Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams

presented by Yingying Tao

Background and Motivation

• Data stream – real-time, continuous, ordered

  Issues
  – May not be wholly stored in memory
  – New items are often more accurate/relevant
  – Query may need consume entire input before giving results

  Solution: sliding window
    time-based
    tuple-based

Background and Motivation

• Two issues for query processing over sliding windows:
  – Re-execution strategies
    • eager re-execution: re-execute query upon new tuple arrival
      – not feasible for streams with high arrival rate
    • lazy re-execution: re-execute query after certain time period
      – delay in generating new results
  – Tuple invalidation procedures
    • eager expiration: remove old tuples upon new tuple arrival
    • lazy expiration: remove old tuples after certain time period
Joins over $n$ data streams with $n$ sliding windows

Assumptions
- Streams consist of relational tuples — one timestamp attr + value attrs
- All windows fit in main memory
- Query plans use extreme right-deep tree
- No intermediate results materialized

Upon new tuple $k$ arrival
- Invalidate expired tuples
- Probe all tuples not expired
- Stream results to user

naive multi-way join

---

Problem with naive multi-way join

Example

<table>
<thead>
<tr>
<th>$t$</th>
<th>$v$</th>
<th>$x$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Solution:
- put newly arrived tuple on top of join order

---

Improved eager multi-way nest-loop joins

Pseudo code

```
// Algorithm Eager Multi-Way Nest-Loop Join
Input: tuple $k$ from window $W_i$, and $n$-way order
Output: $S_i owtie S_{i+1} \bowtie ... \bowtie S_n$

1. $S_i = \{k\}$ and $k$ is in $T_i$
2. for $j = 1$ to $n$
   a. $S_j = S_{j-1}$
   b. for $k$ in $S_j$
      i. if $k$ is expired
         1. Prune $k$
      ii. else
         1. $S_j = S_j \bowtie k$
3. return $S_n$
```
Sliding-Window Join Algorithms

- Lazy multi-way nest-loop joins
  - Adopt same idea as improved eager multi-way NLJs
  - Replace trigger condition “insert new tuple” by re-execute interval

- General lazy multi-way nest-loop joins
  - Remove restriction of “put newly arrived tuples to the outermost for-loop”
  - Timestamp comparisons can only be done in the for-loop of new arrived tuples


definite code

Sliding-Window Join Algorithms

Pseudo codes

- Multi-way hash joins
  - Only scan on hash bucket where the attribute of newly arrived tuple falls in

  \[ \text{Eager version} \]

  \[ \text{Lazy version: similar to Lazy/General lazy multi-way NLJ} \]
Sliding-Window Join Algorithms

- Extension to tuple-based windows
  - Eager re-evaluation: overwrite the oldest tuple by new one
  - Lazy re-evaluation:
    - Maintain a counter for each tuple
    - Verify counter instead of timestamp

Pseudo code

Heuristic based Join Ordering

- Eager re-evaluation
  - Heuristic 1: join with the smallest remaining window first
  - Heuristic 2: join with the window that have the highest selectivity first
  - Heuristic 3: move faster streams up

- Lazy re-evaluation
  - Lazy multi-way NLJs: Considered as straight-forward extension of eager re-evaluation
  - General lazy multi-way NLJs: independently optimize each local join order by applying above heuristics

Heuristic based Join Ordering

- Multi-way hash join
  - Same number of hash bucket in all streams: same as NLJs
  - Various number of hash bucket: compute the average bucket size and apply heuristics

- Other scenarios
  - Hybrid hash-NLJ
  - Expensive predicates
  - Joins on different attributes
  - Fluctuating stream arrival rates
Experiments

• Join processing cost compare
  – Eager multi-way NLJs vs. Naive multi-way NLJs
    50 values each window, time size 100, 1 tuple per unit time
    ![Graph showing comparison between eager and naive multi-way NLJs]
    Observation: eager multi-way NLJs outperform Naive

Experiments (cont’d)

• Join processing cost compare
  – Lazy/General lazy multi-way NLJs vs. Naive multi-way NLJs
    4 windows, same parameters as previous example
    Increase arrival rate of S4 to 10 tuples per unit
    ![Graph showing comparison between lazy and general lazy multi-way NLJs]
    Observation:
    • General NLJs always performs best
    • Lazy NLJs will be beaten when re-evaluation interval is large
    • Almost linear performance for Lazy and General NLJs

Experiments

• Join ordering heuristic validation
  4 windows, arrival rates 1–10 tuple per unit, time size 100–200, # of values 5–500
  ![Table showing validation results]
  Observation:
  • Best plan derived from the heuristics for all above cases
  • Hash join outperforms NLJs
Effect of re-evaluation frequency and number of hash buckets on different algorithms

Experiments

4 windows, arrival rate 1 tuple per unit, time size 100, 50 values each window.

Observations:
• NLJ is the slowest
• The more hash buckets, the better performance
• Very frequent and infrequent re-evaluations are both inefficient

Varying hash table sizes

Experiments

4 windows, time size 100, 50 values each window, re-evaluation rate 5 unit arrival rates 1 tuple per unit for S1, S2, S3, 50 tuples per unit for S4.

Observation: allocate more hash buckets to frequent refreshing window may improve performance

Multi-way NLJ and multi-way hash join proposed can beat naive multi-way NLJ

Heuristics for join ordering can improve performance

System parameters may affect efficiency
• Stream arrival rates
• Tuple expiration policies
• Number of hash buckets

Future work
• Consider query operators other than join
• More heuristics for join ordering
• Better cost estimation strategies

Conclusion
Discussion

- Large, or complex multi-joins?
- Adopting existing query optimization techniques for stream join ordering?
- Windows not be able to fit in main memory?
- Update selectivity for better estimating cost?