Baumstark, Alexander; Paradies, Marcus; Sattler, Kai-Uwe; Kläbe, Steffen; Baumann, Stephan

So far and yet so near - accelerating distributed joins with CXL

Original published in: 20th International Workshop on Data Management on New Hardware (DaMoN 2024) : June 10th 2024. - New York, New York : The Association for Computing Machinery, (2024), art. 7, 9 pp.

Original published: 2024-06-09

ISBN: 979-8-4007-0667-7

DOI: 10.1145/3662010.3663449

[*Visited:* 2025-04-14]

So Far and yet so Near - Accelerating Distributed Joins with CXL

Alexander Baumstark
Marcus Paradies
Kai-Uwe Sattler
TU Ilmenau, Germany
alexander.baumstark@tu-ilmenau.de
marcus.paradies@tu-ilmenau.de
kus@tu-ilmenau.de

Steffen Kläbe
Stephan Baumann
Actian, Germany
steffen.klaebe@actian.com
stephan.baumann@actian.com

ABSTRACT

Distributed partitioned joins are one of the most expensive operators in distributed DBMSs where a major part of the execution is attributed to network transfer costs. Although high-speed network technologies, such as RDMA, can lower this cost, they still come with significantly higher latency than local DRAM access. The emerging CXL interconnect protocol promises to provide direct and cache-coherent access to remote memory while offering byte-addressable memory access without CPU intervention. For short-distance communication in distributed DBMSs, CXL represents an interesting alternative for low-latency requirements. In this work, we explore how CXL can be leveraged for engine-internal communication and data exchange. We discuss and apply communication strategies to distributed joins. We emulate various CXL characteristics based on optimistic and pessimistic assumptions on the real performance of upcoming CXL devices and evaluate their impact on the execution of distributed joins. Our results show that CXL has the potential to improve distributed join performance.

ACM Reference Format:
Alexander Baumstark, Marcus Paradies, Kai-Uwe Sattler, Steffen Kläbe, and Stephan Baumann. 2024. So Far and yet so Near - Accelerating Distributed Joins with CXL. In *20th International Workshop on Data Management on New Hardware (DaMoN '24), June 10, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3662010.3663449

1 INTRODUCTION

Scaling out computation and memory with Massively Parallel Processing (MPP) is one solution to overcome the memory wall in DBMSs. By connecting memory and CPUs of multiple systems, computation can be efficiently parallelized. However, this shifts the problem towards data transfer between distributed CPUs as the main bottleneck [32]. Traditional approaches for data exchange in distributed DBMSs rely on network-based implementations, which often experience high latency and require careful algorithm design to achieve fast communication and low latency. The emergence of interconnect technologies like Remote Direct Memory Access (RDMA) and Compute Express Link (CXL) presents new opportunities to address these challenges. RDMA reduces the latency by bypassing the CPU and network stack to exchange data using a high bandwidth interconnect like InfiniBand, yet access latency on remote data is relatively high compared to memory access [35].

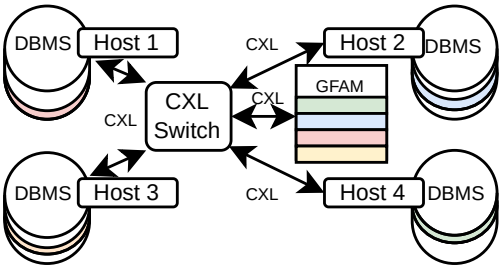

Figure 1: Distributed and partitioned DBMS architecture for hosts interconnected via CXL. Memory is shared via the GFAM feature of CXL 3.0.

CXL is an open standard interconnect for efficient communication between the CPU and devices like accelerators, memory buffers, and I/O devices. The CXL specification cannot only influence the architecture of upcoming systems but also provide solutions to challenges associated with conventional distributed DBMSs. PCIe-carried load and store semantics allow inter-host communication to be conducted similarly to intra-host communication, reducing the overall complexity and potentially enhancing performance [18, 27]. Latency or bandwidth differences in hardware substantially affect performance, necessitating systems to adapt to CXL characteristics for full benefit. CXL 3.0 offers low-latency, memory-semantic communication, allowing interconnected hosts peer-to-peer (P2P) communication and share memory [9].

Moreover, the transition from traditional distributed architectures to shared memory with CXL is accompanied by a paradigm shift: from a distributed sender/receiver model to a shared or disaggregated memory model [18]. This shift entails managing CXL memory, inter-host synchronization, data placement, and different access patterns for higher latencies. This paradigm shift has been initiated by RDMA [19], and various studies have yielded results on application-level cache coherence [11, 35, 39]. RDMA faces challenges like increased read/write amplification, slower synchronization, and extra round trips, all needing mitigation for performance [34]. CXL integrates cache coherence with cache line granularity access over PCIe, which reduces overhead and complexity. Further, insights from handling higher latencies in technologies like Persistent Memory (PMem) can be applied to CXL, due to similar access semantics [6]. Still, the current development of CXL is moving towards an intra-rack solution that requires interconnects like RDMA for inter-rack communication [18, 35].

DaMoN '24, June 10, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0667-7/24/06
https://doi.org/10.1145/3662010.3663449

In this paper, we analyze how existing MPP systems can be seamlessly adapted to CXL and use distributed joins as an example to demonstrate the feasibility of our approach. Distributed joins are significantly influenced by the hardware characteristics of the interconnect linking the distributed hosts. For this, we consider an architecture as illustrated in Figure 1, where several hosts are interconnected via a CXL switch and share memory using the global fabric attached memory (GFAM) feature of CXL 3.0 [9]. Various characteristics can impact distributed join algorithm performance, e.g., the processing performance of data structures like hash tables is influenced by latency, and transfer throughput by the interconnect bandwidth. We investigate how specific characteristics of potential CXL hardware impact distributed hash join algorithms. Given the limited CXL hardware availability, we use hardware emulation to provide insights into the expected performance of future CXL hardware in distributed DBMSs and make the following contributions:

1) We discuss practical communication strategies for hosts interconnected via CXL (Section 3).
2) We implement data exchange primitives for distributed hash joins by leveraging the capabilities of CXL memory (Section 4).
3) We integrate our CXL-based implementation into a commercial DBMS and emulate potential CXL characteristics, evaluating their impact on distributed hash join algorithms (Section 5).

2 BACKGROUND

In the following, we briefly introduce the fundamentals of our work by discussing parallel join strategies, data exchange technologies, and CXL for inter-host communication.

2.1 Parallel and Distributed Joins

Distributed DBMSs commonly use data partitioning to distribute work among hosts and enable parallel and distributed query processing. Data is typically either pre-partitioned (e.g., using hash, range, or random partitioning) or partitioned on the fly [3].

Joining partitioned data must ensure correctness as join partners may be located in different partitions (and on different hosts). Data co-partitioning and co-location are key to avoiding data transfer and achieving fast query execution. For hash partitioning, data co-partitioning of two tables can be achieved by choosing the join key as the partitioning key, and for range partitioning one can choose the same ranges on the join columns to ensure that all join partners are assigned to the same partition. While one can try to co-locate tables that are frequently joined together, designing a partitioning schema for a whole database schema is challenging [5, 14] and will still contain pairs of tables that are not co-located. There are three main approaches to joining tables using hash join algorithms when the build and probe sides are not co-partitioned:

► **Repartition Hash Join (R-HJ):** R-HJ enforces the partition requirement by repartitioning both join sides on the join keys if needed. Afterwards, both sides are co-located and the join can be performed independently on these partitions without further data exchange. The result table is partitioned according to the join keys.
► **Broadcast Hash Join (BC-HJ):** The build side of the join is broadcasted to be available to all workers participating in the join. After broadcasting, independent hash tables are built on each worker and probed independently. This approach is typically reasonable if the build side is small or not partitioned. The result is a table partitioned according to the probe side.
► **Shared Hash Table Hash Join (SHT-HJ):** In the SHT-HJ a shared hash table is built in shared memory and probed concurrently afterwards. Shared memory access granularity may vary in implementations from highly concurrent access of single hash table inserts towards building local hash tables and merging them into shared memory. The result table is partitioned according to the probe side.

The optimizer decision between these approaches is complex and depends on many factors, such as the number of distinct join keys of the probe side (determining the number of hash table collisions), the size of the probe side (BC-HJ and SHT-HJ do not repartition the probe side at all) or the distribution of the probe side (skewness of the partition key vs. the join key). Moreover, subsequent operations might push a partitioning requirement on the join, i.e., subsequent repartitioning might be avoidable depending on the partitioning of the join result. If applied to a distributed DBMS, all described approaches require data exchange or concurrently working on shared memory. Therefore, we investigate how these approaches can be adapted to leverage CXL memory.

2.2 Data Exchange Implementations

Distributed joins aim to enhance parallel processing by utilizing the resources of multiple hosts. Parallel join processing with NUMA architectures has been investigated in [17]. This work shows that resulting performance can benefit from careful implementation based on the underlying system architecture. The work of [23] shows that parallel joins can benefit from additional caching layers like HBM. Similarly, CXL can be seen as a further caching layer. Efficient data exchange is the key to the previously described approaches. Modern solutions range from network-based approaches to disaggregated memory [4, 16]. In the following, we briefly show strategies of data exchange between hosts used in distributed joins.

Message Passing Interface (MPI). is a standardized protocol for inter-host and inter-process network communication [4]. The standard defines routines, syntax, and semantics for writing MPP applications. There are several open-source (OpenMPI, MPICH) and industrial implementations (Intel-MPI). It supports blocking and non-blocking point-to-point and group communication over the network [4]. With this, data can be exchanged via usual network protocols. However, protocol conversions and data buffering in the network stack (OS, NIC, TCP, Ethernet) limit overall performance.

Remote Direct Memory Access (RDMA). is a protocol that enables hosts to access the memory of remote machines directly via the network. Regarding hardware, it is offered by InfiniBand, RDMA over Converged Ethernet, or iWarp. A memory area must be pinned and registered with the network interface card (NIC) to facilitate direct memory access. RDMA-capable NICs bypass the network stack to avoid the OS's context switches and intermediate buffers, which reduces communication overhead and latency [20]. Further, control and data messages are separated. Read or write requests can be executed asynchronously and independently with specialized NICs and without host CPU involvement. RDMA allows *one-sided*

Version	CXL 1.0	CXL 2.0	CXL 3.0
Devices	**Type 1**: Smart NICs **Type 2**: FPGAs, GPUs **Type 3**: Memory Expanders	CXL Single Hierarchy Switch, Logical Devices	CXL Multi Hierarchy Switch
Protocols	Type 1: CXL.io, CXL.cache Type 2: All Type 3: CXL.io, CXL.cache	Switching Fabric, Pooling	P2P Access, GFAM, GIM

Table 1: CXL version history, supported devices and introduced protocols.

communication where data is written to a remote buffer without interaction with the remote host using `READ`, `WRITE` operations or *two-sided* upon a sender/receiver model, where occurring writes are communicated to the other host using `SEND`, `RECV` operations. In [19] the authors study RDMA-based MPI communication for remote shared memory using a polling approach. Our approach, however, adopts multiple strategies via CXL shared memory communication, offering enhanced flexibility and integration. [11] shows an approach where RDMA is used for inter-datacenter communication and shared memory for intra-rack communication. Shared memory access across interconnected hosts necessitates cache coherency to ensure data consistency, as modifications may be stored in local caches [30]. RDMA lacks a cache-coherent protocol, necessitating the implementation of software coherency protocols [24, 39].

Compute eXpress Link (CXL). is an open standard specification that defines several interconnect protocols between processors and different device types built upon PCIe (cf. Table 1). CXL version 1.0 specifies three device types [27] and three protocols [9]: `CXL.io` for PCIe-based I/O setup, `CXL.mem` for memory access via PCIe-carried load/store operations, and `CXL.cache` for cache-coherent access across CXL devices. CXL 2.0 adds CXL switch support, allowing hosts to extend and manage CXL devices within a single hierarchy and introduces memory pooling to multiple host processors.

CXL 3.0 expands cache-coherency semantics for Type 2 and 3 devices to facilitate peer-to-peer communication and memory sharing of the same segment among multiple devices within a cache-coherency domain. The specification supports fabric topologies connecting multiple hosts with Global Fabric Attached Memory (GFAM). This allows disaggregated memory and networking between multiple hosts and devices where the communication is encapsulated in PCIe messages. CXL 3.0 is entirely based on PCIe 6.0, doubling the bandwidth (121 GB/s x16) compared to CXL 2.0 (63 GB/s x16). We focus on the CXL 3.0 specification, which provides cache-coherent P2P communication and shared memory via GFAM.

CXL support is available since the Intel Sapphire Rapids and AMD Epyc Genoa architectures. Research on CXL has been conducted on first 1.0 hardware and emulation [1, 2, 12, 22, 28, 31, 33, 35–38]. In [37] a CXL latency simulator has been proposed, which uses a tracer to add additional CXL latencies. An emulation of CXL with multiple nodes has been shown in [1]. Instead of a single-host emulation, we utilize an emulation environment with separate OS and management. First latency numbers of real hardware are given in [12, 25, 31, 33, 35, 38]. DirectCXL [12] is one of the first prototypes for DRAM memory connected to a host through PCIe. Further, they present first numbers regarding latency ($300 - 500ns$). The work of [25] shows further prototype latency numbers for a Type 3 device (> 500 ns) and bandwidth (< 10 GB/s).

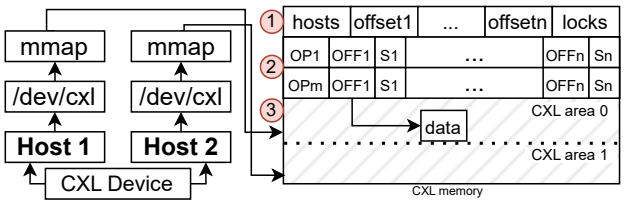

Figure 2: CXL shared memory device runtime (left), and its internal memory organization (right): header (1), data access tables (2) to access data by offset within the CXL area (3).

Software Runtime. CXL Type 3 devices function as headless, remote NUMA nodes, allowing memory expansion devices (Type 3) to integrate additional host-managed device memory (HDM) as main memory [31]. HDM can also operate as a DAX device, enabling direct memory management similar to PMem devices. Figure 2 (left) illustrates the CXL software runtime. In the Linux Kernel, CXL devices are exposed as device entries, enabling the creation of CXL regions using the *cxlctl* tool. Subsequently, *daxctl* can be used to configure the HDM as a *devdax* device. Access to the HDM is facilitated through the exposed path, such as `/dev/cxl`. With this, `mmap` can be utilized to map the HDM to the virtual address space of an application. When connected with other hosts (with CXL 3.0), the same device can be accessed in this way, both hosts using `mmap`. Samsung Scalable Memory Development Kit (SMDK) and SK Hynix's Heterogeneous Memory Software Development Kit (HMDK) provide libraries with a similar software stack [13, 26].

3 INTER-HOST COMMUNICATION VIA CXL

As CXL drivers are in an early development stage, we manage the HDM at the application layer. CXL shared memory has to be managed equally by all hosts. We place a header at the first bytes of the CXL shared memory device (cf. Figure 2 (right)). It includes details such as the number of hosts, the initial offsets of free memory areas, synchronization mechanisms (e.g., spinlocks or barriers), and a reference counter for each area. The remaining CXL memory can be shared globally or allocated among the hosts. However, the applications are responsible for the management, i.e., there is no protection of read/write access, and synchronization is required for concurrent access.

Premises on CXL. Given the current state of CXL software development (frameworks, device drivers) and hardware availability, we make the following assumptions for this work:
(P§1) CXL shared memory acts as a remote NUMA node with uniform NUMA distance for all hosts (PCIe Software View [9]). This arrangement permits atomic operations within an exclusive cache line by a host, visible to other hosts, requiring that all reads and writes conform to cache line sizes (GFAM and GIM [9]).
(P§2) Cache coherency is ensured across a single hierarchy supporting at least 2-4 hosts through CXL, utilizing `CXL.mem` and `CXL.cache`. The advent of CXL 2.0 necessitates an additional software cache coherence protocol like described in [24, 39].
(P§3) Direct host-to-host communication is possible with shared memory as the medium for interaction. This requires sufficient available memory for host-to-device & device-to-host data transfers.

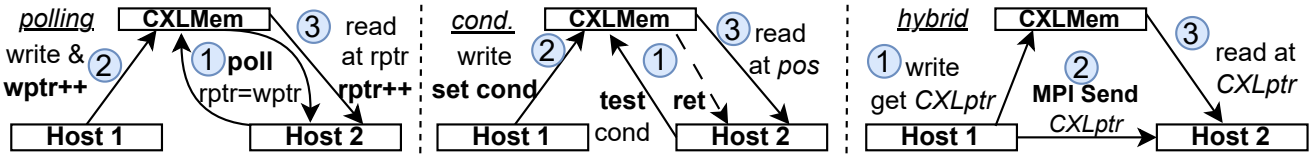

Figure 3: Host communication types via CXL shared memory. For every participating host, the application must ensure a consistent view of the CXL memory in terms of size and (logical) regions. Access to individual addresses is managed through offsets, relative to the starting address of the CXL memory area.

Inter-Host communication. With CXL, communication between hosts relies solely on memory semantics to interact with remote shared memory. We devise three key strategies to ease communication and data transfer among hosts connected via a shared CXL memory device (cf. Figure 3). These strategies map the communication model of MPI into CXL shared memory [19].

► **Polling** checks local read pointers and shared write pointers repeatedly in a loop for a change ①. When a host writes ②, it increments a write pointer in CXL. Polling hosts detect a change, read the data ③, and increment their local pointers. Polling is effective for scenarios with frequent message transfers between hosts, especially when it requires only a few iterations for notification. This strategy is employed once communication has initiated and data is processed in CXL shared memory.
► **Conditional** strategy utilizes a condition variable in CXL to check for changes ① reducing the need for continuous checking. When there are no changes, the thread can proceed with other tasks. Once the writing host notifies by setting the condition ②, the other hosts can detect the change when they check again ③. Using a condition variable for shared memory communication is advantageous in infrequent communication scenarios or for initiating intra-host communication. Additionally, it can be employed to synchronize access to a shared resource, serving as a lock.
► **Hybrid** MPI/CXL combines the MPI approach with CXL by using MPI solely to notify other hosts of changes. Initially, a host writes data to CXL memory ①. Then, it uses MPI to send a pointer (as an offset) pointing to the written CXL data ②. Receiving hosts can read the data through the received pointer ③. The hybrid MPI/CXL strategy is advantageous for initiating communication between hosts, such as launching distributed queries. Placing the data in CXL memory reduces message size, as only the pointer needs to be sent. We use this strategy for host orchestration and management, specifically for exchanging the number of available hosts and additional execution parameters.

Synchronization. Data in CXL memory can be accessed by host-to-device (*H2D*), by device-to-host (*D2H*) transfer, or directly through pointers. The CXL specification guarantees that host-initiated atomic operations can be observed by other hosts, which is fundamental for our synchronization approach [9]. This enables the implementation of various synchronization primitives, similar to those used in standard concurrent synchronization. For this, we implement locking as a spinlock using compare-and-swap and barriers based on an atomic counter and fetch-and-add operation.

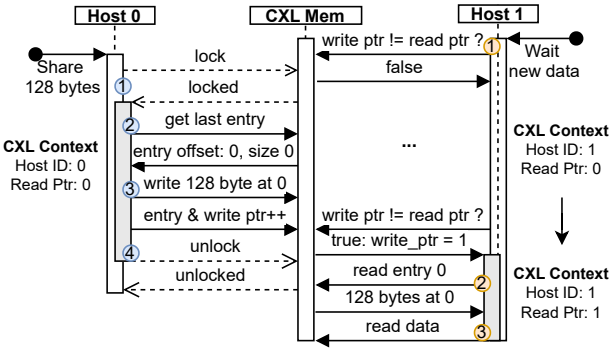

Figure 4: Host 0 writes data in CXL shared memory while Host 1 polls for changes and reads data.

Data Exchange. On the local host side, every host manages a CXL context by itself. The CXL area is managed with data access tables (DAT) (cf. Figure 2). The individual identifier of a host is assigned at startup and communicated via MPI. Shared data in CXL memory is sequentially organized (i.e. as a ring buffer) where individual data can be accessed by DAT entries storing the offsets and sizes of individual data. Further, read and write pointers are stored in CXL memory for each area. A host-managed CXL context, along with its read and write pointers, enables the identification of newly written data by other hosts.

The data exchange procedure between two hosts is illustrated in Figure 4. To share data, a host acquires a lock to write data to a specific CXL area ① before reading the last written entry in the DAT using the write pointer of the CXL area ②. Then, the host writes the data to the appropriate offset, determined by adding the size to the last written offset ③. The header of the area will be updated at the end, and the lock will be released ④. To read the recent data, a host compares its local read pointer with the read pointer of the relevant area ①. If matching ②, the host can read the data at its read pointer ③, increment it, and continue reading until a termination message is encountered. The sender must explicitly signal termination through a final message.

4 JOIN ALGORITHMS ON CXL

The data exchange process is a primary bottleneck in distributed systems, particularly when considering join algorithms in a distributed setting. Individual hosts must wait for data from other hosts, exacerbating the bottleneck with the use of slow interconnects, frequent protocol conversions, and data serialization.

CXL leverages PCIe to provide high bandwidth and low latency and offers memory-like access capabilities, which are ideal for mitigating these challenges. In this section we describe distributed join algorithms in a multi-host setup connected by CXL shared memory. Each host runs a database node with multiple workers, leveraging the approaches described for inter-host communication. The CXL memory area is organized as described in Figure 2 and structured by the number of participating hosts referred to as CXL areas (allowing GFAM or GIM [9]). Each area is (logically) assigned to a single host during query setup. The CXL areas contain a header and a separate DAT for each operator pair (sender, receiver) of a join, simplifying the identification of data belonging to an operator.

4.1 Repartition Hash Join

Data in distributed DBMSs can be partitioned over all participating hosts. Executing a partitioned hash join results in repartitioning the data (R-HJ). The phases of this algorithm are:

(1) Repartitioning of both joins sides on join keys,
(2) data exchange of partitions to the appropriate nodes, and
(3) join on local partitions.

Repartitioning Phase. The purpose of the repartitioning phase is to partition both join sides such that tuples with matching join keys result in the same partition. After co-location, these partitions can be processed independently. Partitioning is done by selecting tuples by the join keys of both join sides. Selected tuples will be serialized and inserted into one or multiple buffers to prepare them before the actual exchange. Serialization is necessary as host-local data cannot be accessed directly from other hosts. Each buffer is tagged by an operator and the host *id* to determine the belonging data. Additionally, the count of tuples inserted is placed in the initial bytes of the buffer, offering additional information to the receiver.

Data Exchange Phase. We adapt the sender-receiver model implemented via the polling approach in CXL for data exchange. The query optimizer enforces the communication pairs of the individual partitions, the number of sender/receiver threads per operator, and the number of buffers per operator.

On the sending side, when the data is partitioned according to the previous step, the partitions (buffers) will be transferred to their determined destination. Sending data is implemented by writing data directly into the assigned CXL area of the destination host. This allows hosts to communicate via CXL shared memory and reduces additional synchronization for concurrent access. To perform the exchange via CXL, the sending host locks the CXL area and performs the data exchange procedure as described in Section 3. For optimal performance, distributed join algorithms need to consider the hardware access latency. For this, we employ two strategies. Tuples for a particular partition can either be placed directly on an allocated buffer in CXL or beforehand a host local buffer and copied afterward to CXL memory. We refer to these strategies as **CXL-D**(irect) and **CXL-C**(opy).

For CXL-D, buffers are allocated using `cxl_malloc` similar to the usual `malloc` but returns a pointer to CXL memory. Data for the buffer can be directly written using `memcpy`. With CXL-C, the data will be first copied to local memory and afterward as one single transfer with `cxl_memcpy` to CXL shared memory. This function additionally manages the DAT of the CXL area. Depending on the size and resulting access latency of the CXL memory one of the methods has to be preferred. When the transfer of tuples to the CXL buffer is complete, the DAT entry with offset position and size will be inserted and the write pointer of this CXL area header will be increased, indicating available data for participating hosts. Then, the cache line of the sender needs to be flushed to make changes visible to other hosts. This will be done for all buffers. After this, a final buffer without tuples will be placed in CXL, containing the number of buffers sent to indicate the end of the transfer.

On the receiving side, the hosts poll the CXL area for changes that start directly at query execution. This is done iteratively by comparing the read pointer of the hosts' CXL context with the write pointer in the CXL area header as described in Section 3. Whenever a change is detected, the buffer can either be read directly (**CXL-D**) or copied to a host local buffer (**CXL-C**). As data can be sent by multiple senders concurrently, the order of the data can be mixed up. The receiver can obtain the belonging data through a tag inserted in the buffer. The receiving side polls until the termination message with the number of messages sent from the sender is obtained.

After the data exchange, both join sides are co-located and partitioned according to the join keys. The remaining join steps can be performed locally on the partitions without further data exchange. The results of the join can either be processed further or merged at later processing.

4.2 Broadcast Hash Join

The main characteristic of the BC-HJ is the broadcast of the build side to all available hosts. The optimizer will enforce which build side will be considered for the broadcast. The phases of the BC-HJ are as follows:

(1) Broadcast of the build side to participating hosts,
(2) build of local hash tables, and
(3) probe on local hash tables.

Broadcast. In pure network-based solutions without the presence of shared memory, broadcast data are transferred multiple times, degrading overall performance due to multiple copies. CXL with shared memory enables shared data placement directly accessible by other hosts.

Instead of transferring buffers from one host to the assigned CXL memory area of the other hosts, like with the R-HJ, broadcasting hosts place the data directly in their assigned CXL memory area. The main advantage of broadcasting via CXL shared memory is that only one transfer per buffer is necessary. Broadcasting can be performed by multiple hosts when data is (pre)partitioned. If a host has no data to broadcast, it places a termination message with 0 as the number of tuples sent. In the same way, as in the R-HJ, transfers can be direct (CXL-D) or through copies from local buffers (CXL-C). With each host writing to its assigned CXL area, additional synchronization is unnecessary, enhancing parallelism. Receiving workers need only to collect the data from all other areas to obtain the complete data. Collecting is again done by polling. As every worker maintains its CXL context and read-pointers for all areas, new data can be recognized by comparing them with the write pointer of the areas.

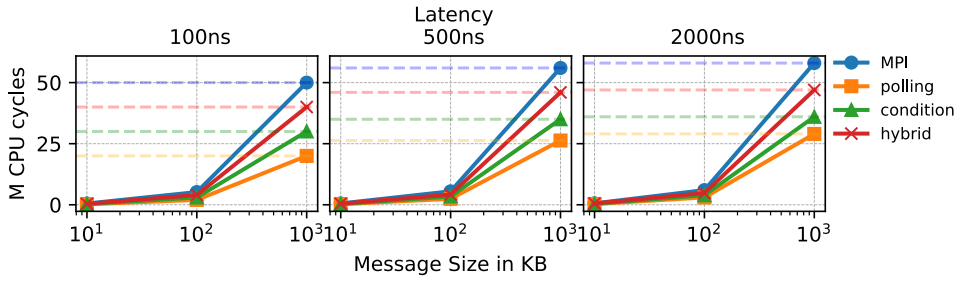

Figure 5: RTT for 100 messages between 2 hosts varies by message size and access latency, with dotted lines indicating 1MB message runtimes.

To enhance polling response, we expanded the polling to include iterations over other areas in each cycle, allowing data access in CXL from the fastest host. The build and probe phase starts as soon as a host reads the final message from the sender(s).

Build & Probe. As soon as the data exchange is complete and each host receives the data hash tables can be constructed locally. Hash table construction utilizes CXL-C or CXL-D, affecting build and probe performance based on CXL characteristics.

4.3 Shared Hash Table Hash Join

In SHT-HJ, hosts operate directly on a hash table placed in CXL memory. The underlying hash table implementation is built upon vectorization [29] and chaining for collision resolution [15].

In contrast to the previous two hash join algorithms, the SHT-HJ operates on a data structure allocated in CXL instead of memory buffers. This implies a more fine-granular and frequent data access which is more affected by the CXL memory latency. The individual steps of the SHT-HJ are:

(1) Allocate a global hash table,
(2) build the global hash table with workers, and
(3) probe them individually.

Hash Table Build. A barrier synchronizes hash table allocation, with the first worker at the barrier allocating a global hash table in its CXL area via `cxl_malloc`. The optimizer determines a suitable allocation size for the hash table. The barrier blocks other hosts and their workers until the allocation is completed. Then, the hosts iterate over the DAT of the CXL areas to find the allocated hash table. The allocated hash table is the first entry of a DAT. All subsequent allocations by other hosts to extend the hash table have to be in the same CXL area which can be achieved by passing the appropriate area to `cxl_malloc`. After this, all workers begin with the building phase of the join. The lock of the CXL area, where the hash table is allocated synchronizes concurrent access when building the hash table in CXL memory. Intra-worker synchronization is not necessary since every thread works on different bucket ranges.

The individual workers can work directly in CXL (CXL-D) or create the buckets first in the host-local memory and copy them afterward to the main table (CXL-C). The latter is useful when there are many tuples and a high CXL access latency. When the build is complete, the workers copy their buckets from memory into CXL shared memory using `cxl_memcpy`.

	Latency			Bandwidth		
	Read	Write	**Emulation**	Read	Write	**Emulation**
Local NUMA	87 ns	93 ns	**< 100 ns**	107 GB/s	98 GB/s	29 GB/s
Remote NUMA	147 ns	155 ns	**< 500 ns**	33 GB/s	29 GB/s	29 GB/s
PMem	411 ns	940 ns	**< 2000 ns**	30 GB/s	12 GB/s	29 GB/s
Emul. Network		3233 ns	**< 4000 ns**		8 GB/s	29 GB/s

Table 2: Latency and bandwidth of supervisor hardware and resulting measures in emulated instances.

Both approaches only differ in data allocation and final merging in the case of CXL-C. On the implementation side, it requires switching pointers from main memory to CXL shared memory. This proves the seamless integration of CXL shared memory into distributed DBMSs. Every host waits on a final barrier for other hosts to finish the build stage.

Probe. For probing, the hash table can be accessed directly in CXL memory (CXL-D) or copied to the local memory of the workers (CXL-C). As data is partitioned and co-located, different workers operate on distinct ranges of buckets. Thus, transfer or access costs can be reduced.

4.4 Query Execution

A query often consists of multiple joins executed on the results of respective previous joins. The usual implementation with MPI provides extended communication by tagging messages, i.e., a message is tagged by operator and worker *id* to identify belonging tuples to an operator. For CXL, we assign each join operator its own DAT (see Figure 2 right side). The number of DATs has to be configured by the query optimizer. Workers of an operator can identify their belonging message from other hosts by accessing their specific DAT in the CXL area identified by the assigned operator identifier.

5 EVALUATION

Our experiments were performed on a system equipped with 4 Intel Xeon 8376H CPUs (56 threads each, across 4 NUMA nodes), 768 GB DDR4 RAM (24 DIMMs), 1537 GB Intel Optane Persistent Memory 200 Series (24 NV-DIMMs of 128 GB), and 6 Intel NVMe SSDs (2.5 TB each), running on Linux 5.4 and Ubuntu 20.04. We integrated the distributed join algorithms into the commercial Actian Vector DBMS [10], built on the X100 [7] analytical database engine. For MPI, we use Intel MPI 2019. For all approaches, we use `mpirun` to start and manage multiple hosts.

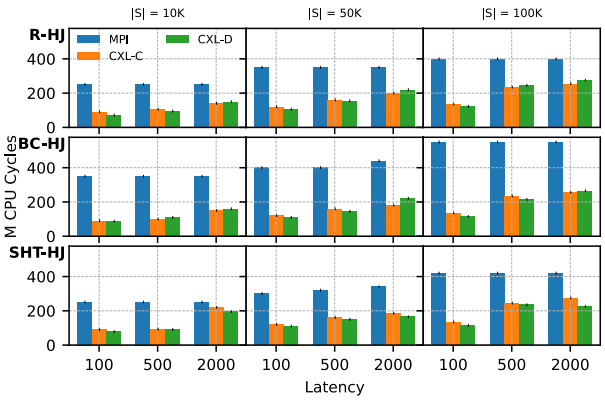

Figure 6: Benchmark with different table sizes for S and latencies on CXL. Run time in million CPU cycles.

Emulation. We use QEMU and NUMA-based emulation of CXL shared memory due to the unavailability of CXL 3.0 hardware [8]. We emulate multiple instances on a single hypervisor. Hence, we implicitly provide hardware-controlled cache coherency for CXL shared memory, as with premise **P§2**. NUMA-based emulation can show more realistic CXL characteristics since prototypes are not optimized for latency [35].

A QEMU instance can be configured with different memory configurations by placing a file-backed memory in an appropriate location, like PMem. We use three hardware latencies to evaluate different latency configurations: optimistic (local NUMA), realistic (remote NUMA), and pessimistic (PMem-like) (cf. Table 2). The resulting memory performance corresponds to the characteristics of the underlying hardware (including some emulation overhead) since syncs and flushes are pushed to the underlying device. In the emulated environment, the CXL device is exposed as a remote NUMA node, allowing the OS to access it as detailed in Section 2.2. For local NUMA latency, emulated instances and the CXL device are set up on the same NUMA node of the hypervisor. For remote NUMA, they are configured on different NUMA nodes using `numactl`. For PMem, we configure the memory on a PMem device. The emulated maximum bandwidth is similar in all settings (PMem-like). We use the hypervisor's CPU with a Kernel-based Virtual Machine (KVM) and measure the run time in CPU cycles with `rdtsc`, `rdtscp`, and `cpuid` instructions to retrieve CPU cycle-accurate results from the emulated instances [21].

Further, we use the hypervisor network stack to provide the highest possible network performance between emulated instances. With this, we can emulate a high-bandwidth and low-latency network for the baseline as network packets are transferred via DRAM between emulated hosts without leaving the NIC. With this setup, we can evaluate different CXL characteristics of possible future hardware and their effects on the resulting DBMS performance.

The CXL specification includes numbers for additional latencies introduced by the protocol, which we also consider in our evaluation [9]. Further, we include added performance numbers which we emulate by explicit cache line flushes to show the worst-case behavior of possible coherence mechanism.

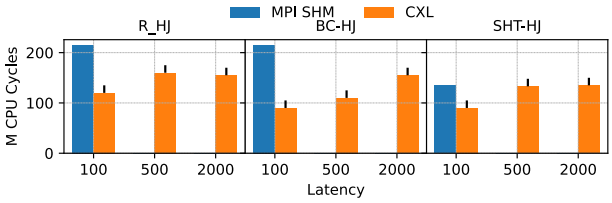

Figure 7: Host-local shared memory execution with MPI vs CXL with |S| = 50k, with 4 hosts and 2 threads each.

Host communication. A benchmark with different memory access latencies for the communication approaches presented in Section 3 is shown in Figure 5. The benchmark measures the round trip time (RTT) of 100 message exchanges between two hosts. In all settings, the polling approach provides the fastest RTT. MPI and hybrid methods, both network-dependent, show similar run times. Sending a pointer that points to data in CXL reduces transfer overhead. The conditional approach, which checks for messages via the server's main loop, performs between polling and hybrid. CXL can utilize the PCIe 4.0 x16 slot's bandwidth of 32 GB/s, surpassing the network bandwidth significantly. Using `strace`, we found CXL-based methods cut system calls by up to 50% and decreased time per call by reducing network stack calls, thus lowering communication costs.

Microbenchmark Joins. For the distributed join microbenchmark, we employ synthetically generated datasets. We benchmark a join $R \bowtie S$, with $R = 100,000$. S varies in size, including $10,000$, $50,000$, and $100,000$ to show performance on a low and higher transfer amount. We conduct the benchmark in a 4-node setting and the data are equally partitioned among the hosts.

Figure 6 depicts the comparison between the distributed join variants with CXL. We compared each join with their equivalent network-based MPI implementation and measured only the query time. For every run, the CXL implementation reduces the execution time. With an increasing latency, the execution time of the CXL approaches increases since data has to be accessed multiple times with H2D and D2H transfers. With the increasing $|S|$, MPI run time degrades since multiple buffer transfers over the network while the CXL implementation utilizes data transfers over PCIe and its bandwidth by using `memcpy`-like transfers. CXL run time increases marginally with increasing transfer size due to higher bandwidth.

Direct vs Copy. We compared the direct CXL memory access against the copy approaches. The results are shown in Figure 6 as CXL-D and CXL-C. For the R-HJ and BC-HJ, the CXL-D approach is more affected by latency than CXL-C. The CXL-D approach is always slower because of the increased access costs in the highest latency setting. However, since the access granularity in the SHT-HJ is smaller (buffer vs tuples), direct access is more beneficial. CXL-D is advantageous for high data volumes and low latency, leading to reduced data transfers. Also, CXL-D can hide latencies efficiently for small data and fewer data accesses. Whenever there is high latency, the CXL-C can hide the latency more effectively. SHT-HJ with CXL-D outperforms R-HJ and B-HJ due to finer access granularity on higher latencies. BC-HJ benefits large data and high latencies with reduced data transfers.

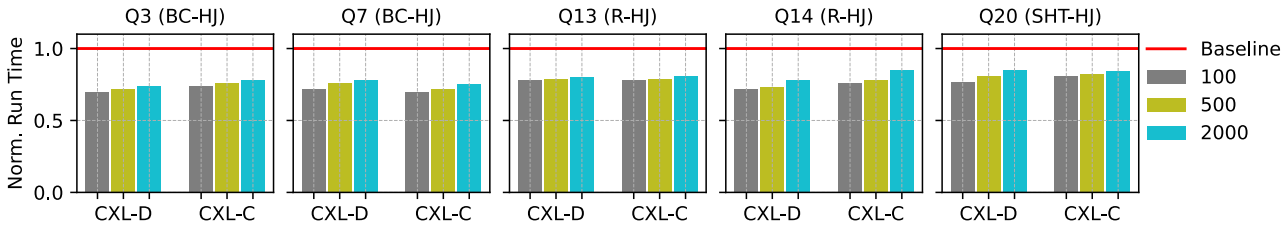

Figure 8: Selected TPC-H queries running on four hosts with 56 threads. Baseline uses an MPI-based network approach for data exchange. The queries are executed with CXL-D and CXL-C and under different emulated latencies (100, 500, 2000).

Intra-host shared memory. We run the MPI baseline with four nodes directly on the supervisor. In this setting, the MPI baseline was configured to transfer messages through shared memory (`I_MPI_SHM` flag). The results in Figure 7 show still less execution time for the CXL-based approaches. The BC-HJ and SHT-HJ show the fastest execution for lower latency, while in higher latencies the different algorithms are comparable. Therefore, CXL can achieve NUMA-like performance for data exchange in a distributed setting. Adding additional CXL protocol latency (≈ 50ns [9]) for every CXL memory access on both hosts affects the run time marginally. The CXL approach achieves comparable performance in algorithms with fewer data transfers like SHT-HJ. Thus, the CXL architecture could offer a NUMA-like behavior for scaled distributed DBMSs while providing distributed resources.

TPC-H. To evaluate the distributed joins with a more realistic workload, we use the well-known TPC-H benchmark. We executed TPC-H queries (SF 500) on four nodes using CXL shared memory, specifically targeting queries Q3, Q7, Q13, Q14, and Q20 known for data transfer-bound hash joins. Queries Q3 and Q7 use the B-HJ, queries Q13 and Q14 leverage the R-HJ, and query Q20 the SHT-HJ. The execution times (normalized to network-based baseline) are shown in Figure 8. The higher bandwidth of a CXL can decrease run time by up to 25% for queries bottlenecked by data exchange. For the BC-HJ in Q3 and Q7, data transfers are reduced to a single data transfer into CXL shared memory. The CXL-C approach hides additional latency by initially buffering data in local memory. R-HJ in Q13 and Q14 is significantly impacted by latency from frequent memory access. Multiple buffers add to processing complexity, making direct access to CXL shared memory advantageous. The SHT-HJ works directly on data structures with fewer data transfers, making CXL-D more beneficial. However, for higher latency, CXL-C can mitigate the effects more effectively.

6 CONCLUSION

In this work, we described possible communication strategies for hosts interconnected via CXL shared memory. Based on these strategies, we present different data exchange approaches for distributed hash join algorithms that leverage CXL shared memory provided by future CXL 3.0 hardware. The distributed hash join algorithms were evaluated using different emulated CXL shared memory setups. This work has demonstrated that while working with CXL offers performance benefits, it also introduces additional challenges that must be effectively addressed to fully exploit CXL.

To summarize, the results of this work are as follows:

1) P2P communication via a CXL shared memory device must be conducted solely through memory semantics. Concurrent data access and coordination must be implemented within the system. We addressed this issue by adapting the communication model from a network-based approach to a shared memory setting, utilizing CXL memory management with read/write pointers.
2) Different communication schemes can be adapted from the sender-receiver model to shared memory with minimal changes to the network-based implementation. The pooling approach offers the fastest communication response, even at higher latencies.
3) In comparison to network-based solutions, CXL reduces the volume of data shared across hosts since every host can access the same memory region. This is particularly advantageous for broadcasting data to all participating hosts like with the BC-HJ.
4) In contrast to RDMA, CXL incorporates a cache coherence protocol, simplifying the overall development process for data sharing. However, data exchange between racks still requires additional interconnects such as RDMA.
5) CXL enables the transformation of the distributed system into one that operates as if it were a single cohesive unit. As demonstrated with the SHT-HJ, every host contributes to computing a data structure, significantly enhancing the level of parallelism.

The results show that CXL has the potential to change future architectures of up-scaled distributed DBMS leading to significant performance enhancements. CXL shared memory allows distributed DBMS to function as a logically unified system.

Future CXL 3.0 hardware might introduce additional latencies due to cache coherency mechanisms and specific protocol characteristics, as well as bandwidth limitations. Furthermore, especially in distributed systems, the bandwidth limitations and increased latencies of the interconnect must be considered.

ACKNOWLEDGMENTS

This work was partially funded by the Actian Corp. and by the German Research Foundation (DFG) in the context of the projects "Hybrid Transactional/Analytical Graph Processing in Modern Memory Hierarchies (#TAG)" and "Processing-In-Memory Primitives for Data Management (PIMPMe)" as part of the priority programs "Scalable Data Management for Future Hardware" (SPP 2037, SA 782/28-2) and "Disruptive Memory Technologies" (SPP 2377, SA 782/31).

REFERENCES

[1] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna T. Malladi, and Yang-Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *International Conference on Management of Data, DaMoN 2022*. 8:1–8:5. https://doi.org/10.1145/3533737.3535090

[2] Moiz Arif, Kevin Assogba, M. Mustafa Rafique, and Sudharshan Vazhkudai. 2022. Exploiting CXL-based Memory for Distributed Deep Learning. In *Proceedings of the 51st International Conference on Parallel Processing, ICPP 2022, Bordeaux, France, 29 August 2022 - 1 September 2022*. ACM, 19:1–19:11. https://doi.org/10.1145/3545008.3545054

[3] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. 168–180.

[4] Claude Barthels, Gustavo Alonso, Torsten Hoefler, Timo Schneider, and Ingo Müller. 2017. Distributed Join Algorithms on Thousands of Cores. *Proc. VLDB Endow.* 10, 5 (2017), 517–528. https://doi.org/10.14778/3055540.3055545

[5] Stephan Baumann, Peter A. Boncz, and Kai-Uwe Sattler. 2016. Bitwise dimensional co-clustering for analytical workloads. *VLDB J.* 25, 3 (2016), 291–316. https://doi.org/10.1007/S00778-015-0417-Y

[6] Lawrence Benson, Marcel Weisgut, and Tilmann Rabl. 2023. What we can learn from persistent memory for cxl. (2023).

[7] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[8] Jonathan Cameron. 2024. QEMU. https://gitlab.com/jic23/qemu. Accessed: 2024-03-21.

[9] CXL Consortium. 2024. Compute Express Link. https://computeexpresslink.org/cxl-specification-landing-page Accessed on 29.02.2024.

[10] Andrei Costea, Adrian Ionescu, Bogdan Răducanu, Michał Switakowski, Cristian Bârca, Juliusz Sompolski, Alicja Łuszczak, Michał Szafrański, Giel de Nijs, and Peter Boncz. 2016. VectorH: Taking SQL-on-Hadoop to the Next Level. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1105–1117. https://doi.org/10.1145/2882903.2903742

[11] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1477–1488. https://doi.org/10.1109/ICDE48307.2020.00131

[12] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. 287–294.

[13] SK Hynix. 2024. HMSDK: Heterogeneous Memory Software Development Kit. https://github.com/skhynix/hmsdk Accessed on 29.02.2024.

[14] Steffen Kläbe and Kai-Uwe Sattler. 2023. Patched Multi-Key Partitioning for Robust Query Performance. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*. 324–336. https://doi.org/10.48786/EDBT.2023.26

[15] Gary D. Knott and Pilar De La Torre. 1989. Hash table collision resolution with direct chaining. *J. Algorithms* 10, 1 (mar 1989), 20–34. https://doi.org/10.1016/0196-6774(89)90021-7

[16] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. 2021. Farview: Disaggregated memory with operator off-loading for database engines. *arXiv preprint arXiv:2106.07102* (2021).

[17] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. 2015. Massively Parallel NUMA-Aware Hash Joins. In *In Memory Data Management and Analysis*, Arun Jagatheesan, Justin Levandoski, Thomas Neumann, and Andrew Pavlo (Eds.). Springer International Publishing, Cham, 3–14.

[18] Alberto Lerner and Gustavo Alonso. 2024. CXL and the Return of Scale-Up Database Engines. arXiv:2401.01150 [cs.DB]

[19] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. 2003. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 17th Annual International Conference on Supercomputing* (San Francisco, CA, USA) *(ICS '03)*. 295–304.

[20] Teng Ma, Kang Chen, Shaonan Ma, Zhuo Song, and Yongwei Wu. 2021. Thinking More about RDMA Memory Semantics. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 456–467. https://doi.org/10.1109/Cluster48925.2021.00033

[21] Gabriele Paoloni. 2010. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. *Intel Corporation* 123, 170 (2010).

[22] S. J. Park, H. Kim, K.-S. Kim, J. So, J. Ahn, W.-J. Lee, D. Kim, Young-Ju Kim, J. Seok, J.-G. Lee, H.-Y. Ryu, C. Y. Lee, J. Prout, K.-C. Ryoo, S.-J. Han, M.-K. Kook, J. S. Choi, J. Gim, Y. S. Ki, S. Ryu, C. Park, D.-G. Lee, J. Cho, H. Song, and J. Y. Lee. 2022. Scaling of Memory Performance and Capacity with CXL Memory Expander. In *2022 IEEE Hot Chips 34 Symposium, HCS 2022, Cupertino, CA, USA, August 21-23, 2022*. 1–27. https://doi.org/10.1109/HCS55958.2022.9895633

[23] Constantin Pohl and Kai-Uwe Sattler. 2018. Joins in a heterogeneous memory hierarchy: exploiting high-bandwidth memory. In *Proceedings of the 14th International Workshop on Data Management on New Hardware* (Houston, Texas) *(DAMON '18)*. Association for Computing Machinery, New York, NY, USA, Article 8, 10 pages. https://doi.org/10.1145/3211922.3211929

[24] Magdalena Pröbstl, Philipp Fent, Maximilian E. Schüle, Moritz Sichert, Thomas Neumann, and Alfons Kemper. 2021. One Buffer Manager to Rule Them All: Using Distributed Memory with Cache Coherence over RDMA. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021*. 17–26.

[25] Niklas Riekenbrauck, Marcel Weisgut, Daniel Lindner, and Tilmann Rabl. 2024. A Three-Tier Buffer Manager Integrating CXL Device Memory for Database Systems. (2024).

[26] Samsung. 2024. OpenMPDK/SMDK: Scalable Memory Development Kit. https://github.com/OpenMPDK/SMDK Accessed on 29.02.2024.

[27] Debendra Das Sharma, Robert Blankenship, and Daniel S. Berger. 2023. An Introduction to the Compute Express Link (CXL) Interconnect. arXiv:2306.11227 [cs.AR]

[28] Joonseop Sim, Soohong Ahn, Taeyoung Ahn, Seungyong Lee, Myunghyun Rhee, Jooyoung Kim, Kwangsik Shin, Donguk Moon, Euiseok Kim, and Kyoung Park. 2023. Computational CXL-Memory Solution for Accelerating Memory-Intensive Applications. *IEEE Comput. Archit. Lett.* 22, 1 (2023), 5–8. https://doi.org/10.1109/LCA.2022.3226482

[29] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. 2011. Vectorization vs. compilation in query execution *(DaMoN '11)*. 33–40. https://doi.org/10.1145/1995441.1995446

[30] Daniel Sorin, Mark Hill, and David Wood. 2011. *A primer on memory consistency and cache coherence*. Morgan & Claypool Publishers.

[31] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023*. 105–121. https://doi.org/10.1145/3613424.3614256

[32] Junjay Tan, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing a cloud DBMS: architectures and tradeoffs. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2170–2182. https://doi.org/10.14778/3352063.3352133

[33] Chenjiu Wang, Ke He, Ruiqi Fan, Xiaonan Wang, Wei Wang, and Qinfen Hao. 2023. CXL over Ethernet: A Novel FPGA-based Memory Disaggregation Design in Data Centers. In *31st IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2023, Marina Del Rey, CA, USA, May 8-11, 2023*. IEEE, 75–82. https://doi.org/10.1109/FCCM57271.2023.00017

[34] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1033–1048. https://doi.org/10.1145/3514221.3517824

[35] Zhonghua Wang, Yixing Guo, Kai Lu, Jiguang Wan, Daohui Wang, Ting Yao, and Huatao Wu. 2024. Rcmp: Reconstructing RDMA-Based Memory Disaggregation via CXL. *ACM Trans. Archit. Code Optim.* 21, 1, Article 15 (jan 2024), 26 pages. https://doi.org/10.1145/3634916

[36] Xingda Wei, Haotian Wang, Tianxia Wang, Rong Chen, Jinyu Gu, Pengfei Zuo, and Haibo Chen. 2023. Transactional Indexes on (RDMA or CXL-based) Disaggregated Memory with Repairable Transaction. *CoRR* abs/2308.02501 (2023). https://doi.org/10.48550/ARXIV.2308.02501 arXiv:2308.02501

[37] Yiwei Yang, Pooneh Safayenikoo, Jiacheng Ma, Tanvir Ahmed Khan, and Andrew Quinn. 2023. CXLMemSim: A pure software simulated CXL. mem for performance characterization. *arXiv preprint arXiv:2303.06153* (2023).

[38] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023*. 658–674. https://doi.org/10.1145/3600006.3613135

[39] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. 685–699.