Parallelization of Massive Multiway Stream Joins on Manycore CPUs

Constantin Pohl, Kai-Uwe Sattler

TU Ilmenau

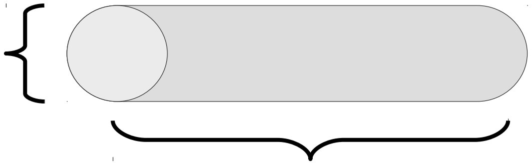
Databases and Information Systems Group

Ilmenau, Germany

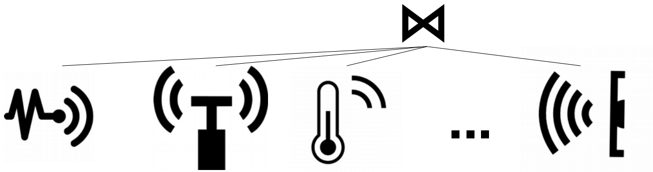
Introduction

Parallelism Opportunities
by Manycore CPUs

Throughput and *Latency* demands
of Data Stream Processing

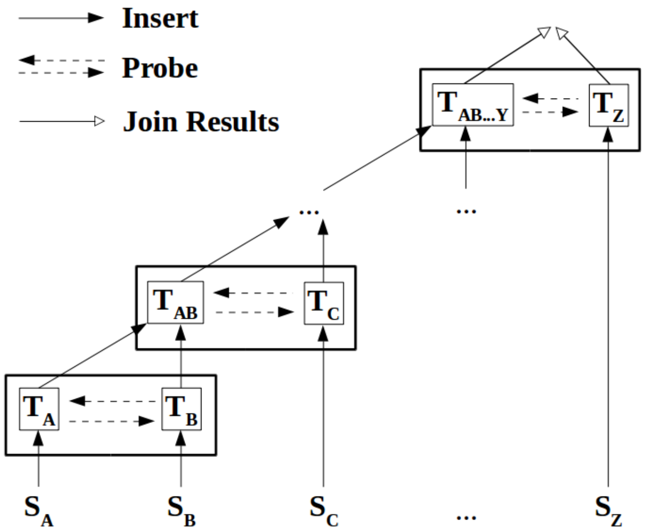


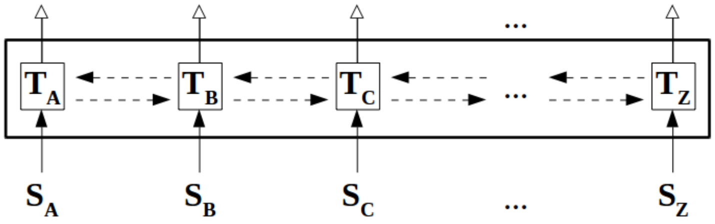
Joining many stream sources
running concurrently



Introduction

Binary Join (tree) vs. Multiway Join



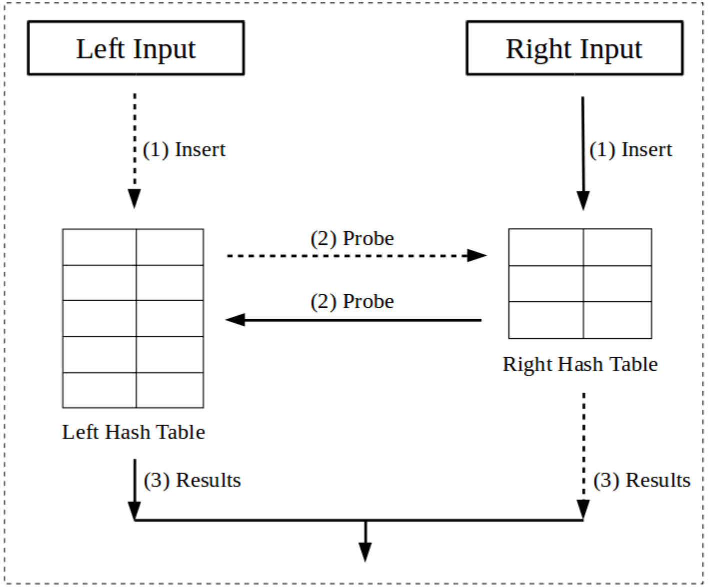


Research Questions

- **Performance**
  - High number of joinable streams
  - Binary join tree vs. single Multiway join
- **Scaling**
  - Manycore CPU
  - Opportunities of Multiway join
- **Synchronization**
  - Shared join execution
  - Data access

Binary: Symmetric Hash Join (SHJ)

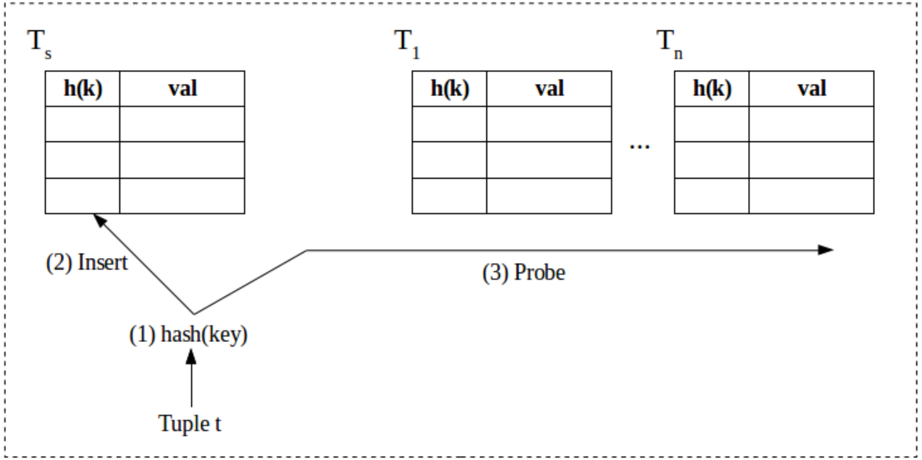
- **Non-Blocking Join**
  - Producing results continuously
- **Low individual tuple latency**
  - Insert, probe, return result(s)
- **Binary**
  - Cascading SHJ operators for 3+ joinable streams (join tree)



conceptual view of SHJ

Multiway: MJoin[1]

- **One hash table T per input stream**
- **Tuple t arrives from stream s:**
  - Hash key,
  - insert into T_s,
  - probe all other $(T-T_s)$

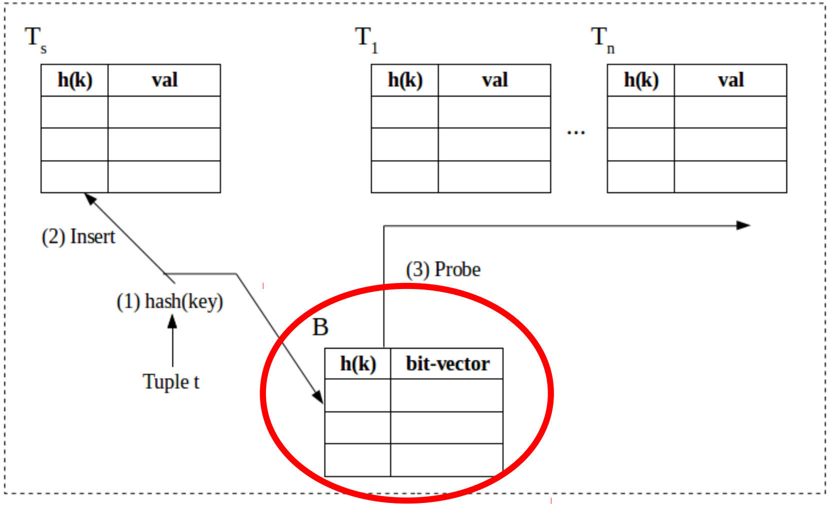


conceptual view of MJoin

[1]Luping Ding, Elke A. Rundensteiner, George T. Heineman: *MJoin: a metadata-aware stream join operator*, DEBS 2003

Multiway: AMJoin[2]

- **Advanced MJoin**
- **<u>Central idea</u>:**
  Avoid unnecessary probes
  - Additional bit-vector hashtable B for tracking key presence:
    - one vector per key hash value,
    - vector length of #streams/tables
  - Whole bit-vector positions set to 1: initiate join execution



conceptual view of AMJoin

[2]Tae-Hyung Kwon, Hyeon Gyu Kim, Myoung-Ho Kim, Jin Hyun Son: *AMJoin: An Advanced Join Algorithm for Multiple Data Streams Using a Bit-Vector Hash Table*, IEICE Transactions 92-D(7): 1429-1434 (2009)

Optimizations: OptAMJoin

- **AMJoin experiments of the paper[2]:**
  - 5 joinable streams (5-way),
  - Intel Core2 Duo 2.66 GHz (2 cores, Win XP, 4GB RAM)

- **Optimizations proposed for 100+ streams & manycore CPU:**
  - Atomic counters vs. bit-vectors
  - Array vs. Hash table for dense key space
  - Lock-free vs. Locks/Latches
  - (Parallelization schema)

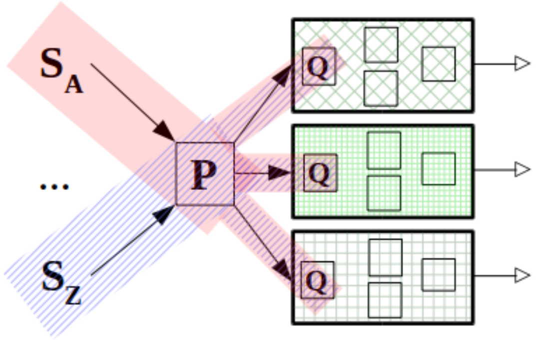
=> OptAMJoin

[2]Tae-Hyung Kwon, Hyeon Gyu Kim, Myoung-Ho Kim, Jin Hyun Son: *AMJoin: An Advanced Join Algorithm for Multiple Data Streams Using a Bit-Vector Hash Table*, IEICE Transactions 92-D(7): 1429-1434 (2009)

Parallelization Strategies

- ***Data Parallelism*:**
  - Routing tuples to partitions by Partitioner P (key range determines partition)
  - Join execution in each partition independent from other partitions (own thread)
  - Tuple exchange with queues Q

- **(Dis-)Advantages:**
  - \+ Scale out
  - \+ Partition synchronization
  - \- Load balancing (key ranges)
  - \- Partitioner overhead
  - \- Queue delay

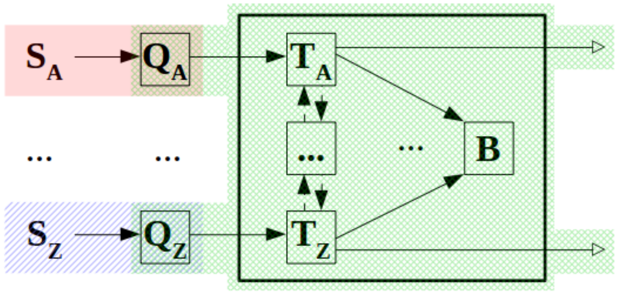


Data Parallelism

Parallelization Strategies

- ***SPSC-Paradigm*:**
  - “Single Producer, Single Consumer”
  - Streams write to own SPSC queue
  - Single join instance collects tuples & manages whole join

- **(Dis-)Advantages:**
  - \+ No internal join synchronization
  - \- Queue delay
  - \- Possible overwhelming of join thread (high tuple arrival rates)

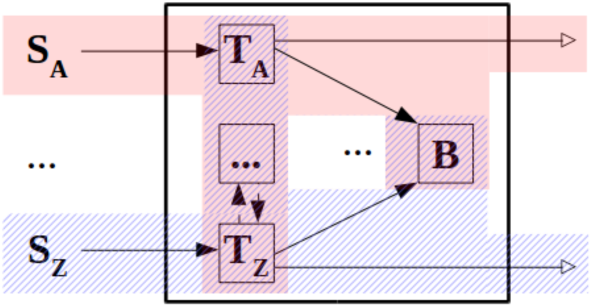


SPSC-Paradigm

Parallelization Strategies

- ***Shared Data Structures*:**
  - Join tables & Bit-vector table shared to all stream threads
  - Stream threads perform join individually

- **(Dis-)Advantages:**
  - \+ No additional efforts (queues, partitioner)
  - \- Scaling increases contention drastically
  - \- Handling of duplicates/out of order tuples



Shared Data Structures

Evaluation Setup (I)

- Xeon Phi KNL 7210, 64 cores (à 4 threads), <1.5GHz, 96GB DDR4 (SNC-4, MCDRAM unused (flat))
- Implemented in Stream Processing Engine PipeFabric[3]
- Main query to execute:

SELECT *
FROM Stream S_1, S_2, ... , S_{N-1}, S_N
SLIDING WINDOW(1000000)
WHERE S_1.key $=$ S_2.key
AND ...
AND S_{N-1}.key $=$ S_N.key

[3]open source, *https://github.com/dbis-ilm/pipefabric*

Evaluation Setup (II)

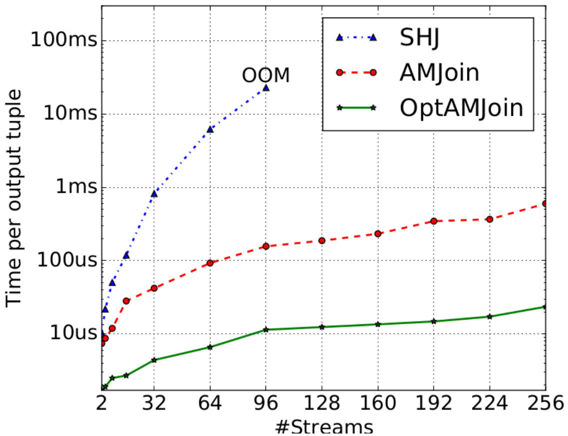
- **Tuples**
  - <key,value> pairs (8+8byte)
  - 1m distinct keys
  - shuffled randomly per stream

- **SHJ**
  - Left-deep tree

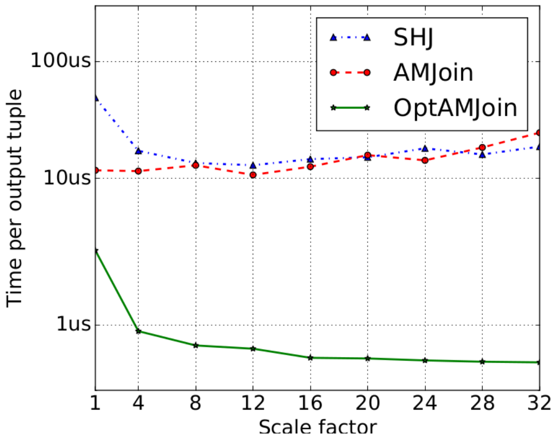
- **Weak & Strong scaling**
  - **weak:** increasing number of joinable streams (=threads)
  - **strong:** 8 streams to join, increasing join instances merging results

Evaluation (I)

SHJ, AMJoin, OptAMJoin with ***Shared Data Structures***
Parallelization



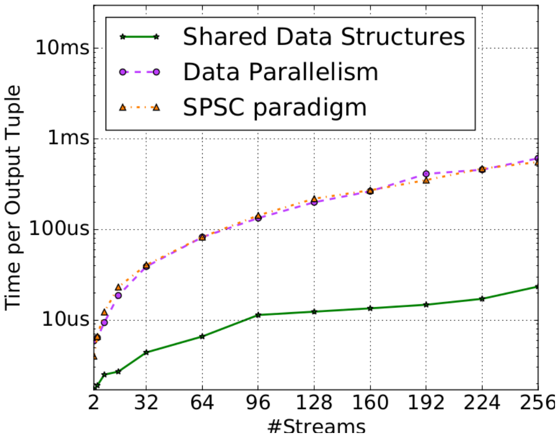
Weak Scaling



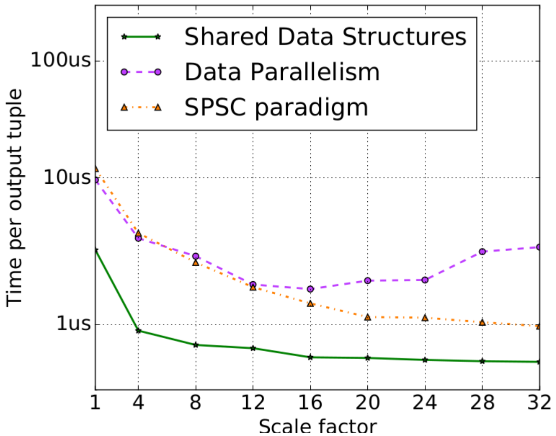
Strong Scaling

Evaluation (II)

OptAMJoin with all three parallelization strategies



Weak Scaling



Strong Scaling

Evaluation (III)

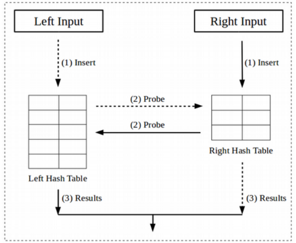
Memory footprints of all three join algorithms in GB

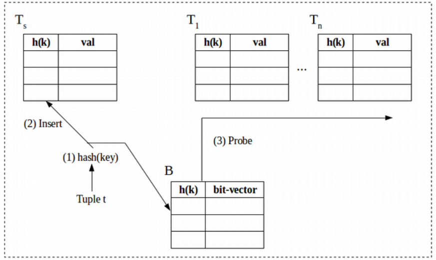
Streams	SHJ	AMJoin	OptAMJoin
2	0.260	0.253	0.106
8	1.646	0.745	0.419
16	4.328	1.400	0.836
64	40.449	5.334	3.339
256	528.253	21.079	13.353

Conclusion

- Multiway join performance is superior to binary join trees
- Shared data structures with lock-free synchronization and no additional buffers (queues) perform best
- May change under heavily-skewed streams (contention)

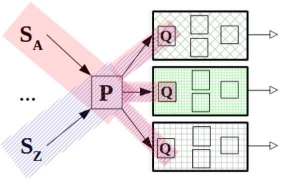
Summary

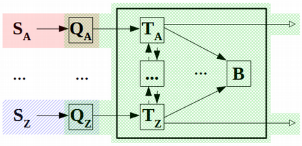


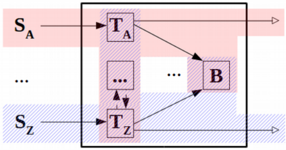


- **Join algorithms**:
  - **SHJ:** Binary
  - **MJoin:** Multiway
  - **AMJoin:** MJoin + bit-vector hash table
  - **OptAMJoin:** AMJoin + optimizations (counter, array, lock-free)

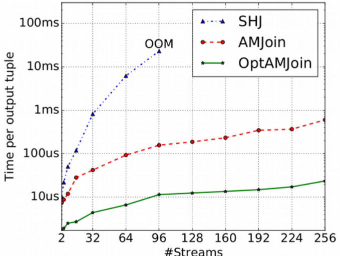
- **Parallelization strategies**:
  - *Data parallelism*
  - *SPSC-paradigm*
  - *Shared data structures*

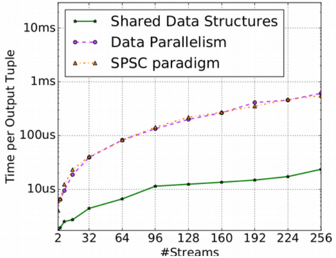






- **Evaluation**:
  - Algorithms
  - Parallelization
  - Memory consumption





Streams	SHJ	AMJoin	OptAMJoin
2	0.260	0.253	0.106
8	1.646	0.745	0.419
16	4.328	1.400	0.836
64	40.449	5.334	3.339
256	528.253	21.079	13.353