Low-Latency Handshake Join

Pratanu Roy
Systems Group, Department of Computer Science
ETH Zürich, Switzerland
pratanu@inf.ethz.ch

Jens Teubner
DBIS Group, Department of Computer Science
TU Dortmund University, Germany
jens.teubner@cs.tu-dortmund.de

Rainer Gemulla
Max-Planck-Institut für Informatik
Saarbrücken, Germany
rgemulla@mpi-inf.mpg.de

ABSTRACT

This work revisits the processing of stream joins on modern hardware architectures. Our work is based on the recently proposed *handshake join* algorithm, which is a mechanism to parallelize the processing of stream joins in a NUMA-aware and hardware-friendly manner. Handshake join achieves high throughput and scalability, but it suffers from a high latency penalty and a non-deterministic ordering of the tuples in the physical result stream. In this paper, we first characterize the latency behavior of the handshake join and then propose a new *low-latency handshake join* algorithm, which substantially reduces latency without sacrificing throughput or scalability. We also present a technique to generate *punctuated result streams* with very little overhead; such punctuations allow the generation of correctly ordered physical output streams with negligible effect on overall throughput and latency.

1. INTRODUCTION

With the ongoing adoption of multi-core machines, the need to devise suitable parallel algorithms keeps increasing. A major challenge is that high degrees of parallelism heavily emphasize locality effects in the hardware. Only very recently, a few algorithms have been suggested that are prepared for the *non-uniform memory access (NUMA)* characteristics of modern hardware (e.g., [16, 20]).

In this paper, we revisit the processing of data streams in multi-core systems. The computation of *sliding-window joins* is among the most critical and resource-intensive operations in this context. The *handshake join* algorithm, which was recently proposed by Teubner and Mueller [20], provides a NUMA-aware solution for general sliding-window joins and achieves high throughput as well as good scalability with the number of compute cores.

The favorable performance characteristics of handshake join come at the cost of *high latency* and *non-deterministic output order*. We perform an in-depth analysis of the worst-case latency of handshake join. We find that its maximum latency depends linearly on the sliding-window size. Our experiments show that even for moderately-sized windows consisting of a few minutes of data from both input streams, the latency of handshake join can be in the range of tens of seconds. Moreover, the order of the output stream produced by handshake join is non-deterministic, and it is not clear how a deterministic output ordering can be enforced without sacrificing throughput or further increasing latency. Both, high latency and non-deterministic output order, limit the usefulness of the handshake join in real-world stream processors.

To alleviate these drawbacks, we propose *low-latency handshake join* (LLHJ) as an alternative to the handshake join algorithm. Low-latency handshake join maintains the favorable throughput and scalability characteristics of handshake join, but reduces latency by multiple orders of magnitude to the millisecond scale. Moreover, low-latency handshake join can produce a deterministically ordered output stream with negligible runtime and space overhead through a *punctuation mechanism*.

In summary, the main contributions of this paper are:

(i) We carefully analyze and experimentally verify the latency characteristics of handshake join, the state of the art in stream join processing.

(ii) We propose low-latency handshake join, which makes use of a *tuple expedition* mechanism to improve upon handshake join. Rather than queuing up tuples in a processing pipeline—the main source of latency in handshake join—, our low-latency handshake join fast-forwards tuples between CPU cores. Low-latency handshake join maintains the NUMA-efficient point-to-point communication pattern as well as the semantic guarantees of handshake join, but reduces latency by multiple orders of magnitude.

(iii) We show how low-latency handshake join can be modified to generate a *punctuated output stream*. This allows for deterministically ordered output streams with negligible space and time overhead.

The remainder of this paper is organized as follows. Section 2 summarizes the stream join problem and its existing solutions. In Section 3, we study the latency characteristics of the handshake join. We then present our low-latency handshake join in Section 4, followed by a discussion on result generation (Section 5) and output order (Section 6).

Proceedings of the VLDB Endowment, Vol. 7, No. 9

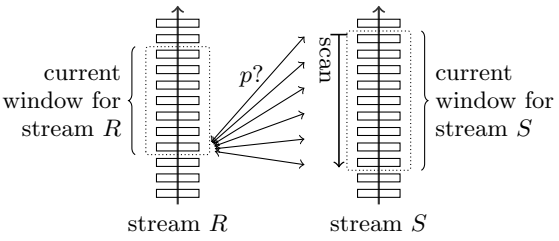


Figure 1: Three-step procedure of Kang *et al.* [10]. Every newly arrived tuple (here: of stream R) initiates a scan of the opposite window, testing the join condition p.

The results of our experimental study as well as relevant related work are summarized in Sections 7 and 8, respectively. Section 9 concludes this paper.

2. BACKGROUND: STREAM JOINS

Input data for streaming operators is, by definition, infinite in size. Hence, it must be cut into finite slices to make operators like joins ($\bowtie$) semantically sound. The prevalent way of slicing an infinite stream is the use of *sliding windows*, where the current slice ("window") at any moment consists of the last-seen portion of the stream. Practical definitions are *tuple-based windows* (the last k tuples that arrived for the stream) and *time-based windows* (tuples that arrived during the last τ time units).

2.1 Kang's Three-Step Procedure

Kang *et al.* [10] were the first to describe a streaming join operator. Each newly arriving tuple r of stream R is processed in three steps:

1. *Scan* the window associated with input stream S and look for matching tuples;
2. *Invalidate* old tuples in both windows (depending on the window specification);
3. *Insert* r into the window associated with R.

The join problem is inherently symmetric, and tuples s arriving from stream S are handled in a symmetric fashion.

We illustrated the *scan* part of the procedure in Figure 1. After arrival of a new tuple r, the window associated with S is scanned and for each tuple the *join predicate* $p(r, s)$ is evaluated to find matching pairs.

Latency Characteristics. The earliest moment when a result tuple $\langle r, s\rangle$ can possibly be produced is when the later of the two tuples r and s has arrived. In this sense, Kang's procedure offers optimal latency characteristics, since it tries to find all possible matches immediately on arrival of any tuple.

2.2 Parallelizing the Three-Step Procedure

Observe that Kang's procedure is inherently sequential. Since window states may change with every arriving tuple, setups that process new arrivals in parallel or in nondeterministic order may lead to different—incorrect—join output.

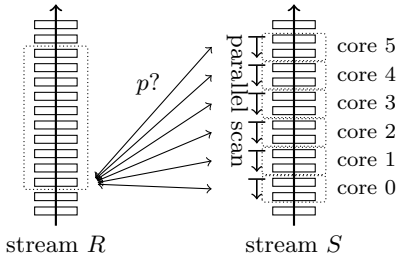


Figure 2: CellJoin [9], a parallel version of Kang's three-step procedure. Upon every arrival of a tuple, the opposing window is *re-partitioned* to perform a parallel scan.

2.2.1 CellJoin

Gedik *et al.* [9] have thus suggested to stick with the three-step procedure even in parallel environments, but parallelize the *scan* task over available processing units.

The idea, realized as the CellJoin algorithm, is illustrated in Figure 2. On arrival of a new tuple (say from stream R), the in-memory window of the opposite input stream (say S) is (re-)partitioned. The resulting partitions S_i are assigned to available CPU cores, which together perform a parallel window scan.

CellJoin inherits favorable latency characteristics from Kang's three-step procedure (with a slight overhead resulting from now-necessary core-to-core communication). The price for the favorable latency, however, is the limited scalability of CellJoin. Windows must be re-partitioned upon every new input tuple, an overhead that grows linearly with the core count and the input stream rate [9]. CellJoin has not been designed with NUMA environments in mind. Window re-partitioning and input tuple replication (to every node) depend on a globally shared memory—a system model known to scale poorly with core counts.

2.2.2 Stream Joins on Heterogeneous Hardware

Data parallelism was used in a similar fashion also in other settings to accelerate the evaluation of stream joins. Karnagel *et al.* [11, 12], for instance, showed how the idea is a good fit to implement stream processing on graphics processors (GPUs). Specifically, their *HELLS*-Join algorithm benefits from the high memory bandwidth available in modern GPUs to scan potentially large tuple windows efficiently.

2.3 Handshake Join

Handshake join [20] replaces the three-step procedure by a *data flow-oriented* setup, illustrated in Figure 3. Both input streams notionally *flow* through the stream processing engine, in opposing directions. The two sliding windows are laid out side by side and predicate evaluations are performed along the windows whenever two tuples *encounter* each other. The mechanism resembles *soccer players* and the handshake ceremony that they perform before important matches (hence the term "handshake join").

The data flow-oriented view lends itself to be processed in parallel. To this end, handshake join's processing pipeline is divided into *segments* which are then assigned to individual processing units [20]. The data flow then becomes a physical

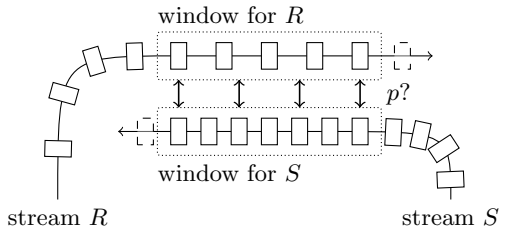


Figure 3: Handshake join mechanism. Streams flow by each other in opposite directions; comparisons (and result generation) happens in parallel as the streams pass by.

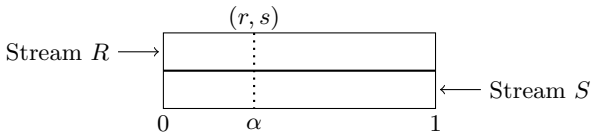


Figure 4: Latency analysis of handshake join.

movement of data from CPU to CPU.

The model is an excellent fit for the characteristics of modern multi-core architectures. Communication is restricted to local point-to-point messages between neighboring cores; all memory accesses are local; and no central coordinator (a potential scalability bottleneck) is needed to orchestrate the cores in the system. In [20], it was shown that handshake join can scale linearly to at least hundreds of processing units (cores).

The down side of this scalability is that tuples may have to queue for longer periods of time before they encounter matching partners. A potentially high *latency* is the consequence. For example, two tuples r and s that arrive at close-by time stamps will both have to travel about half of their join window before $\langle r, s\rangle$ is detected as a join result.

3. ANALYZING HANDSHAKE JOIN

Handshake join was designed with throughput as the primary optimization goal. The resulting latency penalty was largely ignored in previous work [20]. To better understand—and later avoid—the latency cost, in this section we quantify the incurred latency analytically as well as experimentally.

3.1 Latency Model

Consider two streams R and S that flow through a handshake join instance as illustrated in Figure 4. At time stamp T (the current time) two tuples $r \in R$ and $s \in S$ encounter each other at position α along the processing pipeline of handshake join, α ranging from 0 (left end of the pipeline) to 1 (right end). We denote the windows associated with R/S as W_R/W_S and their sizes with $|W_R|/|W_S|$, respectively.

Assuming that the two streams are in a steady flow, at time T, r has travelled the fraction α of the overall R window (t_r is the arrival time stamp of r):

$$T = t_r + \alpha \times |W_R| \ . \tag{1}$$

Likewise, s has left the fraction $(1-\alpha)$ of W_S behind before meeting r at time T:

$$T = t_s + (1-\alpha) \times |W_S| \ . \tag{2}$$

Combined, the two equations yield

$$t_r - t_s = |W_S| - \alpha \times (|W_R| + |W_S|) \ . \tag{3}$$

The reference point for latency is the time stamp of the tuple that arrived later. Suppose r arrived after s (*i.e.*, $t_r - t_s > 0$), then

$$|W_S| > \alpha \times (|W_R| + |W_S|)$$

or

$$\alpha < \frac{|W_S|}{|W_R| + |W_S|} \ . \tag{4}$$

Analogously, if s arrived after r, we obtain

$$\alpha > \frac{|W_S|}{|W_R| + |W_S|} \ . \tag{5}$$

The observed latency is $T - \max(t_r, t_s)$, *i.e.*, the time needed to detect the match, starting from the arrival of the later tuple. By inserting Equation 1 into Equation 4 and Equation 2 into Equation 5, we get

$$T - t_r = \alpha \times |W_R| < \frac{|W_S| \times |W_R|}{|W_R| + |W_S|} \tag{6}$$

$$T - t_s = (1-\alpha) \times |W_S| < \frac{|W_S| \times |W_R|}{|W_R| + |W_S|} \ . \tag{7}$$

Thus, for the observed latency of handshake join we get

$$T - \max(t_r, t_s) < \frac{|W_S| \times |W_R|}{|W_R| + |W_S|} \ . \tag{8}$$

A typical situation is when both windows have the same size, *i.e.*, $|W_R| = |W_S| = |W|$. The expected maximum latency then is $1/2 \times |W|$.

In practice, such latency is significant. To illustrate, for the benchmark scenario considered in [9] and [20], a window size of 15 min will correspond to latencies of up to 7.5 min!

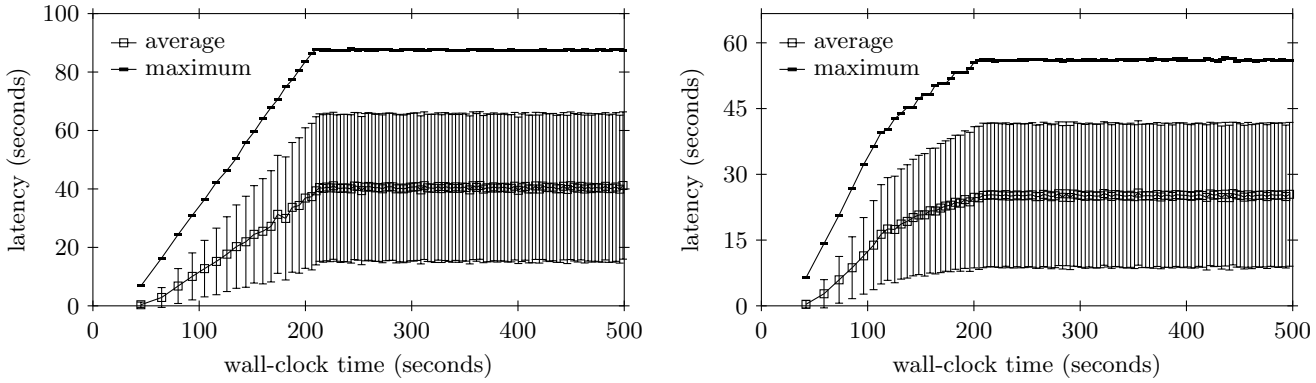
3.2 Experimental Verification

To verify the correctness of our model, we analyzed the latency distribution of handshake join. To this end, we used the original implementation which is publicly available and modified it to generate accurate time stamps for the input and output streams. For our experiment, we used a server machine with an AMD Opteron 6174 processor. We used 40 cores for the experiment.

Generally, latency in handshake join exhibits a very high variance. Hence, in Figure 5 we plotted the observed latency as a moving average, together with the corresponding standard deviation. We also show maximum latencies that we observed.

The x-axis in these figures is the wall-clock time as we run our experiment. At time 0, the experiment starts with empty windows. Latency gradually increases as the join windows fill up (and sufficiently old tuples exist in the windows at all), until both average and maximum latencies reach stable values (here: at $T = 200$ seconds $= \max(|W_R|, |W_S|)$).

Our latency model of Section 3.1 predicts a maximum latency of $\frac{|W_R| \times |W_S|}{|W_R| + |W_S|}$ (*i.e.*, 100 seconds for the configuration in Figure 5(a) and 66.6 seconds for the one in Figure 5(b)). In both figures, the observed latencies are slightly below the predictions. This is because our parameter α assumes an infinite number of CPU cores. For realistic CPU counts, α becomes discrete and we observe a slightly better actual



(a) $|W_R| = |W_S| = 200$ seconds.

(b) $|W_R| = 100$ and $|W_S| = 200$ seconds.

Figure 5: Latency distribution of handshake join. Each data point represents 200,000 output tuples.

latency (handshake join degenerates to Kang's three-step procedure when using just a single CPU core [20]).

A high absolute latency and a high variance are both problematic characteristics of handshake join's latency. The former one is undesirable, because the very purpose of most stream processors is to react in real time. The latter one mixes up the *order* of tuples in the output stream. Such disorder can be corrected, but only at the price of *(a)* even more latency and *(b)* high memory overhead, because tuples must be buffered for re-ordering.

4. LOW-LATENCY HANDSHAKE JOIN

The goal of this work is to remove the latency bottleneck of the handshake join. In this section, we propose our *low-latency handshake join*. The algorithm is semantically equivalent to the handshake join and classical stream join operators with respect to their set of output tuples. It also preserves the architecture awareness of the handshake join but in addition avoids its high latency penalty.

4.1 Overview

As in handshake join, our low-latency alternative considers a chain of processing nodes (CPU cores) in which each node has some local memory store and neighboring nodes are connected by communication channels using FIFO queues. Tuples from input stream R flow through the chain from "left to right", whereas tuples from S flow in the opposite direction, i.e., from "right to left". The relevant state of the windows of R and S is distributed across FIFO queues and local stores. LLHJ proceeds as follows:

① When an input tuple v arrives at one of the input streams, low-latency handshake join decides on its *home node* h_v.

② Tuple v is then *expedited* through the chain of processing nodes, i.e., v does not queue up on each node but becomes forwarded almost immediately after local arrival.

③ When v reaches its home node h_v, v is added to a *node-local window* W_{h_v} and marked as *stored*.

④ As a stored tuple, v *continues* its expedition, until it reaches the end of the processing pipeline, where it is discarded.

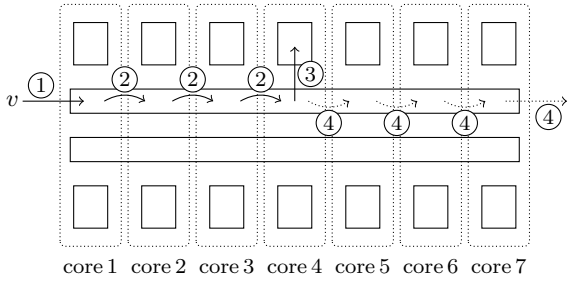


Figure 6: High-level idea of our low-latency handshake join. Input tuple v from R rushes through the pipeline, gets stored in the local window of node h_v = core 4, continues to the pipeline end, and gets discarded.

⑤ Similar to the original algorithm, low-latency handshake join uses *expiry messages* (termed "acknowledgement messages" in [20]) to delete expired tuples from the processing pipeline and from node-local windows.

Figure 6 illustrates the first four steps for an input tuple v from R (the "upper" stream). In Step ①, low-latency handshake join decides on core 4 as the home node of v. Tuple v travels through the pipeline, gets stored on core 4, and continues until core 7, where it gets discarded. The squares boxes at the top and bottom indicate the node-local windows on each core for streams R and S, respectively.

The high latency of the original handshake join algorithm arises due to strict queueing of tuples along the distributed join windows. Instead, in low-latency handshake join, each core immediately forwards ("expedites") incoming tuples such that each tuple will have been seen by *all* involved processing units shortly after its arrival. The moving tuple gets discarded when it has rushed through all nodes, but the copy on its assigned home node remains available for matching until explicitly removed by an expiry message (see Section 4.2.4 for a discussion of Step ⑤).

Other than in the original algorithm, in low-latency handshake join every tuple gets to rest only on a single processing node, its home node. This opens up the possibility to combine low-latency handshake join with local *indexing strate-*

gies, where temporary hash or B-tree indexes are built over the node-local data to accelerate matching.

4.2 Detailed Discussion

4.2.1 Communication Pattern

Low-latency handshake join aims to maintain the positive characteristics of the original algorithm. In particular, as can also be seen in Figure 6, low-latency handshake join inherits the architecture-efficient communication pattern from handshake join. Teubner and Mueller [20] demonstrated that this communication pattern matches, for example, the system topology of the AMD Magny Cours system. Every core communicates only with its immediate neighbors and through dedicated FIFO channels. Such channels are efficient to implement and—more importantly—scale well to large core counts.

4.2.2 Propagating Tuples

In realistic settings, low-latency handshake join processing is highly dynamic: tuples are constantly propagated between cores and are added or removed from node-local windows. This bears a risk of *race conditions*, which may lead to incorrect results when not treated properly. In particular, we want to make sure that when a tuple from r moves through the pipeline, it encounters *exactly once* all concurrent tuples from S—i.e., tuples from S that are either in the pipeline at the time r arrives or arrive themselves before r reaches the end of the pipeline—and vice versa.

Teubner and Mueller [20] pointed out the risk of *missed join pairs*, which may occur when the two tuples are simultaneously "in flight" between the same two neighboring cores (and thus do not encounter each other). The problem can be avoided by using an *acknowledgement mechanism* that ensures that a tuple virtually remains on its sending node until the arrival of the tuple has explicitly been confirmed by the receiving node via an acknowledgement message. It suffices to use such an acknowledgement mechanism on only one of the two input streams to guarantee that "missed" join pairs are detected on at least one side of a pair of neighbors; see [20] for details.

We adapt the above acknowledgement mechanism in low-latency handshake join. This mechanism ensures that each tuple is (virtually) present at exactly one node while travelling, and the encountering of two tuples is detected reliably.

4.2.3 Matching Tuples

With the acknowledgement mechanism in place, we can view the propagation of tuples along the processing pipeline as *data flows*: There is one data flow for the tuples from R (flowing from left to right in our figures) and one for the tuples from S (right to left).

In the original handshake join algorithm, each tuple in a data flow is treated in exactly the same way as any other tuple. In our latency-optimized variant, the procedure is different. In particular, each data flow now consists of what we call *fresh tuples* (② in Figure 6) and *stored tuples* (④). Fresh tuples did not yet pass their home node and are thus not stored in any node-local window. Stored tuples passed their home node so that a *copy* of the tuple is stored in the corresponding node-local window. Note that we distinguish stored tuples (in the processing pipeline) from stored copies (in the node-local window).

A fresh or stored tuple from one flow (say, R) may encounter *(i)* fresh tuples from the other flow (S), *(ii)* stored tuples from the other flow (S), and *(iii)* stored copies in a node-local window (of S). To ensure correct results, low-latency handshake join treats arriving tuples differently depending on their state (fresh or stored) and source (R or S). Possible cases of how tuples can meet (or not meet) are summarized in Table 1. We need to treat each case differently to avoid both missing results and duplicate results. In the discussion below, we write "fresh/stored" for the case that a fresh tuple R "meets" a matching stored tuple from S; the other cases are denoted correspondingly.

Fresh/Fresh. The situation that compares best to the original handshake join algorithm is when two fresh tuples encounter each other. This case is illustrated in Figure 7. The figure shows the state of a single tuple r from R and a single tuple s from S as they flow through the processing pipeline. Solid segments indicate a fresh tuple, dotted segments a stored tuple, and arrows mark the position of the respective home nodes. The two tuples meet at time T (at which both are fresh).

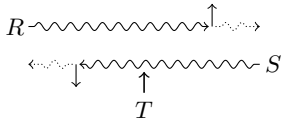


Figure 7: At time T, two fresh tuples meet.

In the fresh/fresh situation, neither tuple has yet found its home node and thus no stored copy of either tuple exists. Hence there will be no future situation where the two tuples might see each others' copies again. We thus perform an immediate predicate evaluation and emit a join result in case of a match (as in the original handshake join).

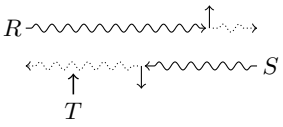


Figure 8: At time T, a fresh tuple r meets a stored tuple s.

Fresh/Stored. Figure 8 illustrates the situation when a fresh tuple $r \in R$ encounters a stored tuple $s \in S$. Since a copy of s has already been placed at its home node h_s, we know that r will see a copy of s somewhere down r's processing pipeline. We thus attempt no match. Instead, whenever a tuple from R arrives at a processing node, we scan the node-local window of S and emit all matching join pairs. This local join is performed in all cases, i.e., whether r is fresh or stored.

Stored/Stored. When two stored tuples meet (Figure 9), we attempt no match as above. The situation is more intricate, however: *both* tuples $r \in R$ and $s \in S$ will see the node-local copies of their counterpart as they travel along. Care must be taken to avoid incorrect *stored/stored double matches* and thus duplicates in the output stream.

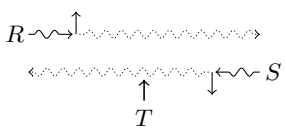


Figure 9: At time T, two stored tuples meet.

In order to perform matching *exactly once*, we need to break the symmetry between R and S and attempt matching for such combinations in only one node-local store (this

Met when travelling?	State r	State s	Evaluate $p(r,s)$ when travelling	Evaluate $p(r,s)$ at h_r	Evaluate $p(r,s)$ at h_s	Expedition flag of r when s arrives at h_r
yes	fresh	fresh	yes	no	no	-
	fresh	stored	no	no	yes	-
	stored	fresh	yes	no	no	Set
	stored	stored	no	no	yes	Set
no	fresh/stored	-	-	-	yes	-
	-	fresh/stored	-	yes	-	Cleared

Table 1: When to evaluate join predicate $p(r,s)$?

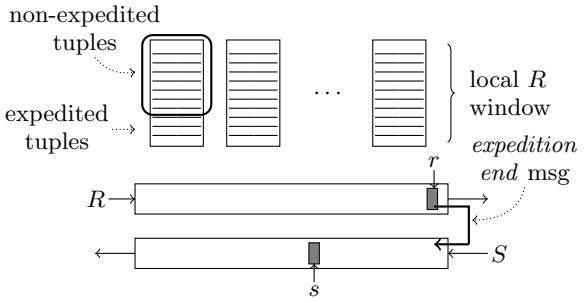


Figure 10: *Expedition end* messages help to maintain the expedition flag in each node-local R-store.

resembles the asymmetric acknowledgement mechanism of [20]). To this end, we extend the node-local windows of R to record an *expedition flag* for each tuple. The flag of tuple r (in local window W_{h_r}) indicates whether or not tuple r has finished its expedition (see below). To avoid double matches, arriving tuples from S are matched not against the entire node-local window from R but only against tuples that have finished their expedition (expedition flag cleared).

To populate the flag, we introduce a new message type, as illustrated in Figure 10. Whenever an r-tuple has reached the end of its pipeline, we insert an *expedition end* message into the flow of S; this message signals r's home node h_r that r has now reached the pipeline end (so h_r can clear r's expedition flag). Since all tuples are processed in order, the flag can be realized as a pointer that separates expedited and non-expedited tuples in h_r's node-local window.

The above mechanism takes advantage of the strict FIFO ordering in the system, *i.e.*, all messages from one node to its neighbor are sent through the same FIFO channel, regardless of the type of the message.

Double matching must only be avoided for the stored/stored situation of Figure 9 (by ignoring "expedited" tuples in node-local R windows). The expedition end message for r is generated at the pipeline end and thus separates the tuples from S that have encountered r from those that have not (because they arrived after r reached the end of the pipeline). In more detail, all tuples from S that are inserted into the flow of S before the expedition end message of r (e.g., tuple s in Figure 10) match with r as it travels (but not with its stored copy). All tuples inserted after the expedition end message match with the stored copy of r (but do not encounter r as it travels).

Stored/Fresh. Finally, a stored tuple $r \in R$ may encounter a fresh tuple $s \in S$. This situation is illustrated in Figure 11 on the right.

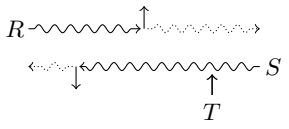


Figure 11: At time T, a stored tuple r meets a fresh tuple s.

Since s encounters r, we know that r is and will still be flagged as "in expedition" on r's home node h_r when s eventually arrives there (the expedition end message of r arrives at node h_r after s). Hence, when s reaches h_r, it will ignore r during the node-local window scan due to the above mechanism for avoiding stored/stored double matches: a *stored/fresh miss* would result.

We resolve this problem by handling fresh/stored and stored/fresh situations in an asymmetric way, too. While the former are simply ignored, we do attempt a match in the latter case, similar to the encountering of two fresh tuples.

Table 1 summarizes the discussion above for any pair $(r,s) \in R \times S$. Low-latency handshake join produces correct results because $p(r,s)$ is evaluated exactly once for each case (row in the table) and the different cases are mutually exclusive.

4.2.4 Tuple Expiration

We directly adapt the tuple expiration mechanism of handshake join [20], which can be flexibly used with arbitrary types of sliding windows. The processing pipeline of the algorithm is oblivious with respect to the specification of the sliding windows and, in particular, the algorithm does not know (or need to know) whether it is processing tuple-based, time-based or any other form of sliding windows.

We briefly summarize the approach here. Low-latency handshake join assumes that there is an external *driver* that is aware of the sliding window specification and determines when tuples enter or leave one of the sliding windows. The driver then submits tuple arrivals and tuple expiration messages to the low-latency handshake join instance. All arriving tuples enter the pipeline as fresh tuples on the left (R) or right (S); expiring tuples are submitted to the opposite ends, i.e., right for R and left for S. Intuitively, all tuples from R that are inserted into the left-to-right queue after the expiration message of some tuple $s \in S$ do not join with s, and vice versa. Similar to tuple arrivals, tuple expiration messages travel along the pipeline; each node simply removes the stored copy of the respective tuple, if present. See [20] for further details.

```
1  Procedure: low_latency_handshake_join ()
2  while true do
3      if message waiting in leftRecvQueue then
4          process_left () ;
5      if message waiting in rightRecvQueue then
6          process_right () ;
```

Figure 12: LLHJ with asynchronous message passing (runs on each core).

```
1  Procedure: process_left ()
   /* code is for core k in the pipeline        */
2  msg ← message from leftRecvQueue ;
3  if msg is an arrival message then
4      r ← extract new tuple from msg ;
5      if k is leftmost core in pipeline then
6          decide on home node h_r and tag r accordingly ;
7      insert r into rightSendQueue ;
8      scan W_Sk and IW_Sk to find tuples that match r ;
9      if k = h_r then
10         insert r into W_Rk (marked as expedited) ;
11     if k is rightmost in pipeline then
12         place expedition end for r in rightRecvQueue ;
13 else if msg is an acknowledgement message then
14     remove acknowledged tuple s from IW_Sk ;
15 else
       /* msg is an expiry message                 */
16     s ← extract expired tuple from msg ;
17     if s found in node-local store W_Sk then
18         remove s from W_Sk ;
19     else
20         insert msg into rightSendQueue ;
```

Figure 13: Processing a message from left input queue.

```
1  Procedure: process_right ()
   /* code is for core k in the pipeline        */
2  msg ← message from rightRecvQueue ;
3  if msg is an arrival then
4      s ← extract tuple from msg ;
5      if k is rightmost core in pipeline then
6          decide on home node h_s and tag s accordingly ;
7      insert s in leftSendQueue ;
       /* avoid stored/stored double matches       */
8      scan non-expedited entries of W_Rk to find tuples
       that match r ;
9      if s is fresh (k > h_s) then
           /* avoid stored/fresh misses            */
10         insert s into IW_Sk ;
11     if k = h_s then
12         insert s into W_Sk ;
13     insert acknowledgement for s into rightSendQueue ;
14 else if msg is an expedition end then
15     r ← extract expedition end r from msg ;
16     if r found in node-local store W_Rk then
17         clear the expedition flag for r from W_Rk ;
18     else
19         insert msg into leftSendQueue ;
20 else
       /* msg is an expiry                         */
21     r ← extract expired tuple from msg ;
22     if r found in node-local store W_Rk then
23         remove r from W_Rk ;
24     else
25         insert msg into leftSendQueue ;
```

Figure 14: Processing a message from right input queue).

4.3 The Complete Algorithm

Figures 12–14 describe the low-latency handshake join algorithm more formally.

The main loop of low-latency handshake join is given by the function `low_latency_handshake_join ()` in Figure 12, which is run on each of the cores in the pipeline. Each core communicates with the core to its left via an input queue (leftRecvQueue) and an output queue (leftSendQueue); similar queues exist for communication with the core to the right. The main event loop simply dispatches incoming messages from either the left or right input channel to its respective event handler.

Each core k maintains a node-local window W_{R_k} of tuples from R, a node-local window W_{S_k} of tuples from S, and a temporary buffer IW_{S_k} of tuples from S that have been forwarded but not yet acknowledged.

Left-to-right messages. The pseudo-code for handling messages that are passed from left to right is shown in Figure 13. There are three types of left-to-right messages: arrival messages of tuples from R, acknowledgement messages of forwarded tuples from S (to avoid missed join pairs), and expiration messages of tuples from S.

We first discuss the case when a new tuple $r \in R$ arrives at node k. If k is the leftmost node in the pipeline, and thus the first node that processes r, it tags r with its home node h_r (lines 5 and 6). In our default implementation, we select home nodes in a round-robin fashion to ensure even load balancing. To minimize latency, each tuple is then immediately forwarded to the next neighbor to the right (line 7), before both the node-local window W_{S_k} and the set IW_{S_k} of forwarded but not-yet-acknowledged tuples are scanned for possible matches (line 8). If k is r's home node ($h_r = k$), then r is stored in the node-local window W_{R_k} (lines 9 and 10). If k is the right-most node of the pipeline, an *expedition end* message for r is generated and sent in the opposite direction (line 12, cf. Section 4.2.3).

The remaining cases are handled as follows: If node k receives an acknowledgement messages of a tuple $s \in S$ it had forwarded (in the opposite direction), it removes s from the buffer IW_{S_k} of not-yet-acknowledged tuples (lines 13 and 14). If node k receives and expiration message of some tuple $s \in S$, it removes the tuple from its node-local window W_{S_k} if present or passes the expiration message along to its right neighbor otherwise.

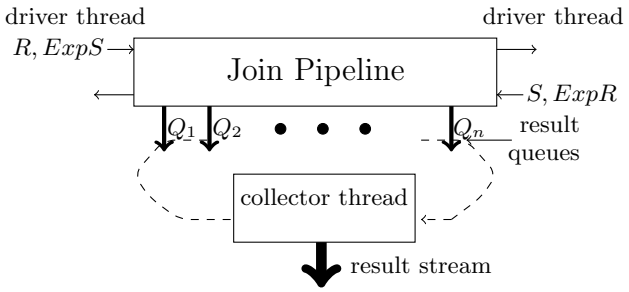


Figure 15: Result assembly in handshake join.

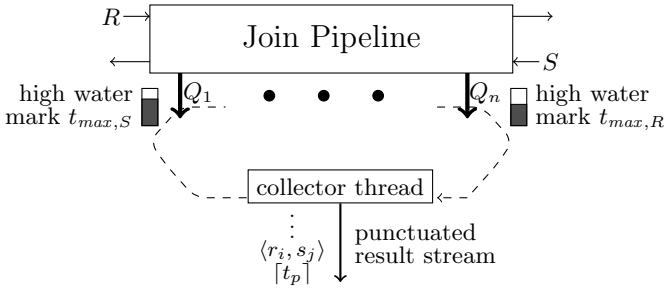


Figure 16: Low-latency handshake join infers time stamp information for punctuations $\lceil t_p \rceil$ is from *high water marks*, which are maintained for both input streams at the respective pipeline end.

Right-to-left messages. The handling of tuples in the opposite direction is not completely symmetric because of the acknowledgement mechanism (Section 4.2.2) and the expedition end mechanism (Section 4.2.3). Messages that travel from right to left are: arrival messages of tuples from S, expiration messages of tuples from R, and expedition end messages of tuples from R (to avoid stored-stored double matches).

When a tuple $s \in S$ arrives at node k, it is tagged (only at right-most node) and forwarded as before. In contrast to the handling of left-to-right tuple arrivals, tuple s is matched with only non-expedited tuples in the local window W_{R_k} to avoid stored-stored double matches. Moreover, s is stored in the temporary buffer IW_{S_k} (only needed if s is still fresh, cf. Table 1) and an acknowledgement message for s is sent to the node to the right.

Lines 14–19 handle expedition end messages of tuple $r \in R$. If r is stored in the node-local window W_{R_k}, its expedition flag is cleared; otherwise, the expedition end message is forwarded to the next node to the left.

Finally, expiration messages from tuples from R are handled as in the left-to-right case.

5. RESULT STREAM

Handshake join finds join matches in a fully distributed manner. The paper of Teubner *et al.* [20] does not, however, elaborate on how result tuples are assembled into one single output stream.

Figure 15 illustrates how result collection is implemented in the publicly available source code of Teubner *et al.*. Every worker thread of the handshake join processing pipeline connects to a dedicated result queue (Q_i in Figure 15). A separate thread (termed *collector*) periodically iterates over all result queues, vacuums them, and produces result tuples to a single output stream.

Strictly speaking, collecting results in this manner breaks with the strict neighbor-to-neighbor communication model of handshake join. The collector thread has to communicate with all nodes in the system and runs the risk to become a bandwidth and merging bottleneck. Oge *et al.* [18] investigated this problem in an FPGA-based implementation and proposes *adaptive merging* to address it. In practical settings, however, stream processing engines are typically used to sieve high-volume data streams and return only low-volume results. In the experimental settings that we considered, we could not saturate the result collection of neither the code of [20] nor of our own code.

6. ORDER OF RESULT TUPLES

As a consequence of their sequential nature, data streams come with a meaningful notion of *order*. And for most data stream operators—including such that use windowing—useful definitions have been suggested that exploit or produce streams with a well-specified order.

On the flip side, precise order semantics is sometimes costly to guarantee while not even necessary for realistic workloads [13]. *Punctuations* [14, 15] were thus suggested as a mechanism to allow for local disorder within the stream. Explicit markers ("punctuations") indicate stream progress as a logical property within the stream. They can be interpreted as a *guarantee* about stream items that precede or follow the punctuation in the physical stream.

Applied to the permission of local disorder, a punctuation $\lceil t_p \rceil$ might, for instance, state the guarantee that there will be no more tuples in the stream with a time stamp earlier than t_p (all tuples v with time stamp $t_v < t_p$ have already occurred in the stream before the punctuation).

6.1 Punctuations in Low-Latency Handshake Join

Low-latency handshake join is—like its original algorithm handshake join—a good example of an algorithm that can benefit from relaxed order requirements and, through the use of parallelism, turn the relaxation into improved performance. But in spite of the algorithm's high potential for parallelism, low-latency handshake join offers a very natural way of producing tight punctuations with only little overhead.

6.1.1 High Water Marks

The mechanism to infer punctuation information in low-latency handshake join is illustrated in Figure 16. For both input streams, we maintain a *high water mark* $t_{max,R}/t_{max,S}$. High water marks record the highest time stamp for any tuple r/s that has reached the respective pipeline end (since time stamps are monotonic, this corresponds to the *last* time stamp seen).

The matching process, *i.e.*, the generation of result tuples along the processing pipeline, is *driven* by expedited tuples that traverse the pipeline ('scan' calls in line 8 of Figures 13 and 14). And since all input data is processed in strict sequential/time stamp order, high water mark values $t_{max,R}/t_{max,S}$ indicate that no tuples r/s with time stamps

$t_r < t_{max,R}$ and $t_s < t_{max,S}$ can drive any further matches.[1]

6.1.2 Result Time Stamps

The time stamp $t_{\langle r,s\rangle}$ of a result tuple $\langle r, s\rangle$ is defined to be the *later* of the two time stamps t_r and t_s, *i.e.*, $t_{\langle r,s\rangle} := \max(t_r, t_s)$ or $t_{\langle r,s\rangle} \geq t_r \wedge t_{\langle r,s\rangle} \geq t_s$.

For result tuples $\langle r, s\rangle$ whose generation is driven by an expedited tuple r, the result time stamp $t_{\langle r,s\rangle}$ must be greater or equal to the time stamp t_r of r, which we know is greater or equal to the high water mark $t_{max,R}$. Likewise, when the generation is driven by an expedited s, we know that $t_{\langle r,s\rangle} \geq t_s \geq t_{max,S}$.

Independent of what kind of tuple drove the output generation, the *minimum high water mark* $\min(t_{max,R}, t_{max,S})$ is a safe bet. Any newly generated tuple $\langle r, s\rangle$ will have a time stamp $t_{\langle r,s\rangle} \geq \min(t_{max,R}, t_{max,S})$.

6.1.3 Collector Thread and Punctuations

We can use this reasoning over time stamps to implement a result collector that generates correct and tight punctuations. To this end, we implement the collector as follows:

1. Read high water marks $t_{max,R}/t_{max,S}$ and determine minimum high water mark $t_p := \min(t_{max,R}, t_{max,S})$.
2. Read out ("vacuum") all result queues Q_i and forward all previously generated result tuples to overall join output stream.
3. Place a punctuation $\lceil t_p \rceil$ in the overall join output stream.
4. Repeat.

6.2 Ordered Result Streams

Generating punctuations this way causes virtually no processing overhead. The effect, however, can be significant.

Original Handshake Join Algorithm. If strict time stamp ordering is required in the overall output stream, either algorithm must temporarily buffer output tuples to fix up the generated disorder. In case of the original handshake join algorithm, we already saw that individual tuple latencies are in the order of the total window size. Thus, to produce correctly sorted output, handshake join will have to delay and buffer generated result tuples for time scales up to the window length ($|W_R|$ or $|W_S|$).

Besides adding another delay to the already-high latency of handshake join, buffering such tuple amounts can cause a significant resource overhead. To illustrate, the benchmark configuration that we consider in Section 7 assumes tuple rates in the order of 3000 tuples/sec, window sizes of 15 minutes, and join hit rates of $1 : 250,000$. This corresponds to an output data rate of more than 60,000 tuples/sec. For a 7.5-minute delay, the system will thus need a buffer of almost 30 million output tuples.

This *space overhead* is complemented by a very high *CPU overhead*, because periodically the full buffer must be (partially) sorted and old tuples must be written to the join output.

[1]It is quite likely, however, that some cores are still busy generating output for tuples r'/s' with $t_{r'} = t_{max,R}$ or $t_{s'} = t_{max,S}$. The fast-forwarding mechanism of low-latency handshake join, in fact, makes this the common case.

Low-Latency Handshake Join. The situation looks radically different when punctuations in low-latency handshake join are used to optimize the sorting of output data. Buffers must now only be maintained until the next punctuation. In practice, this leads to a saving in memory space of several orders of magnitude. In addition, as indicated above, smaller buffers save precious CPU cycles and reduce overall algorithm latency.

We will experimentally verify the effects of punctuations in Section 7.

7. EXPERIMENTS

We ran the experiments on a 2.2 GHz AMD Opteron 6174 "Magny Cours" machine [5], the same machine that is used in [20]. The machine contains 48 real x86-64 cores, distributed over 8 NUMA regions which are connected through a set of point-to-point HyperTransport links. The data flow of the algorithm can be laid out over the available CPU cores such that only short-distance communication is needed and no congestion occurs on any link or at any NUMA site. Our prototype implementation uses *libnuma* library and an asynchronous FIFO implementation similar to that of [4]. The system was running Ubuntu Linux 12.04 LTS. We used GCC 4.6.3 version to compile the code.

7.1 Experimental Setup

For ease of comparison, we used the same benchmark setup that was used to evaluate CellJoin [9] and handshake join [20]. Two streams $R = \langle \mathsf{x} : \mathsf{int}, \mathsf{y} : \mathsf{float}, \mathsf{z} : \mathsf{char}[20] \rangle$ and $S = \langle \mathsf{a} : \mathsf{int}, \mathsf{b} : \mathsf{float}, \mathsf{c} : \mathsf{double}, \mathsf{d} : \mathsf{bool} \rangle$ are joined via the two-dimensional band join

$$\texttt{WHERE } r.\mathsf{x} \texttt{ BETWEEN } s.\mathsf{a} - 10 \texttt{ AND } s.\mathsf{a} + 10$$
$$\texttt{AND } r.\mathsf{y} \texttt{ BETWEEN } s.\mathsf{b} - 10. \texttt{ AND } s.\mathsf{b} + 10. \ .$$

The join attributes contain uniformly distributed random data from the interval 1–10,000, which results in a join hit rate of $1 : 250,000$. As in [9], we ran all experiments with symmetric data rates, that is $|R| = |S|$.[2] All the workers including the driver and collector threads are implemented as Linux threads.

7.2 Throughput and Scalability

Firstly, we want to make sure that low-latency handshake join does not lose the throughput and scalability characteristics of the handshake join. We ran the experiment where we varied the core count from 4 to 40 and for each configuration we determined the maximum throughput that the system could sustain without dropping any data for both of the algorithms. It is shown in the Figure 17.

The experiment shows that the changes made in low-latency handshake join have negligible impact on scalability and absolute throughput. In fact, we found that for homogeneous hardware environments, the assignment of home nodes typically leads to a better load distribution than the self-balancing code of [20]. In Figure 17, this leads to a slight throughput improvement, especially for large core counts.

7.3 Latency

To compare the latency of the low-latency handshake join and handshake join in the same scale, we plot the average

[2]Workload and output data rates grow with $|R| \times |S|$ or *quadratically* with the reported stream rate.

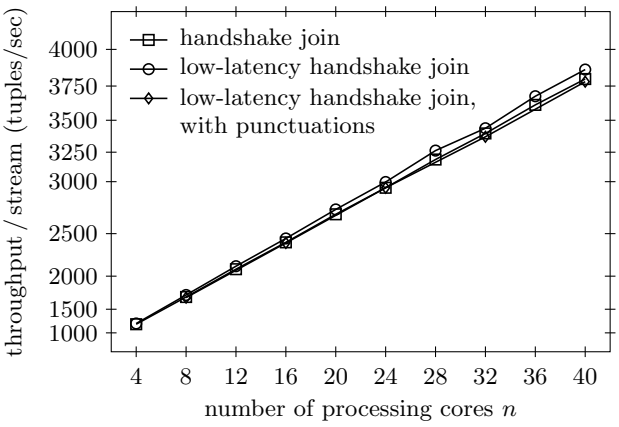


Figure 17: Throughput comparison: original handshake join implementation (⊟); low-latency handshake join (⊖); and low-latency handshake join with generated punctuations (⊸). The improved latency characteristics of low-latency handshake join have no noticeable effect on throughput.

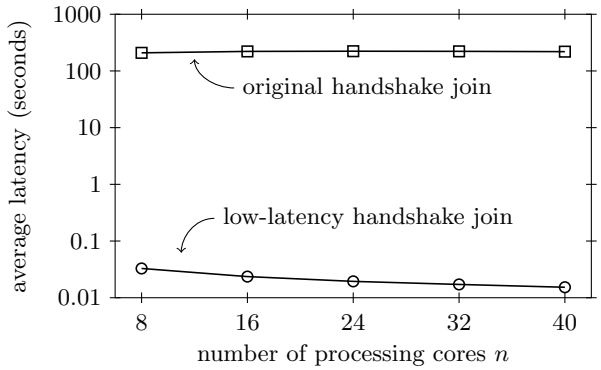


Figure 18: Average latency, original handshake join vs. low-latency handshake join; 15 min window.

latency for both algorithms in Figure 18, computed over a 15-minute window with increasing core count. Low-latency handshake join achieves almost four orders of magnitude better average latency in comparison to handshake join. Increasing number of cores do not have much impact on the average latency of handshake join as the latency mainly depends on the window sizes.

As the core count increases, low-latency handshake join exhibits a noticeable drop in latency. The source of this is a *batching mechanism* in our code (and in that of [20]). The driver thread batches 64 tuples before it pushes them together into the join pipeline. For handshake join the impact of batching is completely hidden because of the high values of the latencies. In contrast, for low-latency handshake join this becomes visible because of the small values of latencies. For a throughput of our 8-core configuration, a new batch is submitted approximately every 46 milliseconds; experimentally, we observed an average latency of 32 ms per tuple. With 40 cores, due to increased throughput, the batch distance drops to about 20 milliseconds, or an average latency of 15 ms.

A more detailed latency analysis is illustrated in Figure 19. The two graphs repeat the earlier experiment of Figure 5, but for low-latency handshake join.

Tuple expedition brought the average latency down below 10 ms, with maximum latencies of 30 ms and below—an improvement of more than three orders of magnitude (note the different axis scales in Figures 19 and 5). Further, latencies have become much less sensitive to the configuration of the join window; both evaluated configurations resulted in comparable latency values. Even in these two cases the main source of latencies is the batching delay (which is about 9 ms on an average).

7.3.1 With Reduced Batching Effect

Latency can further be reduced by minimizing the batching size of the implementation. Vectorized processing (an important source of performance of handshake join [20]) requires a batch size of at least four tuples. Therefore, we reduced the batch size of low-latency handshake join to this value and re-ran the latency analysis experiment (see Figure 20).

In this configuration, a batch will be issued every 1.2 ms. This is consistent with the average result latency, which is about 1 ms in Figure 20; worst-case latency is now around 3–4 ms (spikes in Figure 20 are a result of scheduling effects that are outside the control of our code).

Aside from batching (which still remains the main source of latency), latency is caused by two effects that will add up in practice: *(a)* tuples must be fast-forwarded through the pipeline, with an overall delay that depends on the pipeline length; *(b)* each tuple must, in parallel on all nodes, be compared to node-local windows (this effect is independent of the pipeline length). Both effects are too small to be visible in Figure 20. Core-to-core messaging is extremely fast in modern architectures. Baumann et al. [4], for instance, report a single-hop latency below 1 μs. Even for very long processing pipelines, this latency will not be noticeable.

7.4 Effect of Punctuation

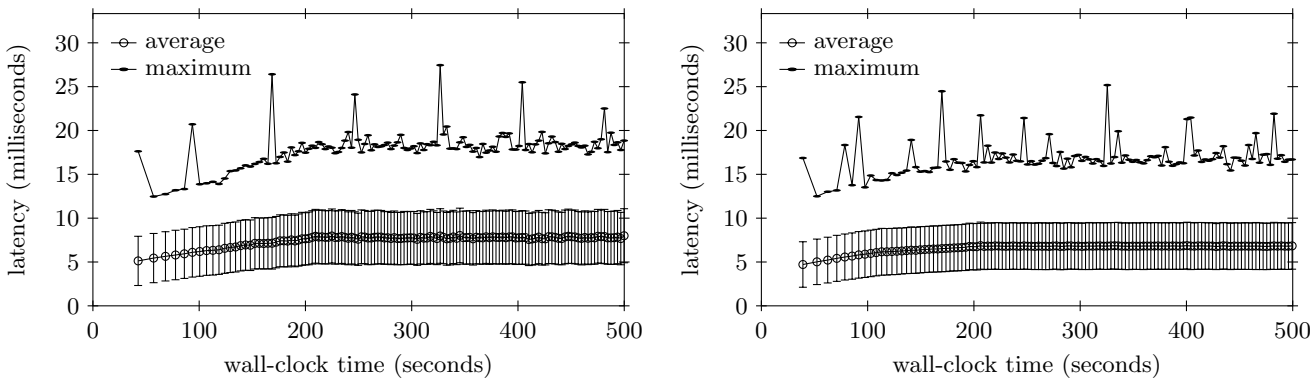
Punctuations incur additional overhead on the join workers. Especially the first and the last core i.e. C_1 and C_n in the join pipeline need to send additional messages in the form of markers to the collector thread. Figure 17 also includes throughput numbers when we turned on the generation of punctuations in low-latency handshake join (shown as ⊸).

With punctuations turned on, throughput stays only marginally below the throughput we achieved with the plain low-latency handshake join code. This confirms our expectations in Section 6.1, where we designed punctuation generation to be a lightweight operation over the low-latency handshake join.

7.5 Generation of Sorted Output

Although the latency variance for output tuples for low-latency handshake join is small, generation of a completely sorted output for the down stream operator from a non-punctuated result stream is a difficult task, as we detailed in Section 6.2.

Even with punctuations, a downstream sorting operator needs to *buffer* tuples until it receives the punctuation. We implemented such a sorting functionality and tracked the



(a) $|W_R| = |W_S| = 200$ seconds.

(b) $|W_R| = 100$ and $|W_S| = 200$ seconds.

Figure 19: Latency distribution of low-latency handshake join. Each data point represents 200,000 output tuples.

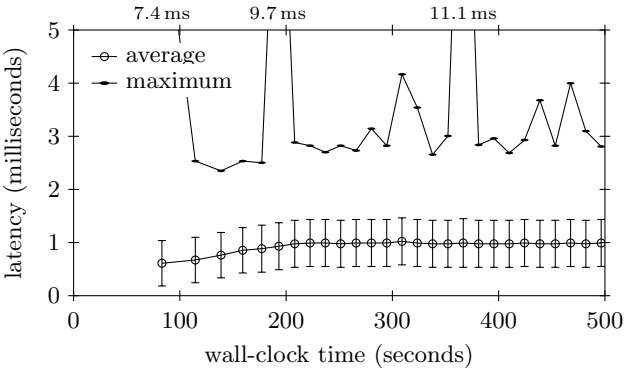


Figure 20: Latency distribution of low-latency handshake join (with a batch size of four); $|W_R| = |W_S| =$ 200 seconds.

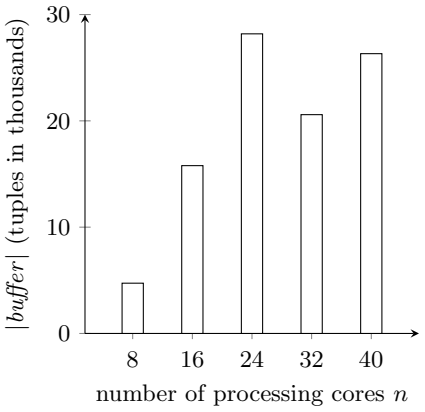


Figure 21: Maximum size of the buffer i.e., $|buffer|$ in tuples with increasing cores.

algorithm	throughput (tuples/sec)
handshake join [20]	5125
low-latency handshake join	5117
low-latency handshake join with index	225234

Table 2: Throughput comparison of the 40 core configuration of handshake join, low-latency handshake join with and without indices; 15 min window.

maximum size of its internal buffer, i.e., the maximum number of tuples that buffer had at any point in time. This is shown in the Figure 21. The figure confirms that sorted output can be generated with only a small overhead (to compare, without low-latency handshake join's punctuation mechanism, the operator would have to buffer and re-sort many millions of tuples).

7.6 Looking Forward: Index Acceleration

A virtue of low-latency handshake join is that it can play well together with node-local index structures to accelerate searching. In ongoing work, we are studying the performance advantages that such access structures may have to speed-up processing whenever join predicates permit the use of an index.

Table 2 hints at the potential of additional index structures. To this end, we changed the join predicate of our benchmark to allow for hash-based processing and implemented a local hash table within each processing node. Table 2 lists the throughput numbers that we achieved for handshake join, low-latency handshake join, and our hash-accelerated prototype.

8. RELATED WORK

With the omnipresence of many-core and heterogeneous machines, it has become more important than ever to design algorithms tuned to the underlying architecture to achieve certain guarantees. A recent work on main memory join [3] has shown that the relational database join algorithm that is carefully tuned to the underlying hardware achieves the

best throughput. Nevertheless, the handshake join [20] algorithm is also carefully tuned to the underlying architecture to achieve the best throughput but largely ignores the optimisation of the latency characteristics. Latency is an important parameter of goodness for streaming application [2, 1]. Especially in trading application such features are of utmost importance [19, 22], as the algorithm should detect and report about anomalies as early as possible. There are also several database join algorithms that try to reduce the latency and produce results as early as possible mainly to support the pipeline query model as well as to support streaming applications [17, 21, 8]. In this work, we achieved both very high throughput and very small latency by carefully tuning the algorithm to the underlying hardware so that the latency and throughput requirement of streaming application can be satisfied.

Punctuations were introduced by Tucker et al. [14] to break the infinite semantics of streaming data to allow certain database operators to avoid infinite memory consumption and indefinite blocking. There have been several works that use punctuations in join to mark the end of appearance of some tuple in the input e.g., PJoin [6]. A subsequent work is PWJoin [7] that improves the performance of PJoin utilizing both the window semantics and punctuation semantics. Low-latency handshake join also uses punctuations in an innovative way to guarantee certain feature of the future result tuples with respect to timestamp.

9. CONCLUSION

We built on top of the handshake join algorithm and present the algorithm *low-latency handshake join* that circumvents the shortcomings of handshake join. The algorithm allows us to get the best of both worlds. It exhibits the scalability and throughput characteristics like handshake join and as well as reduces latency and enables us to produce a punctuated result stream. This, in turn, allows to produce result tuples in complete order. In addition, we have successfully characterized the latency behavior of handshake join algorithm.

As part of our future work, we plan to investigate the performance of both of these algorithms with different kinds of indices. As well as we will explore the impact of different kinds of partitioning on the input data.

Acknowledgements. Pratanu Roy was supported by the European Commission (FP7, Marie Curie Actions; project *Geocrowd*). Jens Teubner was supported by the Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis."

10. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[2] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *Proc. of VLDB*, volume 30, pages 480–491, 2004.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Proc. of ICDE*, pages 362–373, 2013.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proc. of ACM SOSP*, Big Sky, MT, USA, 2009.

[5] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, 30(2):16–29, 2010.

[6] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman. Joining punctuated streams. In *Proc. of EDBT*, pages 587–604. 2004.

[7] L. Ding and E. A. Rundensteiner. Evaluating window joins over punctuated streams. In *Proc. of CIKM*, pages 98–107. ACM, 2004.

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

[9] B. Gedik, P. S. Yu, and R. Bordawekar. CellJoin: A parallel stream join operator for the Cell processor. *The VLDB Journal*, 18(2):501–519, 2009.

[10] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proc. of ICDE*, Bangalore, India, 2003.

[11] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. The HELLS-join: a heterogeneous stream join for extremely large windows. In *Proc. of DaMoN*, New York, NY, USA, June 2013.

[12] T. Karnagel, B. Schlegel, D. Habich, and W. Lehner. Stream join processing on heterogeneous processors. In *BTW Workshops*, pages 17–26, Magdeburg, Germany, Mar. 2013.

[13] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *Proc. of ACM SIGMOD*, pages 1081–1092, Indianapolis, IN, USA, June 2010.

[14] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proc. of ACM SIGMOD*, Baltimore, MD, USA, 2005.

[15] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: a new architecture for high performance stream systems. *Proc. VLDB Endowment (PVLDB)*, 1(1), 2008.

[16] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. Numa-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[17] M. Mokbel, M. Lu, and W. Aref. Hash-merge join: a non-blocking join algorithm for producing fast and early join results. In *Proc. of ICDE*, pages 251–262, 2004.

[18] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga. A fast handshake join implementation on FPGA with adaptive merging network. In *Proc. of SSDBM*, Baltimore, MD, USA, July 2013.

[19] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proc. VLDB Endowment (PVLDB)*, 3(1-2):1525–1528, 2010.

[20] J. Teubner and R. Mueller. How soccer players would do stream joins. In *Proc. of ACM SIGMOD*, Athens, Greece, 2011.

[21] A. N. Wilschut and P. M. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.

[22] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with FPGAs. *Proc. VLDB Endowment (PVLDB)*, 3(1), 2010.