Online Appendix to Exploiting *k*-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams

SHIVNATH BABU, UTKARSH SRIVASTAVA, and JENNIFER WIDOM
Stanford University

A. DETAILED EXAMPLES OF *K*-CONSTRAINTS

One application of a data stream management system is to support traffic monitoring applications for a large network such as the backbone network of an Internet Service Provider (ISP) [Caceres et al. 2000]. Such a system might run continuous queries over streams of packet headers, flow records, and performance measurements to monitor network health, detect equipment failures and attacks, etc. We describe a number of k-constraints that arise in this application.

Example A.1. Some routers on the network are usually configured to report traffic statistics for recently expired flows [NETFLOW 2003]. (Here a flow denotes the collection of packets sent in one TCP connection from a source to a destination.) A typical setting expires a flow and outputs a flow record when a "finish" packet arrives in the flow (voluntary closure) or when no packets arrive in the flow for a timeout interval of 15 s (forced closure). Consider the `stop_time` attribute of the resulting flow record stream which denotes the arrival time of the last packet in the flow. The values of this attribute are nondecreasing over voluntarily closed flows, but forced closures create a *scrambling* of `stop_time` values in the stream as a whole. If the router reports at most n flows every second to limit the bandwidth consumed by monitoring applications, any two `stop_time` values that are out of order in the flow record stream will be at most $15n$ tuples apart, so the flow record stream satisfies OA($15n$) over the `stop_time` attribute.

Example A.2. Network measurement streams are often transmitted via the UDP protocol instead of the more reliable but higher cost TCP protocol to minimize the monitoring load placed on the network [NETFLOW 2003]. Since UDP can deliver packets out of order, it can create some scrambling in values of stream attributes that are otherwise ordered. For example, if the minimum and maximum network delay from the data collection device to the stream processing system are d_{min} and d_{max} seconds, respectively, and the device limits its bandwidth consumption to n tuples per second, then any two tuples that arrive out of transmission timestamp order at the processing system will be at most $(d_{max} - d_{min})n$ tuples apart, creating an OA($(d_{max} - d_{min})n$) constraint on the transmission timestamp.

Example A.3. An interesting continuous query in network monitoring, termed *trajectory sampling*, maintains a summary of routes taken by packets through the network [Duffield and Grossglauser 2000]. To support this query, devices on links across the network sample packets continuously with the property that a packet chosen by any one device will be chosen by all other devices that observe the packet [Duffield and Grossglauser 2000]. Consider the resulting merged stream of tuples with schema (`pkt_id`,`link_id`) sent by these devices. `pkt_id` is a unique identifier for a packet and `link_id` represents a link where the packet was observed [Duffield and Grossglauser 2000]. If there are m devices sampling at the rate of s packets per second, and d represents the maximum delay of packets through the network, then any two tuples in the stream with the same value of `pkt_id` are separated by no more than $m \times s \times d$ tuples with a different value of `pkt_id`, creating a CA($m \times s \times d$) constraint on the `pkt_id` attribute.

Example A.4. If the amount of traffic destined to a *peer* ISP on a network link L exceeds a certain threshold, a network analyst might want to drill down into a sample of this traffic. A continuous query for this purpose joins two streams: S_1(`pkt_hdr`, `peer_id`, `timestamp`) and S_2(`peer_id`, `num_bytes`, `timestamp`). S_1 is a stream of packets sampled from L containing the packet header (`pkt_hdr`), the destination peer ISP (`peer_id`), and the packet arrival time at the granularity of 5-min intervals (`timestamp`). S_2 is a stream of measurements containing the total observed traffic (`num_bytes`) on L destined to peer ISP (`peer_id`) for each 5-min interval (`timestamp`). Clearly each packet on S_1 is destined to a unique peer and arrives at a unique 5-min interval making $S_1 \bowtie S_2$ a many-one natural join. Furthermore, if the number of peer ISPs is less than 25, ignoring the effects of computational and network latency for simplicity, the unique joining S_2 tuple of any tuple $s_1 \in S_1$ will have arrived within 25 S_2 tuples that arrive after s_1, providing the basis for a RIDS(k) constraint over the many-one join from S_1 to S_2.

B. BASIC QUERY PROCESSING ALGORITHM FOR DAG-SHAPED JOIN GRAPHS

We describe how the basic query processing algorithm for DT-shaped joins graphs needs to be extended to handle DAG-shaped join graphs. Theorem 3.2 does not hold when a query's join graph is DAG-shaped instead of DT-shaped, as illustrated by the following simple example.

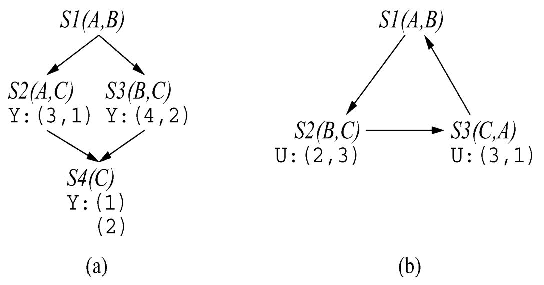

Fig. 18. Join graphs and synopses used in examples for DAG-shaped and cyclic join graphs.

```
/* Insert tuple $s$ into the synopsis of stream $S$ */
1. Procedure $\mathcal{S}(S).InsertTuple(s)$ {
2 – 16. Lines 2 – 16 from Figure 2
17.   /* Otherwise, $S$ has no children, or all child tuples of $s$ are present in the respective
18.     $Yes$ components. Handle directed acyclic subgraphs in $G_S(Q)$ */
19.   For each stream $X \in G_S(Q)$, $X \notin \{S \cup Children(S)\}$ {
20.     For each pair of vertex disjoint paths $P_1, P_2$ from $S$ to $X$ {
21.       Let $P_1 = S, A_1, A_2, \ldots, A_n, X = A_{n+1},\ n \geq 1$,
22.           $P_2 = S, B_1, B_2, \ldots, B_m, X = B_{m+1},\ m \geq 1$,
23.           with $A_i \neq B_j, 1 \leq i \leq n, 1 \leq j \leq m$;
24.       Tuple $x_1 = s$;
25.       For (int $i = 1$; $i \leq n+1$; $i$ ++) {
26.         if($x_1 \rightarrow \mathcal{S}(A_i).Yes \neq \phi$)
27.           $x_1 = x_1 \rightarrow \mathcal{S}(A_i).Yes$;
28.         else $x_1 = x_1 \rightarrow \mathcal{S}(A_i).Unknown$; }
29.       Tuple $x_2 = s$;
30.       For (int $i = 1$; $i \leq m+1$; $i$ ++) {
31.         if($x_2 \rightarrow \mathcal{S}(B_i).Yes \neq \phi$)
32.           $x_2 = x_2 \rightarrow \mathcal{S}(B_i).Yes$;
33.         else $x_2 = x_2 \rightarrow \mathcal{S}(B_i).Unknown$; }
34.       if($x_1 \neq x_2$) {
35.         $\mathcal{S}(S).No.InsertTuple(s)$; return; }}}
36.   $\mathcal{S}(S).Yes.InsertTuple(s)$; }
```

Fig. 19. Procedure invoked when a tuple s arrives in stream S in a DAG-shaped join graph.

Example B.1. Consider a query Q with the DAG-shaped join graph and synopses shown in Figure 18(a). Suppose tuple $s = (3, 4)$ arrives in stream S_1. Although both child tuples of s are in the respective *Yes* components, clearly $s \rhd\!\!< G_{S_1}(Q)$ is empty.

If there exist two or more vertex-disjoint paths from stream S to stream $X \in G_S(Q)$, denoted $P_1, P_2, \ldots, P_l$, $l \geq 2$, then $s \rhd\!\!< G_S(Q)$ is nonempty only if s joins with the same tuple $x \in X$ for each of these chains of many-one joins P_i, $1 \leq i \leq l$, from S to X. Notice that each pair of these vertex-disjoint paths P_i, P_j, $i \neq j$, produces a directed acyclic subgraph in $G_S(Q)$.

The modified algorithm for synopsis maintenance in DAG-shaped graphs is shown in Figure 19. Lines 17–35 in Figure 19 provide the extra checks to handle directed acyclic subgraphs of $G_S(Q)$. These checks are invoked only if all of s's child tuples are in the corresponding child *Yes* components. The procedures in Figures 3, 4, and 5 remain unchanged for DAG-shaped join graphs. Result generation proceeds exactly as in the DT-shaped case since Theorems 3.4 and 3.5 also hold for DAG-shaped join graphs.

C. BASIC QUERY PROCESSING ALGORITHM FOR CYCLIC JOIN GRAPHS

We use the following example to illustrate the problems that cyclic join graphs introduce.

Example C.1. Consider the cyclic join graph and synopses in Figure 18(b). Tuple $s_3 = (3, 1)$ is in $\mathcal{S}(S_3).Unknown$ since its child tuple in S_1 has not arrived yet, and tuple $s_2 = (2, 3)$ is in $\mathcal{S}(S_2).Unknown$ since its child tuple $s_3 \in \mathcal{S}(S_3).Unknown$. Suppose tuple $s_1 = (1, 2)$ arrives in S_1. If we follow the basic query processing algorithm for DT-shaped or DAG-shaped join graphs, s_1 will be inserted into $\mathcal{S}(S_1).Unknown$ since s_1's child tuple $s_2 \in \mathcal{S}(S_2).Unknown$. This step would lead to a deadlock because of the cyclic dependency among tuples s_1, s_2, and s_3. Notice that s_1, s_2, and s_3 join with each other resulting in $s_1 \rightarrow G_{S_1}(Q) \neq \phi$, $s_2 \rightarrow G_{S_2}(Q) \neq \phi$, and $s_3 \rightarrow G_{S_3}(Q) \neq \phi$. Thus, s_1, s_2, and s_3 should be inserted into the respective *Yes* components by Definition 3.1, and result tuple $(1, 2, 3)$ must be produced.

Now suppose tuple $s'_1 = (4, 2)$ arrives in S_1. Although s'_1's child tuple is in *Yes*, clearly $s'_1 \rightarrow G_{S_1}(Q) = \phi$. This problem is similar to the problem introduced by DAG-shaped join graphs (Example B.1).

Figure 20 gives the extended query processing algorithm that handles cyclic join graphs. The main modification is that before we insert a tuple s into $\mathcal{S}(S).Unknown$ because its child tuple s' is in $\mathcal{S}(S').Unknown$, we need to check if a cyclic relationship exists between S and S' that permits us to move all tuples in the cycle to their respective *Yes* or *No* synopsis components. An additional modification that is similar in spirit to the modification required for DAG-shaped graphs is needed to handle the fact that a cycle introduces a pair of vertex-disjoint paths from S to S for each stream S in the cycle. For result generation, there may be more than one minimal cover in a cyclic join graph, so we need to compute the minimal covers and then apply Theorem 3.4 which holds for cyclic join graphs as well.

D. PROOFS OF THEOREMS

D.1 Some Useful Lemmas

LEMMA D.1. *A tuple $s \in S$ joins with at most one tuple in each stream $R \in G_S(Q)$, $R \neq S$.*

PROOF. By induction on the length of the unique directed path from S to R, denoted $l_{S\rightarrow R}$. Clearly, $l_{S\rightarrow R} \geq 1$. If $l_{S\rightarrow R} = 1$, then R is a child of S, and the theorem holds because of the many-one join from S to R. This step forms the basis of the induction. As the induction hypothesis, suppose the theorem holds whenever $l_{S\rightarrow R} < n$. If $l_{S\rightarrow R} = n$, $n > 1$, consider the first stream T in the

```
/* Insert tuple $s$ into the synopsis of stream $S$ */
1.  Procedure $\mathcal{S}(S).InsertTuple(s)$ {
2 − 8. Lines 2 − 8 from Figure 2
9.      For each stream $R \in Children(S)$ {
10.         /* If the child tuple of $s$ in child stream $R$ is present in $\mathcal{S}(R)$, we need to check
11.             for cyclic dependencies */
12.         if ($s \rightarrow (\mathcal{S}(R).Yes \cup \mathcal{S}(R).Unknown) \neq \phi$) {
13.             if (there exists a directed path from $R$ to $S$) {
14.                 Boolean missing_child_tuple = false;
15.                 For each path from $R$ to $S$:
16.                         $R_1 = R, R_2, \ldots, R_n, R_{n+1} = S$ {
17.                     Tuple $x = s$;
18.                     For (int $i = 1$; $i \leq n$; $i$++) {
19.                         if($x \rightarrow \mathcal{S}(R_i).Yes \neq \phi$)
20.                             $x = x \rightarrow \mathcal{S}(R_i).Yes$;
21.                         else if($x \rightarrow \mathcal{S}(R_i).Unknown \neq \phi$)
22.                             $x = x \rightarrow \mathcal{S}(R_i).Unknown$;
23.                         else {
24.             /* The cyclic dependency cannot be resolved because the tuple in $R_i$
25.                 joining with $s$ is yet to arrive */
26.                             missing_child_tuple = true; break; } }
27.             /* if $s$ completes the set of tuples in this cyclic dependency, then check
28.                 whether the tuples join. */
29.                     if ($i = n + 1$ and $x \rightarrow s = \phi$) {
30.                         $\mathcal{S}(S).No.InsertTuple(s)$; return; } }
31.                 if (missing_child_tuple = true) {
32.             /* The cyclic dependency cannot be resolved because one or more tuples
33.                 are yet to arrive */
34.                     $\mathcal{S}(S).Unknown.InsertTuple(s)$; return; } }
35.             else {
36.             /* There is no cycle involving $S \rightarrow R$ */
37.                 if ($s \rightarrow \mathcal{S}(R).Unknown \neq \phi$) {
38.                 /* the child tuple is present in $\mathcal{S}(R).Unknown$ */
39.                     $\mathcal{S}(S).Unknown.InsertTuple(s)$; return; } } }
40.         /* else the child tuple of $s$ in $R$ has not arrived. */
41.         else { $\mathcal{S}(S).Unknown.InsertTuple(s)$; return; }}
42.     /* Handle directed acyclic subgraphs in $G_S(Q)$ */
43 − 59. Lines 19 − 35 from Figure 19
60.     $\mathcal{S}(S).Yes.InsertTuple(s)$; }
```

Fig. 20. Procedure invoked when a tuple s arrives in stream S in a cyclic join graph.

directed path from S to R. A tuple $s \in S$ can join with at most one tuple $t \in T$. Since $l_{T \rightarrow R} < n$, by the induction hypothesis t joins with at most one tuple in R. By transitivity, s can join with at most one tuple in R. Hence, the theorem holds for $l_{S \rightarrow R} = n$. □

LEMMA D.2. *If a tuple $s \in S$ is part of a query result tuple, then $s \in \mathcal{S}(S).Yes$.*

PROOF. Let t be the query result tuple that s is part of. Consider the projection of t onto the streams in $G_S(Q)$, denoted α_s. The existence of α_s shows that $s \ltimes G_S(Q) \neq \phi$, which means that $s \in \mathcal{S}(S).Yes$ by Definition 3.1. □

LEMMA D.3. *Consider a stream R that is reachable from a stream S in $G(Q)$ by following directed edges. The insertion of a tuple $s \in S$ into $\mathcal{S}(S)$.Yes cannot happen before the insertion of its unique joining tuple $r \in R$ into $\mathcal{S}(R)$.Yes. (We say an event e_1 happens before an event e_2 if the set of tuples that have been processed completely when e_1 happens is a strict subset of the set of tuples that have been processed completely when e_2 happens.)*

PROOF. The proof follows from Definition 3.1 of *Yes* synopsis components. □

D.2 Proof of Theorem 3.2

Definition 3.1 says that tuple $s \in \mathcal{S}(S)$.*Yes* for a stream S if $s \rhd\!\!< G_S(Q) \neq \phi$. Theorem 3.2 says that $s \in \mathcal{S}(S)$.*Yes* for a DT-shaped join graph $G_S(Q)$ if s satisfies all filter predicates on S, and all children of s are in the respective *Yes* components. We have to prove that the statements in Definition 3.1 and those in Theorem 3.2 are equivalent for DT-shaped join graphs.

We will first prove the forward direction: If $s \rhd\!\!< G_S(Q) \neq \phi$, then s satisfies all filter predicates on S, and all child tuples of s are in the respective *Yes* components. If $s \rhd\!\!< G_S(Q) \neq \phi$, then clearly s satisfies all filter predicates on S. The rest of the proof assumes this fact. The proof is by induction on the length of the longest directed path starting at S, denoted l_S. If $l_S = 0$, then S has no children, and the claim holds. This step forms the basis of the induction. As the induction hypothesis, let the claim hold whenever $l_S < n$. We now consider a stream S with $l_S = n$, and a tuple $s \in S$ such that $s \rhd\!\!< G_S(Q) \neq \phi$. By Lemma D.1, s joins with a unique tuple in each stream $R \in G_S(Q)$, $R \neq S$. Therefore, s must generate a unique result tuple in $G_S(Q)$, denoted α_s (α_s is a joined tuple containing one tuple each from all streams in $G_S(Q)$). Now consider stream $R \in Children(S)$. If $r \in R$ is the unique child tuple of s, then α_s must contain r as its component tuple from R. We claim $r \rhd\!\!< G_R(Q) \neq \phi$. The proof is straightforward. By definition, $G_R(Q) \subset G_S(Q)$ for DT-shaped graphs. Thus, r will join with the same tuples in streams $T \in G_R(Q)$, that are contained in α_s. Also, $l_R < n$ by property of DT-shaped graphs. Given $r \rhd\!\!< G_R(Q) \neq \phi$ and $l_R < n$, by the induction hypothesis we know that all child tuples of R are in the respective *Yes* components, which puts $r \in \mathcal{S}(R)$.*Yes*. Thus, we have proved that all child tuples of s are in the respective *Yes* components.

We will now prove the reverse direction: If s satisfies all filter predicates on S, and all child tuples of s are in the respective *Yes* components, then $s \rhd\!\!< G_S(Q) \neq \phi$. Again the proof is by induction on the length of the longest directed path starting at S, denoted l_S. If $l_S = 0$ the claim clearly holds. This step forms the basis of the induction. As the induction hypothesis, let the claim hold whenever $l_S < n$. We now consider a stream S with $l_S = n$. Let $R_1, R_2, \ldots, R_m$ be the children of S. Consider a tuple $s \in S$ that satisfies all filter predicates on S, and all child tuples of s are in the respective *Yes* components. For any child tuple $r_i \in R_i$ of s, $1 \leq i \leq m$, $r_i \in \mathcal{S}(R_i)$.*Yes* means that all child tuples of r_i are their *Yes* components. Since $l_{R_i} < n$ by property of DT-shaped graphs, by the induction hypothesis we know $r_i \rhd\!\!< G_{R_i}(Q) \neq \phi$. Consider the joined tuple t consisting of $s, \alpha_{r_1}, \alpha_{r_2}, \ldots, \alpha_{r_m}$, where α_{r_i} is the unique result tuple generated by r_i in $G_{R_i}(Q)$ (recall Lemma D.1). We claim that t is a result tuple of $G_S(Q)$.

The proof is straightforward. By property of DT-shaped graphs, no stream is common between $G_{R_i}(Q)$ and $G_{R_j}(Q)$, for $1 \leq i \leq m$, $1 \leq j \leq m$, and $i \neq j$, and there are no join predicates involving a stream in $G_{R_i}(Q)$ and a stream in $G_{R_j}(Q)$. Also, the union of all streams in $G_{R_i}(Q)$, $1 \leq i \leq m$, and S together constitute all streams in $G_S(Q)$. Since $r_i \rhd\!\!< G_{R_i}(Q) \neq \phi$, $1 \leq i \leq m$, we know that α_{r_i} satisfies the filter and join conditions over streams in $G_{R_i}(Q)$. The remaining filter predicates in $G_S(Q)$ are those over S, which are given to be satisfied by s. The remaining join predicates in $G_S(Q)$ are those involving S and one of its children, all of which are satisfied by $s, r_1, r_2, \ldots, r_m$ since $r_1, r_2, \ldots, r_m$ are the child tuples of s. Thus, t is a result tuple of $G_S(Q)$ which implies $s \rhd\!\!< G_S(Q) \neq \phi$. □

D.3 Proof of Theorem 3.3

We will prove that for a tuple $s \in S(\tau)$, if s fails a filter predicate on S or if a child tuple of s is in the respective *No* component (i.e., if Theorem 3.3 adds s to $\mathcal{S}(S)$.*No*), then $s \rhd\!\!< G_S(Q) = \phi$ at all times $\geq \tau$. Clearly, if s fails a filter predicate on S, $s \rhd\!\!< G_S(Q) = \phi$. We will assume this fact in the rest of the proof.

The proof is by induction on the length of the longest directed path starting at S, denoted l_S. If $l_S = 0$ the claim clearly holds. This step forms the basis of the induction. As the induction hypothesis, let the claim hold whenever $l_S < n$. We now consider a stream S with $l_S = n$. Let $r \in R(\tau)$ be the child tuple of s such that $r \in \mathcal{S}(R)$.*No* at time τ either because r fails a filter predicate on R or because a child tuple of r is in the respective *No* component. Since $l_R < n$ by property of DT-shaped graphs, the claim holds for $r \in R$. Thus, $r \rhd\!\!< G_R(Q) = \phi$ for all times $\geq \tau$. We will prove by contradiction that $s \rhd\!\!< G_S(Q) = \phi$ at all times $\geq \tau$. Suppose $s \rhd\!\!< G_S(Q) \neq \phi$ at time $\tau' \geq \tau$. Let α_s be the unique result tuple that s generates in $G_S(Q)$ at time τ' (recall from Section D.2 that α_s is a joined tuple containing one tuple each from all streams in $G_S(Q)$). By the property of DT-shaped graphs, $G_R(Q) \subset G_S(Q)$. Thus, the existence of α_s implies the existence of α_r, which will be the projection of tuple α_s on to the streams in $G_R(Q)$. The existence of α_r contradicts the fact that $r \rhd\!\!< G_R(Q) = \phi$. Thus, we have shown by contradiction that $s \rhd\!\!< G_S(Q) = \phi$ at all times $\geq \tau$, which completes the proof. □

D.4 Proof of Theorem 3.4

The proof is by contradiction. Suppose a query result tuple t is generated when a tuple s is inserted into the *Yes* synopsis component of a stream S such that S is not part of any minimal cover. Consider a minimal cover of Q, denoted ρ. Let R be a stream in ρ such that S is reachable from R. (R is not reachable from S. Otherwise, $\rho - \{R\} \cup \{S\}$ would be a minimal cover, which would give a contradiction.) Let r and s be the component tuples in t from streams R and S respectively. By Lemma D.2 $r \in \mathcal{S}(R)$.*Yes* and $s \in \mathcal{S}(S)$.*Yes*. By Lemma D.3 we know that the insertion of r into $\mathcal{S}(R)$.*Yes* cannot happen before the insertion of s into $\mathcal{S}(S)$.*Yes* which contradicts the fact that t is generated when s is inserted. (Given our definition of "happens before" in Lemma D.3, it is possible that neither the insertion of r into $\mathcal{S}(R)$.*Yes* nor the insertion of s into $\mathcal{S}(S)$.*Yes* happens before the other. However, since R is not reachable from S, we will always have to infer $s \in \mathcal{S}(S).Yes$ before we can infer $r \in \mathcal{S}(R).Yes$.) $\square$

D.5 Proof of Theorem 3.5

Theorem 3.5 has a straightforward proof from graph theory. $\square$

D.6 Proof of Theorem 5.2

We will prove that if the three conditions in Theorem 5.2 are satisfied for a tuple $s \in S$, then no future result tuple can have s as its component tuple from S. The proof is by contradiction. Assume that a future result tuple t has s as its component tuple from S. Since t is a future result tuple, t must contain at least one tuple that arrived after the conditions in Theorem 5.2 were satisfied. Without loss of generality, let this tuple be $r \in R$. By Lemma D.2, all component tuples of t belong to the respective *Yes* components of the streams, including all component tuples of t from streams in the minimal cover ρ in Theorem 5.2. From Condition C3 in Theorem 5.2, we can infer that $R \notin \rho$. Since ρ is a cover of $G(Q)$, there exists a stream $U \in \rho$ such that R is reachable from U. Let tuple $u \in U$ be the component tuple of t from U. From Conditions C2 and C3 in Theorem 5.2 we can infer that u was inserted into $\mathcal{S}(U).Yes$ before the arrival of r which contradicts Lemma D.3.

The astute reader might have noticed that the proof did not use the fact that $\rho \subseteq Parents(S)$. Although this condition is not necessary for Theorem 5.2 to hold, it gives us an efficient way to evaluate Condition C3 using CA(k) or OAP(k) constraints on attributes in *Parents*(S) involved in joins with S. $\square$