SplitJoin: A Scalable, Low-latency Stream Join Architecture with Adjustable Ordering Precision

Mohammadreza Najafi

Joint work with:
Mohammad Sadoghi
Hans-Arno Jacobsen

Middleware Systems Research Group
Department of Informatics
Technical University of Munich

m.najafi@in.tum.de
USENIX ATC, June, 2016
Stream Processing Challenges

Applications

Algorithmic Trading  Targeted Advertising

Intrusion Detection

Stream join is a fundamental operation for relating information from different streams
Stream Join Semantic

Sliding window concept for unbounded streams
Stream Join Semantic

Sliding window concept for unbounded streams
Stream Join Semantic

Sliding window concept for unbounded streams
Basic Join Core Architecture

Large window buffers: expensive processing
Conventional Parallel Stream Join Architecture

Latency (average visiting latency):
Amount of time that it takes for two consecutive tuples (one from each stream) to be compared
Conventional Parallel Stream Join Architecture

Incurs inter-core communication overhead among join cores
Conventional Parallel Stream Join Architecture

Incurs inter-core communication overhead among join cores
Requires complex logic to avoid race conditions

Pratanu Roy, Jens Teubner, and Rainer Gemulla.
Low-latency handshake join. *PVLDB*’14
Conventional Parallel Stream Join Architecture

Incurs inter-core communication overhead among join cores
Requires complex logic to avoid race conditions
Incurs high latency due to bi-directional flow and chaining

Pratnu Roy, Jens Teubner, and Rainer Gemulla.
Low-latency handshake join. *PVLDB'*14
Conventional Parallel Stream Join Architecture

- **Join Core 1**
  - Requires complex logic to avoid race conditions.
  - Incurs high latency due to bi-directional flow and chaining.
  - Requires explicit expiration messages using a complex coordination unit.

- **Join Core 2**
  - Window-R_N/N

- **Join Core N**
  - Incurs inter-core communication overhead among join cores.

---

Pratanu Roy, Jens Teubner, and Rainer Gemulla. **Low-latency handshake join.** *PVLDB*’14

SPLITJOIN, USENIX ATC 2016
Novel (Parallel) Stream Join: SplitJoin

Splitting of the join computation into independent storage and processing steps

JC stands for Join Core
SplitJoin Independent Storage & Processing Concept

Splitting of the join computation into independent storage and processing steps
SplitJoin Parallelization

Break each window into smaller sub-windows
Join cores take turns in storing tuples
SplitJoin Data-flow

bi-directional data-flow

Coordinator

JJC

replicat-forward

top-down data-flow

Distributor

JJC

No inter-core communication
No race conditions
Significant reduction in latency
No expiry messages required
SplitJoin Abstract Architecture

//Creating SplitJoin join cores---------------------
JoinCore* joincores[THREAD_COUNT];
for(int i=0; i < THREAD_COUNT; i++){
    joincores[i] = new JoinCore(i, -1);
    joincores[i]->Rand_Init();}

//Create threads & linking them to the join cores---
pthread_t thread_ID[THREAD_COUNT];
for(int i=0; i < THREAD_COUNT; i++){
    pthread_create(&thread_ID[i], NULL, &JoinCore::JoinCore_Engine_helper, joincores[i]);
}

//Collecting the resulting tuples-------------------
pthread_t collector_thread_ID;
pthread_create(&collector_thread_ID, NULL, Collector, regions);

//Feeding the incoming streams to SplitJoin regions-
pthread_t feeder_thread_ID;
pthread_create(&feeder_thread_ID, NULL, Feeder, regions);
Query Processing Example

SELECT *
FROM R, S
WHERE R.ID == S.ID AND
    R.Price < S.Price
WITHIN 10 Minutes

In-order distribution and processing
Query Processing Example

SELECT *
FROM R, S
WHERE R.ID == S.ID AND
   R.Price < S.Price
WITHIN 10 Minutes

In-order distribution and processing
Query Processing Example

```
SELECT * 
FROM R, S 
WHERE R.ID == S.ID AND 
  R.Price < S.Price 
WITHIN 10 Minutes
```

In-order distribution and processing
Query Processing Example

SELECT *
FROM R, S
WHERE R.ID == S.ID AND
    R.Price < S.Price
WITHIN 10 Minutes

In-order distribution and processing
Query Processing Example

```
SELECT *  
FROM R, S  
WHERE R.ID == S.ID AND  
  R.Price < S.Price  
WITHIN 10 Minutes
```

In-order distribution and processing
Query Processing Example

```
SELECT *
FROM R, S
WHERE R.ID == S.ID AND
     R.Price < S.Price
WITHIN 10 Minutes
```

In-order distribution and processing
**Result Gathering Network** (Including Ordering Punctuations)

Join result ordering:
Necessity depends on application’s requirements

**Join result ordering precision:**
Parameterized strictness guarantee for join result ordering

**Punctuation mark:**
A marker placed into result stream to separate tuples into groups

Scalable result collection

Adjustable ordering precision and cost
Result Gathering Network (Including Ordering Punctuations)

Scalable result collection
Adjustable ordering precision and cost
Comparison (Conventional Solutions vs. SplitJoin)

- Bi-directional data-flow
  - Inter-core communication
    - Complex join core
    - Race prevention mechanism is needed
  - No simultaneous neighbor to neighbor communication
    - Reduces communication channel utilization
  - Global coordinator
    - Requires an expiry message per tuple
    - Reduces communication channel utilization
    - Increases the complexity of the system
  - Tuples’ arrival ordering is not preserved
- No ordering mechanism
  - Need to buffer

- Top-down data-flow
  - No inter-core communication
    - Simple join core
    - Higher performance
    - No race condition
      - Tuples have only one path to each join core
      - Full utilization of communication channels
  - No global coordination is required
    - No need for expiry messages
  - Tuples’ arrival order is inherently preserved
- Punctuated ordering
  - Adjustable precision
- Failure resilience
Experimental Setup

- A 32-core Dell PowerEdge R820 featuring 4 × Intel E5-4650 processors
  - Ubuntu 14.04.2 LTS, Docker container

- Two streams \( R = (x:\text{int}, y:\text{float}, z:\text{char}[20]) \) and \( S = (a:\text{int}, b:\text{float}, c:\text{double}, d:\text{bool}) \) are joined via a two-dimensional band join, as follows:

\[
\text{WHERE } r.x \text{ BETWEEN } s.a-10 \text{ AND } s.a+10 \\
\text{AND } r.y \text{ BETWEEN } s.b-10 \text{ AND } s.b+10
\]

- For time-based window, we used `gettime()` to generate time-stamps
  - Using synthetic timing improves the overall performance by about 15%
Throughput Comparison

Significant performance improvement ~60%
Latency Comparison

More than 3 times reduction

Substantial latency reduction
SplitJoin Processing Pipeline Latency Measurements

SplitJoin detailed latency reports
Ordering Precision Overhead

Ordering precision effect on throughput
Conclusions

- **SplitJoin** is a scalable stream join architecture that removes inter-core communications and dependencies
  - A splitting abstraction to “process” and “store” incoming data streams concurrently and independently
  - A top-down data flow for a coordination-free distribution and parallelization of stream join processing
  - A distribution tree with logarithmic access latency

- **SplitJoin** breaks the stream join with window size of $X$ into smaller (but similar) stream joins with window sizes of $X/N$
  - As a result, other stream join algorithms (e.g., indexed-based stream join) are applicable to each individual join core without affecting the architecture of **SplitJoin**

- **SplitJoin** is accompanied by an adjustable punctuated ordering mechanism