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

presented by Yingying Tao

Background and Motivation

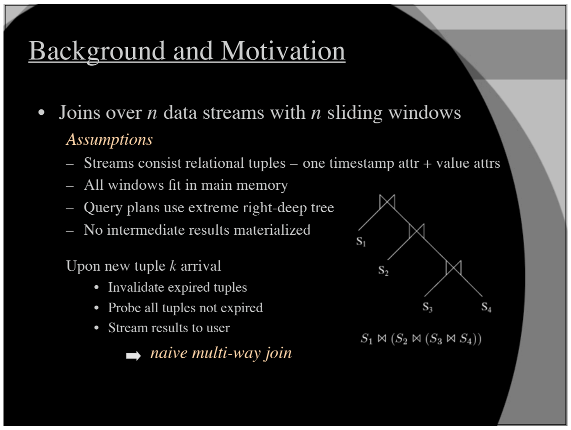
- Data stream – real-time, continuous, ordered

  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-evaluation*: re-execute query upon new tuple arrival
      - not feasible for streams with high arrival rate
    - *lazy re-evaluation*: re-execute query after certain time period
      - delay on generating new results
  - Tuple invalidation procedures
    - *eager expiration*: remove old tuples upon new tuple arrival
    - *lazy expiration*: remove old tuples after certain time period



Background and Motivation

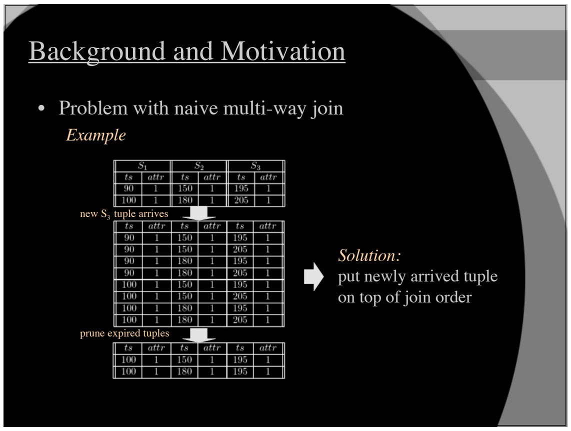
- Joins over *n* data streams with *n* sliding windows

  Assumptions
  - Streams consist 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*

$S_1 \bowtie (S_2 \bowtie (S_3 \bowtie S_4))$



Background and Motivation

- Problem with naive multi-way join

  Example

S_1		S_2		S_3	
ts	*attr*	*ts*	*attr*	*ts*	*attr*
90	1	150	1	195	1
100	1	180	1	205	1

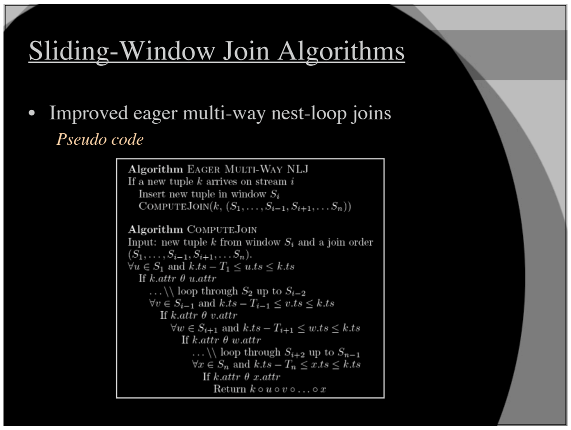
new S_3 tuple arrives

ts	*attr*	*ts*	*attr*	*ts*	*attr*
90	1	150	1	195	1
90	1	150	1	205	1
90	1	180	1	195	1
90	1	180	1	205	1
100	1	150	1	195	1
100	1	150	1	205	1
100	1	180	1	195	1
100	1	180	1	205	1

prune expired tuples

ts	*attr*	*ts*	*attr*	*ts*	*attr*
100	1	150	1	195	1
100	1	180	1	195	1

➡ *Solution:*
put newly arrived tuple
on top of join order



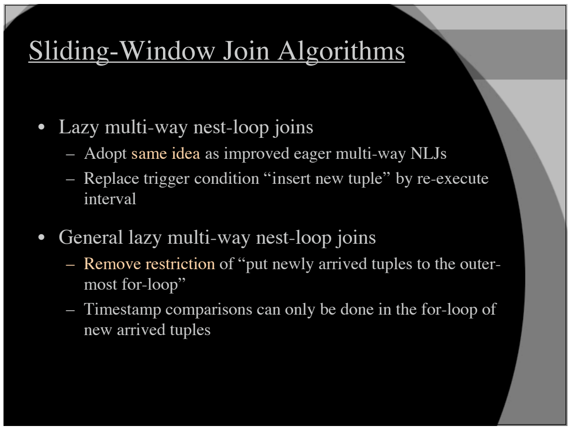
Sliding-Window Join Algorithms

- Improved eager multi-way nest-loop joins

  Pseudo code

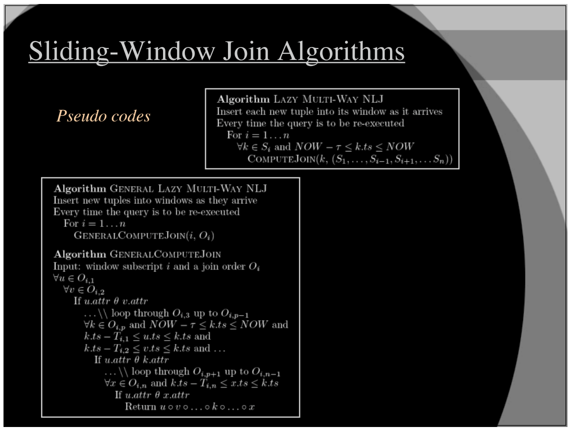
```
**Algorithm** EAGER MULTI-WAY NLJ
If a new tuple $k$ arrives on stream $i$
  Insert new tuple in window $S_i$
  COMPUTEJOIN($k$, $(S_1, \dots, S_{i-1}, S_{i+1}, \dots S_n)$)

**Algorithm** COMPUTEJOIN
Input: new tuple $k$ from window $S_i$ and a join order
$(S_1, \dots, S_{i-1}, S_{i+1}, \dots S_n)$.
$\forall u \in S_1$ and $k.ts - T_1 \leq u.ts \leq k.ts$
  If $k.attr\ \theta\ u.attr$
    $\dots$ \\ loop through $S_2$ up to $S_{i-2}$
    $\forall v \in S_{i-1}$ and $k.ts - T_{i-1} \leq v.ts \leq k.ts$
      If $k.attr\ \theta\ v.attr$
        $\forall w \in S_{i+1}$ and $k.ts - T_{i+1} \leq w.ts \leq k.ts$
          If $k.attr\ \theta\ w.attr$
            $\dots$ \\ loop through $S_{i+2}$ up to $S_{n-1}$
            $\forall x \in S_n$ and $k.ts - T_n \leq x.ts \leq k.ts$
              If $k.attr\ \theta\ x.attr$
                Return $k \circ u \circ v \circ \dots \circ x$
```



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 outer-most for-loop"
  - Timestamp comparisons can only be done in the for-loop of new arrived tuples



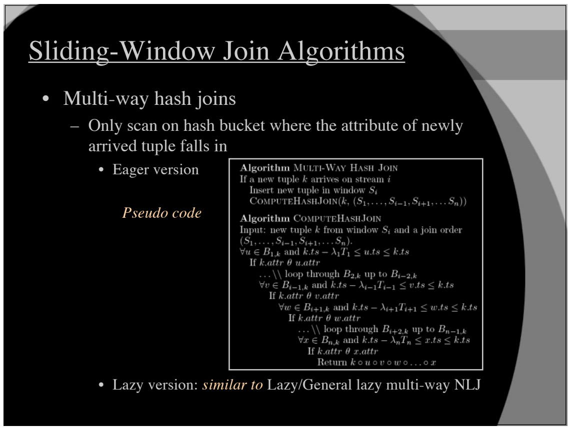
Sliding-Window Join Algorithms

Pseudo codes

```
Algorithm LAZY MULTI-WAY NLJ
Insert each new tuple into its window as it arrives
Every time the query is to be re-executed
  For $i = 1 \ldots n$
    $\forall k \in S_i$ and $NOW - \tau \leq k.ts \leq NOW$
      COMPUTEJOIN($k$, $(S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n)$)
```

```
Algorithm GENERAL LAZY MULTI-WAY NLJ
Insert new tuples into windows as they arrive
Every time the query is to be re-executed
  For $i = 1 \ldots n$
    GENERALCOMPUTEJOIN($i$, $O_i$)

Algorithm GENERALCOMPUTEJOIN
Input: window subscript $i$ and a join order $O_i$
$\forall u \in O_{i,1}$
  $\forall v \in O_{i,2}$
    If $u.attr\ \theta\ v.attr$
      ... \\ loop through $O_{i,3}$ up to $O_{i,p-1}$
      $\forall k \in O_{i,p}$ and $NOW - \tau \leq k.ts \leq NOW$ and
      $k.ts - T_{i,1} \leq u.ts \leq k.ts$ and
      $k.ts - T_{i,2} \leq v.ts \leq k.ts$ and ...
        If $u.attr\ \theta\ k.attr$
          ... \\ loop through $O_{i,p+1}$ up to $O_{i,n-1}$
          $\forall x \in O_{i,n}$ and $k.ts - T_{i,n} \leq x.ts \leq k.ts$
            If $u.attr\ \theta\ x.attr$
              Return $u \circ v \circ \ldots \circ k \circ \ldots \circ x$
```



Sliding-Window Join Algorithms

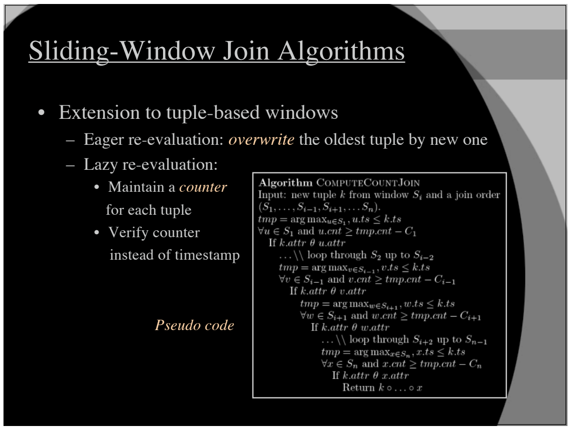
- Multi-way hash joins
  - Only scan on hash bucket where the attribute of newly arrived tuple falls in
    - Eager version

Pseudo code

```
Algorithm MULTI-WAY HASH JOIN
If a new tuple $k$ arrives on stream $i$
  Insert new tuple in window $S_i$
  COMPUTEHASHJOIN($k$, $(S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n)$)

Algorithm COMPUTEHASHJOIN
Input: new tuple $k$ from window $S_i$ and a join order
$(S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n)$.
$\forall u \in B_{1,k}$ and $k.ts - \lambda_1 T_1 \leq u.ts \leq k.ts$
  If $k.attr\ \theta\ u.attr$
    ... \\ loop through $B_{2,k}$ up to $B_{i-2,k}$
    $\forall v \in B_{i-1,k}$ and $k.ts - \lambda_{i-1} T_{i-1} \leq v.ts \leq k.ts$
      If $k.attr\ \theta\ v.attr$
        $\forall w \in B_{i+1,k}$ and $k.ts - \lambda_{i+1} T_{i+1} \leq w.ts \leq k.ts$
          If $k.attr\ \theta\ w.attr$
            ... \\ loop through $B_{i+2,k}$ up to $B_{n-1,k}$
            $\forall x \in B_{n,k}$ and $k.ts - \lambda_n T_n \leq x.ts \leq k.ts$
              If $k.attr\ \theta\ x.attr$
                Return $k \circ u \circ v \circ w \circ \ldots \circ x$
```

- Lazy version: *similar to* Lazy/General lazy multi-way NLJ

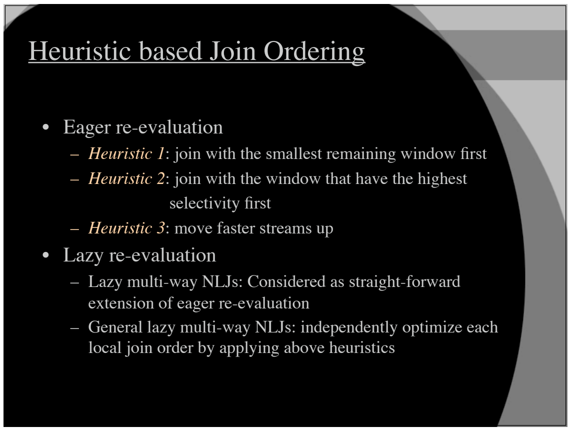


Sliding-Window Join Algorithms

- 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

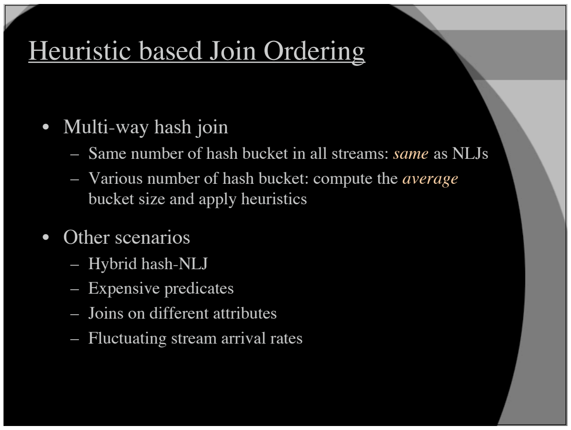
Pseudo code

```
Algorithm ComputeCountJoin
Input: new tuple $k$ from window $S_i$ and a join order
$(S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n)$.
$tmp = \arg\max_{u \in S_1}, u.ts \leq k.ts$
$\forall u \in S_1$ and $u.cnt \geq tmp.cnt - C_1$
  If $k.attr\ \theta\ u.attr$
    $\ldots$ \\ loop through $S_2$ up to $S_{i-2}$
    $tmp = \arg\max_{v \in S_{i-1}}, v.ts \leq k.ts$
    $\forall v \in S_{i-1}$ and $v.cnt \geq tmp.cnt - C_{i-1}$
      If $k.attr\ \theta\ v.attr$
        $tmp = \arg\max_{w \in S_{i+1}}, w.ts \leq k.ts$
        $\forall w \in S_{i+1}$ and $w.cnt \geq tmp.cnt - C_{i+1}$
          If $k.attr\ \theta\ w.attr$
            $\ldots$ \\ loop through $S_{i+2}$ up to $S_{n-1}$
            $tmp = \arg\max_{x \in S_n}, x.ts \leq k.ts$
            $\forall x \in S_n$ and $x.cnt \geq tmp.cnt - C_n$
              If $k.attr\ \theta\ x.attr$
                Return $k \circ \ldots \circ x$
```



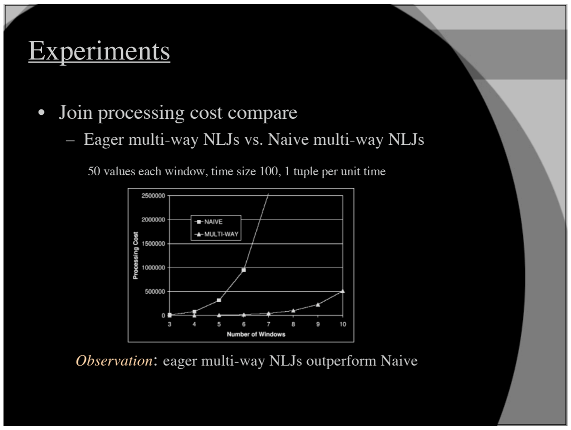
Heuristic based Join Ordering

- 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

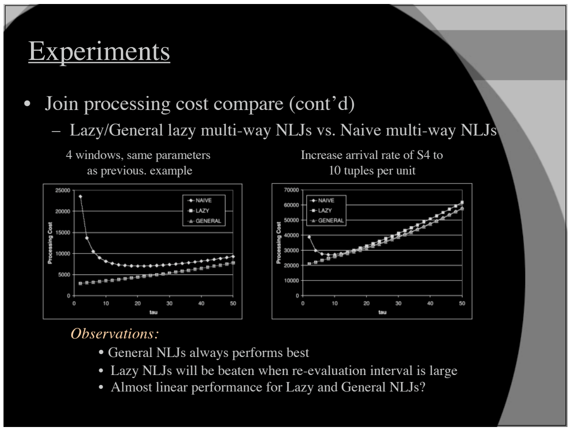


Experiments

- 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

Observation: eager multi-way NLJs outperform Naive



Experiments

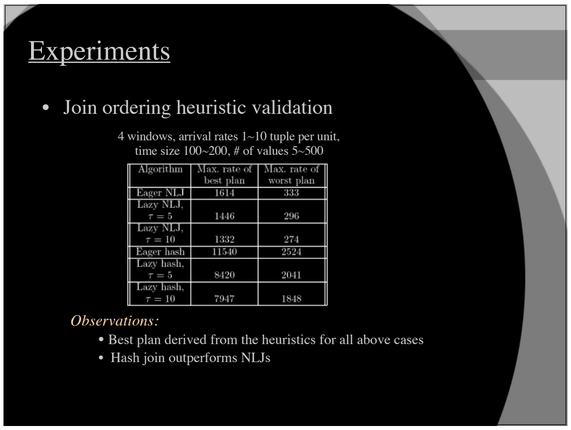
- Join processing cost compare (cont’d)
  - Lazy/General lazy multi-way NLJs vs. Naive multi-way NLJs

4 windows, same parameters
as previous. example

Increase arrival rate of S4 to
10 tuples per unit

Observations:

- 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

Algorithm	Max. rate of best plan	Max. rate of worst plan
Eager NLJ	1614	333
Lazy NLJ, $\tau = 5$	1446	296
Lazy NLJ, $\tau = 10$	1332	274
Eager hash	11540	2524
Lazy hash, $\tau = 5$	8420	2041
Lazy hash, $\tau = 10$	7947	1848

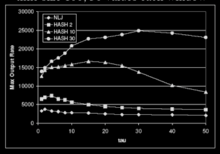
Observations:

- Best plan derived from the heuristics for all above cases
- Hash join outperforms NLJs

Experiments

- Effect of re-evaluation frequency and number of hash buckets on different algorithms

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

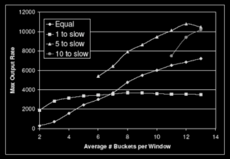


Observations:
- NLJ is the slowest
- The more hash buckets, the better performance
- Very frequent and infrequent re-evaluations are both inefficient

Experiments

- Varying hash table sizes

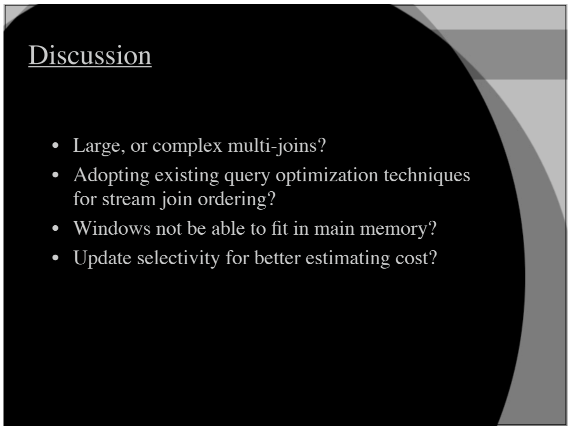
4 windows, time size 100, 50 values each window, re-evaluation rate 5 unit
arrival rates 1 tuple per unit for S_1, S_2, S_3, 50 tuple per unit for S_4



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

Conclusion

- 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



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?