**OVERVIEW**

- Let's start with an example
- Understanding the challenges involved
- How do they exploit the stream properties
- An overview of k-Mon architecture
- The Basic Query execution algorithm
- Extensions to the above algorithm
- Results of each of the tricks used above
- A summary of the tricks
- Venues for further research

**EXAMPLE**

Incoming traffic (C)

Customer

ISP

Link to ISP's backbone (B)

Outgoing traffic (O)

Select sum(C.size)
From C[Range 10 min], B[Range 10 min], O[Range 10 min]
Where C.pid = B.pid and B.pid = O.pid

**Challenges in Exploiting Stream Properties**

- Is there a set of properties that can be useful across a wide variety of applications?
  - Basic constraints identified: many-one joins, stream based referential integrity, ordering and clustering.
- How to handle delays and reordering in the network traffic?
  - The notion of k-constraints, an adherence parameter
- Stream properties may change during the lifetime of a long running continuous query
  - Developed a k-mon architecture to continuously monitor and detect k useful constraints

**Stream Constraints**

- Join Constraints
  - Many-one joins
  - Referential integrity constraint on joins
- Ordered-Arrival Constraints
  - Have a rough ordering in one of the attributes
- Clustered-Arrival Constraints
  - Stream of tuples are roughly clustered on an attribute

**k-Mon Architecture**

New and continuous query

Query Registration

Initial query plan

Add/drop constraints, adjust k

Constraint Monitoring

Query Execution

Changes in k for monitored constraints

Potentially-useful constraints
Basic Query Execution Algorithm

- Notations
  - \( S_1 S_2 \), \( S_1 \) is the parent stream and \( S_2 \) the child stream
  - \( s_1 S_2 \), \( s_1 \) is the parent tuple and \( s_2 \) a unique tuple in \( S_2 \) as many-one join
  - \( G(Q) \) is a join graph

- Definitions
  - Consider a time \( T \) and a tuple \( s \). The synopsis \( S(S) \) has three components
    - Yes – \( s \) will contribute to the result set
    - No – \( s \) will never contribute to the result set
    - Unknown – we do not know

Synopsis Reduction

- Eliminating Tuples
  - If the stream \( S \) forms a minimal cover for \( G(Q) \), then any tuple \( s \) that belongs to \( S \) can be just “logically” inserted into \( S(S) \). Yes
  - Always eliminate No component for root stream synopses

- Eliminating Columns
  - Store only those attributes of \( S \) that are involved in joins with other streams
  - In \( S(S) \) store only those attributes involved in joins with Parents(\( S \))

How do we extend this to sliding windows

- Need to keep track of the order of tuples
  - Maintain the order information by tagging each tuple with a sequence number
  - By this we can calculate the following metrics useful for further filtering
    - Clustering distance: gives information about clusters within a stream
    - Scrambling distance: gives ordering information within a stream
    - Join distance: provides information for referential integrity between streams involved in a join
  - When a tuple drops out of a window, it can be either discarded or moved to \( S(S) \). No

Trick 1: Referential Integrity
Constraints over data streams (RIDS(\( k \)))

- Idea: On a many-one join \( S_1 S_2 \). When a tuple \( s_1 \) arrives on \( S_1 \) then its joining child tuple has arrived in \( S_2 \) or will arrive within \( k \) tuple arrivals on \( S_2 \).
- Implementing RIDS(\( k \)) Usage
  - Maintain a counter for the tuples that arrive in each stream
  - An extra sequence number to index \( S(S) \). Unknown for garbage collection
- Monitoring RIDS(\( k \))
  - If we overestimate \( k \) we get no gains in terms of memory usage
  - If we underestimate \( k \) we get false negatives
  - As data streams in “learn” the value of \( k \)

Experimental Analysis for RIDS(\( k \))

RIDS allows us to eliminate No components without any risk

Trick 2: Clustered Arrival
Constraint over data streams (CA(\( k \)))

- Idea: CA(\( k \)) holds on attribute \( A \) in stream \( S \) if, for every pair of tuples that belong to \( S \), the clustering distance over \( A \) between them is at most \( k \).
- Implementing CA(\( k \))
  - Need to maintain state for each parent \( S \) that is involved in a join and the corresponding cluster distance
  - Counter for the tuples that have arrived on \( S \)
- Monitoring CA(\( k \)) : Similar to monitoring and learning clustering distances within streams rather than join distances across streams
Experimental Analysis for CA(\(k\))

CA helps in eliminating tuples from Unknown and Yes components.

Experimental Analysis for OA(\(k\))

In the absence of RIDS, OA can help in eliminating dangling tuples.

Experimental analysis for \(k\)

<table>
<thead>
<tr>
<th>Alg.</th>
<th>0</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
<th>20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIDS</td>
<td>20/24</td>
<td>40/24</td>
<td>41/24</td>
<td>45/25</td>
<td>46/28</td>
</tr>
<tr>
<td>CA</td>
<td>22/24</td>
<td>42/24</td>
<td>25/25</td>
<td>27/25</td>
<td>28/24</td>
</tr>
<tr>
<td>OAP</td>
<td>21/21</td>
<td>23/21</td>
<td>24/22</td>
<td>25/25</td>
<td>27/25</td>
</tr>
<tr>
<td>OAC</td>
<td>20/18</td>
<td>20/18</td>
<td>20/18</td>
<td>21/18</td>
<td>22/18</td>
</tr>
</tbody>
</table>

Tuple-processing time in micro seconds for different values of \(k\).

Summary of tricks

- \(k\)-constraint for \(S_1 \rightarrow S_2\):
  - Default: \(S(S_1)\), Yes if \(S(S_1)\) is a cover and \(S(S_1)\) No if \(S(S_1)\) is not stream
  - RIDS: \(S(S_2)\), No and \(S(S_1)\), Unknown
  - CA on \(S_1\), A: \(S(S_1)\), Yes and non-dangling tuples in \(S(S_2)\), Yes \(\cup\) No \(\cup\) Unknown
  - OAP on \(S_1\), A: \(S(S_1)\), Yes and \(S(S_2)\), Yes \(\cup\) No \(\cup\) Unknown
  - OAC on \(S_2\), A: \(S(S_2)\), No and \(S(S_1)\), Unknown

Venues for further research

- Exploit time-based constraints in streams
- Constraint selection problem: What is the best combination of constraints?
- Develop a framework that adapts a combination of constraints