SCALING DISTRIBUTED DATABASE JOINS BY DECOUPLING COMPUTATION AND COMMUNICATION

Abhirup Chakraborty

ACM Member

ABSTRACT

To process a large volume of data, modern data management systems use a collection of machines connected through a network. This paper proposes frameworks and algorithms for processing distributed joins—a compute- and communication-intensive workload in modern data-intensive systems. By exploiting multiple processing cores within the individual machines, we implement a system to process database joins that parallelizes computation within each node, pipelines the computation with communication, parallelizes the communication by allowing multiple simultaneous data transfers (send/receive). Our experimental results show that using only four threads per node the framework achieves a 3.5x gains in intra-node performance while compared with a single-threaded counterpart. Moreover, with the join processing workload the cluster-wide performance (and speedup) is observed to be dictated by the intra-node computational loads; this property brings a near-linear speedup with increasing nodes in the system, a feature much desired in modern large-scale data processing system.

KEYWORDS

Distributed joins; Multi-core; Database; Pipelining; Parallel processing;

1. INTRODUCTION

Analyzing massive datasets is essential for many real world applications from diverse domains; data analytics, distributed databases, video analytics, graph analytics, machine learning are a few examples of such application domains. Join operation is a basic and important one common to all these applications. Thus an efficient join processing algorithm will benefit all the application. Researchers have worked towards enhancing the join algorithms for a single multi-core processor [1, 2, 3, 4, 5, 6]. However, a single machine can not handle the loads for a join over the massive data from the emerging applications [7, 8, 9, 10].

A distributed join over a massive dataset requires shuffling a large volume of data among the processing nodes. Continual advancements in processor and interconnection network technologies result, respectively, in a larger number of processing cores per chip and a higher network bandwidth. Such improvements in both processing capacities within a socket and the network bandwidth bring the opportunity to improve the performance of compute- and communication-intensive applications.

In this paper, we use a multi-threaded framework to increase compute and communication efficiency while supporting a data-intensive application (i.e., a distributed join over partitioned data) within a shared-nothing system. Processing distributed joins requires shuffling of data across nodes, which needs synchronization across the nodes. A multicore node can shuffle data

across multiple nodes in parallel; however, to process a distributed join over a network of multicore machines, the nodes should orchestrate the computation and communication loads within each node. In this paper, we propose a framework to process distributed joins by decomposing the join processing loads within a node into a number of sub-tasks; and we exploit multi-threading to support synchronization-free computation by unbundling the join loads into computation and communication tasks and by scheduling the sub-tasks in orderly fashion. The key contributions of the work are as follows:

1. We decouple computation and communication with a process, and parallelize the compute and communication tasks using an event-based scheduling across a number of threads. Isolating computation from communication allows overlapping the tasks, thereby maximizing the performance. Also, the framework can maximize communication throughput across the nodes by allowing multiple concurrent data transfers (or sockets) across the processes.

2. We devise a mechanism to pipeline computation with simultaneous communicationin a node. By using a thread-local store, called HashTable Frame (HTF), we present mechanisms to reduce concurrency overheads (in receiver threads) while maintaining data received from remote nodes.

3. We develop methodologies to reduce concurrency overheads while accessing theshared data within a node. Using a small pool of memory (mini-buffer) within each compute threads, we develop a two-level method to maintain a global shared list of output tuples, denoted as the Result List. A compute thread merges the output results with the global Result List only when the local mini-buffer is full or when the thread is about to terminate.

4. We present the performance results to show the speedup and performance gain within each node and in the system as a whole.

The rest of the paper is organized as follows. Section 2 introduces the join processing workload considered in this paper. Section 3 outlines the multi-threaded framework to process joins forgoing any synchronization barriers across the nodes. Section 4 describes in details the techniques and algorithms to process joins within a shared-nothing system. Section 5 presents the experimental results. Section 6 covers the related work, and Section 7 concludes the paper and presents a number of open issues and directions for future work.

2. DISTRIBUTED JOINS

We consider processing the binary join $(R \bowtie S)$ between two relations R and S, where the relations are partitioned across a set of nodes connected through a high-bandwidth network. Each of the relations R and S consists of N disjoint partitions, and each node $i \in N$ stores one partition from each relation (i.e., R_i and S_i). Each node reads the partitions from the disk, forms hashtables for the partitions and stores the hashtables (with M buckets) in main memory (Figure 1). To join the two relations, we should shuffle the partitions across the nodes. Depending on the nature of the joins, we can use two different types of shuffling of the partitions.

In case of a non-equijoin, each node sends the partition of the outer relation (which happens to be the smaller relation) to all other nodes in the system. Hence, the communication pattern in an all-to-all broadcast of the partitions stored in the nodes. In case of an equijoin, we can use a hash distribution scheme that assigns a subset of the hash buckets ($m_i \in M$) to a node i. Now,

all the nodes in the system should send to node i only the buckets m_i assigned to that node. Hence, the communication pattern is an all-to-all personalized broadcast.

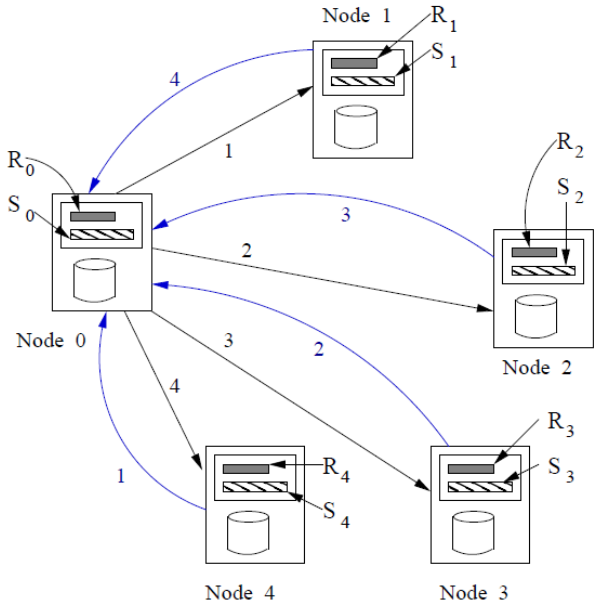

Figure 1: Distributed joins of two tables partitioned over a shared-nothing system

The shuffling of data by a sender node proceeds in a round of phases. Figure 1 shows the detailed approach in shuffling the data across the nodes. The nodes are logically arranged in a circular ring. In each phase, a node sends its data to a receiver node and receives the data from another sender node. A node chooses the receivers and the senders, respectively, in clockwise and counter-clockwise order. For example, in the first phase, node 0 sends its data to node 1, and receives data from node 4. In the second phase, the node 0 sends to node 2 and receives from node 3; in the third phase, the it sends to node 2 and receives from node 2; in the fourth phase (for node 0), 3 is the receiver and 0 the sender. For a system with n nodes, there are a total of $n-1$ phases of communication within each node.

Algorithm 1: DISTRIBUTEDJOIN(R_i, S_i, n)

Data: Partition R_i and S_i in Host H_i, a parameter n

Result: T_i contains the result of joins among all the partitions of the outer relation R and the local partition S_i of the inner relation S

begin

1 $T_i \longleftarrow R_i \ltimes S_i$

2 **for** $k \leftarrow 1$ *to* $n-1$ **do**

3 $r \longleftarrow (i+k)\%n$

4 $B_{snd} \longleftarrow \text{SELECT}_r(R_i, S_i)$

5 $\text{SEND}(j, B_{snd})$

6 $s \longleftarrow (i-k+n)\%n)$

7 $B_{rcv} \longleftarrow \text{RECEIVE}(s)$

8 $T_i \longleftarrow T_i \cup (B_{rcv} \ltimes S_i)$

9 Barrier()

Algorithm 1 shows the pseudo-code for the distributed join processing algorithm. The iterations in line 2 corresponds to the phases of communication. The Select method

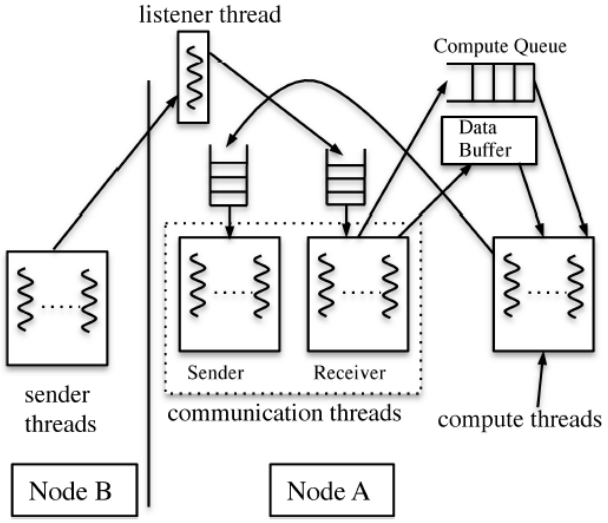

Figure 2: Organization of threads within a node

selects the content to send in a phase: in case of an equijoin (all-to-all broadcast), it picks the content of the buckets (from both R_i and S_i) assigned to the receiver r; and in case of a non-equijoin (hash-based distribution), the method pulls only the partition of the outer relation R_i.

3. MULTI-THREADED FRAMEWORK

In this section, we describe the multi-threaded framework to support data-intensive applications within a network of distributed, many-core machines. The framework provides fine-grained, parallel computations, and improves all-to-all data shuffling by supporting multiple, simultaneous data transfer channels across the distributed nodes. A node can initiate ad-hoc, asynchronous transfer of data without any pre-defined communication sequence. A node exchanges control information by passing metadata, and regulates the execution within the node based on the metadata received from the remote nodes. In short, the framework targets parallel computation within a node, efficient data transfer across the nodes, and any-to-any, asynchronous communication across the nodes. Each node in the system supports a number of threads that can be categorized into two types—computation and communication threads. Communication threads can be divided into three sub-types: listener thread, sender threads and receiver threads. These threads communicates control information among each other by using three queues: *Compute Queue* (Q_c), *Send Queue* (Q_s) and *Receive Queue* (Q_r). Each of the queues uses *Mutexes* and *Condition variables* to address the bounded-buffer problem [11]. Figure 2 shows the organization of the threads and queues within a node.

Compute threads within a node provide fine-grained parallelism within a node by executing multiple tasks simultaneously. These threads pulls tasks from a compute queue. These threads initiates communication with the remote nodes by passing a *send event* to the send threads via the send queue (Q_s).

A *listener thread* within a node listens to a predefined server port (*sport*) in the node and allows the remote nodes to setup ad-hoc communication channels with the node. Any node can initiate connection with a remote node by using the *sport* and the IP address of the remote node. Upon successfully receiving a connection from a remote node, the listener thread assigns a socket descriptor (*sockD*) to the channel, and passes the socket information to a receiver thread within the node by pushing a new record to the *Receive Queue* (Q_r).

A *sender thread* receives *send events/tasks* from the send queue (Q_s), and initiates connection with the remote node by using the IP address and *sport* of the remote node. The node completes the operation specified in the send event by passing control information and metadata to the remote node. We don't persist the sockets created by the sender threads while serving a send event. The participating nodes destroys the socket once the event has been processed. In a data-intensive application, each send event requires a significant volume of data transfer across the participating nodes, minimizing the relative overhead in setting up sockets. For applications requiring frequent transfers to short messages, we need to persist a few sockets to provide a fixed communication topology to exchange the short messages; such an issue is orthogonal to the problem studied in this paper, and is a topic of future work.

A *receiver thread* pulls *receive events/tasks* from the receive queue (Q_r). Using the socket descriptor (sockD) created by the listener thread, a receiver thread receives the data and the tasks/events from the remote node. It stores the control events/tasks the compute queue (Q_c), and the received data in a the data buffer as HTFs. A *receiver thread* is blocked when the shared memory pool in the data buffer is empty. An HTF is an skeleton of the remote hashtable, and it does not fully materialize the remote hashtable, as the buckets are continually purged by the compute tasks, that are pipelined with the data reception (by the receiver thread).

4. DISTRIBUTED JOIN PROCESSING

In this section, we describe the mechanism to process joins over a shared nothing system using the multi-threaded framework, as given in Section . The system uses three phases to process the join: input data shuffling, in-node join computation, and result collection at the sink. Threads within the nodes pass control messages to notify various tasks and to signal the changes of the phases. Using the control messages, system computes the join without any barrier synchronization across the networked nodes. This section begins with a description of the join processing mechanism within a node. We then describe the state transition diagram showing the control messages, and present the algorithms deployed in each of the sender, receiver and the compute threads.

4.1. In-Node computation

We describe the mechanism used within a node to process the join between two relations. We load the input data from disk and store the local partition in memory as hash tables. Multiple *sender threads* can simultaneously read data from the local partition and send the data to remote nodes; on the other hand, multiple *receiver threads* receives data from the remote nodes and store the data locally as separate hash-tables, referred to as HashTable Frames (HTF), that share a common memory pool in data buffer. If the data is already partitioned across nodes using a hash distribution scheme, then the number of buckets in a HTF would be equal to the number of buckets pinned to the node; if the table data within nodes are not already hash-partitioned (i.e., every node can contain data tuples that might belong to any buckets of the hash function), then the data will be partitioned using a hash table and the total buckets in a HTF would be the same as that in the hash table within a node. Using a separate HTF for each sender node facilitates the computation of lineage of the output result without modifying the incoming data to tag each record with the source node of the data.
Figure 3 shows the system details of processing within a node. After receiving data from a remote node, the receiving node adds, for each incoming bucket, a compute record

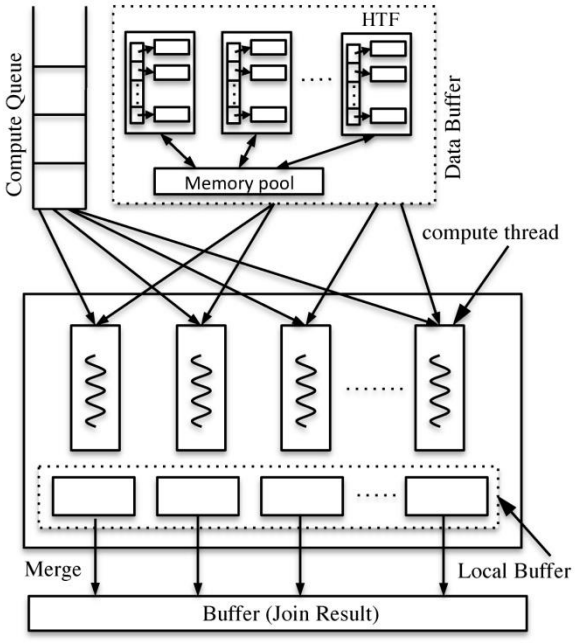

Figure 3: Join processing within a node

r_c = h*type, bI, htfI, table*Ii in the compute queue. The attributes in the compute record describes the compute task: *Bucket index* (bI) denotes the bucket (in the hashtable frame) to be joined to the other relation(s), *hashtable frame index* (htfI) gives the hashtable frame of the joining bucket, and table index (tableI) denotes the global input relation/table (which the hashtable denoted by *htfI* is a part of). A compute thread pulls the records from the compute queue and processes the tasks. If the task is of type join, the handler routine (subsection) joins the buckets with the respective bucket from the joining relation. Each thread merges the output tuples with the *result buffer*. Directly accessing the result buffer for each result tuple creates contention among the threads. We provide local buffer within each thread to reduce the thread contention. Each thread stores the result in the local buffer, and merges the local buffer with global result buffer when the local buffer is full or has at least one block. Such a merge happens at the block level and the whole block from the local buffer is appended to the result buffer, which minimizes the contention overhead. Each thread merges the partially-filled block, if any, within the local buffer after joining all the incoming buckets.

4.2. State Transitions

Threads within each node in the system pass control/event records to convey communication and computation tasks. Using the queues and the control records, the threads change their computation and communication states and keep track of the phases of the join computation. Such an event-based mechanism avoids any global barrier synchronization across the participating nodes. Figure 4 presents the event diagram depicting the transition of states within each of the threads.

A *shuffle scheduler* thread generates the communication schedules and tasks within each of the node (**Step 1**). The scheduler threads could be the main thread or any pre-assigned compute thread (i.e., compute thread 0). For a system with N nodes, the communication schedule for a node i ($0 \leq i < N$) consists of send records h partitionReady,IP_d, $sport_d$ i for each of the $N-1$ destination nodes $d = (i+1)\%N,...,(i-1+N)\%N$. Here, IP_d

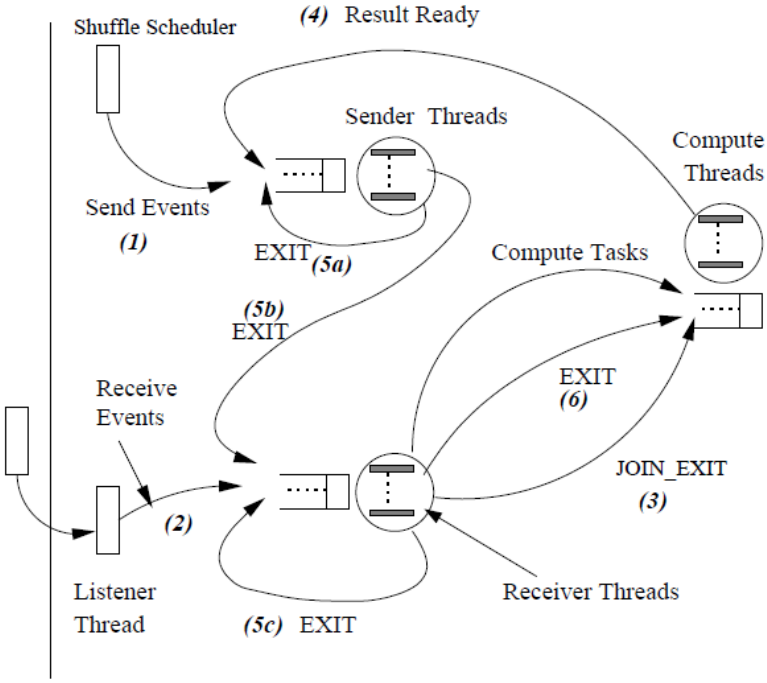

Figure 4: Event diagram showing the flow of events across the threads within a node

and $sport_d$ are, respectively, the IP address and server port of the destination node *d*, and the partitionReady indicates the *type* of the event. Here, we denote a compute, send or receive event by the type of the record. Also, we use the term event, task and records interchangeably where the context is explicit. The sender threads pulls the send tasks from the send queue (Q_s) and opens socket with the remote destination node.

The listener thread receives the connections from the remote nodes and generate receive events hpartitionReady,*sockD* i in the receive queue, Q_r (**Step 2**). The receiver threads maintains a counter to keep track of the number of nodes that have transferred the partition to the node. Once the node has received data from all other nodes in the system, the receive thread produces a compute event hjoinExit, , , i in the compute queue Q_c (**Step 3**). The receive thread generates the joinExit events only after all the data from the remote nodes have been received and the respective compute records have been added to the Q_c; hence, there will not be any compute event of type join, that comes after the joinExit event. When a compute thread gets a compute event of type joinExit, the primary compute thread (i.e., compute thread 0) produces a send event of *type* resultReady to signal the sender threads to transfer the result to the sink node (**Step 4**). After sending the result to the sink node, a sender thread produces a send event hexit, , i for the Q_s and a receive event hexit, _i for the Q_r (**Step 5a** and **5b**). These exit events terminates the sender and the receiver threads. The primary receive thread generates a compute event hexit, , , i to indicate that the communication tasks (input data shuffling, result transfer) have been completed, and that it is now safe to close the compute threads within the sink node (**Step 6**). Note that the compute threads within the non-sink nodes are closed after receiving the joinExit event.

4.3. Algorithms

This section describes the procedures used by the compute, sender and receiver threads. As mentioned earlier, threads of the same type shares a queue that stores the events received from

different threads in the system. Each thread fetches records from the respective queue and processes the events in parallel with the other threads.

Algorithm 2: COMPUTEHANDLER()

```
1  flag ⟵ True
2  while flag do
3      r_c ⟸= Q_c
4      switch r_c.type do
5          case JOIN:
6              JOINBUCKET(r_c.bI, r_c.htfI, r_c.tableI)
7              FREE(r_c.bI, r_c.htfI)
8              if processed all buckets of r_c.htfI then
9                  FREE(r_c.htfI)
10         case JOINEXIT:
11             Res_i ⟵ Res_i ∪ LB[threadID]
12             BARRIER()
13             if threadID = 0 then
14                 Q_s ⟸= (RESULTREADY, SINK)
15                 if the node is not a SINK then
16                     flag = FALSE
               else
17                 flag = FALSE
18         case EXIT:
19             PRINTRESULT(Res)
20             flag = FALSE
```

4.3.1. Compute Handler

Each of the compute threads uses the Algorithm 2 as the handler routine. If the event type is join, Line 3 fetches a compute record from the compute queue (Q_c). Line 6 joins the bucket with the relevant bucket(s) from the other joining relation(s). Line 7 releases the memory in the relevant bucket within the hashtable frame, and line 9 frees up the memory occupied by the hashtable frame when all the buckets within the hashtable frame are processed (i.e., joined with buckets from the joining relations).

If the event r_c fetched from the compute queue (Q_c) is of type joinExit, the handler algorithm merges the result in local buffer (LB) with the global result Res_i in the node (Line 11). The compute threads wait for a local synchronization barrier in line 12. Upon the synchronization, the primary compute thread (i.e., thread 0) signals a event hresultReady,sinki to the send queue (line 14, indicating that the local join result within the node is available to be sent to the sink node. The primary compute thread within a non-sink node is terminated in line 16, whereas line 17 terminates the secondary compute threads (i.e, non-zero threadIDs) in all nodes (sink and non-sink). The primary compute thread within the sink node is terminated in line 20, when it processes the exit event received from the primary receive thread. Line 19 prints the output in the output device. Note that only the primary compute thread in the sink node receives the exit event from the primary receive thread.

4.3.2. Send Handler

The handler procedure for the send threads is given in Algorithm 3. The main loop in the procedure fetches events (line 3 from the send queue and processes the events. Line 6 han-

Algorithm 3: SENDHANDLER()

```
1  flag ⟵ True
2  while flag do
3      r_s ⟸ Q_s
4      switch r_s.type do
5          case RESULTREADY or PARTITIONREADY:
6              HANDLEOUTBOUNDREQ(r_s)
7              if r_s.type = RESULTREADY then          /* Result has been sent */
8                  Q_s ⟸ (EXIT, _)
9                  if the node is not a SINK then
10                     Q_r ⟸ (EXIT, _)
11         case EXIT:
12             Q_s ⟸ r_s
13             flag = FALSE
```

dles the resultReady and partitionReady events. This method (HandleInBoundReq) sends the input partition or the output result to the destination node using the socket given in the event record r_s. We develop simple mechanisms and protocols to transfer or shuffle various data structures (e.g, hash-tables for input partition, result list for output results) over TCP sockets; these protocols properly serialize (or de-serialize) the data structures at the sending (or receiving) ends. The HandleInBoundReq method handles the resultReady event in a non-sink node by sending the local results to the sink node, whereas in a sink-node the method simply ignores the resultReady event. If the fetched record r_s in a send thread is a resultReady event, the thread initiates an exit event (in line 8) to close all the send threads. Note that the send queue might have a few pending events (e.g., partitionReady), which must be processed before terminating the send threads. Line 12–13 handle the exit event by signaling the termination event to other threads and by halting the loop by setting the flag to false.

4.4. Receive Handler

The handler routine for the receive threads (Algorithm 4) processes the receive events within the receive queue (Q_r). The helper method handleInboundReq handles the two data events:partitionReady and resultReady). The method returns two boolean flags (shuffleFlag and resFlag) indicating if the input data has been received from all source nodes (i.e., all partitionReady events are processed) or the result has been received (at the sink). The method uses an atomic count to trace the number of partitionReady events already processed. The shuffleFlag is set to true when the counter value equals the number of (sender) nodes in the system. Upon receiving the data from all source nodes (i.e., shuffleFlag=true), the receive handler sends joinExit events for all compute threads (Line 10), which signals the completion of the join phase within the node. Note that the resultReady event appears in receive queue only within the sink node.

The receive handler can close the receiver threads when both the result transmission and the input data shuffling are complete (line 14). Line 18 terminates the receive thread and line 19 signals other receive threads to terminate. As noted in section , the primary compute thread (in the sink node) remains alive even after the completion of the join phase (i.e., after joinExit completes). The primary receive handler thread (in the sink)

Algorithm 4: RECVHANDLER()

```
1  flag ⟵ True
2  while flag do
3      r_r ⟸ Q_r
4      switch r_s.type do
5          case RESULTREADY or PARTITIONREADY:
6              (shuffleFlag, resFlag)⟵ HANDLEINBOUNDREQ(r_r.socket)
7              if shuffleFlag then                          /* data is shuffled */
8                  gShuffleFlag = TRUE
9                  for i = 1 to n_ct do
10                     Q_c ⟸ (JOINEXIT, _)
11             if resFlag then                              /* result is received */
12                 gResFlag = TRUE
13             if gResFlag and gShuffleFlag then
14                 Q_r ⟸ (EXIT, _)
15         case EXIT:
16             if node is the SINK and threadID = 0 then
17                 Q_c ⟸ (EXIT, _)
18             flag = FALSE
19             Q_r ⟸ r_r
```

Table 1: Default parameters

Parameter	Defaults	Description
p	8k	page size
R_i	400000	partition size (in tuples) of relation R
D	800000	Domain of join attribute
N_B	1200	Total buckets for the hash table
Stup	128	Size of the tuples in the join relation
N	10	total nodes in the system
n_c	2	compute threads
ncom	2	communication (send and receive) threads
nlis	1	Listener thread

sends the exit event to the compute queue, in line 17.

5. EXPERIMENTS

In this section, we present the experimental data on the performance of the join processing algorithm within the multi-threaded framework implemented in a shared-nothing system. All the experiments are carried out in a 5-node cluster of virtual machines; each node (or VM) within a cluster has a dual-core processor, 1 GB of RAM, runs 64-bit Red Hat Enterprise Linux. The physical machines are connected via a 1 Gbps Ethernet network. We implement the system in C++. We show the performance of the join algorithm within the framework by collecting a few

metrics: *join span, intra-node gains,* and *speedup* due to parallelism in the shared nothing system. The *join span* is the total time to complete the join processing phase in the system; this time is recorded at the sink node when it received the notification of join-phase completion from all the nodes in the system. The *intra-node gains* denotes the savings within a node due to intra-node parallelism in processing and communication (send and receive) loads. Such gains are derived from the parallelization of both communication loads (e.g., a *send* task overlaps with a *receive* one) and computation loads as observed within a node. Formally, we can define the intra-node gains within a node as,

$$Intra\text{-}node\ gain = \frac{total\ loads\ within\ the\ node}{join\ span\ within\ the\ node}$$

Here, *total loads* indicates both the communication and computation times as observed during the join phase within the node. The metric *speedup* derived from *N* nodes in the system is given as,

$$Speedup\ = \frac{Join\ span\ with\ a\ single\ node}{Join\ span\ with\ N\ nodes}$$

The metric *intra-node gains* indicates the effectiveness of a node (within the networked system) in dealing with processing and communication overheads, whereas the *speedup* of a system indicates its join processing time while compared with the single node execution of the equivalent load.

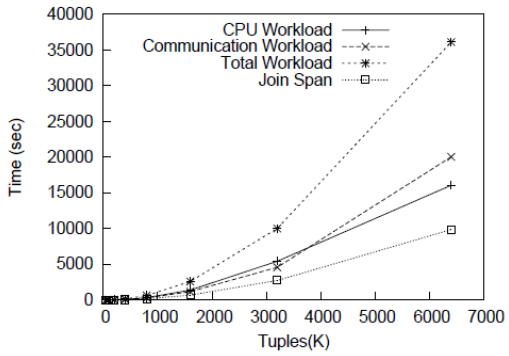

Figure 5: Computation loads, communication loads and join spans with varying table sizes

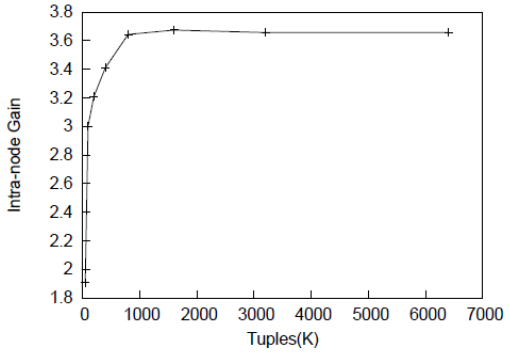

Figure6: Intra-node gains with varying table sizes

We do not show the time or delay for the result transfer phase. A join operator is usually followed by an aggregation and sampling operation; therefore, in a distributed database setting, each worker node locally stores the join results and only sends to the sink node either the aggregation result or a sample of the join output. Moreover, our framework can easily reduce the time to collect results from the worker nodes by applying multiple communication threads at the sink node, which can be readily realized. In the experiments, we focus on the system performance on handling the join processing and the associated all-to-all shuffling loads across the nodes.

As input to the join algorithms, we use synthetic relations generated using a the PQRS algorithm [12]. The PQRS algorithm, that is used to capture spatio-temporal locality in real traffic data (e.g., block access patterns in disk of file system) can be applied to generate the join attribute values (from the domain of the attribute) for the tuples in the join relation. The default values for various parameters in the system are given in table 1

5.1. Varying loads or table size

Figure 5 shows the computation and communication loads and the join span within a node in the system. As shown in the figure, the join span is almost half of both the communication loads and computation loads taken separately. This implies that not only the computation loads overlap with the communication loads, but also the two types of communication loads (i.e., send and receive) are parallelized with each other. Such a parallelism in computation and communication loads imparts a significant performance gain within a node, and as the figure shows the join span value is significantly lower than the total workloads observed within the node. We note that communication overhead play no role on the join span, which is almost dictated by the computational (cpu) loads within a node in the cluster, i.e., join span is nearly half (we have used two compute threads) the cpu loads within a node. Figure 6 shows the performance gain within a node, which stays around 3.6 for a partition size above 400K (tuples) within the node. The gain is low for a smaller partition size due to two factors. First, the overhead due to connection setup and wait time (when no receive thread is available at the receiver end, thus blocking the sender) is significant while compared to actual data transfer time for a low partition size. Second, the computation threads have low amortization opportunity while scanning memory blocks during bucket joins. Using just 2 processing threads and 2 communication threads (one sender and one receiver), each node attains a gain of 3.6 given given a substantial load. As we increase the load, the gain saturates around 3.6.

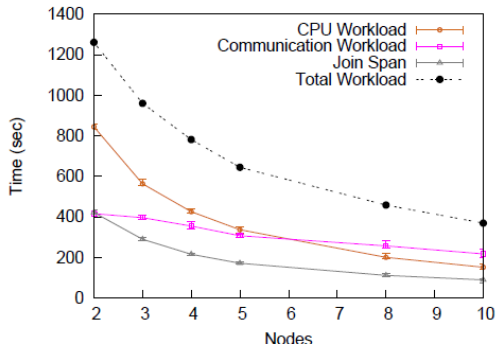

Figure 7: Computation loads, communication loads and the join span varying nodes

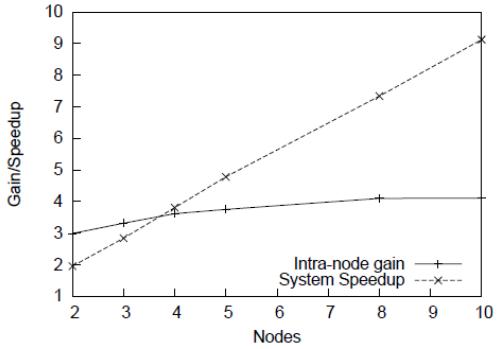

Figure 8: Intra-node gains and system speedup with varying nodes

5.2. Varying Nodes

We study the performance implication of scaling out the number of nodes in the system. We consider a fixed tuple size of 1.6 millions, and equally partition the tuples across the

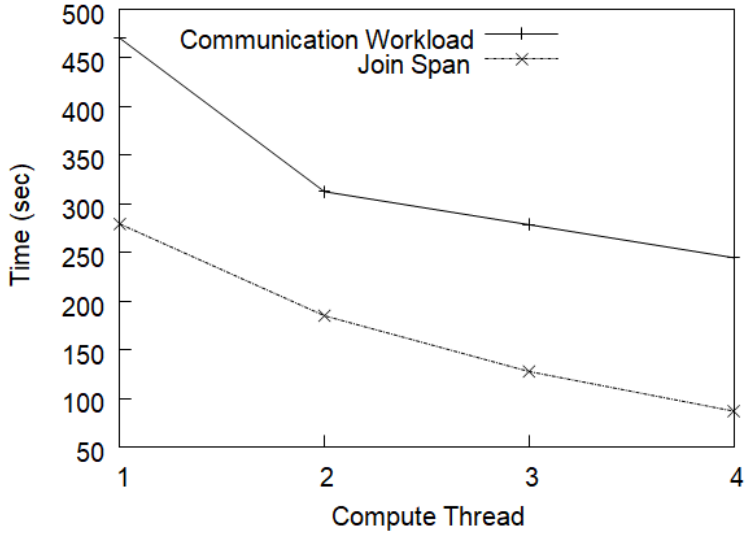

Figure 9: join spans and communication workload with varying compute thread

nodes in the system(e.g., the partition size (within a node) for a 2-node system is 0.8 million). Figure 7 shows the computation load, communication load and join span with varying nodes in the shared-nothing system. The computation load in a node decreases linearly as we scale-out the system. As we increase the number of nodes, data is split across higher number of partitions; For a bucket of the hash table within a node, the node should handle the multiple buckets of data received from the remote nodes. Such fragmentation of buckets leads to random memory accesses within the nodes. The process of reordering the join tasks does not eliminate such random accesses, because the buckets from the remote nodes arrive at a different point in time. Due to this phenomenon, the computation loads decrease more sharply as we add additional nodes to a system with a lower number of nodes.

Contrary to the characteristics of a traditional system, the communication load decreases as we scale out the system. This is due to increased parallelism of data transfer within and across the nodes; A node can receive and send data simultaneously, which in turn unblocks the remote the nodes waiting for sending data to the node, increasing the concurrent data transfer across the nodes in the system. Also, as we increase the number of nodes, the total volume of data that crosses the inter-node link decreases. For example, if the size of the probing relation is $|R|$ and the the total nodes in the system is n, the partition size within a node is $\frac{|R|}{n}$ (the tuples in the relation R is splitted equally across the nodes). Now, the total data volume that a node sends to the other ($n-1$) nodes is given as $S_n = \frac{|R|}{n}(n-1) = |R|(1-\frac{1}{n})$.So, the value of S_n decreases as we increase the n, the total nodes in the system. Due to these phenomena, the communication time (or workload) decreases slightly as scale-out the system.

Figure 8 shows the intra-node gains and system speedup with varying nodes in the system. The intra-node gain increase slightly as we scale out the system. Such an increase in the gain is due to fact that the communication workload changes only a little, whereas both the computation workload and join span decrease linearly. As shown in the figure, the speedup of the system increases linearly with the increase in the number of nodes. Such a linear increase in speedup is due to the elimination of the synchronization barriers, which renders the join span of sink node (i.e., system-wide join span) almost equal to the join span an arbitrary node (evident from the narrow error bar in Figure 7).

5.3. Varying Compute Threads

Figure 9 shows the performance metrics with varying compute threads. With the increase in compute threads, the join span decreases initially, but it increases later on due to overheads in the form of context switches among the threads. The significant reduction in the communication overhead on the left (when the compute threads is changed from 1 to 2) is due to the reduction in blocking (or wait time) within both the send and the receive threads during data transfer. A send thread is blocked when the remote node is busy in receiving from another node, and a receive thread is blocked (for memory) when the memory pool used by the HTFs is exhausted; the receive thread is unblocked when the compute threads release the memory within a bucket after the join operation. So, increasing the compute threads reduces the blocking time for both the send and receive threads.

Supporting data-intensive workloads in a distributed system is a topic of active research. There are a number of systems to support data-intensive applications in a distributed system. Researchers also have developed a number of systems to support various communication intensive algorithms from different domains, e.g., graph processing, database joins, sorting, array processing, etc.

MapReduce and Dryad are two popular distributed data-processing systems to support data-intensive applications[13, 14]. These systems are suitable for data-parallel applications and do not fare well for communication-intensive applications with complex communication patterns (all-to-all, broadcast), stateful operations, or both. Such limitations led to the development of a few domain-specific systems, e.g., Pregel [15] and graphLab [16] for processing graph, Spark [17] for improving iterative performance by supporting inmemory caching and for providing fault tolerance by quickly generating missing or lost data from its lineage.

Using modern multi-core processors, researchers have developed frameworks to support a few communication-intensive algorithms within a shared-nothing system. CloudRamSort [18] and proposes a distributed sorting mechanism in a shared-nothing system by exploiting multi-core processors and SIMD (single instruction multiple data) units within the individual nodes in the system. The framework uses multiple compute threads to process the data, reserves a single thread to communicate data across the nodes using MPI. It divides both the communication and the computation tasks into several stages, and overlaps computation and communication by pipelining the in-node computation tasks (e.g., partitioning, local sorting, merge, payload rearrangement) with intra-node communication tasks (e.g., key transfer, payload transfer). Satish et al. [19] proposes a distributed graph traversal mechanism in a similar shared-nothing system using MPI. Similar to [18], the paper uses a single communication thread (for MPI) and multiple computation threads, and overlaps communication with in-node computations. To reduce communication overheads, it compresses the node sets before sending through MPI. MPI precludes many desirable features like processing without global barrier synchronization, isolating local failures in computation and communication, supporting multiple simultaneous communication channels (i.e., multiple sockets in multiple threads within a process) per node to parallelize the data transfer within a node.

Presto [20] is a distributed framework to process sparse-matrix operation using an arraybased language R [21]. Unlike MapReduce and Dryad, Presto supports iterative and stateful computations using point-point communications across the nodes in the cluster.

The framework scales computations over large datasets by sharing data across the processes within a node (using a shared-memory abstraction within the multi-core machines) and by

dynamically repartitioning data across the processes to balance the loads (i.e., execution times for the tasks).

A few computational frameworks—for example, Condor [22] and WorkQueue [23]—support data-intensive, scientific applications over wide-area computational grid or cluster, using a star (master-worker) or DAG (directed-acyclic graph) topology of communication graph, where all inter-node transfers are supported via the master node; in such a framework, a master (or an intermediate node) with a moderate fan-out stalls the workers (or children) while transferring data to/from the worker. Also, these frameworks does not support parallel communication links in a node; therefore, these frameworks are not suitable for communication-intensive applications with a complex communication pattern (e.g., all-toall or broadcast).

[24] reduces communication cost in parallel query processing by separating communication and computation within a query operator. The work adds new operators (e.g., *pair*, *merge operator*) in the logical query plan, and optimize the plan by pushing up computation or pushing down communication in the operator tree.The computation push-up and communication push-down reduce the intermediate result by processing over a larger input data from the downstream operators. Manciti et al. [25] proposes a heuristic-based algorithm to optimize joins over a large number of input tables. Contrary to query processing with multiple operators, our work optimizes the execution of a single join operator over massive input tables.

A few research works propose join algorithms for GPU-based systems. Reference [8] devises join algorithms using RDMA in a multi-GPU cluster. Reference [9] uses high-bandwidth NVLink to cope with limited bandwidth between main memory and GPUs, and presents algorithms to process joins in GPUs by accessing the join states from main memory. A few research works consider limited bandwidth between GPUs and CPU core (PCI-E bus) and presents join algorithms for a single server node with multiple GPUs [26, 27]. MG-Join [28] uses a multi-hop routing mechanism to optimize hash-join algorithms for multi-GPUs in a single machine.

Researchers have done significant work on processing database joins in a multi-core machine. *No-partition joins* [2] parallelizes the canonical hash join in a multi-core machine without partitioning the data. Teubner et al. [1] and Jha et al. [29] study the performance implications of numerous hardware parameters (e.g., TLB size, SIMD width, core size, relation sizes, etc.) within a multi-core machine and show the need for hardware-aware optimizations for the join algorithms. Blanas et al. [30] studies the memory footprints consumed by various hash- and sort-based join algorithms in a multi-core machine. Leis et al. [31] modifies the parallelization framework in volcano [32] by removing the exchange operator the query plan and instead using a NUMA-aware task dispatcher or scheduler; the dispatcher forms tasks by segmenting input data, and sends a task to an available thread that processes input data against a common query plan. Barber et al. [33] uses a memory-efficient data structure called Concise Hash Table (CHT) to process joins; the algorithm minimizes random memory accesses during probes and avoids partitioning the outer (or probe) table. Barthels et al. [7] and Frey et al. [34] implement parallel join algorithms over distributed data using network hardware with Remote Direct Memory Access (RDMA) within the nodes [34]. Contrary to above work, our approach supports concurrent communication to or from a node using multiple TCP sockets using any available underlying network hardware. At the same time, out approach exploits the computational resources within a node to parallelize the processing tasks.

Addressing the issues in skewed workloads, researchers have proposed numerous approaches to balance the loads while joining relations with skewed join attribute values (e.g., [35, 36, 37, 38, 39, 40, 41, 10]. In this paper, we consider the communication and computation efficiency and synchronization overheads while processing distributed joins. Handling skew is an orthogonal

issue that can be easily tacked by integrating any skew handling algorithm with the data-shuffling module (in a system node) of the proposed framework.

Increasing the degree of parallelism in a large-scale data management system imparts adverse effects on the performance and the speedup, due to increase in both the volume of shuffled data and the overhead due to synchronization barriers. Therefore, as we scale out such a system, maintaining a near-linear speedup is a challenging issue. Considering the issue of processing distributed joins, we have implemented a framework that reduces the network overhead, increase the intra-node performance and achieves a linear speed as we scale out the system. We have decomposed the join operation into a number of computeand communication tasks and devised a methodology to marshal the tasks among the threads using a state-transition mechanism within the threads. Each thread processes the (compute or communication) tasks and coordinates with other threads by generating events to signal a state change. Such a mechanism increases intra-node performance and precludes the costly synchronization barriers among the nodes, and brings the opportunity of parallelizing the data transfer at a finer granularity (i.e., sending to multiple destinations, while receiving from multiple sources). We implemented the framework in a shared-nothing system and observed around 3.5x reduction in intra-node join spans compared to a single-threaded counterpart (with equal number of nodes). More importantly, the framework achieves a linear speedup with an increase in the degree of parallelism in the shared-nothing system.

The paper opens up a few avenues for future work. First, this paper considers a flat topology for the nodes in the distributed systems; a join methodology with hierarchical topology of the sytsem is necessary to scale the system. Second, the notion of separation of computation and communication can be applied to other areas—for example, high performance computing—that processes high throughtput jobs or tasks—that access a large volume of data—in a large cluster of nodes. Third, addressing fault tolerance and recovery a for long-running join operator is an interesting problem; rather than restarting the join operation upon a failure, the system should resume the join processing using the save-points or replication or both.

REFERENCES

[1] J. Teubner, G. Alonso, C. Balkesen, and M. T. Ozsu, "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in *Proc. of the 2013 IEEE Int. Conf.on Data Engineering (ICDE 2013)*, Washington, DC, USA, 2013, pp. 362–373.

[2] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core cpus," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, New York, NY, USA, 2011, pp. 37–48.

[3] M. Bandle, J. Giceva, and T. Neumann, "To partition, or not to partition, that is the join question in a real system," in *Proc. of the 2021 Int. Conf. on Management of Data*, ser. SIGMOD '21. Association for Computing Machinery, 2021, pp. 168–180.

[4] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1378–1389, Aug. 2009.

[5] P. Garcia and H. F. Korth, "Pipelined hash-join on multithreaded architectures," in *Proc. 3rd Int. Workshop on Data Management on New Hardware*, ser. DaMoN '07. New York, NY, USA: ACM, 2007, pp. 1:1–1:8.

[6] ——, "Database hash-join algorithms on multithreaded computer architectures," in *Proc. 3rd Conf. on Computing Frontiers*, ser. CF '06, 2006, pp. 241–252.

[7] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann, "Rack-scale in-memory join processing using rdma," in *Proc. of the 2015 Int. Conf. on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 1463–1475.

[8] C. Guo, H. Chen, F. Zhang, and C. Li, "Distributed join algorithms on multi-cpu clusters with gpudirect rdma," in *Proc. of the 48th Int. Conf. on Parallel Processing*, ser. ICPP '19. New York, NY, USA: Association for Computing Machinery, 2019.

[9] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl, "Triton join: Efficiently scaling to a large join state on gpus with fast interconnects," in *Int. Conf. on Management of Data*, Z. Ives, A. Bonifati, and A. E. Abbadi, Eds. ACM, 2022, pp. 1017–1032.

[10] A. Metwally, "Scaling equi-joins," in *Proc. of the 2022 Int. Conf. on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 2163–2176.

[11] A. Silberschatz, P. Galvin, and G. Gagne, *Operating Systems Concepts*. John Wiley and Sons, Inc., Eighth Edition, 2009.

[12] M. Wang, A. Ailamaki, and C. Faloutsos, "Capturing the spatio-temporal behavior of real traffic data," in *IFIP Intl. Symp. on Computer Performance Modeling, Measurement and Evaluation*, Rome, Italy, September 2002.

[13] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. 2nd ACM SIGOPS/EuroSys European Conf. on Computer Systems*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 59–72.

[15] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. Int. Conf. on Management of data (SIGMOD)*, New York, NY, USA, 2010, pp. 135–146.

[16] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[17] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. of the 9th USENIX Conf. on Networked Systems Design and Implementation (NSDI)*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012.

[18] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani, "Cloudramsort: Fast and efficient large-scale distributed ram sort on shared-nothing cluster," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, New York, NY, USA, 2012, pp. 841–850.

[19] N. Satish, C. Kim, J. Chhugani, and P. Dubey, "Large-scale energy-efficient graph traversal: A path to efficient data-intensive supercomputing," in *Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, Los Alamitos, CA, USA, 2012, pp. 14:1–14:11.

[20] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: Distributed machine learning and graph processing with sparse matrices," in *Proc. of the 8th ACM European Conf. on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 197–210.

[21] *The R project for statistical computing*, http://www.r-project.org.

[22] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The condor experience: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 2-4, pp. 323–356, Feb. 2005.

[23] M. Albrecht, D. Rajan, and D. Thain, "Making work queue cluster-friendly for data intensive scientific applications," in *Proc. Int. Conf. on Cluster Computing*, September 2013, pp. 1–8.

[24] H. Zhang, J. X. Yu, Y. Zhang, and K. Zhao, "Parallel query processing: To separate communication from computation," in *Proc. Int. Conf. on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1447–1461.

[25] R. Mancini, S. Karthik, B. Chandra, V. Mageirakos, and A. Ailamaki, "Efficient massively parallel join optimization for large queries," in *Int. Conf. on Management of Data*, Z. Ives, A. Bonifati, and A. E. Abbadi, Eds. ACM, 2022, pp. 122–135.

[26] R. Rui, H. Li, and Y.-C. Tu, "Efficient join algorithms for large database tables in a multi-gpu environment," *Proc. VLDB Endow.*, vol. 14, no. 4, pp. 708–720, feb 2021.

[27] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, "Hardware-conscious hash-joins on gpus," in *IEEE 35th Int. Conf. on Data Engineering (ICDE)*, 2019, pp. 698–709.

[28] J. Paul, S. Lu, B. He, and C. T. Lau, "Mg-join: A scalable join for massively parallel multi-gpu architectures," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1413–1425.

[29] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh, "Improving main memory hash joins on intel xeon phi processors: An experimental approach," *Proc. VLDB Endow.*, vol. 8, no. 6, pp. 642–653, Feb. 2015.

[30] S. Blanas and J. M. Patel, "Memory footprint matters: Efficient equi-join algorithms for main memory data processing," in *Proc. 4th Annual Symposium on Cloud Computing*, ser. SOCC '13, New York, NY, USA, 2013, pp. 19:1–19:16.

[31] V. Leis, P. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: A numaaware query evaluation framework for the many-core age," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, New York, NY, USA, 2014, pp. 743–754.

[32] G. Graefe, "Encapsulation of parallelism in the volcano query processing system," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, New York, NY, USA, 1990, pp. 102–111.

[33] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe, "Memory-efficient hash joins," *Proc. VLDB Endow.*, vol. 8, no. 4, pp. 353–364, Dec. 2014.

[34] P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner, "A spinning join that does not get dizzy," in *2010 Int. Conf. on Distributed Computing Systems, ICDCS 2010, Genova, Italy, June 21-25, 2010*, 2010, pp. 283–292.

[35] Y. Xu, P. Kostamaa, X. Zhou, and L. J. Chen, "Handling data skew in parallel joins in shared-nothing systems," 2008.

[36] K. A. Hua and C. Lee, "Handling data skew in multiprocessor database computers using partition tuning," in *Proc. 17th Int'l Conf. on Very Large Data Bases*, ser. VLDB '91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 525–535.

[37] N. Bruno, Y. Kwon, and M.-C. Wu, "Advanced join strategies for large-scale distributed computation," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1484–1495, Aug. 2014.

[38] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Robust and skewresistant parallel joins in shared-nothing systems," in *Proc. of the 23rd ACM Int. Conf. on Information and Knowledge Management*, ser. CIKM '14. New York, NY, USA: ACM, 2014, pp. 1399–1408.

[39] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," in *Proc. Intl. Conf. on Very Large Databases (VLDB)*, 1992, pp. 27–40.

[40] A. Vitorovic, M. Elseidy, and C. Koch, "Load balancing and skew resilience for parallel joins," in *IEEE Int. Conf. on Data Engineering (ICDE)*, 2016, pp. 313–324.

[41] X. Zhou, , and M. E. Orlowska, "Handling data skew in parallel hash join computation using two-phase scheduling," in *Proc. Intl. Conf. on Algorithm and Architecture for Parallel Processing*, 1995, pp. 527–536.