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

Paper By: Lukasz Golab
M. Tamer Ozsu

CS 561 Presentation

WPI 11th March, 2004

Students: Malav shah

Professor: Elke Rundenstainer

VLDB 2003, Berlin, Germany.

INDEX

- Introduction
- Problem Description
- Sliding Window Join algorithms
- Cost Analysis
- Join Ordering Heuristics
- Experimental Results
- Related Work
- Conclusion and Future Work

Introduction

- **What is Data Streams?**

  A real-time, continuous, ordered (explicitly by timestamps or implicitly by arrival time) sequence of items.

- **How can you query such type of streams?**

  running a query continually over a period of time and generating new results.

  continuous, standing, or persistent queries.

Applications

- **Sensor Data Processing**
- **Internet Traffic analysis**
- **Financial Ticker**
- **Analysis of various transaction logs such as Web server logs and telephone records**

Issues

- Unbounded streams may not wholly stored in bounded memory.
- New items are often more accurate or more relevant than older items.
- Blocking operators may not be useful as they must consume entire input before any results produced.

Common Solution

- **Define Sliding-Window**

  Restrict the range of continuous queries to a sliding-window that contains the last T items or those items that contains last t time units.

- **Count Based Window (Sequence Based)**

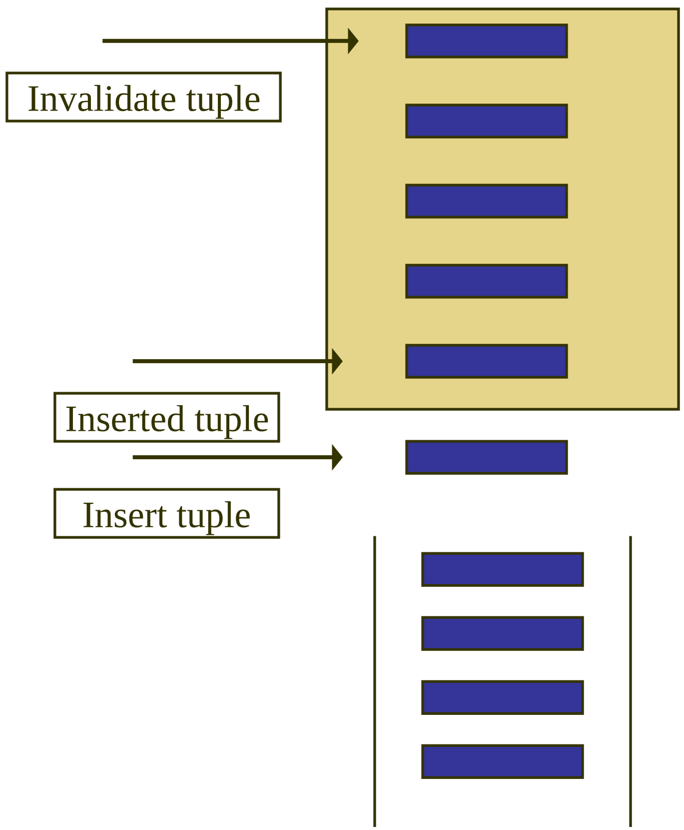
- **Time Based Window (Timestamps Based)**

Issues: using sliding window

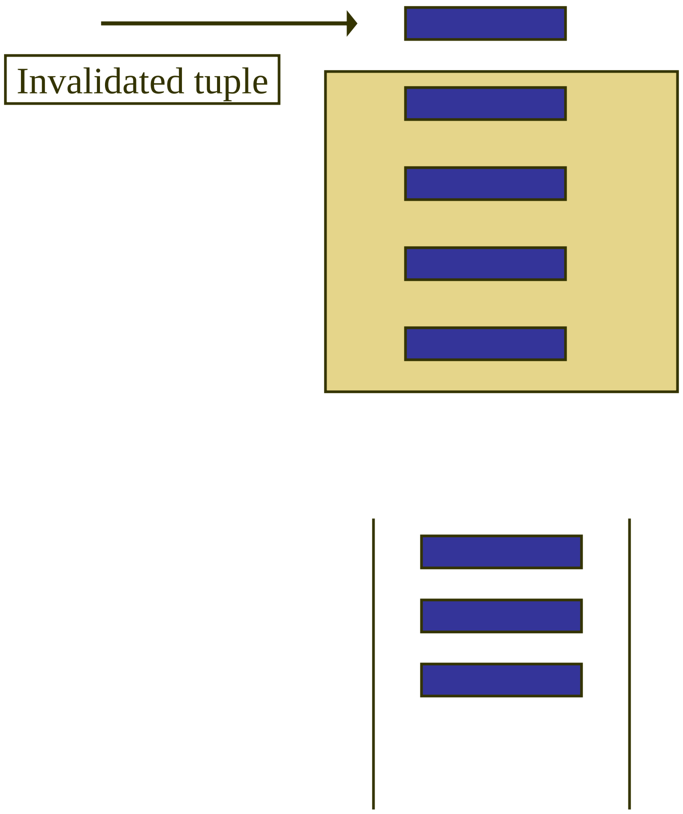
- **Re-Execution Strategies**
  - **Eager re-execution strategy**
  - **Lazy re-execution strategy**

- **Tuple Invalidation Procedures**
  - **Eager expiration**
  - **Lazy expiration**

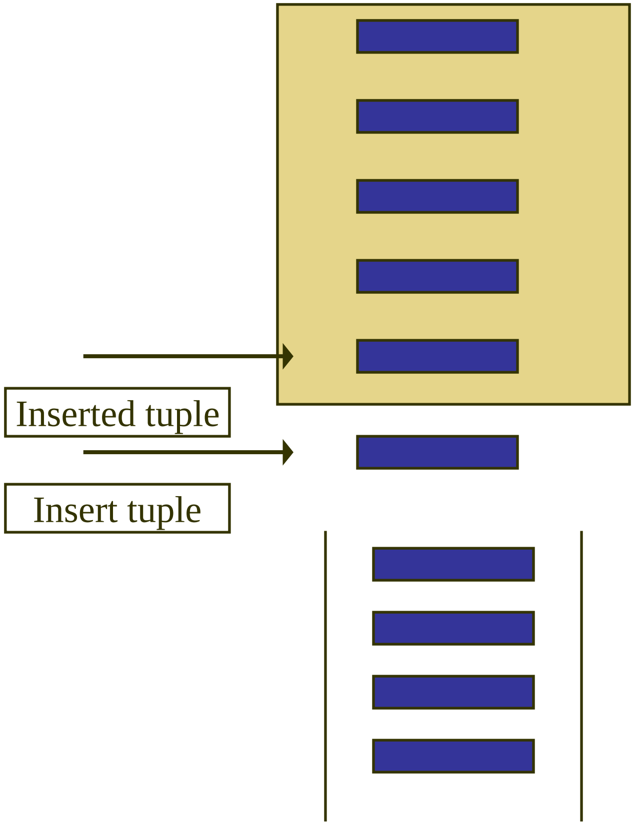
Example of Eager Re-execution and Expiration



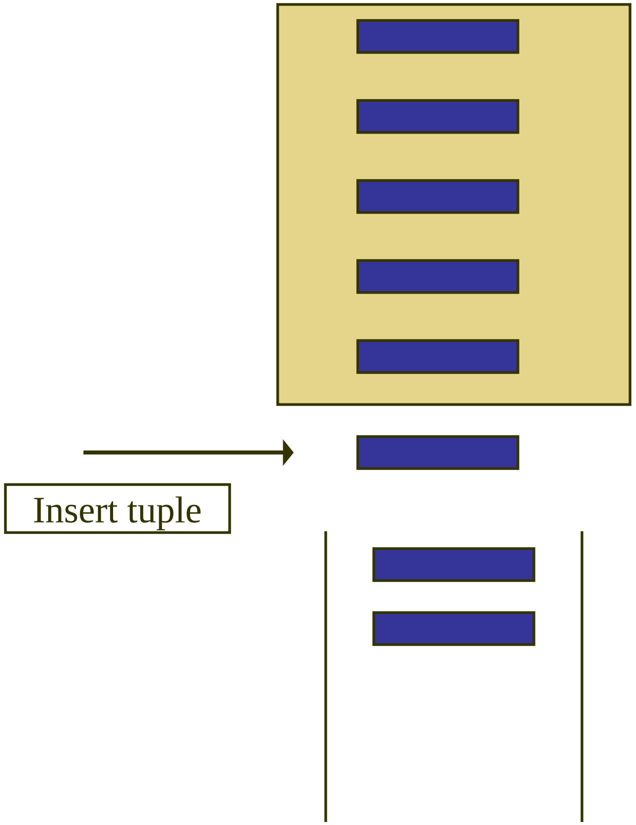
Example of Eager Re-execution and Expiration



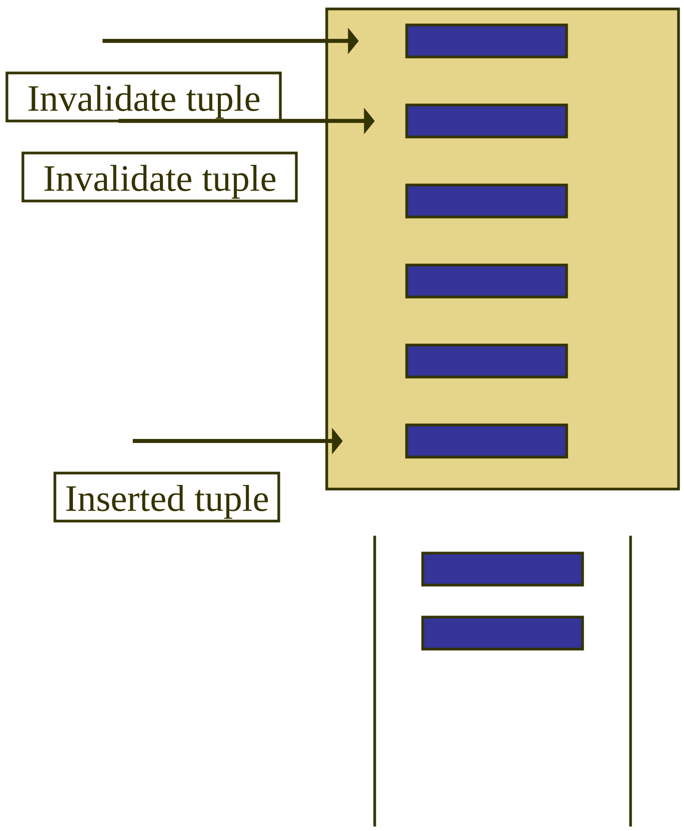
Example of Lazy Expiration



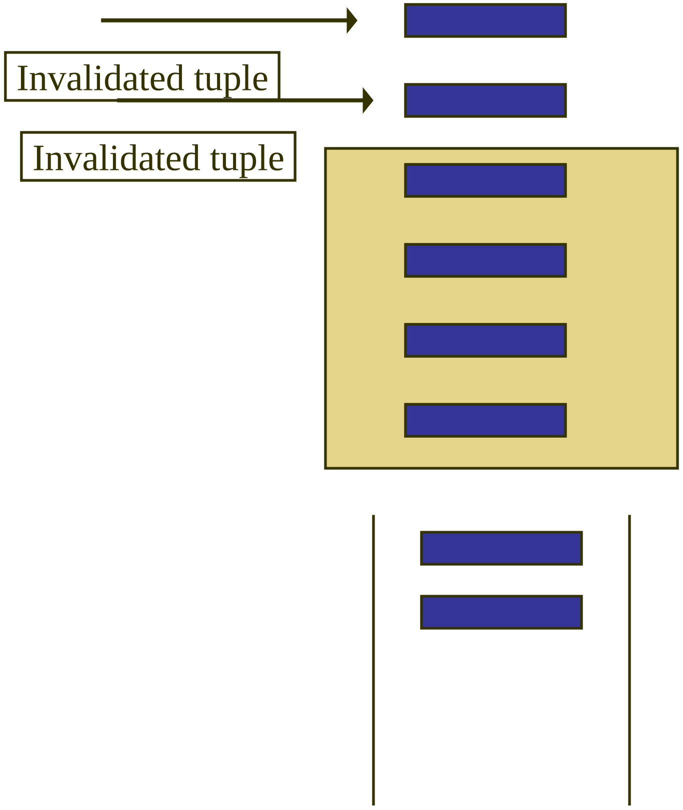
Example of Lazy Expiration



Example of Lazy Expiration



Example of Lazy Expiration



INDEX

- ✓ **Introduction**
- ➢ **Problem Description**
- ➢ **Sliding Window Join algorithms**
- ➢ **Cost Analysis**
- ➢ **Join Ordering Heuristics**
- ➢ **Experimental Results**
- ➢ **Related Work**
- ➢ **Conclusion and Future Work**

Problem Description

- **N Data Streams**
- **N corresponding sliding window**
- **Continuously evaluate exact join of all N window**

Assumption

- **Each stream is consist of relational tuple with schema <timestamp ts, attributes attr>**
- **All windows fit into main memory**
- **All query plans use extreme right-deep join trees that do not materialize any intermediate results**
- **Do not permit time-lagged windows**

Explanation of symbols

λ_i	Arrival rate of stream i in tuples per unit time
S_j	Sliding window corresponding to stream j
T_j	Time size of the j^{th} time-based window
C_j	Number of tuples in S_j
v_j	Number of distinct values in S_j
b_j	Number of hash buckets in the hash index of S_j, if such an index exists
τ	Continuous query re-execution interval
$a \circ b$	Concatenation of tuples a and b
θ	Arithmetic comparison predicate, e.g. $=$

Convention for Join Ordering

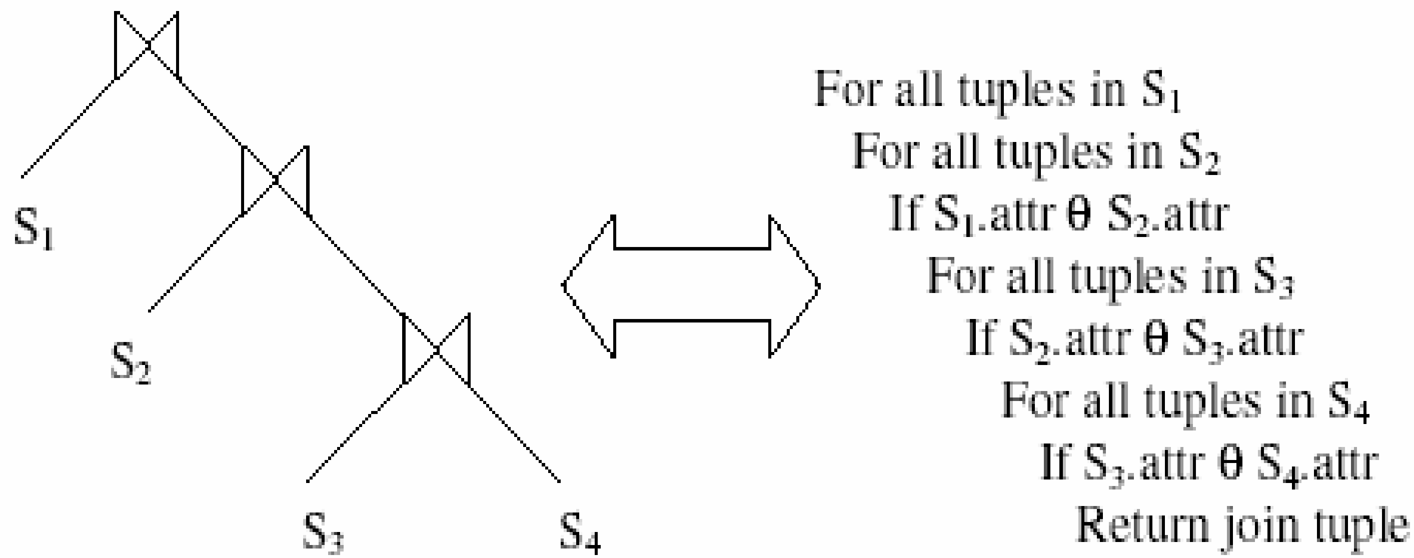


Figure 1: Join order $S_1 \bowtie (S_2 \bowtie (S_3 \bowtie S_4))$ expressed as a join tree (left) and as a series of for-loops (right).

TOP-DOWN Approach

INDEX

- ✓ **Introduction**
- ✓ **Problem Description**
- **Sliding Window Join algorithms**
- **Cost Analysis**
- **Join Ordering Heuristics**
- **Experimental Results**
- **Related Work**
- **Conclusion and Future Work**

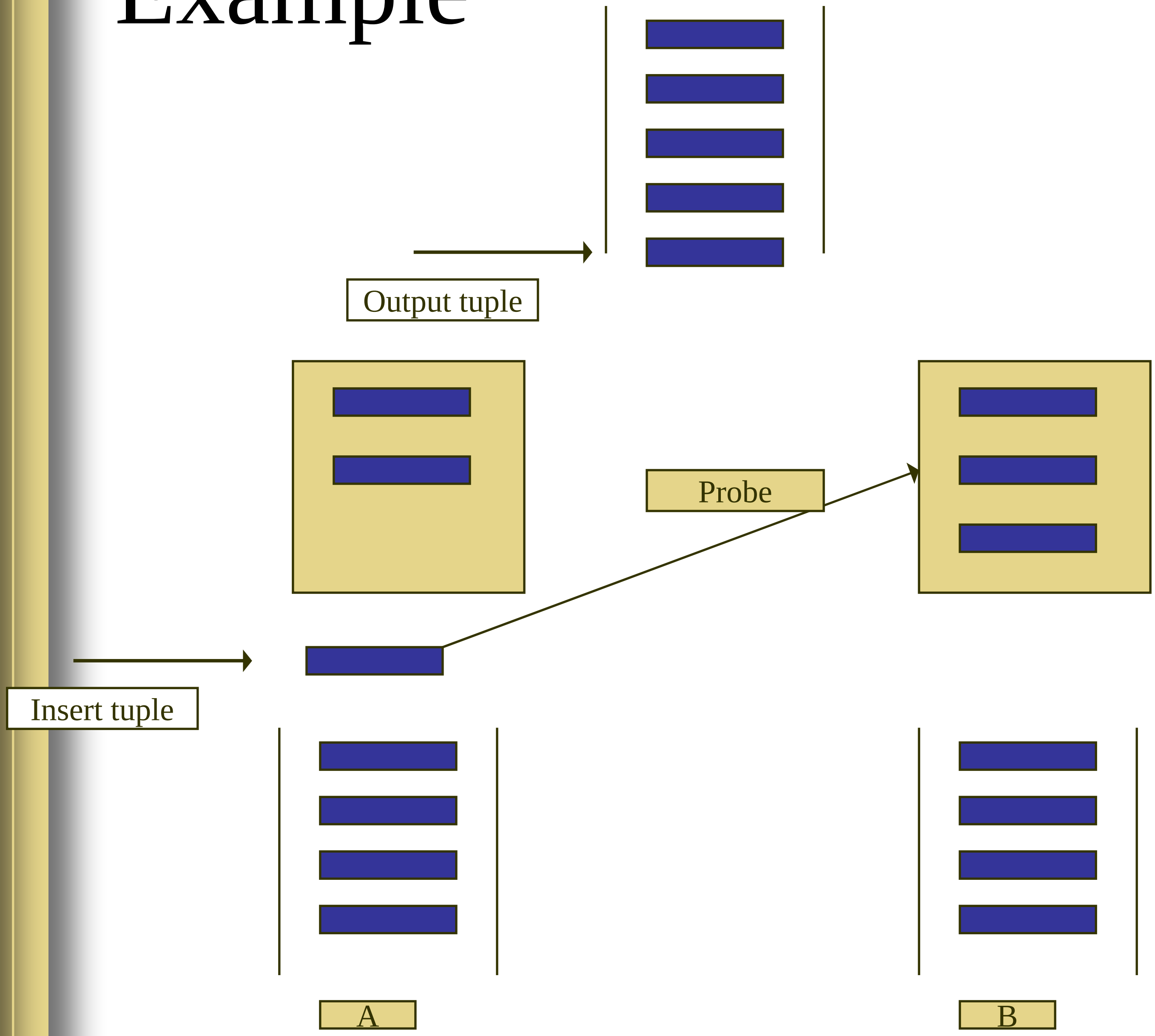
Binary Incremental NLJ

- **Proposed by Kang**

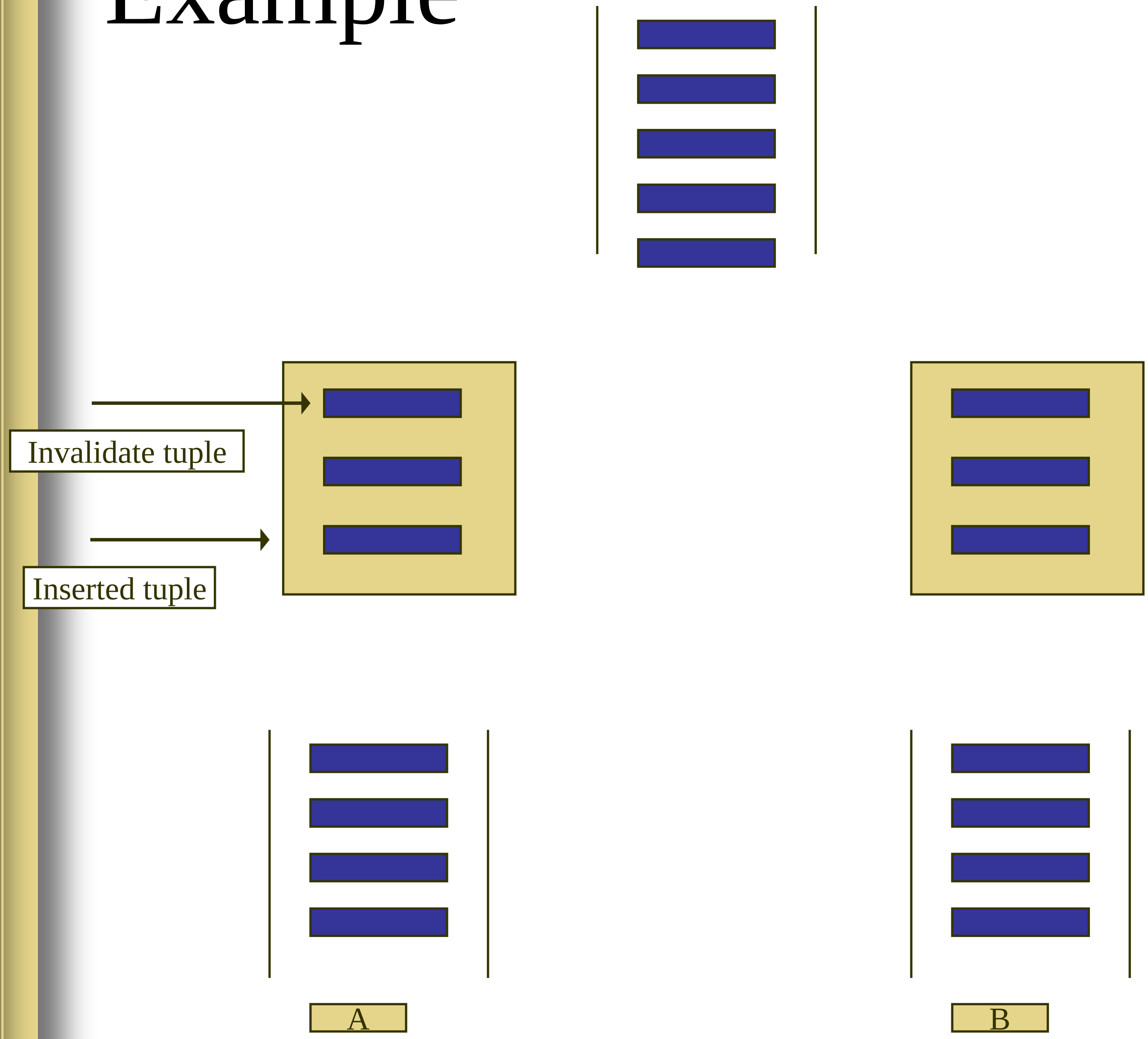
- **Strategy**

  Let S1 and S2 be two sliding windows to be joined. For each newly arrived S1-tuple, we scan S2 and return all matching tuples. We then insert the new tuple into S1 and in-validate expired tuples. We follow the same procedure for each newly arrived S2-tuple.

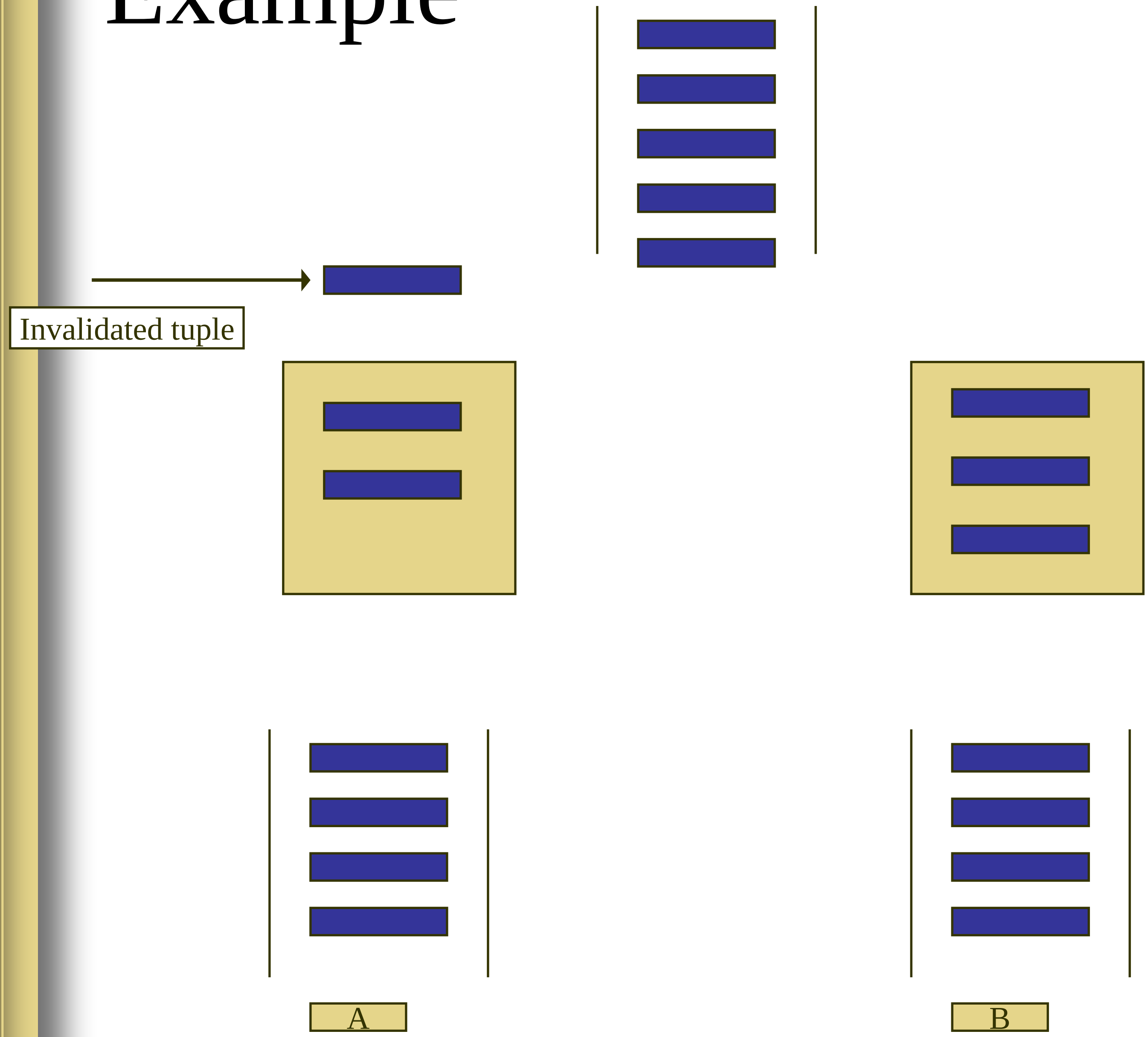
Example



Example



Example



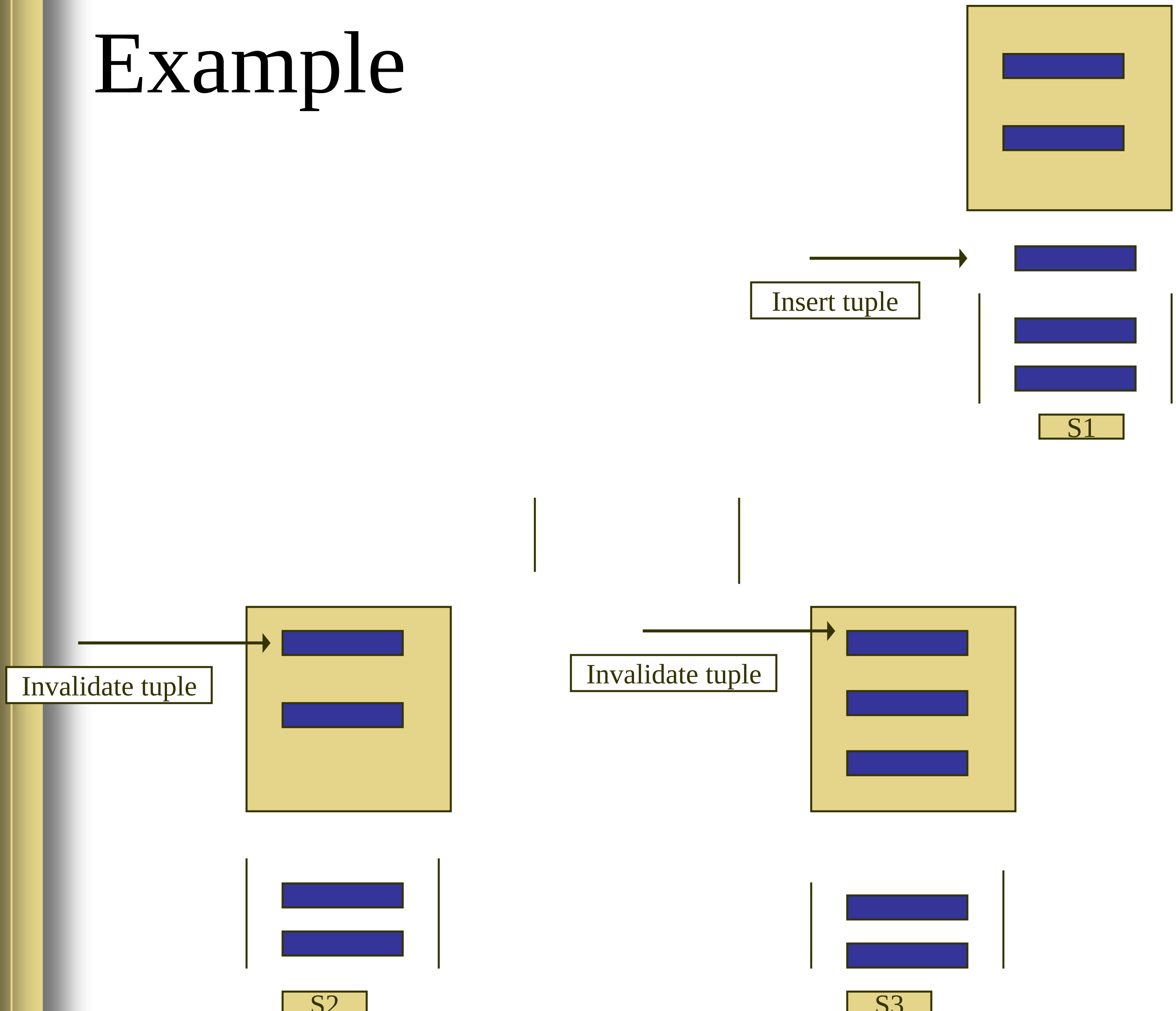
Naïve Multi-way NLJ

- **Extension to Binary Incremental NLJ**
- **Strategy**

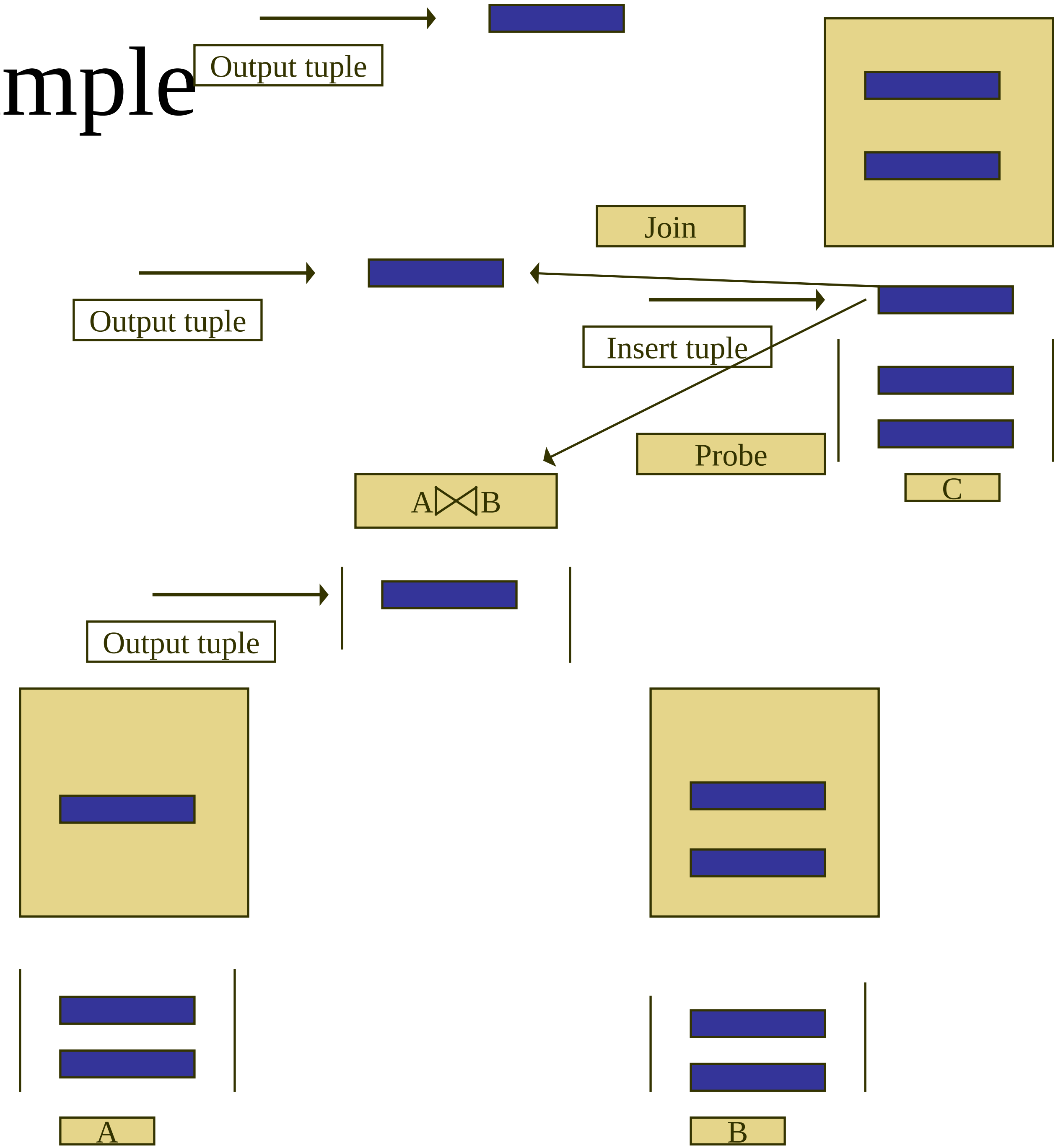
  For each newly arrived tuple k, we execute the join sequence in the order prescribed by the query plan, but we only include k in the join process (not the entire window that contain k).

- In this algorithm we invalidate the expired tuples first
- Extending Naïve Multi-way NLJ to support lazy re-evaluation is easy. Re-execute the join every T time units, first joining new s1-tuples with other sliding windows, then new s2-tuples and so on. (must ensure not to include expired tuples in result)

Example

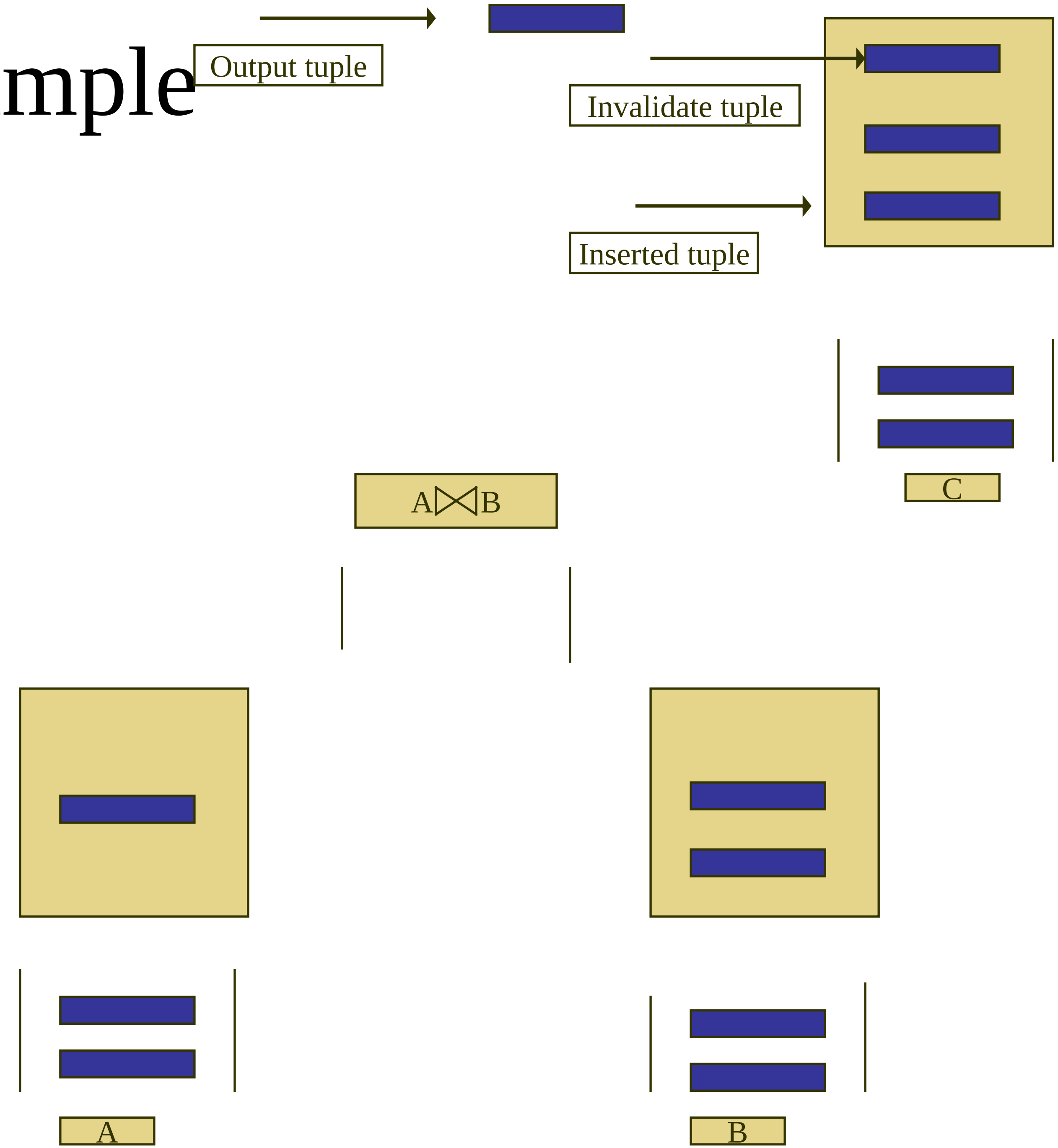


Example

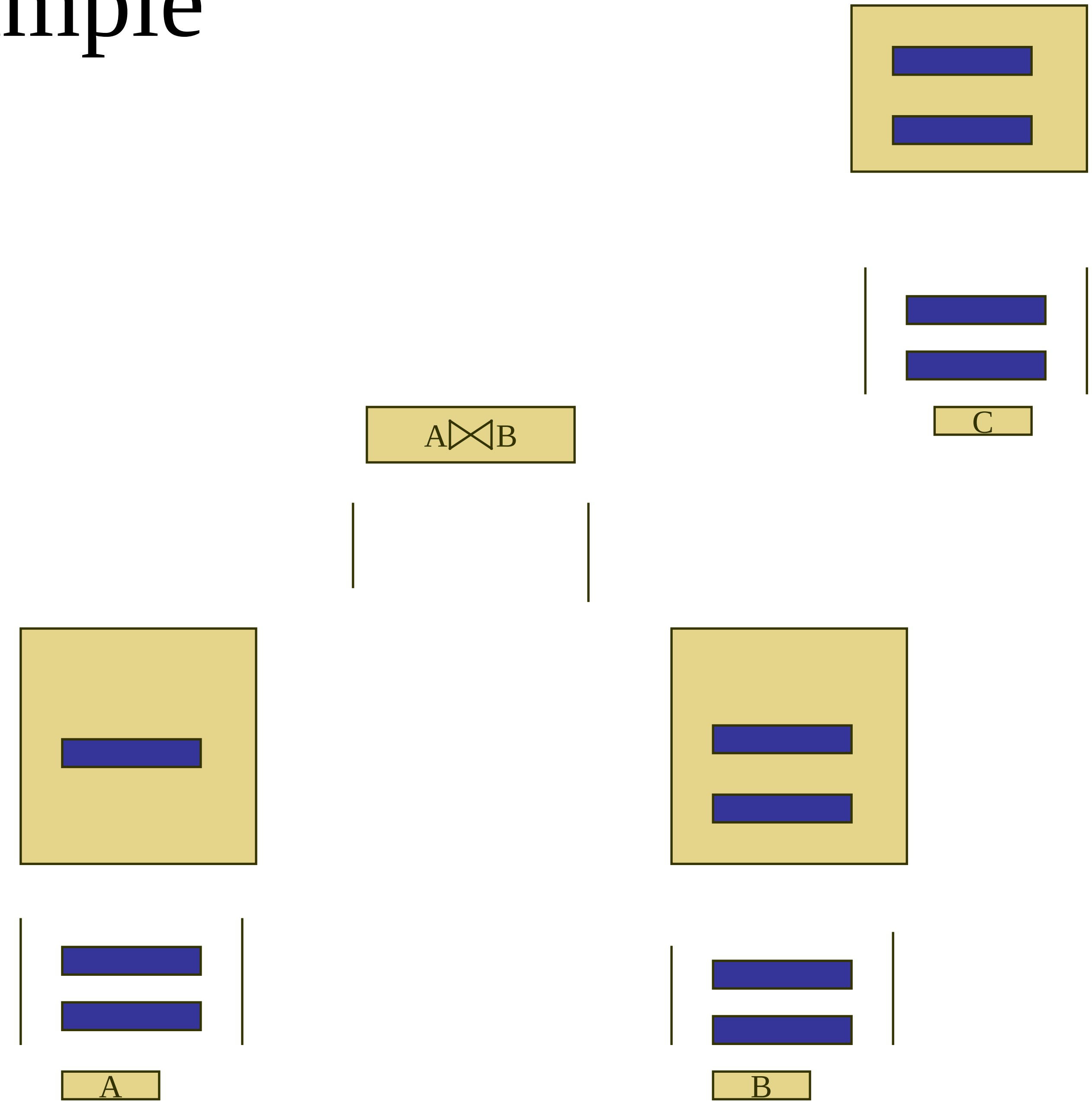


Output tuple
Join
Output tuple
Insert tuple
Probe
C
A⋈B
Output tuple
A
B

Example



Example



Improved eager Multi-way NLJ

- **Problems with Naïve eager Multi-way Join**
  - When a new tuple arrives at the stream which is not first in the order list then we compute the join for both second and third stream for all tuples in first ordered stream. This results in unnecessary work when a new tuple arrives at stream which is not first in the join tree.
  - Why not select only those tuples from s1 which joins with s3-tuple, and make scan of s2 for only those tuples.
  - In worst case when all tuples in s1 joins with newly arrived tuple, we've to scan s2 for every tuple in s1, otherwise it'll be less.

Algorithm for eager Multi-way Join

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

Algorithm COMPUTEJOIN
Input: new tuple k from window S_i and a join order $(S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n)$.
$\forall u \in S_1$ and $k.ts - T_1 \leq u.ts \leq k.ts$
   If $k.attr \ \theta \ u.attr$
      ...\\ 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 $k.attr \ \theta \ w.attr$
                  ...\\ 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 $k \circ u \circ v \circ \ldots \circ x$

Lazy Multi-Way Join

- **Straightforward adaptation to eager multi-way join**
  - **Process in the outer most for-loop all the new tuples which have been arrived since last re-execution**
- **Algorithm**

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 $NOW - \tau \leq k.ts \leq NOW$
      $\textsc{ComputeJoin}(k, (S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n))$

General Lazy Multi-Way Join

- **We can make the lazy multi-way join more general if newly arrived tuples are not restricted to the outer-most for loop.**
- **Accepts arbitrary join order.**
- **Algorithm**

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 $u \circ v \circ \ldots \circ k \circ \ldots \circ x$

Multi-Way Hash Join

- We scan only one hash bucket instead of the entire window at each for loop.
- Algorithm
- Notation: B(i,k) = hi(k.attr) for Ith window

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 $k.attr\ \theta\ u.attr$
    $\ldots$ \\ 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 $k.attr\ \theta\ w.attr$
            $\ldots$ \\ 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 $k \circ u \circ v \circ w \circ \ldots \circ x$

Extension to Count-Based Windows

- **Eager expiration is straightforward:**
  - **Implement window(or hash bucket) as circular arrays**
  - **We can perform insertion and invalidation in one step by overwriting oldest tuple**
- **Lazy expiration is interesting:**
  - **Implement circular counter and assign positions to each element in sliding window(call them cnt)**
  - **When probing for tuples to join with a new tuple k, instead of comparing timestamps, we ensure that each tuples counter cnt has not expired at time k.ts.**
  - **To do this, for each sliding window we find counter with the largest timestamps not exceeding k.ts and subtract window length from this counter (call it tmp) and ensure that we join only those tuples with counter greater than tmp.**

Algorithm

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 $k \circ \ldots \circ x$

INDEX

- ✓ **Introduction**
- ✓ **Problem Description**
- ✓ **Sliding Window Join algorithms**
- ➢ **Cost Analysis**
- ➢ **Join Ordering Heuristics**
- ➢ **Experimental Results**
- ➢ **Related Work**
- ➢ **Conclusion and Future Work**

Cost Analysis

- **Insertion and Expiration cost**
  - All NLJ based algorithms incur a constant insertion cost per tuple: a new tuple is simply appended to its window
  - In hash based algorithm requires more work: need to compute hash function and add tuple in hash table (insertion cost slightly higher)
  - Actual insertion and expiration costs are implementation-dependent
  - If invalidation is too frequent, some sliding window may not contain any state tuples, but we'll still pay the cost to access it (same case with hash joins)
  - Very frequent expiration is too costly, especially in hash joins.

Join Processing Cost

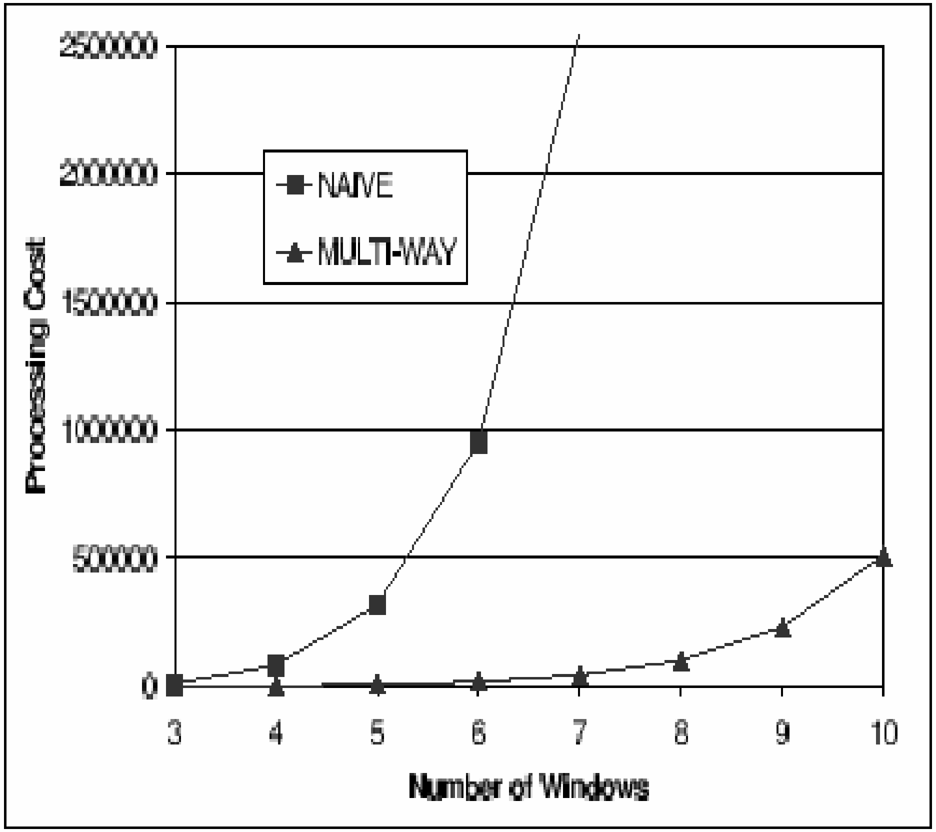
- **Used per-unit-time cost model, developed by kang**

- **When estimating join sizes, standard assumptions regarding containment of value sets and uniform distribution of attribute values are considered**

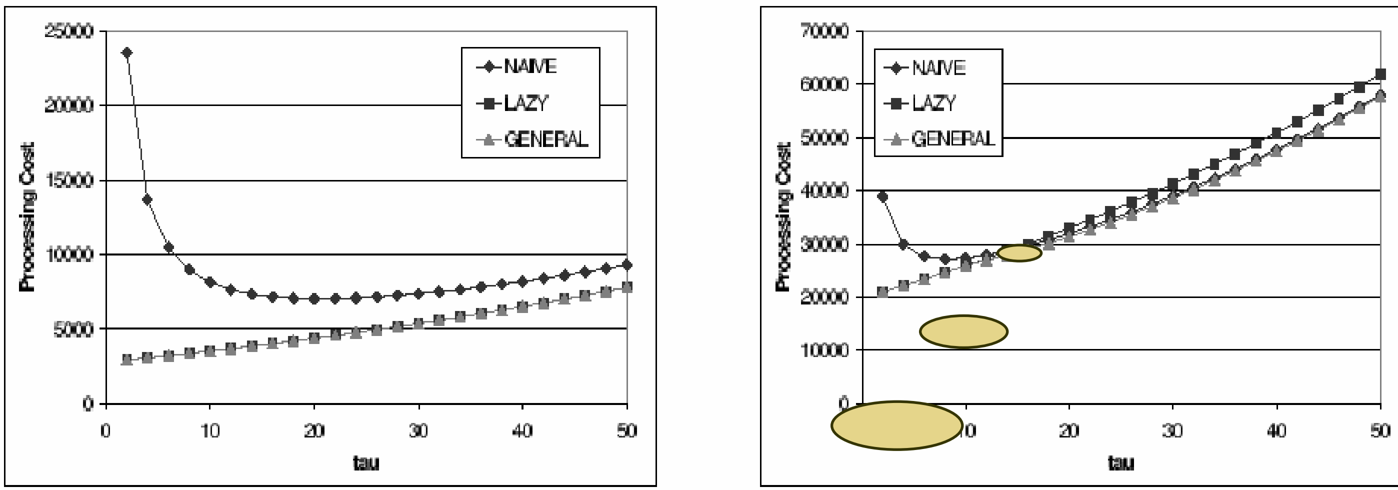
Comparison between Naïve and Proposed Multi-way Joins.

Given equivalent ordering
All window has same window size, all streams has same arrival rate, each window has same distinct value.

Proposed multi-way scales better than Naïve Multi-way Joins.



Comparison between different lazy multi-way join algorithms



(a)

Figure 3: Performance comp… …thm. Derivations of the cost formulas for each algorith… …on of this paper [16].

Lazy does not perform well some time

INDEX

- ✓ Introduction
- ✓ Problem Description
- ✓ Sliding Window Join algorithms
- ✓ Cost Analysis
- ➢ Join Ordering Heuristics
- ➢ Experimental Results
- ➢ Related Work
- ➢ Conclusion and Future Work

Effect of Join Ordering

- **Eager re-execution**
  - **If each window has same number of distinct values, then it is sensible to globally order the joins in ascending order of the window sizes (in tuples), or average hash bucket sizes**
  - **In general, it is sensible (but not optimal always) heuristic is to assemble the joins in descending order of the binary join selectivities, leaving as little work as possible for inner for-loops**
  - **We define a join predicate p1 to be more selective then p2, if p1 produces small result set then p1.**

Example

- **For the example given, the results are as follows**
- **For order s1,s2,s3,s4 processing time is 16000**
- **S2,s1,s3,s4 has cost of 19600**
- **Worst cost plan is 90000.**

Stream 1	$\lambda_1 = 10,\ T_1 = 100,\ v_1 = 500$
Stream 2	$\lambda_2 = 1,\ T_2 = 100,\ v_2 = 50$
Stream 3	$\lambda_3 = 1,\ T_3 = 200,\ v_3 = 40$
Stream 4	$\lambda_4 = 3,\ T_4 = 100,\ v_4 = 5$

Example when two streams faster

- For the example given, the results are as follows
- For order s1,s2,s3,s4 processing time68200
- S2,s1,s3,s4 has cost of 79000
- S3,s1,s4,s2 has cost of 47977 (optimal)
- So it's not the always case that moving all faster streams upward is optimal

Stream 1	$\lambda_1 = 11,\ T_1 = 100,\ v_1 = 200$
Stream 2	$\lambda_2 = 10,\ T_2 = 100,\ v_2 = 100$
Stream 3	$\lambda_3 = 1,\ T_3 = 100,\ v_3 = 65$
Stream 4	$\lambda_4 = 1,\ T_4 = 100,\ v_4 = 20$

Ordering heuristics for Lazy Re-evaluation

- **Recall that lazy multi-way join is as efficient as general multi-way join for small T.**
- **If this is the case then we may use same ordering heuristics as algorithm Lazy Multi-way Join is a straightforward extension of its eager version.**
- **General Multi-way Join is more efficient if a good join-ordering is chosen**
- **General Multi-way join chooses join ordering arbitrarily depending on the origin of the new tuples that are being processed**

Ordering heuristics for Multi-way Hash Join

- **If each hash table has same number of buckets, the ordering problem is same as NLJ. Why?**

- **Hash join so configured operates in nested-loop fashion like NLJ, except in each loop only one hash bucket is scanned instead of entire window.**

Join ordering in other scenarios

- **Hybrid Hash-NLJ: a simple heuristic is to place all the windows that contain hash indices in the inner for-loop.**
- **Expensive Predicates: Those may be ordered near the top of the index tree.**
- **Joins on different attributes: we cannot arbitrarily re-order the join tree. It may still be efficient to place the window from which new tuple arrived at the outer-most for-loop.**
- **Fluctuating Stream arrival rates: If feasible, we re-execute the ordering heuristic whenever stream rates changes beyond some threshold, or we can place the streams which expected to change widely near the top.**

INDEX

- ✓ **Introduction**
- ✓ **Problem Description**
- ✓ **Sliding Window Join algorithms**
- ✓ **Cost Analysis**
- ✓ **Join Ordering Heuristics**
- ➢ **Experimental Results**
- ➢ **Related Work**
- ➢ **Conclusion and Future Work**

Experimental Setting

- Build a simple prototype of algorithms using SUN Microsystems JDK 1.3.1
- Windows PC with 1.2 AMD Processor and 256 MB RAM
- Implemented sliding windows and hash buckets as singly linked list
- All hash functions are simple modular division by the number of hash buckets
- Tuple schema <int ts, int attr>
- Expiration does not delete the tuple, instead java garbage collector do that task
- Tuple generation is simple continuous for-loop which generates tuples randomly from specified distinct values

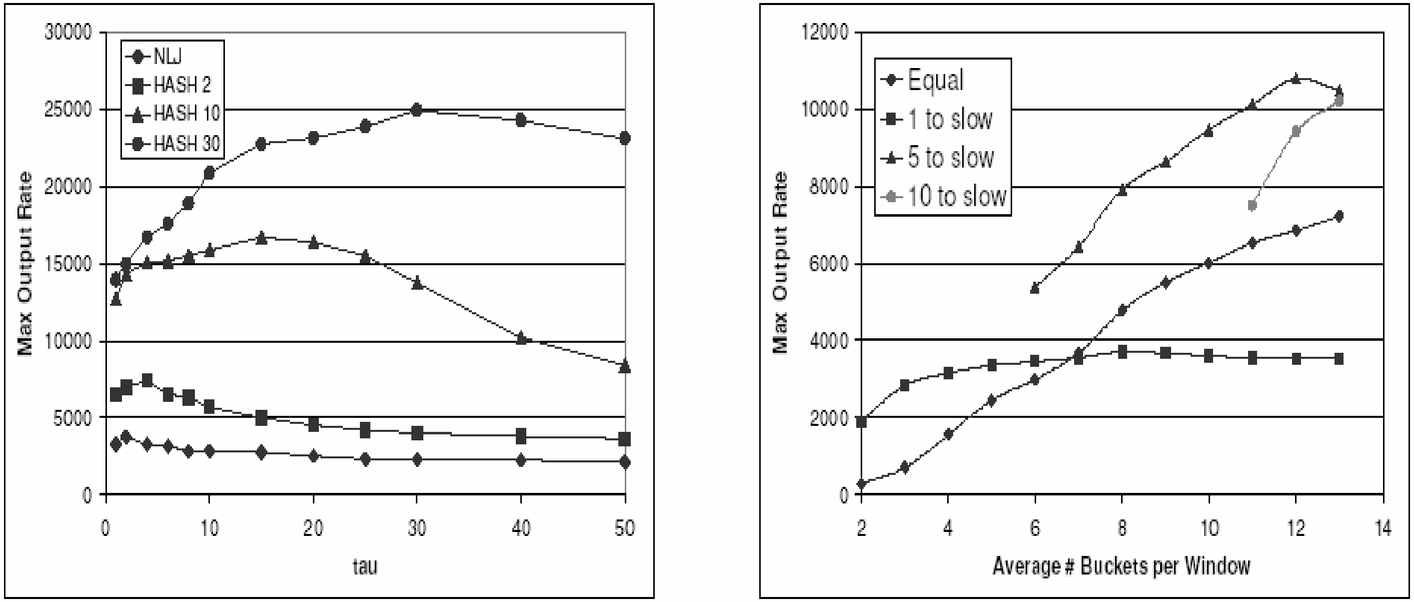
Validation of cost model and Join ordering heuristics

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

Effect of query Re-Evaluation and Expiration Frequencies on Processing Time

- **Eager expirations incurs cost of updating linked list on every arrival of tuple, while lazy expiration performs fewer operations, but allows the window to grow between updates, causing long Join evaluation time.**
- **For both NLJ and hash join short expiration intervals are preferred as cost of advancing pointer is lower than processing larger windows.**
- **Very frequent expiration and re-evaluation are inefficient.**

Varying Hash Table Sizes



(a) (b)

Figure 4: Performance comparison of our algorithms with respect to a) increasing the re-evaluation and expiration interval and b) building large hash tables on fast streams.

INDEX

- ✓ **Introduction**
- ✓ **Problem Description**
- ✓ **Sliding Window Join algorithms**
- ✓ **Cost Analysis**
- ✓ **Join Ordering Heuristics**
- ✓ **Experimental Results**
- ➢ **Related Work**
- ➢ **Conclusion and Future Work**

Related Work

- **Couger: Distributed sensor processing inside the sensor network**
- **Aurora: Allows user to create query plans by visually arranging query operators using boxes**
- **TelegraphCQ: For adaptive query processing**
- **STREAM: Addresses all aspects of data stream management, also proposed (CQL)**
- **Detar: uses combination of window and stream summary**
- **Babu and Widom uses stream constraints**
- **Some related work towards Join processing: XJoin, Hash-Join, Ripple Join, Multi-way XJoin called MJoin.**

INDEX

Conclusion

- **Presented and analyzed incremental, multi-way join algorithms for sliding window over data streams.**
- **Using per-unit-time based model, developed a join heuristic that finds a good join order without iterating over entire search space**
- **With experiments showed that hash-based joins performs better than NLJs and also discovered allocating more hash buckets to larger windows is a promising strategy**

Future Work

- **Goal is to develop a sliding window query processor that is functional, efficient, and scalable**
- **Functionality: intended to design efficient algorithms for other query operators as well.**
- **Efficiency: low-overhead indices for indexing window contents and also exploit constraints to minimize state**
- **Scalability: indexing query predicates, storing materialized views, and returning approximate answers if exact answers are too expensive to compute**

INDEX

Thank You

Malav Shah