ScaleJoin: a Deterministic, Disjoint-Parallel and Skew-Resilient Stream Join

Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, Philippas Tsigas

Chalmers University of technology
Agenda

• What is a stream join?
• Which are the challenges of a parallel stream join?
• Why ScaleJoin?
• How well does ScaleJoin addresses stream joins’ challenges?
• Conclusions
Agenda

• What is a stream join?
  • Which are the challenges of a parallel stream join?
  • Why ScaleJoin?
  • How well does ScaleJoin addresses stream joins’ challenges?
• Conclusions
Motivation

Applications in sensor networks, cyber-physical systems:
• large and fluctuating volumes of data generated continuously demand for:
  • Continuous processing of data streams
  • In a real-time fashion

Store-then-process is not feasible!!!
What is a stream join?

Data stream: unbounded sequence of tuples

Window size $WS$

Predicate $P$

Sliding window

$W_R$

$W_S$
Why parallel stream joins?

- WS = 600 seconds
- R receives 500 tuples/second
- S receives 500 tuples/second
- \( W_R \) will contain 300,000 tuples
- \( W_S \) will contain 300,000 tuples
- Each new tuple from R gets compared with all the tuples in \( W_S \)
- Each new tuple from S gets compared with all the tuples in \( W_R \)

... **300,000,000** comparisons/second!
Agenda

• What is a stream join?
• Which are the challenges of a parallel stream join?
• Why ScaleJoin?
• How well does ScaleJoin addresses stream joins’ challenges?
• Conclusions
Which are the challenges of a parallel stream join?

- Determinism
- Scalability
- High throughput
- Low latency
- Disjoint parallelism
- Skew resilience
Agenda

• What is a stream join?
• Which are the challenges of a parallel stream join?
• Why ScaleJoin?
• How well does ScaleJoin addresses stream joins’ challenges?
• Conclusions
The 3-step procedure (sequential stream join)

1. For each incoming tuple $t$:
   1. compare $t$ with all tuples in opposite window given predicate $P$
   2. add $t$ to its window
   3. remove stale tuples from $t$’s window

We assume each producer delivers tuples in timestamp order.
The 3-step procedure, is it enough?

- Determinism
- Low latency
- Scalability
- Disjoint parallelism
- High throughput
- Skew resilience
Enforcing determinism in sequential stream joins

• Next tuple to process = earliest(t_S,t_R)

• The earliest(t_S,t_R) tuple is referred to as the next *ready* tuple

• Process *ready* tuples in timestamp order → Determinism
Deterministic 3-step procedure

Pick the next ready tuple t:
1. compare t with all tuples in opposite window given predicate P
2. add t to its window
3. remove stale tuples from t’s window
Shared-nothing parallel stream join (state-of-the-art)

**Pick the next ready tuple t:**
1. compare t with all tuples in opposite window given P
2. add t to its window
3. remove stale tuples from t’s window
Shared-nothing parallel stream join
(state-of-the-art)
From coarse-grained to fine-grained synchronization

Prod S

Prod R

PU₁

PU₂

PUₙ

Cons
addTuple(tuple, sourceID) allows a tuple from sourceID to be merged by ScaleGate in the resulting timestamp-sorted stream of ready tuples.

getNextReadyTuple(readerID) provides to readerID the next earliest ready tuple that has not been yet consumed by the former.

https://github.com/dcs-chalmers/ScaleGate_Java
ScaleJoin

Steps for PU

Get next ready input tuple from \( SG_{in} \)
1. compare \( t \) with all tuples in opposite window given \( P \)
2. add \( t \) to its window \textit{in a round-robin fashion}
3. remove stale tuples from \( t \)’s window

Get next ready output tuple from \( SG_{out} \)
Sequential stream join:

ScaleJoin with 3 PUs:
ScaleJoin

Steps for PU_i

- Get next ready input tuple from SG_{in}
- compare t with all tuples in opposite window given P
- add t to its window in a round robin fashion
- remove stale tuples from t’s window
Agenda

- What is a stream join?
- Which are the challenges of a parallel stream join?
- Why ScaleJoin?
- How well does ScaleJoin addresses stream joins’ challenges?
- Conclusions
Evaluation setup

- Common benchmark

- Implemented in Java

- Evaluation platform
  - NUMA architecture: 2.6 GHz AMD Opteron 6230 (48 cores over 4 sockets), 64 GB of memory
  - Architecture with Hyper Threading: 2.0 GHz Intel Xeon E5-2650 (16 cores over 2 sockets), 64 GB of memory
ScaleJoin Scalability – comparisons/second

![Graph showing ScaleJoin Scalability across different numbers of PUs and execution times.](image)
ScaleJoin latency – milliseconds

Number of PUs vs. Latency (milliseconds)

- 5 Min.
- 10 Min.
- 15 Min.

2015-10-30
ScaleJoin skew-resilience
Constant distinct rates with peaks
Agenda

• What is a stream join?
• Which are the challenges of a parallel stream join?
• Why ScaleJoin?
• How well does ScaleJoin addresses stream joins’ challenges?
• Conclusions
Conclusions

• ScaleJoin: a Deterministic, Disjoint-Parallel and Skew-Resilient Stream Join

• Challenges of parallel stream joins

• Fine-grained synchronization (ScaleGate)

• 4 billion comparisons/second, with latency lower than 60 milliseconds
ScaleJoin: a Deterministic, Disjoint-Parallel and Skew-Resilient Stream Join

Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, Philippas Tsigas

Thank you! Questions?