Streaming HyperCube: A Massively Parallel Stream Join Algorithm

Yuan Qiu
HKUST
yqiuac@cse.ust.hk

Serafeim Papadias
HKUST
spapadias@cse.ust.hk

Ke Yi
HKUST
yike@cse.ust.hk

ABSTRACT

Streaming join is an essential operation in many real-time applications. A lot of research has been devoted to the study of distributed streaming join algorithms. However, existing solutions rely on heuristics and cannot handle data skew optimally. On non-streaming, static data, the HyperCube algorithm ensures a balanced load across all processors in an optimal way. In this paper, we extend this algorithm to the streaming setting, which can adapt the HyperCube configuration depending on the current data distribution. Some preliminary experimental results are provided to demonstrate the efficiency our algorithm.

1 INTRODUCTION

The prevalence of applications in financial trading, sensor networks, traffic analysis and web data management has brought attention to query processing over data streams. Various Distributed Stream Processing Systems (DSPSs) have emerged such as Flink [1], Spark Streaming [2] and Apache Storm [3]. As one of the most critical operations in database management systems, join has been extensively studied in the literature both in static models [5–8, 14, 16] and streaming models [11, 12, 17, 18].

Low latency and high throughput are two essentials for an efficient streaming join algorithm. In a massively distributed system, a key to achieving these goals is to ensure a good load balance among the workers. Over static data, the HyperCube algorithm [6] and its extensions [8] achieve an optimal load balance. However, this algorithm crucially depends on the data statistics, in particular, the set of heavy hitters and their frequencies. In a dynamic setting like streaming data, these statistics are changing all the time, which renders the HyperCube algorithm inapplicable.

In this paper, we propose Streaming HyperCube (SHC), an adaption of the HyperCube algorithm to the streaming setting. A key challenge in designing SHC is how to adaptively change the HyperCube configuration as the data statistics change over time. On the one hand, we want to keep the configuration as close as possible to the optimal setting the HyperCube uses for static data. On the other hand, we also want to avoid too frequent configuration changes, since each configuration change requires migration of states, which introduces communication overhead and stalls streaming processing. The SHC algorithm supports both full-stream joins as well as joins over a sliding window.

Below, we first review the related work in Section 2, in particular the static HyperCube algorithm. In Section 3 we describe the Streaming HyperCube algorithm. We have implemented the algorithm in Flink. In Section 4 we describe its implementation and show some preliminary experimental results comparing with some state-of-the-art stream join algorithms.

2 RELATED WORK

2.1 Parallel Hash Join

The Parallel Hash Join (PHJ) algorithm is the default join algorithm used in many state-of-the-art DSPSs [2, 3, 10, 19], including Flink. Consider a (natural) join $R(A, B) \bowtie S(B, C)$. Let p be the number of workers in the system, numbered $1, 2, \dots, p$. The algorithm utilizes a random hash function $h : B \to [p] = \{1, \dots, p\}$ to assign tuples $(a, b) \in R$ and $(b, c) \in S$ to worker $h(b)$. Each worker is responsible for producing the join results on all tuples assigned to it. More precisely, it maintains two hash tables built on attribute B, one for tuples from R, one for tuples from S. Upon the arrival of a tuple $(a, b) \in R$, it inserts it to the hash table on R, and probes the hash table on S with key b to find all tuples in S that can join with (a, b).

When there is no skew, this algorithm performs relatively well. However, when there are heavy hitters, namely, many tuples in R or S share the same value on attribute B, the load may no longer be balanced, since all tuples with the same b value must be sent to the same worker. In the worst case, if all tuples have the same value on b, the join degenerates into a Cartesian product, and one worker will be doing all the work while the other $p - 1$ workers are idle.

2.2 HyperCube

The worst case scenario of the aforementioned PHJ algorithm happens when there is only a single value $b \in B$. In this case, the join degenerates into a Cartesian product. This problem was solved in [6] by the following HyperCube algorithm.

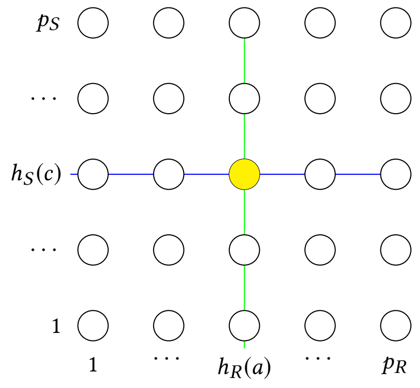

Figure 1: The HyperCube algorithm. Tuple (a, b) is sent to all workers in the column $h_R(a)$, and the tuple (b, c) is sent to all workers in the row $h_S(c)$. The worker at coordinate $(h_R(a), h_S(c))$ produces the join result (a, b, c).

The p workers are organized into a $p_R \times p_S$ grid (see Figure 1), where $p_R = |R|\sqrt{\frac{p}{|R|\cdot|S|}}$, $p_S = |S|\sqrt{\frac{p}{|R|\cdot|S|}}$. Two hash functions $h_R : A \to [p_R]$, $h_S : C \to [p_S]$ are used. Each tuple $(a, b) \in R$ is sent to all workers with coordinates $(h_R(a), *)$ and each $(b, c) \in S$

is sent to all workers $(*, h_S(c))$. As figure 1 shows, each output tuple is handled by exactly one worker. With high probability, each worker receives

$$\tilde{O}\left(\frac{|R|}{p_R}+\frac{|S|}{p_S}\right)=\tilde{O}\left(\sqrt{\frac{OUT}{p}}\right)$$

tuples, where the output size $OUT = |R|\cdot|S|$ for Cartesian product. Assuming $|R| = |S|$, this is better than the $O(|R|)$ load of the PHJ algorithm by an $O(\sqrt{p})$ factor.

Beame et al. [8] have generalized this algorithm to general joins. Let $R(b) = \sigma_{B=b}R$, and similarly define $S(b)$. The idea is to first partition the join attribute B into heavy hitters B^H and light hitters B^L. Let N denote the total number of tuples:

$$B^H = \left\{ b \in B \;\middle|\; |R(b)| > \frac{N}{p} \text{ or } |S(b)| > \frac{N}{p} \right\}$$

and B^L is the remaining attribute values. The light hitters B^L will be handled by the PHJ algorithm. Since there is no skew in the light hitters, the load is bounded by $\tilde{O}(N/p)$ for any worker. For each heavy value $b \in B^H$, $p_{(b)}$ workers are allocated to handle it using the *shares* algorithm, where

$$p_{(b)} = \frac{OUT(b)}{OUT^H}\cdot p = \frac{|R(b)|\cdot|S(b)|}{OUT^H}\cdot p$$

and $OUT^H = \sum_{b\in B^H} OUT(b)$ denotes the output size generated by all heavy hitters. The load of each worker for handling heavy hitters is upper bounded by $\tilde{O}(\sqrt{OUT/p})$. Therefore, the total load is $\tilde{O}\left(\frac{N}{p}+\sqrt{\frac{OUT}{p}}\right)$, which has been shown to be optimal.

However, this algorithm does not work in the streaming model, because it needs all the heavy hitter information to decide the configuration of each cube, namely how to set $p(b)$ for all the b's, and how to set the dimensions of each cube. Furthermore, the heavy hitter set may change throughout stream processing, which requires the HyperCube configuration to change correspondingly. These will be solved by our algorithm described in section 3.

2.3 BiStream Join

Lin et al. [17] proposed the Join-Biclique (JB) algorithm for computing joins over distributed streaming data. Their BiStream system divides the join into two stages, a routing stage for storage and a joining stage for producing outputs. Their work outperforms existing systems [11] in terms of memory consumption, scalability, latency and throughput. However, the algorithm relies on heuristics and does not work well on highly skewed data. We will compare SHC with a Flink implementation of the Join-Biclique algorithm in Section 4.

2.4 Cell Join

Cell Join presented by Gedik et al. [12] solves the windowed streaming join problem specifically on Cell processors. Following the three-step procedure described by Kang et al. [15], the algorithm parallelizes the scan task of streams over available workers to achieve high performance. However, since re-partition is needed whenever a new tuple arrives, this algorithm suffers from scalability issues. Although it supports band joins and can be extended to multi-way joins, its framework is solely designed for the IBM's cell processor.

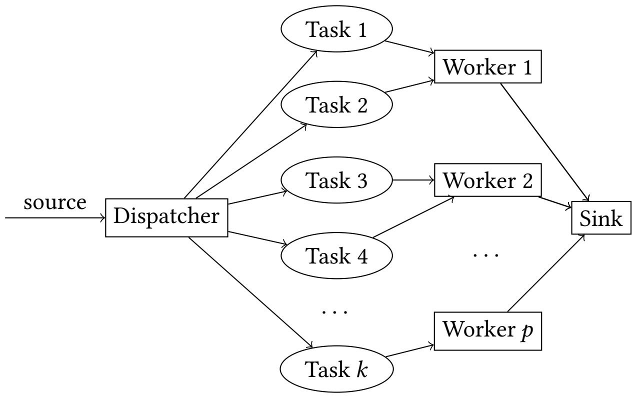

Figure 2: System Architecture.

2.5 Handshake Join

Roy et al. [18] presented a highly parallelizable handshake join for window streams. Hardware acceleration was implemented to achieve high data throughput. The algorithm uses similar window semantics to the ones in [15] by dividing the window into subwindows. The two streams flow in opposite directions so that each pair of subwindows has a handshake. This algorithm adopts the batch-processing model and may introduce disorder in the output tuple sequence, meaning it is not suitable for applications requiring instant outputs.

3 STREAMING HYPERCUBE

3.1 System Architecture

Figure 2 describes the architecture of our system. There are p workers along with a dispatcher and a sink. The dispatcher preprocesses input tuples after receiving them. The tuples are partitioned into *tasks*, which are then dynamically allocated to physical workers at runtime. The workers compute join results on-the-fly and pass them to the sink. Our system adopts the event-triggered model of Flink, where the output is updated immediately after the arrival of each tuple to ensure low latency.

3.2 Algorithm

The dispatcher runs an approximate heavy hitters tracking algorithm [9] to keep track of the heavy hitters and their approximate frequencies. Specifically, it classifies a join value $b \in B$ to be heavy hitter if $N(b) = |R(b)| + |S(b)| > \frac{N}{p}$ and light hitter if $N(b) < (\frac{1}{p} - \epsilon)N$, N is the current stream length (for full stream joins) or window length (for sliding window joins), and ϵ is some small constant. A value with frequency in between may be either heavy or light. As long as $\epsilon \le \frac{1}{2p}$, there are at most $2p$ heavy hitters. Note that the heavy hitter tracking algorithm also maintains approximated frequencies $|R(b)|$ and $|S(b)|$ for each heavy hitter b, each with at most ϵN additive error. Therefore the corresponding output size $OUT(b) = |R(b)|\cdot|S(b)|$ can be estimated with at most $2\epsilon N$ additive error. Summing over all the heavy values, we obtain an estimate of OUT^H, the output size of all heavy hitters, with no more than $2p\epsilon N$ additive error.

Similar to the HyperCube algorithm, the dispatcher treats heavy and light hitters differently. The light hitters are handled by the PHJ algorithm using p tasks and an associated hash function

$h^L : B \to [p]$.[1] Each heavy value b occupies $p_{(b)}$ tasks organized into a $p_{R(b)} \times p_{S(b)}$ grid. Two hash functions $h_{R(b)} : A \to [p_{R(b)}]$ and $h_{S(b)} : C \to [p_{S(b)}]$ are involved.

Suppose a tuple $(a, b) \in R$ arrives at time t. The dispatcher first reflects the update to the heavy-hitter tracking algorithm, so that the heavy hitter set and its statics are still valid approximations. If b is a light hitter, it is sent to task $h^L(b)$ for light hitters and the assigned worker produces the output. If b is a heavy hitter, it is sent to all tasks labeled $(h_{R(b)}(a), *)$ in the $p_{R(b)} \times p_{S(b)}$ grid allocated to b.

A key difference between SHC and the static HyperCube algorithm is that the number of tasks allocated to each heavy hitter, as well as the grid dimensions, may change over time. This is done by a *state migration* between tasks. To see why this is necessary, consider a value b, which starts light but continuously receives tuples over time. If only a single task is allocated to handle it, that worker will have a huge load. For efficient state migration, the dispatcher stores the current number of tasks $p_{R(b)} \times p_{S(b)}$ allocated for each heavy hitter b. According to the estimated statistics, it can also calculate a *desired* number of tasks for b, namely

$$p^d_{R(b)} = \frac{|R(b)|}{\sqrt{OUT^H}}\sqrt{p}, \text{ and } p^d_{S(b)} = \frac{|S(b)|}{\sqrt{OUT^H}}\sqrt{p}$$

When the estimates of OUT^H, $|R(b)|$, and $|S(b)|$ change over time, the dispatcher will compare the currently assigned grid dimension with the desired one. If $p_{R(b)} > 2p^d_{R(b)}$, we reduce $p_{R(b)}$ to half. Specifically, each task

$$(x, y), x = \frac{1}{2}p_{R(b)} + 1, \ldots, p_{R(b)}, y = 1 \ldots, p_{S(b)}$$

sends all its tuples in $R(b)$ to task $(x - \frac{1}{2}p_{R(b)}, y)$ and then gets released. If $p_{R(b)} < \frac{1}{2}p^d_{R(b)}$, double $p_{R(b)}$. Specifically, create $p_{(b)}$ new tasks and extend the grid to $2p_{R(b)} \times p_{S(b)}$. Each task

$$(x, y), x = 1, \ldots, p_{R(b)}, y = 1 \ldots, p_{S(b)}$$

sends all its tuples in $S(b)$ to $(x + p_{R(b)}, y)$. For each tuple $(a', b) \in R(b)$ in the task, apply a new hash function $h' : A \to \{0, 1\}$ to it, where 0 means that the tuple stays, and 1 means the tuple gets sent to task $(x + p_{R(b)}, y)$. The new hash function associated with $p_{R(b)}$ is $h_{R(b)} + p_{R(b)} \cdot h' : A \to 2p_{R(b)}$. It is similar for $p_{S(b)}$.

Finally, the dispatcher manages assignment of tasks to workers. Since the tasks are almost balanced, the dispatcher keeps a count on the total number of tasks each worker is currently handling. When a new task is created, it will be sent to the worker with the minimum number of tasks. When a task is released, the dispatcher decreases the count of the corresponding worker by 1. Note that some tasks may be assigned to the same worker, so that they are only logically separated, but the communication between them can be easily achieved. The worker nodes receiving tasks compute join results, and send it to the sink.

3.3 A Running Example

Figure 3 gives a running example of state migration. Suppose at some point of the algorithm, 500 tuples have been received, where 200 tuples contain b_1 and 250 tuples contain b_2. There are 50 other tuples of light hitters. The output size from heavy hitters is

$$OUT^H = |R(b_1)| \cdot |S(b_1)| + |R(b_2)| \cdot |S(b_2)| = 20k$$

[1] $[p] = \{1, \ldots, p\}$

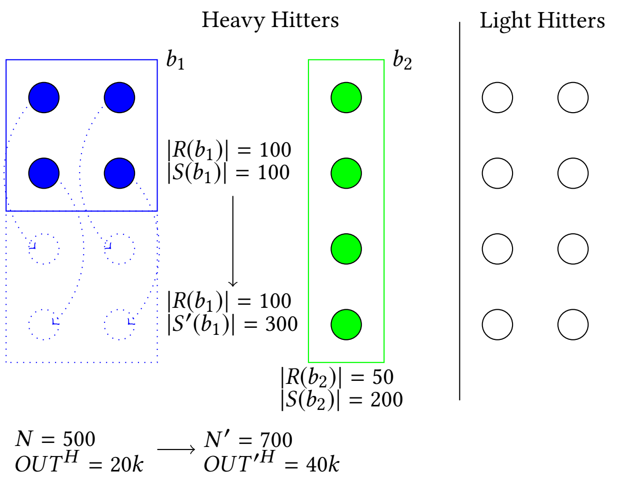

Figure 3: Example task allocation for $p = 8$. There are 8 tasks for light hitters. 2 heavy values b_1 and b_2 each occupies 4 tasks. The dotted circles represents the new tasks created in state migration.

Also assume task allocation is optimal at this time. Indeed, $p_{R(b_1)} = 2, p_{S(b_1)} = 2, p_{R(b_2)} = 1, p_{S(b_2)} = 4$. Note that the task layouts are different between two heavy hitters because they have different internal skew.

Suppose after this state, we received 300 more tuples from $S(b_1)$ and no other tuples. Consider b_1, the dispatcher calculates the new desired number of tasks as:

$$\begin{aligned} p^d_{R(b_1)} &= \frac{|R(b_1)|}{\sqrt{OUT^H}}\sqrt{p} \\ &= \frac{100}{\sqrt{40,000}}\sqrt{8} = 1.414 \end{aligned}$$

Since $2 = p_{R(b_1)} \le 2p^d_{R(b_1)}$, state migration does not need to be performed for $p_R(b_1)$. However,

$$\begin{aligned} p^d_{S(b_1)} &= \frac{|S(b_1)|}{\sqrt{OUT^H}}\sqrt{p} \\ &= \frac{300}{\sqrt{40,000}}\sqrt{8} = 4.243 \end{aligned}$$

indicates that $2 = p_{S(b_1)} < \frac{1}{2}p^d_{S(b_1)}$. The dispatcher decides $p_S(b_1)$ should be doubled, and creates 4 more tasks to handle it. As Figure 3 illustrates, the 2×2 grid is extended to a 2×4 one, where arrows denote the flow of tuples.

For b_2, the desired numbers are 0.707 and 2.828 respectively. No migration is needed since the original 1×4 grid is sill an optimal configuration, up to a factor of 2.

4 EVALUATION

In this section, we discuss implementation details and provide comparative evaluations of our algorithm with the widely implemented PHJ algorithm and state-of-the-art Join-Biclique (JB) [17] algorithm.

4.1 Implementation

In the current implementation of our SHC algorithm, we maintain a copy of each stream in the dispatcher node in the form of a hash table. The dispatcher is responsible for redistributing

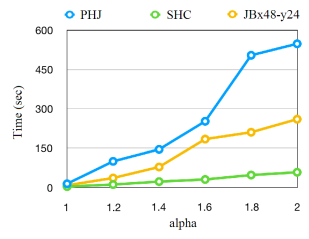

Figure 4: Zipf, varying α.

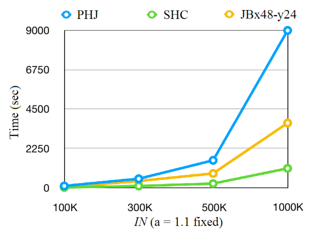

Figure 5: Zipf, varying IN.

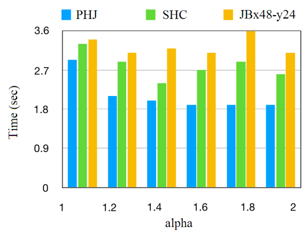

Figure 6: TPC-H, varying α.

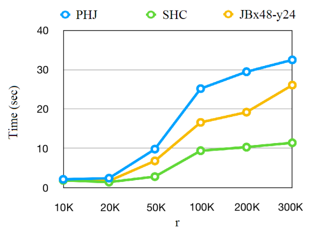

Figure 7: COREL, varying r.

certain parts of the state hashtables to the join processing workers according to state migration policies. In addition, the dispatcher may invalidate the state of certain heavy hitters from specific workers.

4.2 Experimental Setup

Environment. We conduct all experiments on an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz server that has 12 cores with 4 threads each and runs Red Hat 4.8.5. The server has 256 GB of memory, which suffices for maintaining all raw and intermediate data even if all tuples are sent to one processor only. Flink 1.4.2 is installed on the server and the implemented algorithms have maximum parallelism of $p = 48$ processing units.

Data sets. Our preliminary experiments focus on full-stream joins. We use two synthetic and one real data sets to evaluate our algorithm.

The first synthetic data set is generated from the Zipf distribution with varying value for α, which controls the level of the skewness. The second synthetic data set is the TPC-H benchmark [4]. Specifically, we choose to experiment on the Q5.

For the real data set, we use the same COREL dataset as in [13] for applying SRHC on similarity joins. The data set contains a set of 15,000 points with dimentionality 64. Each point corresponds to a histogram of a unique color image collected from the COREL repository. Based on a provided similarity threshold we apply a locality-sensitive hash function on every point, and afterwards we perform a self-join on the produced hash codes.

4.3 Performance Analysis

4.3.1 Zipf Distribution. Figure 4 and 5 demonstrates the performance of three algorithms on the generated Zipf data set, with respect to varying skewness and input size. For the JB algorithm, we present the performance of the best setting, namely JBx48-y24. The y parameter determines the number of groups that the processing units are organized into. SHC clearly outperforms both existing methods in terms of execution time.

4.3.2 TPC-H Benchmark. All TPC-H queries consists of *primary key-foreign key* joins, i.e., the join key is not skewed in one side of each join. Under this scenario, SHC will only assign a $1 \times p_{(b)}$ grid for each heavy value, which is simply broadcasting the primary key tuple to all allocations of the other stream. For JB, we observe that the more groups used, the less efficient the algorithm is. PHJ seems to yield the best performance. We stress that SHC algorithm is better on many-many joins, where skewness exists in both relations.

4.3.3 COREL data set. We perform self similarity join on this high dimensional data set utilizing a $(r, 10r, 0.9, 0.1)$-sensitive family of hash functions, i.e. points within r distance are hashed to the same bucket with at least 0.9 probability and points with more than $10r$ distance have only 0.1 probability to be hashed to the same bucket. This similarity join is a many-many join. Again, SHC algorithm outperforms the other two in terms of execution time. It is better especially when r is large, meaning a higher skewness.

5 CONCLUDING REMARKS

In this paper, we proposed the Streaming HyperCube algorithm that extends the static HyperCube algorithm to the streaming setting. The algorithm continuously maintains an approximation (up to a factor of 2) of the optimal HyperCube configuration, while inuring small state migration overhead. In the future, we plan to conduct a more thorough theoretical analysis and extend it to multi-way joins.

REFERENCES

[1] [n. d.]. Apache Flink Project. https://flink.apache.org/
[2] [n. d.]. Apache Spark Project. https://spark.apache.org/
[3] [n. d.]. Apache Storm Project. https://storm.apache.org/
[4] [n. d.]. The TPC-H benchmark. http://www.tpc.org/tpch/
[5] Foto Afrati, Manas Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D Ullman. 2014. GYM: A multiround join algorithm in mapreduce. *arXiv preprint arXiv:1410.4156* (2014).
[6] Foto N Afrati and Jeffrey D Ullman. 2010. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*. ACM, 99–110.
[7] Foto N Afrati and Jeffrey D Ullman. 2011. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering* 23, 9 (2011), 1282–1298.
[8] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in parallel query processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 212–223.
[9] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding frequent items in data streams. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1530–1541.
[10] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[11] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and adaptive online joins. *Proceedings of the VLDB Endowment* 7, 6 (2014), 441–452.
[12] Buğra Gedik, Rajesh R Bordawekar, and Philip S Yu. 2009. CellJoin: a parallel stream join operator for the cell processor. *The VLDB Journal—The International Journal on Very Large Data Bases* 18, 2 (2009), 501–519.
[13] Aristides Gionis et al. [n. d.]. Similarity search in high dimensions via hashing.
[14] Xiao Hu, Yufei Tao, and Ke Yi. 2017. Output-optimal parallel algorithms for similarity joins. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 79–90.
[15] Jaewoo Kang, Jeffrey F Naughton, and Stratis D Viglas. 2003. Evaluating window joins over unbounded streams. In *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 341–352.
[16] Paraschos Koutris, Paul Beame, and Dan Suciu. 2016. Worst-case optimal algorithms for parallel query processing. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 48. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[17] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable distributed stream join processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 811–825.
[18] Pratanu Roy, Jens Teubner, and Rainer Gemulla. 2014. Low-latency handshake join. *Proceedings of the VLDB Endowment* 7, 9 (2014), 709–720.
[19] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, et al. 2017. The Myria Big Data Management and Analytics System and Cloud Services.. In *CIDR*.