Flow-Join: Adaptive Skew Handling for Distributed Joins over High-Speed Networks

Wolf Rödiger*†, Sam Idicula‡, Alfons Kemper*, Thomas Neumann*

*Technische Universität München
Munich, Germany
{roediger, kemper, neumann}@in.tum.de

‡Oracle Labs
Redwood Shores, CA, USA
sam.idicula@oracle.com

***Abstract*—Modern InfiniBand interconnects offer link speeds of several gigabytes per second and a remote direct memory access (RDMA) paradigm for zero-copy network communication. Both are crucial for parallel database systems to achieve scalable distributed query processing where adding a server to the cluster increases performance. However, the scalability of distributed joins is threatened by unexpected data characteristics: Skew can cause a severe load imbalance such that a single server has to process a much larger part of the input than its fair share and by this slows down the entire distributed query.**

We introduce *Flow-Join*, a novel distributed join algorithm that handles attribute value skew with *minimal overhead*. Flow-Join detects heavy hitters *at runtime* using small approximate histograms and adapts the redistribution scheme to resolve load imbalances before they impact the join performance. Previous approaches often involve expensive analysis phases, which slow down distributed join processing for non-skewed workloads. This is especially the case for modern high-speed interconnects, which are too fast to hide the extra computation. Other skew handling approaches require detailed statistics, which are often not available or overly inaccurate for intermediate results. In contrast, Flow-Join uses our novel lightweight skew handling scheme to execute at the full network speed of more than 6 GB/s for InfiniBand 4×FDR, joining a skewed input at 11.5 billion tuples/s with 32 servers. This is 6.8× faster than a standard distributed hash join using the same hardware. At the same time, Flow-Join does not compromise the join performance for non-skewed workloads.

I. Introduction

Today's many-core servers offer unprecedented single-server query performance and main-memory capacities in the terabytes. Yet, a scale-out to a cluster is still necessary to increase the main-memory capacity beyond a few terabytes. For example, Walmart—the world's largest company by revenue—uses a 16 server cluster with a total of 64 TiB of main memory to perform analytical queries on their business data [21].

Data is commonly partitioned across servers so that users can utilize the combined main-memory capacity of the cluster. Consequently, query processing requires network communication between servers. Network speed used to be a bottleneck for distributed query processing such that a cluster actually performed worse than a single server. Previous work has thus focussed on avoiding communication as much as possible [25], [22]. However, the economic viability of high-bandwidth networks has changed the game: Modern high-bandwidth interconnects with link speeds of several gigabytes per second enable scalable query distributed processing where adding servers to the cluster in fact improves performance [2], [24].

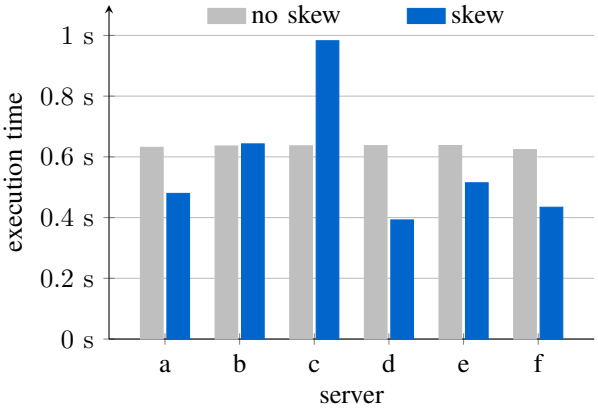


Figure 1. A skewed input causes a load imbalance for a standard distributed hash join (6 servers, InfiniBand 4×QDR at 4 GB/s, Zipf factor $z = 1.25$)

However, skew can cause a load imbalance during data shuffling and thus again threatens the scalability of distributed query processing as highlighted by Figure 1. The experiment shows the join execution time for each individual server in a 6-machine cluster comparing skewed and non-skewed inputs. The cluster is connected via InfiniBand 4×QDR, which offers a theoretical throughput of 4 GB/s. The skewed input causes server c to process many more tuples than the others and by this increases the overall join execution time by 54 %.

Heavy hitter skew is particularly harmful for partitioning-based operators (e.g., join and aggregation), as all tuples with the same partitioning key are assigned to the same server. While distributed aggregations can handle heavy hitter skew effectively using a fast in-cache pre-aggregation, there is no simple remedy for distributed joins. A distributed join partitions both inputs into as many partitions as there are servers. Only tuples from corresponding partitions will join and these partition pairs are thus assigned to servers. As tuples with the same key are assigned to the same server, heavy hitters cause a serious load imbalance as shown in Figure 1. In the example, a much larger number of tuples is assigned to

†Work conducted while employed at Oracle, Redwood Shores, CA, USA.

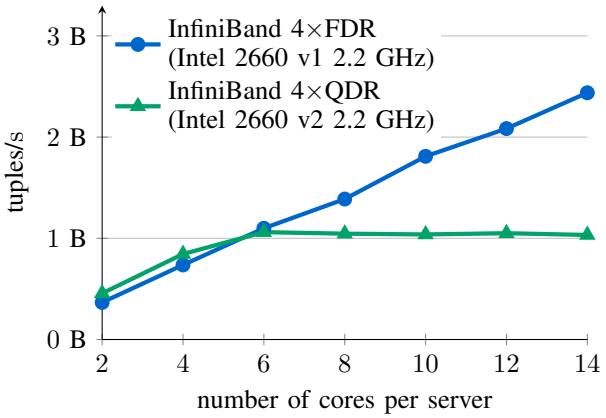


Figure 2. CPU-bound vs. network-bound distributed join processing (6 servers, no skew, 5040 build and 105 M probe tuples per server)

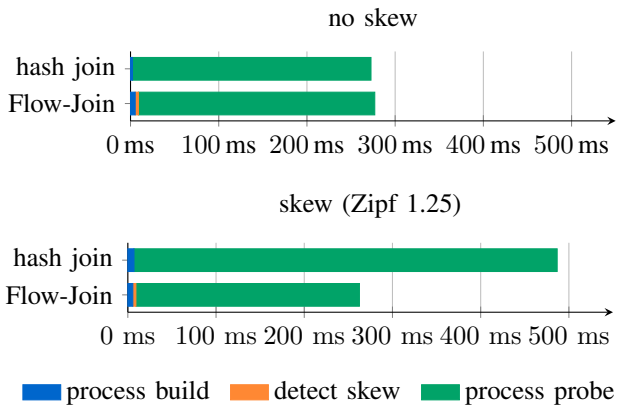


Figure 3. Breakdown of execution time for hash join and Flow-Join (6 servers, 4×FDR at 6.8 GB/s, 5040 build and 105 M probe tuples per server)

the third server. This impacts performance in two ways: First, communication becomes irregular, congesting the link to this server. Second, the third server must process many more tuples than the others. Ultimately, the overloaded server takes much longer to process its part of the input and thus slows down the entire join. With more servers the situation gets even worse: The larger the cluster, the larger the negative impact of skew.

Heavy hitter skew causes load imbalances during distributed query processing whether the network is slow or fast. However, for slow networks the actual join computation accounts for such a small fraction of the total execution time that even expensive skew handling approaches are a viable choice. For example, a distributed join over Gigabit Ethernet using 6 servers with 5040 build and 105 M probe tuples per server has a runtime of 13.2 s. The actual join computation takes less than 250 ms while the remaining time is spent waiting for network transfers. In this case, a preceding skew detection phase would increase runtime by only 2 % even if it performs as much computational work as the actual join.

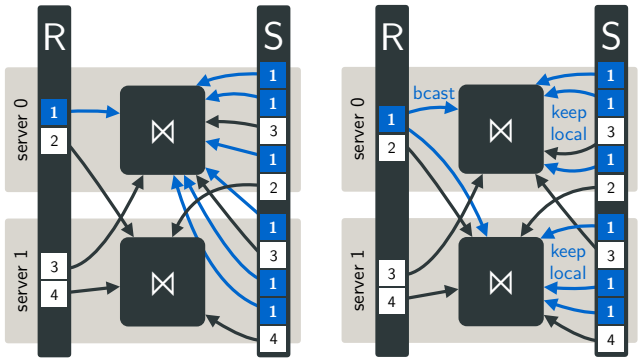
Modern high-speed interconnects have closed the gap between network bandwidth and compute speed: A distributed join is not necessarily network-bound anymore. Figure 2 shows that using additional cores for join processing keeps improving the join performance of a 6-server cluster that is connected via InfiniBand 4×FDR. This stands in contrast to the slower 4×QDR where six cores already suffice to saturate the available network bandwidth. In this experiment the size of the build input is chosen to fit into cache for fastest join processing. Even so, InfiniBand 4×FDR is still not the bottleneck. The experiment shows that skew detection cannot be hidden behind slow network transfers anymore. Instead, any additional work will directly translate into a visible increase in execution time. Modern high-speed networks thus require fast skew detection. This applies even more so for the upcoming InfiniBand EDR hardware, which will offer almost twice the bandwidth of FDR.

Previous approaches to handle skew depend either on detailed statistics [6], [34], which are often not available or overly inaccurate for intermediate results, or an extra analysis phase [15], [14], [33], [25], which is difficult to hide when InfiniBand is used. Flow-Join instead performs a lightweight heavy hitter detection alongside partitioning. It avoids load imbalances by broadcasting tuples that join with heavy hitters.

Flow-Join's skew detection and handling techniques add almost no overhead even for InfiniBand 4×FDR, which is more than 50× faster than Gigabit Ethernet: Figure 3 compares Flow-Join to a standard hash join for skewed and non-skewed inputs. The experiment runs on a 6-server cluster connected via 4×FDR at 6.8 GB/s using 5040 build and 105 M probe tuples per server. Both join algorithms perform similar for the non-skewed input with Flow-Join's skew detection adding only a minimal overhead of 1.6 %. For skewed inputs, the hash join takes 79 % longer, while Flow-Join actually performs 5 % faster compared to the non-skewed input as it keeps heavy hitter tuples local, thereby reducing network communication.

Flow-Join does not have a separate skew detection phase. It detects skew while it partitions the first 1 % of the probe input, after which a global consensus on skew values is formed. The implementation of both algorithms overlaps computation (partitioning, build, and probe) with network communication so that the join finishes shortly after the last network transfer. Our adaptive skew handling approach enables pipelined execution of joins and thus avoids the materialization of the probe side, reducing main-memory consumption significantly. In particular, this paper makes the following contributions:

1) A novel mechanism to detect heavy hitters alongside partitioning of the inputs that incurs only minimal overhead, comparing successively-refined implementations of the employed approximate histograms.
2) A method to adapt the redistribution scheme at runtime for a subset of the keys identified as heavy hitters. We broadcast corresponding tuples to avoid that all heavy hitter tuples are assigned to one server.
3) Based on these two techniques, a highly-scalable implementation of Flow-Join that utilizes RDMA to shuffle data at full network speed and distinguishes between local and distributed parallelism to avoid the inflexibility of the classic exchange operator model.
4) A generalization of Flow-Join beyond key/foreign-key equi joins that employs the Symmetric Fragment Replicate redistribution scheme to optimally handle correlated skew in both inputs.
5) Finally, an extensive evaluation including a scalability experiment using Zipf-generated data as well as a real workload from a large commercial vendor.



(a) A standard hash join assigns the heavy hitter to a single server, causing it to become the bottleneck

(b) Selective Broadcast keeps the skewed probe tuples local and replicates the corresponding build tuple

Figure 4. Selective Broadcast avoids the load imbalance caused by skew

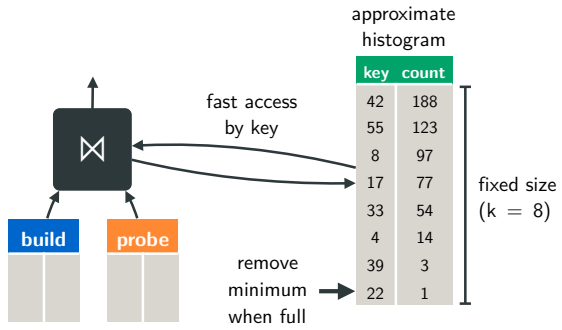


Figure 5. Flow-Join detects and adapts to heavy hitter skew at runtime using small approximate histograms that impose only a minimal processing overhead

II. Flow-Join

It is hard to detect skew at runtime without causing a significant overhead for non-skewed workloads. Materializing all tuples and computing histograms to decide on an optimal assignment takes time and will provide no benefit when the input is not skewed. While the additional computation might not add a noticeable overhead to the overall query execution time for slow interconnects such as Gigabit Ethernet, this is no longer the case for InfiniBand $4\times$FDR, which is more than $50\times$ faster. Any substantial computation in addition to core join processing is likely to increase query execution time when such high-speed networks are used.

Flow-Join is a skew-resilient distributed join algorithm with negligible overhead even for high-speed interconnects. It computes small, approximate histograms alongside partitioning to detect skew early. When a small percentage of the input has been processed, the frequencies in the histograms are checked. Servers exchange the approximate counts for join key values when they exceed a skew threshold—i.e., the expected frequency—by a large factor. Afterwards, all servers in the cluster know the heavy hitters. The tuples that join with heavy hitters are broadcast to avoid the load imbalance before it arises. The heavy hitters can be refined after more tuples have been processed as they might vary over time.

To simplify the presentation we restrict the discussion in this section to the common case of key/foreign-key equi joins. Section IV generalizes Flow-Join to equi joins beyond key/foreign-key relationships as well as non-equi joins. For key/foreign-key joins, heavy hitter skew is by definition limited to the foreign-key side as the attribute values of the other side are necessarily unique as a consequence of the primary key property. Therefore, correlated skew—i.e., both inputs are skewed on the same join key value—is also not covered here but as part of Section IV.

A. Selective Broadcast

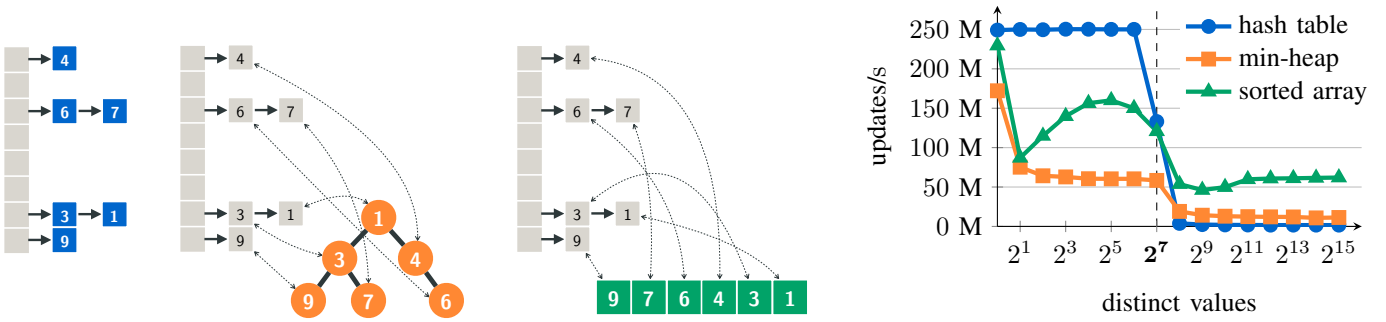
A standard hash join redistributes tuples between servers according to the hash value of the join key. Tuples with the same join key value will thus all end up at the same target server. An example is shown in Figure 4(a): Server 0 is assigned all tuples with the skewed join key 1 and as a result receives $4\times$ more tuples than server 1. This impacts the performance in two ways: First, network communication becomes irregular as most data is sent over the link to server 0. Second, server 0 must process many more tuples than its fair share. The execution time of distributed operators is determined by the slowest server. Consequently, the increased load at server 0 will slow down the entire query. This also affects the scalability of the system: Adding more servers to the cluster will not improve the performance as expected. The heavy hitter is still assigned to a single server and the query execution time thus remains largely unchanged.

An effective way to handle heavy hitter skew is the Selective Broadcast [34], [25], [22] redistribution method for distributed hash joins, which is known as Subset-Replicate [6] in the case of range partitioning. The key idea of Selective Broadcast is to keep tuples with skewed join keys local to avoid overloading a single server. Instead, the corresponding tuples of the other input are broadcast to all servers so that the heavy hitter tuples can be processed locally. This is shown in Figure 4(b) where the R tuples with join key 1 are broadcast, while S tuples with join key 1 are kept local. An equi-join algorithm that uses Selective Broadcast still computes the correct join result and at the same time avoids the load imbalance caused by a standard hash join (for the required changes to support non-equi joins see Section IV-C). Selective Broadcast yields performance results for skewed workloads that are on par with those for non-skewed inputs. For current systems, a distributed join using Selective Broadcast will in fact achieve higher performance for skewed than for non-skewed workloads as the heavy hitter tuples can be kept local and do not need to be materialized and sent over the network.

B. Heavy Hitter Detection

Even though Selective Broadcast seems to solve the problem of heavy hitter skew, there is one important caveat: The heavy hitter values have to be known beforehand. Previous approaches either assumed that heavy hitter elements can be deduced from existing statistics [6], [34]—which is often difficult and expensive for intermediate results—or are computed during a separate analysis phase that requires a complete materialization of both join inputs and additional processing [15], [14], [33], [25].

Flow-Join identifies heavy hitters alongside partitioning and thus does not need detailed statistics or a separate analysis



(a) Hash table (b) Hash table + heap (c) Hash table + sorted array (d) Single-threaded update rate, $k = 128$

Figure 6. Comparison of implementation alternatives for the approximate histogram that is used by Flow-Join to detect heavy hitters alongside partitioning

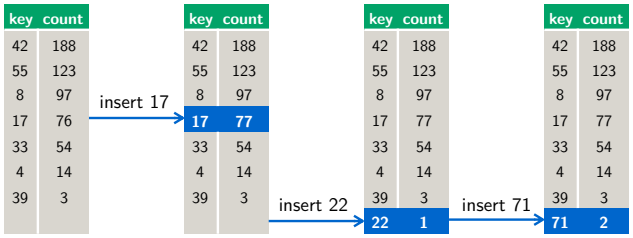


Figure 7. Example for the SpaceSaving algorithm using capacity k = 8 and replacing the entry with the minimum count once the histogram is full

phase. This enables pipelined join processing and avoids materialization of the probe side. The high-level idea of Flow-Join's algorithm to detect heavy hitters is shown in Figure 5. The join maintains approximate—and therefore extremely efficient—histograms for the probe input. Histograms are updated during join processing by incrementing the count for each probe tuple as it is processed. The tuple is only forwarded to the target server when its count is below the skew threshold. Otherwise, the probe tuple is kept local and the corresponding build tuple is broadcast to ensure the correct result.

1) Frequent Items Problem: Detecting heavy hitters during distributed join processing corresponds to finding frequent items in a data stream. This is known as the *frequent items problem*. The subsequent definitions are adapted from a recent survey [4] on algorithms that compute frequent items:

Definition 1: EXACT FREQUENT ITEMS PROBLEM: Given a stream $\mathcal{S}$ of n items $t_1, ..., t_n$, the frequency of item i is $f_i = |\{j \mid t_j = i\}|$, i.e., the number of indices j where the jth item is i. The exact ϕ-frequent items for a frequency threshold of ϕn are then defined as the set $\{i \mid f_i > \phi n\}$ that contains all items that occur more than ϕn times.

However, solving the frequent items problem *exactly* has been shown to require space linear in the size of the input [4]. Basically, a counter per join value would be required to detect the k most frequent items. Apart from the large amount of main memory needed, this would also be very compute-intensive to maintain due to cache misses and related issues. Thus, we instead focus on the *approximate* version of the frequent items problem with error tolerance ϵ:

Definition 2: APPROXIMATE FREQ. ITEMS PROBLEM: Given a stream $\mathcal{S}$ of n items, compute the set F of approximate frequent items such that each reported item $i \in F$ has frequency $f_i > (\phi - \epsilon)n$. This allows a reported item to occur ϵn times less often in the stream than the specified threshold of ϕn. Further, there should be no $i \notin F$ such that $f_i > \phi n$, i.e., all items i with a frequency larger than ϕn have to be reported.

Flow-Join's heavy hitter detection is based on the Space-Saving algorithm [19], which solves the frequency estimation problem with error $\epsilon = 1/k$, where k is the histogram size. The frequency of a reported item is off by at most a factor of $1/k$. Solving the frequent items problem *approximately* with the SpaceSaving algorithm reduces the space requirements significantly compared to the exact version. For example, a histogram with 100 entries is sufficient to compute all items with frequency 1 % or higher. This already suffices to detect all heavy hitters that could potentially impact the performance of a cluster with dozens of machines.

SpaceSaving reports all frequent items. However, there is no guarantee that a reported item is indeed frequent. This is tolerable for our use case, as broadcasting a few additional tuples will not impact performance noticeably. The aforementioned survey on algorithms for the frequent items problem has shown that SpaceSaving is faster, more accurate, and requires less space than other counter-, quantile-, and sketch-based alternatives [4], which we will thus not consider. Figure 7 shows an exemplary execution of the SpaceSaving algorithm for a histogram with space for $k = 8$ elements. In the example there are already 7 elements in the histogram from the beginning. The insertion of the existing key 17 simply increments its count. The subsequent insertion of the new value 22 fills the last remaining free slot. Consequently, there is no free slot for key 71, which instead replaces the element with the smallest count. The count of the previous element 22 is kept and incremented by one, yielding a total count of 2 for the new element 71. Keeping the count of the previous element when a new element is inserted into a full histogram is necessary to ensure an item is never underestimated. This approach bounds the error for the frequency estimation problem to $1/k$ [19].

2) Data Structures: The main challenge introduced by the SpaceSaving algorithm is that elements are accessed in two different ways: via their *key* for updates and their *count* to remove the minimum. The approximate histogram has to support both operations in a very efficient manner. To this end, we compared three successively-refined data structures characterized in Figure 6. The first data structure, depicted

in Figure 6(a), is a hash table, which enables key access in $O(1)$. However, removing the minimum is expensive as it incurs a full scan of the hash table with $O(k)$ operations. Figure 6(b) combines the hash table with a heap to reduce the cost for accessing the minimum. However, the heap requires $\log k$ moves for every update and removal. Thus, insert, update, and remove minimum are now all in $O(\log k)$. Figure 6(c) replaces the heap with a sorted array. This enables a remove minimum that is in $O(1)$. While "increment key" is now worst case $O(k)$, this only occurs for the rightmost element when all elements have the same count. In practice, however, an element rarely moves more than one position to the left.

The experiment shown in Figure 6(d) inserts 10 million random keys for a uniform distribution without skew into histograms that have capacity for $k = 128$ elements. The number of distinct values increases along the x-axis. Note that the x-axis is logarithmic so the hash table outperforms the alternatives only as long as there are very few distinct values. The update rate of the hash table drops severely as soon as the number of distinct values exceeds the capacity $k = 128$. At this point the remove minimum operation becomes necessary, which incurs an expensive scan of the hash table. Combining the hash table with a heap to support a fast remove minimum operation improves performance by up to $7\times$ once there are more than $k = 128$ distinct values. The sorted array implementation further improves over this and outperforms the hash table by at least $28\times$ and up to $37\times$ when there are more than $k = 128$ distinct values. Increasing k shifts this limit but does not change the qualitative result. k should be chosen reasonably small to avoid expensive cache misses.

The sorted-array implementation is very fast, processing about 60 million keys/s *per thread*. The implementation of Flow-Join uses one thread per core to leverage the available parallelism of modern many-core servers. Worker threads do not share a single approximate histogram but instead each use their own private histogram. This avoids the cost of locking or atomic operations required when sharing a data structure. Flow-Join uses the sorted-array implementation with a capacity of $k = 128$ elements and 256 hash-table entries. This results in a hash table load factor of at most $1/2$ and a memory footprint of only $2.5\,\mathrm{KiB}$, ensuring fast in-cache processing.

C. Early and Iterative Detection

Early detection of heavy hitter values is important to adapt the redistribution scheme as soon as possible so that imbalances in network communication and CPU utilization can be resolved before they impact the join performance. Flow-Join combines histograms on a per-server level after processing approximately $1\,\%$ of the probe input. Afterwards, the servers combine the resulting skew lists on the cluster level to reach a global consensus. This is quite cheap: Each server sums up the counts for the join key values of its local per-thread approximate histograms. It then forwards those join key values together with their counts that still exceed the skew threshold to one of the servers. This one server again adds up the counts and thus determines the global list of heavy hitter values as those join key values that still exceed the threshold. It notifies the servers responsible for the corresponding build tuples (identified via the join's hash function). These servers then broadcast the build tuples across the cluster.

Sampling the first $1\,\%$ of the input ensures the early detection of the heavy hitter values. Once the build tuples are broadcast, the materialized skewed probe tuples can be joined. This process can be repeated periodically during join processing to iteratively refine the heavy hitter detection in case the heavy hitters in the probe input vary over time.

D. Fetch on Demand

Combining all histograms in the cluster did not introduce a noticeable overhead in our experiments even for 32 servers. Still, the global consensus requires synchronization between servers. It is possible to avoid synchronization and reduce the materialization of heavy hitters to a minimum: Instead of creating a shared global list of skewed join values, servers (or even individual worker threads) can decide on the heavy hitter values on their own. Once a join key value exceeds the skew threshold, they fetch the corresponding build tuple asynchronously from the responsible server (identified via the join's hash function) similar to the PRPQ scheme by Cheng et al. [3]. A requesting server does not even need to wait for the response but can continue processing its input and simply materialize the probe tuples for the outstanding heavy hitter. Once the remote server has responded with the build tuple, the requesting server can join the materialized probe tuples. We call this refined approach *fetch on demand*.

Fetch on demand has a second benefit apart from avoiding the synchronization of the global consensus approach. Different servers (or even worker threads) could encounter different heavy hitters in their part of the input. Fetch on demand allows them to request only the build tuples for heavy hitters relevant to them while global consensus always broadcasts the build tuples for all heavy hitters across the whole cluster.

E. Hash Join Algorithm

There are different ways to implement a distributed hash join including the classic Grace-style hash join [16] and the distributed radix join [1]. Both suffer from the load imbalance caused by heavy hitter skew as they have to assign each heavy hitter join key value to a single server. Barthels et al. [1] have shown that a distributed radix join experiences a $3.3\times$ slow down for a skewed workload with Zipf factor $z = 1.2$ on an 8-server cluster. We observed similar results for the standard partitioned hash join, which suffers from a $2.1\times$ slow down for Zipf factor 1.25. The effect of skew further intensifies with the cluster size—a standard hash join slows down by $5.2\times$ on a 32-server cluster for a skewed workload with Zipf factor 1.25. This highlights the need for low-overhead skew handling techniques whether the underlying join algorithm is a standard Grace-style hash join or a distributed radix join.

Flow-Join is independent of the underlying hash join algorithm used. During partitioning of the tuples Flow-Join uses approximate histograms to detect heavy hitters and Selective Broadcast to handle them. This applies to Grace-style hash joins [16] as well as radix joins [1]. We decided to implement the former to evaluate Flow-Join. In the experiments we focus on the worst case for Flow-Join, i.e., the build relation fits into cache. In this setting, the cost of skew detection and handling has the largest visible impact as the join performs fastest. A larger build side would cause cache misses, slowing down the join and thus making the overhead of Flow-Join less visible.

Table I. COMPARISON OF NETWORK DATA LINK STANDARDS

	GbE	InfiniBand (4×)				
		SDR	DDR	QDR	**FDR**	EDR
bandwidth [GB/s]	0.125	1	2	4	**6.8**	12.1
compared to GbE	1×	8×	16×	32×	**54×**	97×
latency [µs]	340	5	2.5	1.3	**0.7**	0.5
introduction	1998	2003	2005	2007	**2011**	2014

III. IMPLEMENTATION DETAILS

Our implementation of Flow-Join uses remote direct memory access (RDMA) for network communication to utilize all the available bandwidth offered by modern high-speed interconnects such as InfiniBand 4×FDR. Flow-Join combines exchange operators for distributed processing with work stealing across cores and NUMA regions for local processing to scale join processing to large clusters of many-core servers.

A. High-Speed Networks

InfiniBand is a high-bandwidth, low-latency cluster interconnect offering several different data rates, which have been standardized over the years since its introduction in 2001 as shown in Table I. Most experiments in this paper were performed on a 32-server cluster that is connected via InfiniBand 4×FDR hardware providing more than 50× the bandwidth of Gigabit Ethernet and latencies as low as 0.7 µs.

InfiniBand offers the choice between two transport protocols: standard TCP via *IP over InfiniBand* (IPoIB) and an InfiniBand-native *ibverbs* interface, which enables remote direct memory access (RDMA). Figure 8 compares the throughput of TCP/IP over Gigabit Ethernet and InfiniBand with that of RDMA for a full-duplex stream of 512 KiB messages. The experiment shows that RDMA is necessary to achieve a network throughput near the theoretical maximum of 6.8 GB/s for InfiniBand 4×FDR. This stands in contrast to TCP/IP, which causes significant CPU load—fully occupying one core—and requires complex tuning as well as the use of multiple data streams each using a dedicated core to come close to the throughput of RDMA [24]. For the 32-server cluster used in the evaluation of this paper with 8 cores per CPU and a fast 4×FDR interconnect, using all the available bandwidth with TCP/IP would likely occupy most of the cores of one of the two CPUs—compared to virtually no CPU cost for RDMA. Flow-Join's implementation thus uses RDMA instead of standard TCP/IP for network communication.

InfiniBand's *ibverbs* interface is inherently asynchronous and thus requires a distinctly different application design. An application has to post work requests to the work queues of the InfiniBand card, which operates completely asynchronously. The card inserts a notification to a completion queue once it has finished a transfer. As its name suggests, RDMA reads and writes memory without involving the operating system or application during transfers. The performance-critical data transfer path thus involves no CPU cost at all. The application has to manage network buffers explicitly to enable these zero-copy network transfers. Registering the network buffers with the network card before communication is necessary as the kernel has to pin them to main memory to avoid swapping to disk. Network buffers should be reused as much as possible as registration is a costly operation [9]. We use a message pool to reuse buffers and avoid the cost of repeated memory allocation and registration with the InfiniBand card.

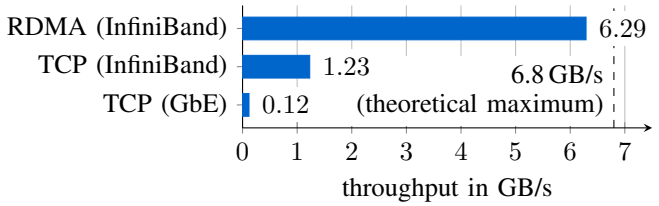


Figure 8. Only remote direct memory access (RDMA) can fully leverage InfiniBand (512 KiB messages, duplex communication, one stream, 4×FDR)

RDMA further offers two different application semantics: Memory semantics allow the sender to read and write the receiver's memory without its involvement, requiring that both exchange information beforehand to identify the target memory region. With channel semantics both sender and receiver post work requests that specify the memory regions. This renders the preceding information exchange unnecessary. Channel semantics offer the additional benefit that not only the sender but also the receiver is notified when the data transfer has finished. For these reasons, the implementation of Flow-Join's communication multiplexer uses channel semantics.

B. Local and Distributed Parallelism

The exchange operator [12] is a landmark idea as it allows systems to encapsulate parallelism inside an operator. All other relational operators are kept oblivious to parallel execution, making it straightforward to parallelize existing non-parallel systems. The exchange operator is commonly used to introduce parallelism both inside a single machine and between servers (e.g., Vectorwise Vortex [5] and Teradata [34]). However, the classic exchange operator has several disadvantages as it introduces unnecessary materialization cost during local processing, is inflexible in dealing with load imbalances and thus especially vulnerable to attribute value skew, and further faces scalability problems due to the large number of connections required for large clusters of many-core servers [24].

Flow-Join instead implements local and distributed parallelism differently. On the local level, instead of using traditional exchange operators, we parallelize Flow-Join similar to the morsel-driven approach by Leis et al. [17]. Workers process small NUMA-local chunks of tuples to avoid expensive remote memory accesses across CPU sockets. Work stealing allows faster workers to steal work units from workers that lag behind. This effectively resolves local load imbalances. The impact of work stealing grows with the number of cores. Threads are pinned to cores to avoid expensive thread migrations. On the global level, exchange operators are connected via RDMA-based communication multiplexers instead of directly to each other. The multiplexer ensures that there is always at least one packet in flight for every target server to use all the available bandwidth of the InfiniBand network. Both levels of parallelism are seamlessly integrated into a new approach that avoids unnecessary materialization, is flexible in dealing with load imbalances, and offers near-linear scalability in both the number of cores and servers in the cluster.

IV. Generalized Flow-Join

This section generalizes Flow-Join beyond key/foreign-key equi joins. The high-level idea is shown in Figure 9: Tuples with a join value that is neither skewed in R nor in S are partitioned, R tuples that join with heavy hitters in S are broadcast and S tuples that join with heavy hitters in R are handled similarly. Tuples with a join value that is both a heavy hitter in R and in S are redistributed according to the Symmetric Fragment Replicate (SFR) [28] data shuffling scheme as described in Section IV-B.

A. Algorithm

In the following we will describe the generalized Flow-Join in more detail. We denote the build input as R and the probe input as S. Typically, the smaller input is used as build side to keep the hash table for the local join small, we therefore follow this convention. However, Flow-Join's skew detection and handling does not require this. The build input R is distributed before the probe input S to enable pipelined probing. Flow-Join avoids materialization as much as possible to minimize memory consumption. In detail, the generalized Flow-Join proceeds as follows:

1) **Exchange R:** For each tuple in R:
   - Update approximate histogram
   - Compare heavy hitter count to threshold:
     a) *skewed:* insert tuple into local hash table
     b) *otherwise:* send tuple to target server
   - Create global list of heavy hitters in R
2) **Exchange S:** For each tuple in S:
   - Update approximate histogram
   - Compare heavy hitter count to threshold:
     a) *skewed in S:* materialize S tuple as the corresponding R tuple is not broadcast at this point in time
     b) *skewed in R (but not S):* broadcast S tuple to all servers
     c) *otherwise:* send S tuple to target server
   - Create global list of heavy hitters in S
3) **Handle skew:** (necessary if skew is detected in S)
   a) Broadcast R tuples that join with heavy hitters in S that are not also heavy hitters in R, join the corresponding materialized S tuples
   b) Redistribute tuples whose join key is a heavy hitter in both R and S via the Symmetric Fragment Replicate redistribution scheme

The generalized Flow-Join computes the correct join result as explained in the following paragraphs:

- The algorithm partitions tuples with join key values that are **not skewed** in R nor in S in step 1(b) and 2(c). These tuples are joined correctly at their target server.
- R tuples for heavy hitters **skewed only in R** are kept local after the skew threshold is met in step 1(a) while before that they were sent to the target server in step 1(b). In both cases the tuples are joined correctly as the corresponding S tuple for the heavy hitter is broadcast in step 2(b).

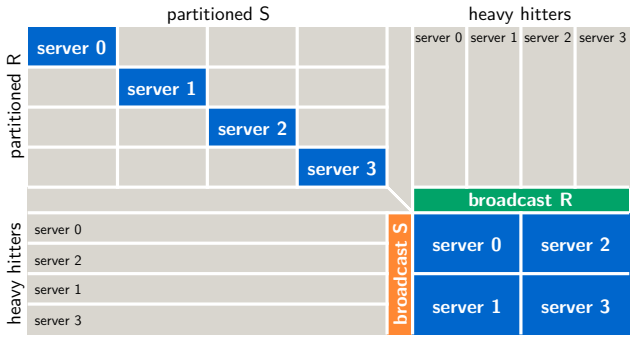


Figure 9. The generalized Flow-Join detects and handles skew in both inputs, using Symmetric Fragment Replicate (SFR) to deal with correlated skew

- S tuples for heavy hitters **skewed only in S** are materialized after the skew threshold is met in step 2(a) while before that they were sent to the target server in step 2(c). The S tuples sent to the target server are joined correctly as the corresponding R tuple was previously sent there in step 1(b). The materialized S tuples are probed locally after the corresponding R partition has been broadcast in step 3(a).
- R tuples for heavy hitters **skewed in both R and S** are kept local after the skew threshold is met in step 1(a) while before that they were sent to the target server in step 1(b). In both cases the tuples are redistributed using SFR in step 3(b). Note that these tuples were not replicated in step 3(a) and exist only on a single server, avoiding spurious results.
- S tuples for heavy hitters **skewed in both R and S** are materialized after the skew threshold is met in step 2(a). Note that S tuples broadcast in step 2(b) before the skew threshold was met were probed correctly and are not considered in this step. The materialized S tuples and corresponding R tuples are redistributed and joined via SFR in step 3(b).

The following section describes step 3(b) and the handling of correlated skew with SFR in detail.

B. Correlated Skew

A join key value that is a heavy hitter for both inputs is even more problematic than tuples skewed for only one input. The join for such a heavy hitter is basically a cross product and thus causes not only a severe load imbalance during data redistribution but also generates a quadratic number of result tuples on a single server. While broadcasting the tuples from one of the two inputs—as proposed by previous approaches [6], [34]—balances data redistribution and result generation across servers, it still incurs a significant cost for broadcasting a large number of heavy hitters.

The Symmetric Fragment Replicate (SFR) [28] data shuffling scheme can be used to handle correlated skew at least as good as a broadcast. In many cases SFR is able to reduce query execution time significantly. Instead of assigning all heavy hitter tuples to a single server or broadcasting them, the servers are logically arranged in a grid. Servers replicate heavy hitters for one input across rows, while those of the other

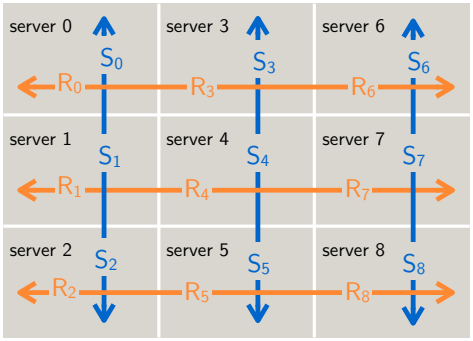


Figure 10. The Symmetric Fragment Replicate redistribution scheme logically organizes the servers of the cluster in a rectangle; heavy hitter tuples are replicated across rows for one input and across columns for the other input

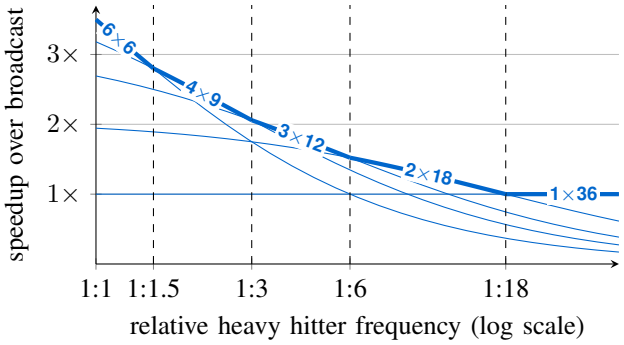


Figure 11. The optimal SFR rectangle depends on the relative frequency of the heavy hitter in the two inputs (36-servers)

input are replicated across columns. An example with $n = 9$ servers is shown in Figure 10. This scheme ensures that every heavy hitter tuple from one input is joined exactly once with every heavy hitter tuple of the other input. The join site for two heavy hitter tuples from relation R and S, respectively, is the server at the intersection of the corresponding row and column. SFR reduces query execution time and network traffic by up to a factor of $(\sqrt{n}+1)/2$ compared to broadcasting.

Assuming a cluster of n servers and a heavy hitter that occurs x times in R and y times in S per server: Assigning the heavy hitter to a server costs $(n-1)(x+y)$ time units as every server has to send the $x+y$ heavy hitter tuples to the target server (except the target server itself), causing a congestion on the receive link of the target server. Broadcasting R costs $(n-1)x$ time units as every server sends its x heavy hitters of the smaller side R to every other server using all network links in parallel. SFR reduces this to $(n_1-1)x+(n_2-1)y$ time units, where n_1 and n_2 are the number of rows and columns of the SFR rectangle (with $n_1 \times n_2 = n$): The heavy hitter tuples of R are replicated across n_1 rows and those of S across n_2 columns. Again, all links of the network are used in parallel.

The optimal grid shape depends only on the relative frequency of the heavy hitter in both inputs, e.g., a quadratic shape is best when the heavy hitter occurs in both inputs with roughly the same frequency. The other extreme is a rectangle with only one row (or column) for a heavy hitter that occurs mostly in one of the two inputs. In this case SFR degenerates to a broadcast. Figure 11 shows the optimal rectangles for $n = 36$ servers for different heavy hitter frequency ratios and the corresponding speedup of SFR over a broadcast. For example, when frequencies differ only by up to a factor of 1.5, the quadratic 6×6 shape is the best choice and improves performance by up to $3.5\times$ (general case: $\frac{\sqrt{n}-1}{2}\times$) compared to broadcasting. On the other hand, when the frequency for one input is more than $18\times$ (general case: $\frac{n}{2}\times$) larger than for the other, SFR degenerates to a broadcast with a 1×36 shape (general case: $1 \times n$). The potential rectangles are limited to the integer divisors of the number of servers n. For example, a quadratic shape is only possible when the number of servers is a square number.

SFR reduces the load imbalance, i.e., query execution time, but not necessarily the amount of network traffic. While SFR never sends more tuples than broadcasting, the assignment of the heavy hitter to a single server can reduce the network traffic at the cost of a significantly increased execution time. The assignment of the heavy hitter to a single server transfers $(n-1)(x+y)$ tuples as every server sends their heavy hitter tuples to the target server (except the target server itself). A broadcast transfers $n(n-1)x$ tuples as every server sends the heavy hitter tuples for the smaller input to every other server. SFR reduces network traffic to $n((n_1-1)x+(n_2-1)y)$ tuples as every server sends its heavy hitter tuples for one input across the n_1 rows and for the other input across n_2 columns.

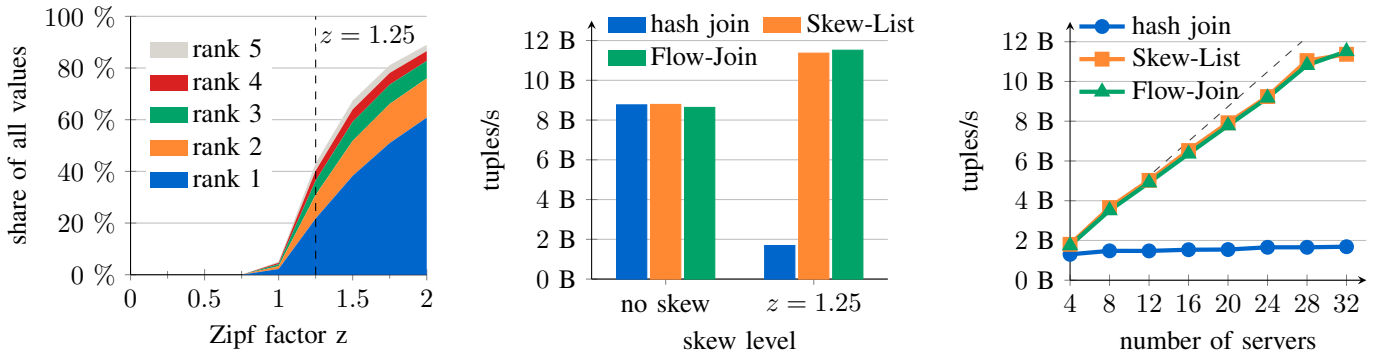
C. Non-Equi Joins

Similar to a broadcast join [8], which replicates the smaller input across all servers while it keeps the larger one fragmented, Flow-Join has to be adapted to compute the correct result for semi, anti, and outer joins. This is owed to the Selective Broadcast and Symmetric Fragment Replicate [28] redistribution schemes, which replicate tuples and can thus create spurious and duplicate results.

The semi join may produce duplicate result tuples at different join sites. The result therefore has to be redistributed to eliminate these duplicates. The anti join will produce valid results m times, once for each of the m join sites the tuple was replicated to. A tuple may find a join partner at only a subset of its join sites and thus cause spurious results at the remaining sites, where it does not find a join partner. The result needs to be redistributed and a tuple is to be included in the result only if it occurs exactly m times, i.e., if it found no join partner at any of its join sites. The outer join may produce duplicates and false results similar to the anti join. Dangling tuples have to be redistributed and counted. Only dangling tuples that found no partner across all of their join sites have to be kept in the final result. The following table lists the required changes for Flow-Join to support non-equi joins:

	$R \ltimes S$	$R \rtimes S$	$R \triangleright S$	$R \triangleleft S$	$R \bowtie S$	$R \bowtie S$
hash join	✓	✓	✓	✓	✓	✓
Flow-Join	✓[1]	✓[1]	✓[1]	✓[1]	✓[2]	✓[2]

[1]Requires redistribution of the result.
[2]Requires redistribution of dangling tuples.



(a) Ratio of the five most frequent elements for an increasing Zipf factor z (64 bit integer keys)

(b) Flow-Join incurs minimal overhead for non-skewed inputs, is 6.8× faster for skew (32 servers)

(c) Flow-Join scales near-linearly before the switch becomes a bottleneck (Zipf factor $z = 1.25$)

Figure 12. Flow-Join detects heavy hitter values at runtime and still performs as well as the Skew-List algorithm, which takes a list of heavy hitters as input

V. Evaluation

This section evaluates our approach with two scenarios: In the first scenario, we generate skewed data that follows a Zipf distribution. This allows us to scale the input size easily with the number of servers for a scalability experiment. The second scenario uses a real workload from a large commercial vendor, demonstrating that Flow-Join can improve query processing performance in practice. At last, we will evaluate several parameters, including the build input size, the Zipf factor z, the skew threshold, and the number of reported heavy hitters.

A. Experimental Setup

The experimental setup is illustrated in Figure 13 where the line width of each connection corresponds to its available bandwidth. We conducted most experiments in this paper on a cluster of 32 servers connected via Connect-IB InfiniBand cards at a 4× fast data rate (FDR) resulting in a theoretical network bandwidth of 6.8 GB/s per incoming and outgoing link. The aggregate bandwidth of the cluster for all-to-all data shuffles is thus 218 GB/s. Each Linux server is equipped with two Intel Xeon E5-2660 CPUs clocked at 2.2 GHz with 8 physical cores each (16 hardware contexts per CPU due to hyper-threading) and 64 GiB of main memory per CPU—resulting in a total of 512 cores (1024 hardware contexts) and 4 TiB of main memory in the cluster.

B. Scalability

The first scenario depicted in Figure 12 compares Flow-Join to a standard hash join and Skew-List. Skew-List is an omniscient variant of Flow-Join that knows all heavy hitter values beforehand. It takes a predefined list of heavy hitter values as part of its input instead of performing skew detection at runtime. Tuples consist of 64 bit key and 64 bit payload. The keys of the probe tuples follow a Zipf distribution with Zipf factor $z = 1.25$. The Zipf distribution is known to model real world data accurately, including the size of cities and word frequencies [13]. Given n elements ranked by their frequency, a Zipf distribution with skew factor z denotes that the most frequent item with rank 1 accounts for $x = 1/H_{(n,z)}$ of all values, where $H_{(n,z)} = \sum_{n}^{i=0} 1/i^z$ is the nth generalized harmonic number. The element with rank r occurs x/r^z times. Figure 12(a) illustrates the impact of the Zipf factor z on the share of the five most frequent elements for 64 bit integers. For $z = 1$, these five values occur in 5 % of all tuples, increasing to 43 % for $z = 1.25$, 67 % for $z = 1.5$ and 89 % for the extreme case of $z = 2$. Our experiments use Zipf factor $z = 1.25$. The input consists of 5040 build and 105 M probe tuples per server. A build input that fits into cache represents the worst case for Flow-Join: It ensures fastest processing, exposing any overhead incurred by skew detection and handling.

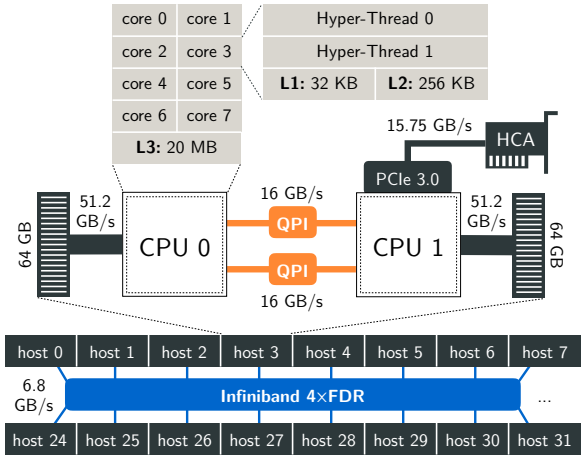
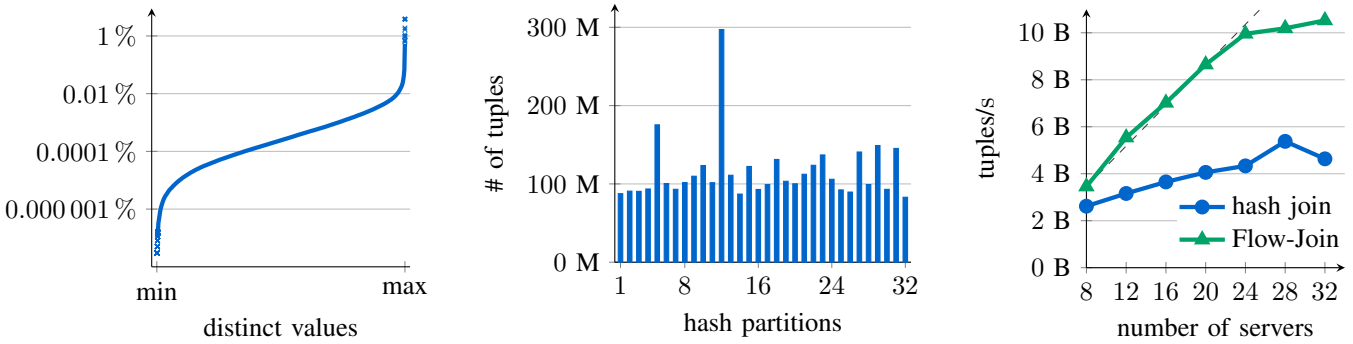


Figure 13. Experimental setup: Fully-connected InfiniBand 4×FDR cluster with 32 servers, resulting in a total of 512 cores and 4 TiB of main memory

Figure 12(b) compares the three distributed join algorithms using all 32 servers of the cluster. The experiment reveals two important findings: First, Flow-Join imposes only an insignificant overhead of 1.5 % for non-skewed workloads, joining 8.6 billion tuples/s (408 ms) compared to 8.8 billion tuples/s (402 ms) for the hash join and the Skew-List algorithm. Second, Flow-Join performs 6.8× better than the standard hash join for a skewed input with Zipf factor $z = 1.25$, joining 11.5 billion tuples/s (305 ms) compared to the 1.7 billion tuples/s of the standard hash join (2.1 s). There is no measurable performance overhead compared to the omniscient Skew-List algorithm, which knows the heavy hitter values beforehand. Flow-Join's performance will improve with the degree of skew as an increasing number of heavy hitters can be kept local and thus are neither materialized nor sent over the network.



(a) Data distribution of the case study: the most common join key occurs in 3.8 % of the tuples

(b) The largest probe partition has 2.6× more tuples than the expected 116 million (32 servers)

(c) Flow-Join scales near-linearly up to 24 servers (outlier due to hash function)

Figure 14. Flow-Join outperforms the standard hash join by up to 2.3× for a real workload from a large commercial vendor

Figure 15. The query plan for the case study; the highlighted join is subject to skew in its probe input (the subtree on the right); skew is in this case hard to infer from statistics as the probe input is an intermediate result

Figure 12(c) evaluates the scalability of the three join algorithms for the skewed workload by increasing the number of servers in the cluster. The standard hash join does not scale as it assigns all tuples of a heavy hitter join key to a single server and thus causes a severe load imbalance during data redistribution. Adding more servers improves performance only minimally. In contrast, Flow-Join scales near-linearly up to 28 servers. At this point, the switch becomes the bottleneck and throughput drops. Uncoordinated all-to-all communication reduces the aggregate bandwidth due to negative effects such as credit starvation for InfiniBand. Network scheduling [24] is required to scale beyond 32 servers.

C. Real Workload

The second scenario evaluates Flow-Join for a real workload from a large commercial vendor. The query plan of the use case is shown in Figure 15. The query plan follows the convention that the left input is used as build input for the local hash join while the right input is used to probe the hash table. The highlighted join operator is subject to skew on the probe side. It is difficult to anticipate the skew from table statistics alone in this and similar cases as the probe input is not a base relation but another join. The input consists of 53 000 build and 3.7 billion probe tuples, with a combined size of 55 GiB.

Figure 14(a) depicts the join key distribution for the probe input. The five most frequent join values occur in 9.4 % of all tuples. This moderately skewed input already leads to a huge imbalance during hash repartitioning as illustrated in Figure 14(b) for 32 servers: The largest partition has 2.6× more tuples than the expected $1/32$ of the input. This limits the scalability of a standard hash join as this one large partition has to be assigned to a single server, which consequently slows down the entire join. Flow-Join instead scales much better due to its low-overhead skew handling scheme as shown in Figure 14(c). It processes the 55 GiB input in only 350 ms using 32 servers. However, in this experiment network congestion already kicks in at 28 servers due to the lower degree of skew, which causes a larger amount of data to be shuffled across the network. Again, network scheduling would be necessary to keep up near-linear scalability. The outlier for the hash join at 28 servers is due to the hash function, which in this case assigns heavy hitters more evenly across partitions.

D. Impact of Workload Characteristics and Parameters

The experiments in this paper use a build input that fits into cache as this represents the worst case for Flow-Join, revealing any overheads caused by skew detection and handling. We also conducted experiments that vary the size of the build and probe input. Our experiments revealed that the join duration increases proportionally with the size of the probe input as expected, when the size of the build input is fixed. Using a larger build input that exceeds the L3 cache increases the cost for probing the hash table, resulting in a 2.5× longer runtime—independent of whether skew detection is performed or not. However, as a consequence the relative overhead for skew detection and handling in fact decreases: The actual join processing becomes slower, but the cost for Flow-Join's heavy hitter detection stays the same. When the build input exceeds the cache, a partitioning join such as [18], [29] could be used. Flow-Join's techniques to detect and handle skew still apply.

The skewness of the data is another important workload characteristic. We varied the Zip factor z between 0 and 2. The runtime of the standard hash join increases proportionally with the size of the largest probe partition as expected. The

size of the largest probe partition is directly determined by the most frequent heavy hitter. The effect is even higher when several heavy hitters are assigned to a single partition.

Flow-Join has two important tuning parameters: The maximum number of heavy hitters h reported per node during global consensus (by default h is unlimited) and the skew threshold (defaults to $0.01\,\%$). Together they determine how many heavy hitters are reported during skew detection. A node reports up to h heavy hitters that exceed the skew threshold. Lowering the skew threshold respectively increasing h will report more heavy hitters and thus broadcast more build tuples across the cluster. We conducted an experiment with 8 servers, Zipf factor 1.25, and unlimited h that varies the skew threshold from $0.01\,\%$ to $100\,\%$ in steps of $10\times$. For $0.01\,\%$ 121 heavy hitters are reported and the join executes in $252\,\mathrm{ms}$. Using a threshold of $0.1\,\%$ detects 49 heavy hitters, increasing the runtime by $1.2\,\%$ to $255\,\mathrm{ms}$. A threshold of $1\,\%$ reveals 12 heavy hitters, resulting in a $12\,\%$ higher execution time of $282\,\mathrm{ms}$. A threshold of $10\,\%$ discovers only one heavy hitter, increasing the runtime by $38\,\%$ to $349\,\mathrm{ms}$. A threshold of $100\,\%$ does not detect any heavy hitters, giving a $134\,\%$ longer runtime of $592\,\mathrm{ms}$. We observed similar results when the skew threshold is fixed to $0.01\,\%$ and we instead vary the maximum number of heavy hitters reported per node: Choosing $h = 128$ results in a runtime of $252\,\mathrm{ms}$, $h = 64$ increases this by $2\,\%$ to $257\,\mathrm{ms}$, $h = 16$ leads to a $15\,\%$ higher execution time of $289\,\mathrm{ms}$, $h = 1$ gives a $40\,\%$ longer runtime of $353\,\mathrm{ms}$, and $h = 0$ results in a $135\,\%$ increased runtime of $593\,\mathrm{ms}$.

VI. Related Work

Making distributed joins resilient to attribute value skew is a popular topic in the database literature [15], [14], [6], [33], [27], [34], [25], [23]. Yet, most approaches add significant overheads for non-skewed workloads—especially for high-speed interconnects, which do not allow to hide the extra computation. Often the inputs are materialized and scanned completely [15], [14], [33], [25]. Other approaches require detailed statistics [6], [34], which might be overly inaccurate or unavailable for intermediate results. Flow-Join instead detects heavy hitters during partitioning and does not rely on statistics.

DeWitt et al. [6] first introduced the idea to replicate tuples that join with heavy hitters for range partitioning and called it Subset-Replicate. Xu et al. [34] applied the idea to hash partitioning, calling it Partial Redistribution & Partial Duplication (PRPD). PRPD requires a list of skewed values based on collected statistics in contrast to Flow-Join, which detects heavy hitters at runtime. PRPD uses every hardware context as a separate parallel unit, i.e., there is no work stealing inside a single server. Skew effects are thus much higher as heavy hitter values are sent to a single parallel unit.

Eddy [31] and Flux [27] are dedicated operators for adaptive stream processing. The distributed Eddy routes tuples between operators—which are considered to reside on a server of their own—to address imbalances between operators. Flux is a modified exchange operator that creates many more partitions than servers to shift partitions from overloaded to underutilized servers. However, it cannot split a large partition that essentially consists of a single heavy hitter value and thus does not solve the scalability problem caused by skew.

The comparatively low bandwidth of standard network interconnects such as Gigabit Ethernet creates a bottleneck for distributed query processing. Consequently, recent research focused on the network cost: Neo-Join [25] computes an optimal assignment of partitions to servers that minimizes the *network duration*. It handles skew using Selective Broadcast on a partition level. However, it materializes and scans both inputs to generate histograms and has to solve a compute-intensive linear program. Track-Join [22] redistributes join keys in a dedicated track phase to decide on the join location for each join key value separately and by this achieves minimal *network traffic*—excluding the track phase. However, the track phase itself is a separate distributed join that is sensitive to skew and materializes both inputs.

Modern interconnects close the gap between network bandwidth and compute speed. This enables a cluster to outperform a single server and scale the query performance when servers are added to the cluster [2], [24]. Frey et al. [10] designed the Cyclo Join using RDMA over 10 Gigabit Ethernet for join processing within a ring topology. Goncalves and Kersten [11] extended MonetDB with a novel distributed query processing scheme based on continuously rotating data over a modern high-speed network with a ring topology. However, ring topologies, by design, use only a fraction of the available network bandwidth in a fully-connected network. Mühleisen et al. [20] use RDMA to utilize remote memory for temporary database files in MonetDB. Costea and Ionescu [5] extended Vectorwise, which originated from the MonetDB/X100 project [35], to a distributed system using MPI over InfiniBand. Barthels et al. [1] implemented a distributed radix join using RDMA over InfiniBand, providing a detailed analysis that includes experiments with skewed workloads. They measured a $3.3\times$ slow down for Zipf factor 1.2 on an 8-server cluster, which is in line with our results and further highlights the need for low-overhead skew handling.

Flow-Join detects heavy hitters using the SpaceSaving algorithm [19], which solves the approximate frequent items problem and was shown to outperform alternatives [4]. Roy et al. [26] designed a pre-filtering stage that speeds up SpaceSaving by a factor of 10 for skewed inputs. Teubner et al. [30] employed FPGAs to solve the frequent items problem in hardware, processing 110 million items per second. Both pre-filter and hardware acceleration can be applied to Flow-Join to reduce the overhead of skew detection even further.

Elseidy et al. [7] propose a mechanism to dynamically change the grid of the Symmetric Fragment Replicate redistribution scheme [28] according to running cardinality estimates. However, an additional random redistribution of the input tuples doubles the network traffic. They use a symmetric join algorithm [32] that materializes both inputs in memory and processes every tuple twice (once for build and once for probe).

VII. Concluding Remarks

The scalability of distributed joins is threatened by unexpected data characteristics: Skew can cause a severe load imbalance such that one server in the cluster has to process a much larger part of the input than its fair share—slowing down the entire query. Previous approaches often require expensive analysis phases, which slow down join processing for non-

skewed workloads. This is especially the case when high-speed interconnects such as InfiniBand are used, which are too fast to hide the extra computation. Other approaches depend on detailed statistics, which are often not available or overly inaccurate for intermediate results.

Flow-Join is a novel join algorithm that detects and adapts to heavy hitter skew alongside partitioning with minimal overhead and without relying on existing statistics. It uses approximate histograms to detect skew and adapts the redistribution scheme at runtime for the subset of the keys that were identified as heavy hitters. The combination of decoupled exchange operators connected via an RDMA-based communication multiplexer for distributed processing and a work-stealing-based approach for local processing enables Flow-Join to scale near-linearly with the number of servers in the cluster.

Our evaluation with Zipf-generated data sets as well as a real workload from a large commercial vendor has shown that Flow-Join performs as good as an optimal omniscient approach for skewed workloads and at the same time does not compromise join performance when skew is absent from the workload. The overhead of Flow-Join's adaptive skew handling mechanism is indeed small enough to process skewed and non-skewed inputs at the full network speed of InfiniBand $4\times$FDR at more than 6 GB/s per link, joining a skewed workload at 11.5 billion tuples/s with 32 servers—$6.8\times$ faster than a standard hash join. Flow-Join overlaps computation with the network communication so that the join finishes shortly after the last network transfer. Detecting and handling skew at runtime enables the pipelined execution of joins and thus avoids the materialization of the probe side, reducing main-memory consumption significantly.

Finally, we generalized Flow-Join beyond the common case of key/foreign-key equi joins. The generalized Flow-Join applies the Symmetric Fragment Replicate redistribution scheme for heavy hitters that occur in both inputs, reducing the query execution time by up to a factor of $(\sqrt{n}+1)/2$ for correlated skew in a cluster with n servers.

VIII. Acknowledgments

Wolf Rödiger is a recipient of the Oracle External Research Fellowship. This work has further been partially sponsored by the German Federal Ministry of Education and Research (BMBF) grant RTBI 01IS12057.

References

[1] C. Barthels, S. Loesing, D. Kossmann, and G. Alonso. Rack-scale in-memory join processing using RDMA. In *SIGMOD*, pages 1463–1475, 2015.

[2] C. Binnig, U. Çetintemel, A. Crotty, A. Galakatos, T. Kraska, E. Zamanian, and S. B. Zdonik. The end of slow networks: It's time for a redesign. *CoRR*, abs/1504.01048, 2015.

[3] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos. Robust and skew-resistant parallel joins in shared-nothing systems. In *CIKM*, pages 1399–1408, 2014.

[4] G. Cormode and M. Hadjieleftheriou. Methods for finding frequent items in data streams. *VLDB J.*, 19(1):3–20, 2010.

[5] A. Costea and A. Ionescu. Query optimization and execution in Vectorwise MPP. Master's thesis, Vrije Universiteit, Amsterdam, Netherlands, 2012.

[6] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, pages 27–40, 1992.

[7] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. *PVLDB*, 7(6):441–452, 2014.

[8] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *SIGMOD*, pages 169–180, 1978.

[9] P. W. Frey. *Zero-copy network communication*. PhD thesis, ETH Zürich, Zurich, Switzerland, 2010.

[10] P. W. Frey, R. Goncalves, M. Kersten, and J. Teubner. Spinning relations: high-speed networks for distributed join processing. In *DaMoN*, pages 27–33, 2009.

[11] R. Goncalves and M. Kersten. The Data Cyclotron query processing scheme. *TODS*, 36(4), 2011.

[12] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, pages 102–111, 1990.

[13] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. *SIGMOD Record*, 23(2):243–252, 1994.

[14] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *VLDB*, pages 525–535, 1991.

[15] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the Super Database Computer (SDC). In *VLDB*, pages 210–221, 1990.

[16] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *NGC*, 1(1):63–74, 1983.

[17] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[18] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *TKDE*, 14(4):709–730, 2002.

[19] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, pages 398–412, 2005.

[20] H. Mühleisen, R. Gonçalves, and M. Kersten. Peak performance: Remote memory revisited. In *DaMoN*, 2013.

[21] H. Plattner. The impact of in-memory databases on applications. Talk, July 7, 2014.

[22] O. Polychroniou, R. Sen, and K. A. Ross. Track join: Distributed joins with minimal network traffic. In *SIGMOD*, pages 1483–1494, 2014.

[23] S. Ray, B. Simion, A. D. Brown, and R. Johnson. Skew-resistant parallel in-memory spatial join. In *SSDBM*, 2014.

[24] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *PVLDB*, 9(4):228–239, 2015.

[25] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *ICDE*, pages 592–603, 2014.

[26] P. Roy, J. Teubner, and G. Alonso. Efficient frequent item counting in multi-core hardware. In *KDD*, pages 1451–1459, 2012.

[27] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.

[28] J. W. Stamos and H. C. Young. A symmetric fragment and replicate algorithm for distributed joins. *TPDS*, 4(12):1345–1354, 1993.

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

[30] J. Teubner, R. Müller, and G. Alonso. Frequent item computation on a chip. *TKDE*, 23(8):1169–1181, 2011.

[31] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.

[32] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS*, pages 68–77, 1991.

[33] J. L. Wolf, P. S. Yu, J. Turek, and D. M. Dias. A parallel hash join algorithm for managing data skew. *TPDS*, 4(12):1355–1371, 1993.

[34] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD*, pages 1043–1052, 2008.

[35] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical DBMS. In *ICDE*, pages 1349–1350, 2012.