27, 31, 32]. Some of them are shared. However, they solve an orthogonal problem. Namely, they enable shared aggregation given different windows, predicates, and group-by clauses [14, 17, 7, 20]. Thus, they could be plugged into our approach as described in Section 7. However, in contrast to SHARON, many of them aggregate only *raw input events for single-stream queries* [14, 20, 19]. Others evaluate simple Select-Project-Join queries with window semantics over data streams [17]. They do not support CEP-specific operators such as event sequence that treat the order of events as

a first-class citizen. Typically, they require the *construction of join results* prior to their aggregation.

**Multi-Query Optimization** techniques include materialized views [9] and common sub-expression sharing [8, 12] in relational databases. However, these approaches do not have the temporal aspect prevalent for CEP queries. Thus, they neither focus on event sequence computation nor their aggregation. Furthermore, they assume that the data is statically stored on disk prior to processing. They neither target in-memory execution nor real-time responsiveness.

## 10 Conclusions and Future Work

Our SHARON approach is the first to enable *shared online* event sequence aggregation. The SHARON optimizer encodes sharing candidates, their benefits and conflicts among them into the SHARON graph. Based on the graph, we define effective candidate pruning principles to reduce the search space of sharing plans. Our sharing plan finder returns an optimal plan to guide the executor at runtime. SHARON achieves an 18-fold speed-up compared to state-of-the-art approaches.

In the future, we plan to further investigate event sequence aggregation sharing for dynamic workloads to produce a new optimal sharing plan on the fly and migrate from the old to the new sharing plan with minimal overhead. Another interesting direction for future work is to leverage modern distributed multi-core clusters of machines to further improve the scalability of shared online event sequence aggregation.

## References

- [1] Apache Flink. <https://flink.apache.org/>.
- [2] Uber Releases Hourly Ride Numbers In New York City To Fight De Blasio. <https://techcrunch.com/2015/07/22/uber-releases-hourly-ride-numbers-in-new-york-city-to-fight-de-blasio/>.
- [3] Unified New York City Taxi and Uber data. <https://github.com/toddwschneider/nyc-taxi-data>.
- [4] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.
- [5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994.
- [6] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In *VLDB*, pages 480–491, 2004.
- [7] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

- 
- [8] U. S. Chakravarthy and J. Minker. Multiple query processing in deductive databases using query graphs. In *VLDB*, pages 384–391, 1986.
  - [9] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190 – 200, 1995.
  - [10] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.
  - [11] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid. Incremental evaluation of sliding-window queries over data streams. *IEEE Trans. on Knowl. and Data Eng.*, 19(1):57–72, Jan. 2007.
  - [12] G. Giannikis, P. Unterbrunner, J. Meyer, G. Alonso, D. Fauser, and D. Kossmann. Crescendo. In *SIGMOD*, pages 1227–1230, 2010.
  - [13] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.
  - [14] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Three-level processing of multiple aggregate continuous queries. In *ICDE*, pages 929–940, 2012.
  - [15] M. Hong, M. Riedewald, C. Koch, J. Gehrke, and A. Demers. Rule-based multi-query optimization. In *EDBT*, pages 120–131, 2009.
  - [16] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
  - [17] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.
  - [18] C. Lei and E. A. Rundensteiner. Robust distributed query processing for streaming data. *ACM Trans. Database Syst.*, 39(2):17:1–17:45, 2014.
  - [19] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322, 2005.
  - [20] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: Efficient evaluation of sliding window aggregates over data streams. In *SIGMOD*, pages 39–44, 2005.
  - [21] M. Liu, E. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *SIGMOD*, pages 889–900, 2011.
  - [22] Y. Mei and S. Madden. ZStream: A cost-based processor for adaptively detecting composite events. In *SIGMOD*, pages 193–206, 2009.

- [23] O. Poppe, C. Lei, E. A. Rundensteiner, and D. Maier. GRETA: Graph-based Real-time Event Trend Aggregation. In *VLDB*, pages 80–92, 2018.
- [24] Y. Qi, L. Cao, M. Ray, and E. A. Rundensteiner. Complex event analytics: Online aggregation of stream sequence patterns. In *SIGMOD*, pages 229–240, 2014.
- [25] M. Ray, C. Lei, and E. A. Rundensteiner. Scalable pattern sharing on event streams. In *SIGMOD*, pages 495–510, 2016.
- [26] S. Sakai, M. Togasaki, and K. Yamazaki. A note on greedy algorithms for the maximum weighted independent set problem. *Discrete Appl. Math.*, 126(2-3):313–322, 2003.
- [27] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. In *VLDB*, pages 702–713, 2015.
- [28] E. Wu, Y. Diao, and S. Rizvi. High-performance Complex Event Processing over streams. In *SIGMOD*, pages 407–418, 2006.
- [29] H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in CEP. In *SIGMOD*, pages 217–228, 2014.
- [30] J. Zhang, Y. Wang, and D. Yang. CCSpan: Mining closed contiguous sequential patterns. *Knowledge-Based Systems*, 89:1–13, 2015.
- [31] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *SIGMOD*, pages 299–310, 2005.
- [32] R. Zhang, N. Koudas, B. C. Ooi, D. Srivastava, and P. Zhou. Streaming multiple aggregations using phantoms. In *VLDB*, pages 557–583, 2010.
- [33] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, pages 431–442, 2004.

## Appendix

### A Sharable Pattern Detection

To detect sharable patterns (Definition 3), we deploy a version of the CCSpan algorithm [30]. In this section, we first describe the original CCSpan algorithm. We then justify our changes and provide the modified algorithm.

**Original CCSpan Algorithm.** CCSpan stands for Closed Contiguous Sequential Pattern mining. A pattern is considered to be *frequent* if it appears in more input sequences than the given support. A pattern is *contiguous* if it is not interrupted by other patterns in the input sequences. Lastly, a pattern is called *closed* if it cannot be further extended.

---

**Algorithm 7** Modified CCSpan algorithm
 

---

**Input:** A query workload  $Q$

**Output:** A hash table  $S$  mapping a sharable pattern  $p$  to a set of queries in  $Q_p$  that contain  $p$

```

1:  $H, S \leftarrow$  empty hash tables
2: for each  $q$  in  $Q$  do
3:    $l \leftarrow q.pattern.length$ 
4:   for each  $end = 0; end \leq l; end++$  do
5:     for each  $start = 0; start \leq end; start++$  do
6:        $p = q.pattern.substring(start, end)$ 
7:       if  $p.length > 1$  then
8:          $Q_p \leftarrow H.get(p); Q_p.add(q); H.put(p, Q_p)$ 
9: for each  $p$  in  $H$  do
10:   $Q_p \leftarrow H.get(p)$ 
11:  if  $Q_p.size > 1$  then  $S.put(p, Q_p)$ 
12: return  $S$ 

```

---

CCSpan adopts a pattern growth algorithm that records the pattern's occurrence in the input sequences. That is, starting from length 1, the algorithm recursively extends the patterns to their maximal length. CCSpan has linear time complexity in the number of input sequences assuming that the maximal length of input sequences is a small constant.

**Modified CCSpan Algorithm.** Since shorter sequences can be shared between more queries than longer sequences, we detect not only frequent closed (or longest) sequences but also their sub-sequences. However, sharing a sequence of length one is not beneficial. Thus, we alter the original CCSpan algorithm to detect all *frequent contiguous sequential patterns* of length  $l > 1$ . A pattern is considered to be *frequent* if it appears in more than one query.

The modified CCSpan algorithm (Algorithm 7) consumes the query workload  $Q$  and returns a hash table  $S$  that maps each sharable pattern  $p$  to the set of queries  $Q_p \subseteq Q$  that contain  $p$ . Two empty hash tables  $H$  and  $S$  are initialized in Line 1.  $S$  contains all sharable patterns, while  $H$  maintains all patterns, i.e.,  $S \subseteq H$ . For each query  $q \in Q$ , the algorithm considers each sub-pattern  $p$  of the pattern of  $q$  (Lines 3–6). If the length of  $p$  is greater than one, the query  $q$  is added to the set of queries  $Q_p$  that  $p$  appears within (Line 7–8). Lastly, we access each pattern  $p$  in the hash table  $H$  and if  $p$  appears more than one query, the pattern  $p$  and its respective queries  $Q_p$  are added to the result  $S$  (Lines 9–11). The hash table  $S$  is returned in Line 12.

**Complexity Analysis.** Let  $n$  be the number of queries in  $Q$  and  $l$  be the maximal length of a pattern. Then, the three for-loops in Lines 2–8 are called  $O(nl^2)$  times. In addition, the for-loop in Lines 9–11 iterates  $O(nl)$  times. All operations on the hash tables  $H$  and  $S$  happen in constant time. In summary, the time complexity is linear in  $n$ , i.e.,  $O(nl^2) + O(nl) = O(n)$  since  $l$  is a small constant in practice. The space complexity is determined by the size of the hash tables  $H$  and  $S$ . The number of stored patterns is  $O(nl)$ . Each pattern is mapped to  $O(n)$  queries. In summary, the space complexity is quadratic in  $n$ , i.e.,  $O(n^2l) = O(n^2)$ .



---

**Algorithm 8** GWMIN algorithm

---

**Input:** A weighted graph  $G = (V, E)$ 
**Output:** An independent set  $IS$ 

```

1:  $IS \leftarrow \emptyset$ ;  $i \leftarrow 0$ ;  $G_i \leftarrow G$ 
2: while  $V(G_i) \neq \emptyset$  do
3:    $max \leftarrow 0$ 
4:   for each  $v$  in  $V(G_i)$  do
5:      $new\_max \leftarrow \frac{weight(v)}{degree_{G_i}(v)+1}$ 
6:     if  $new\_max > max$  then
7:        $max \leftarrow new\_max$ ;  $v_i \leftarrow v$ 
8:    $IS \leftarrow IS \cup \{v_i\}$ ;  $G_{i+1} \leftarrow G_i[V(G_i) \setminus \mathcal{N}_{G_i}^+(v_i)]$ ;  $i++$ 
9: return  $IS$ 

```

---

## B GWMIN Algorithm

The GWMIN algorithm finds the MWIS in a graph [26]. GWMIN stands for Greedy Minimum degree algorithm for Weighted graphs. Algorithm 8 consumes a weighted graph  $G = (V, E)$  and returns its independent set  $IS$ . At the beginning, the set  $IS$  is empty, the iteration counter  $i = 0$ , and the graph in  $i^{th}$  iteration  $G_i = G$  (Line 1). In each iteration, the algorithm selects the vertex  $v$  with the maximal ratio

$$\frac{weight(v)}{degree_{G_i}(v) + 1}$$

where  $weight(v)$  is the weight and  $degree_{G_i}(v)$  is the degree of  $v$  in  $i^{th}$  iteration (Lines 3–7). This vertex  $v$  is added to the independent set  $IS$ ,  $v$  and its neighbors are removed from the graph  $G$  (Line 8). Once the graph  $G$  is empty, the algorithm returns the independent set  $IS$  (Lines 2, 9).

**Complexity Analysis.** The for-loop in Lines 4–7 is called  $O(|V|)$  times. The time complexity of removing a vertex  $v$  and its neighbors from the graph is  $O(|E|)$ . Thus, the time complexity of one iteration of the while-loop in Lines 2–9 is  $O(|V| + |E|)$ . This while-loop is called  $O(|V|)$  times. In summary, the time complexity is  $O(|V|(|V| + |E|))$ . The space complexity is linear in the size of the graph  $G$  and its independent set  $IS$ , i.e.,  $O(|V| + |E|)$ .