Parallelizing Intra-Window Join on Multicores: An Experimental Study

Shuhao Zhang

Singapore University of Technology and Design

shuhao.zhang@sutd.edu.sg

March 25, 2021

Overview

Motivation Example: Joining of Streams

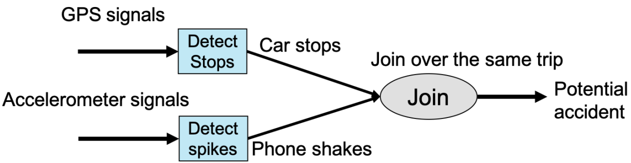


Figure: How Uber Detects on Trip Car Crashes – Nicolas Anderson & Jin Yang, Uber (Flink Forward, Oct, 2019)

- **Task:** Combines GPS signals and accelerometer signals to infer accidents
- **Implementation:** The combination needs to be performed over data streams – stream join

Window-based Stream Join

- **Windows**: bounded subsets of streams
- **Inter-Window join**: joins over a series of windows
  - Most studies pay more attention to this type of stream join.
  - They especially focus on the cases where there are windows overlap.

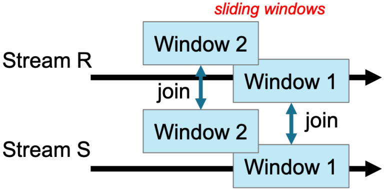


Figure: Inter-Window Join

The Case of Intra-Window Join

- **Intra-window join:** joins over one window of data streams
- It concerns the case when users are only interested at one particular subset of streams
  - One example is the Uber's detection of car accidents: at each time, the focus is to join streams over a *duration of the same trip* (i.e., one window).
  - Another example is at Pinterest, developers join the activation record per user for a *single time window of three days* [Pinterest19].

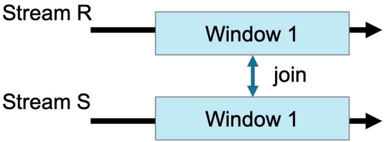


Figure: Intra-Window Join (also called "online join" [Elseidy14] and "Full-history join" [Lin15])

Literature Gaps

- Early work of Intra-Window Join historically focuses on its single-thread execution efficiency and is no longer suitable on modern multicore architectures.
- Recent works on parallelizing stream join processing are geared towards Inter-Window Join. Only two works concern parallelizing Intra-Window Join [Elseidy14] [Lin15].
- We observe many alternative ways to implement parallel Intra-Window Join and it remains unclear which way is the most efficient approach.

Challenges

There are particularly three main challenges in resolving the aforementioned literature gaps.

- C1 A large design space: lazy/eager; hashing/sorting; etc.
- C2 Conflicting performance metrics: throughput, latency, and progressiveness.
- C3 Modern hardware features: SIMD, multicore parallelism and NUMA effects

Methodology Overview

- Prior studies [Elseidy14, Lin15] are usually biased at algorithm, metrics, hardware architectures.
- We propose the first comprehensive benchmark suite: 8 different algorithms 4 real world datasets & 1 carefully designed synthetic dataset.
- We evaluate the benchmark suite on a recent (2017 Q4) modern multicore server with different hardware settings.

Algorithm Design Aspects

- It is difficult and unnecessary to implement *all* join algorithms in our benchmark suite.
- Instead, we propose to select a *representative subset* of algorithms so that they can well cover the following key algorithm design aspects.

Key Algorithm Features

- Execution Approach
- Join Method
- Partition Schemes

Execution Approaches

- **Lazy**: waits for all input tuples of the concerned window from both input streams to arrive, and then joins a complete set of tuples.
- **Eager**: aggressively joins upon the arrival of only a subset of input tuples.

Join Methods

- **Hash**: construct hash table (build) and query the hash table for generating matches (probe).
- **Sort**: sort input relation (sort) and merge two sorted relations while generating matches (merge & match)

Partition Schemes

- **W/ or W/o physical partition**: whether the algorithm replicates input relations among threads before execution.
- **Content-sensitive/insensitive distribution**: whether the algorithm shuffle or key-based partitioning input relations among threads.

Selected Algorithms

Table: Summary of studied join algorithms

Name	Approach	Join Method	Partitioning Schemes
NPJ [Blanas11]	Lazy	Hash	No physical partitioning
PRJ [Kim09]	Lazy	Hash	Cache size-aware replication
MWAY [Chhugani08]	Lazy	Sort	Equisized range partitioning
MPASS [Balkesen13]	Lazy	Sort	Equisized range partitioning
SHJ^{JM} [Wilschut91]+[Elseidy14]	Eager	Hash	Content-insensitive stream distribution
SHJ^{JB} [Wilschut91]+[Lin15]	Eager	Hash	Content-sensitive stream distribution
PMJ^{JM} [Dittrich02]+[Elseidy14]	Eager	Sort	Content-insensitive stream distribution
PMJ^{JB} [Dittrich02]+[Lin15]	Eager	Sort	Content-sensitive stream distribution

Remark

Covers all the design aspects.

Benchmark Datasets

Table: Statistics of four real-world workloads (Window length w=1sec)

	Arrival rate (tuples/ms)	Key duplicates	Key skewness (Zipf)	Number of tuples
Stock	υ_R=61, υ_S=77	$dupe(R)\approx 67.7$, $dupe(S)\approx 78.5$	$skew_{key}(R)=0.112$, $skew_{key}(S)=0.158$	$\|R\|$ $(\|S\|) = \upsilon_{R(S)} \cdot w$
Rovio	$\upsilon_R\approx 3\cdot 10^3$, $\upsilon_S\approx 3\cdot 10^3$	$dupe(R)=dupe(S)=17960.0$	$skew_{key}(R)=0.042$, $skew_{key}(S)=0.042$	$\|R\|$ $(\|S\|) = \upsilon_{R(S)} \cdot w$
YSB	$\upsilon_R=\infty$, $\upsilon_S\approx 10^4$	$dupe(R)=1$, $dupe(S)=10^5$	$skew_{key}(R)=0.033$, $skew_{key}(S)=0.032$	$\|R\|=1000$, $\|S\|=\upsilon_S \cdot w$
DEBS	$\upsilon_R=\infty$, $\upsilon_S=\infty$	$dupe(R)\approx 172.6$, $dupe(S)\approx 111.5$	$skew_{key}(R)=0.003$, $skew_{key}(S)=0.011$	$\|R\|=10^6$, $\|S\|=10^6$

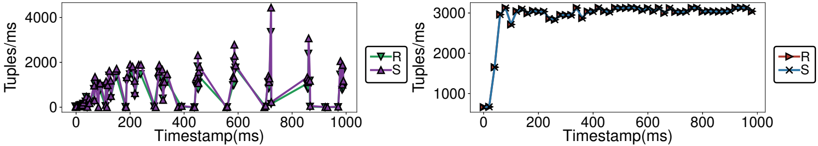


Figure: Time distribution of Stock and Rovio. Other uniform arrival datasets are shown as horizontal lines and omitted.

Implementations

- **Algorithm Implementation:** Re-implement 8 algorithms according to the original papers in the same codebase (C++)
- **Dataset Structure:** Assume a narrow $\langle$ key, payload $\rangle$ tuple configuration with 64 bits length.
- **Profiling Methods:** Read Time-Stamp Counter (RDTSC) to measure progress, Intel PCM and Perf to gather architectural statistics

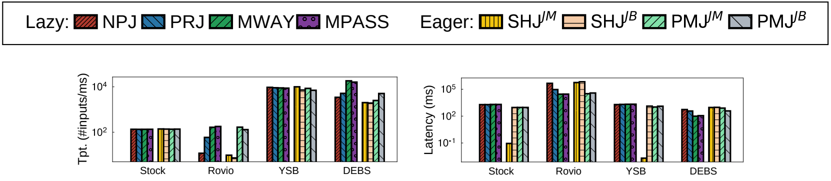
Environment Settings

Table: Specification of our evaluation platform

Component	Description
Processor (w/o HT)	Intel(R) Xeon(R) Gold 6126 CPU, 2 (socket) * 12 * 2.6GHz
L3 cache size	19MB
Memory	64GB, DDR4 2666 MHz
OS & Compiler	Linux 4.15.0, compile with g++ O3

To exclude the impact of NUMA, we use only one socket for our experiments. We leave the evaluation of NUMA in future work.

Overall Comparison on Throughput and Latency

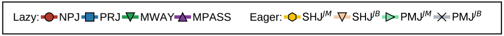


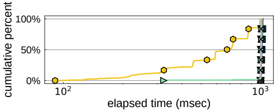
(a) Throughput (b) Latency

Figure: Throughput and latency comparison.

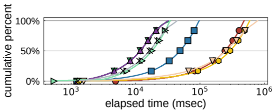
Lazy approach brings higher throughput; while eager approach achieves lower latency in some workloads.

Overall Progressiveness Comparison

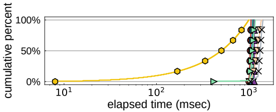




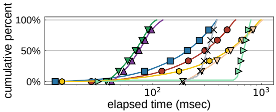
(a) Stock



(b) Rovio



(c) YSB



(d) DEBS

Figure: Progressiveness comparison.

Eager approach can not guarantee faster progress.

Where Does Time Go?

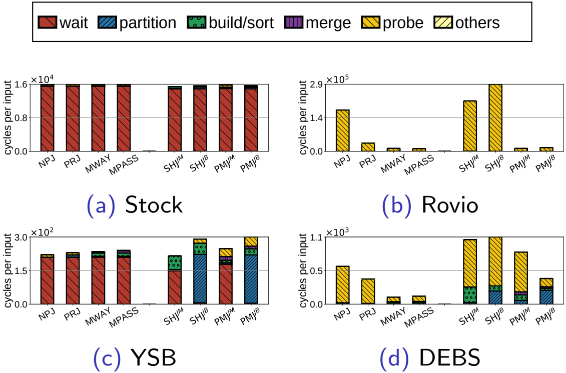
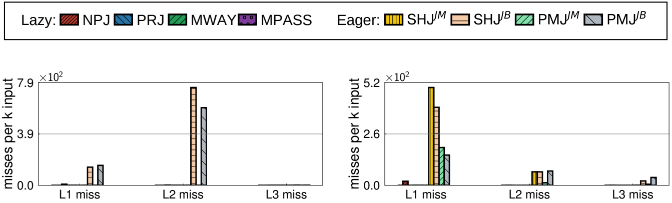


Figure: Execution time breakdown.

Lazy approach does spend more in waiting, but eager approach spends more in others, especially in *partition* and *probe* phase.

What Happened During Partition and Probe?



(a) Partition (b) Probe

Figure: Execution time breakdown.

- Partition: the uncontrolled random access of content-sensitive partition scheme (JB) leads to high cache misses.
- Probe: frequent interleave access to two input streams results in severe cache trashing for all eager algorithms.

Other Experiments

More experimental results are discussed in our paper.

- Impact of Workloads
- Impact of Algorithm Configurations
- Impact of Multicore and SIMD

Open Sourced Benchmark

For more details, please checkout our benchmark at
`https://github.com/ShuhaoZhangTony/AllianceDB`

The Guide to Appropriate Algorithm

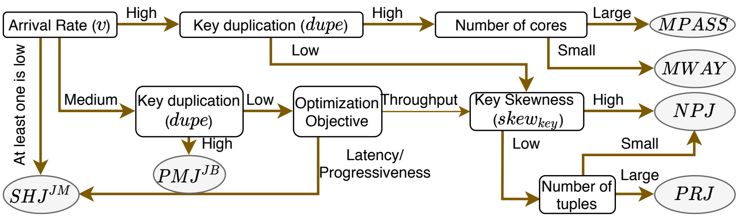


Figure: Decision tree for picking an appropriate algorithm. The root node of the tree is the arrival rate node.

Remark

In a nut shell, no one size fits all.

Conclusion and Future Work

1. Adaptive Intra-Window Join algorithm that considers all the factors including workload, metrics and hardware is needed.
2. It is important to further extend this study to include more hardware architectures such as NUMA, HBM, GPUs, and FPGAs.
3. Joint efforts from relational DB and stream processing communities for other important operations, e.g., stream aggregation.

References I

Pinterest (2019)
Real-time-experiment-analytics-at-pinterest-using-apache-flink
https://medium.com/pinterest-engineering/real-time-experiment-analytics-at-pinterest-using-apache-flink-841

Q.Lin, B.C.Ooi, Z.Wang, and C.Yu (2015)
Scalable distributed stream join processing.
In Proc. SIGMOD, pages 841–852.

M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch (2014)
Scalable and adaptive online joins
Proc. VLDB Endow., 7(6):441–452.

References II

S.Blanas, Y.Li, and J.M.Patel. (2011)
Design and evaluation of main memory hash join algorithms for multi-core cpus
In Proc. SIGMOD, page 37–48.

C.Kim, T.Kaldewey, V.W.Lee, E.Sedlar, A.D.Nguyen, N.Satish, J.Chhugani, A. Di Blas, and P. Dubey. (2009)
Sort vs. hash revisited: Fast join implementation on modern multi-core cpus
Proc. VLDB Endow., 2(2):1378–1389.

J.Chhugani, A.D.Nguyen, V.W.Lee,W.Macy, M.Hagog, Y.-K.Chen, A.Baransi, S. Kumar, and P. Dubey. (2008)
Efficient implementation of sorting on multi-core simd cpu architecture.
Proc. VLDB Endow., 1(2):1313–1324.

References III

C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. (2013)
Multi-core, main-memory joins: Sort vs. hash revisited
Proc. VLDB Endow., 7(1):85–96.

A. N. Wilschut and P. M. G. Apers. (1991)
Dataflow query execution in a parallel main-memory environment
In Proc. ICPADS, pages 68–77.

J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer (2002)
Progressive merge join: A generic and non-blocking sort-based join algorithm
In Proc. VLDB, pages 299–310.

The End