# One Buffer Manager to Rule Them All

## Using Distributed Memory with Cache Coherence over RDMA

Magdalena Pröbstl
magdalena.proebstl@tum.de

Philipp Fent
philipp.fent@tum.de

Maximilian E. Schüle
m.schuele@tum.de

Moritz Sichert
moritz.sichert@tum.de

Thomas Neumann
neumann@in.tum.de

Alfons Kemper
kemper@in.tum.de

Technical University of Munich

## ABSTRACT

Remote direct memory access (RDMA) allows query processing on distributed systems when data exceeds the size of local memory on a single machine. Cache coherence protocols generalize distributed memory as they abstract remote data access and cache data locally. Unfortunately, existing protocols are limited to main memory only.

In this work, we extend a recently presented cache-coherence protocol to deal with background storage as well. With its memory input/output primitives, this protocol serves as basis for a distributed buffer manager that is capable of pinning and evicting pages to frames of remote nodes. Whereas locking is part of the cache coherence protocol, the buffer manager is responsible for a distributed buffer pool. This results in two-stage caching with almost no overhead, due to the quickly converging performance characteristics of SSDs and RDMA capable networks like InfiniBand to not waste compute performance. Benchmarking the input/output primitives results in a eviction strategy of fully utilizing the distributed buffer pool before swapping out pages.

## 1. INTRODUCTION

Data sets are constantly growing, quickly reaching sizes of multiple terabytes. While modern in-memory databases are becoming popular, growth of main memory capacities has slowed down significantly in recent years. Scaling-up fast storage is also limited, since each additional NVMe solid state disk (SSD) in use blocks PCIe lanes of the processor. This results in a fundamental limit of storage capacity on a single node system and poses a problem to both traditional disk-oriented or a modern in-memory database systems and makes it necessary to scale-out to multiple nodes.

Scale-out systems employ a distributed architecture for multiple nodes [18]. Popular state-of-the-art systems use remote direct memory access (RDMA) to directly expose
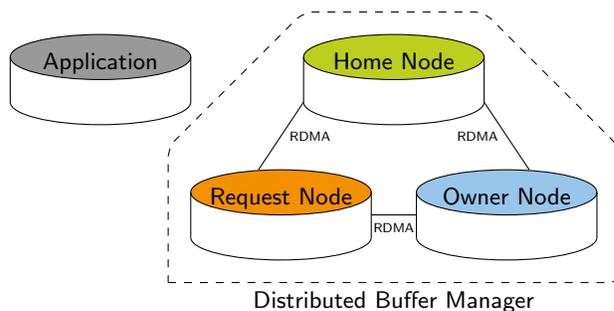


Figure 1: Sketch of a distributed buffer manager: The buffer manager hides the memory layout from the application's perspective. It loads pages either from a home or remote SSD first into frames of the home node. If frames run out, frames of the remote node are taken and the page is cached locally according to the cache coherence protocol.

local RAM to their scale-out cluster. RDMA allows low-latency communication between nodes and memory access to remote nodes that behaves similarly to local non-uniform memory access (NUMA) [15]. RDMA has proven to be the best technology for a distributed database system [9, 13, 23]. Various research systems enhance in-memory join processing [2], build distributed index structures [19, 29] or have decentralized lock management [28].

Performance advantages of RDMA stem largely from by-passing the remote operating system and CPU by directly accessing remote memory with hardware support of the network interface card. Traditionally, RDMA was used to accelerate specialized data structures but current research by Cai et al. [3] generalizes distributed main memory using a cache coherence protocol. For a complete storage scale-out, distributed memory access is but one building block for a distributed beyond main memory database system [8, 10, 20, 21, 22, 24]. One link to traditional systems is still missing: *How to combine distributed memory with traditional page caching in a buffer manager?*

In our work, we answer this question with a design and implementation of a distributed buffer manager (see Figure 1). It combines a canonical buffer pool [5, 4, 16] with distributed memory access, resulting in two-stage caching with almost no overhead, due to the quickly converging performance characteristics of SSDs [17] and RDMA capable networks like InfiniBand. With our study, we show that

it is possible to connect the traditional world of relational database management systems to efficient distributed data processing. The source code has been made available at `https://github.com/tum-db/cachecoherence`.

This work's contributions are as follows:

1. We introduce and evaluate the concept of distributed memory management using a cache coherence protocol (Section 3).
2. We design an efficient implementation of a buffer manager over distributed memory (Section 5).
3. We evaluate the performance of the buffer manager using microbenchmarks (Section 6).

This paper reports the concept and evaluation of an RDMA-based distributed buffer manager. It therefore referes to related work on RDMA for operations of distributed database systems and a framework for RDMA-based primitives. We proceed by introducing a cache coherence protocol that covers—unlike previous studies—operations on both main memory and background storage. Hereby, it provides an abstraction of distributed main memory using caching that can be accessed uniformly and allows global access to distributed background storage without wasting compute performance on remote servers. Finally, we introduce our distributed buffer manager, which is based on an extension of a disk-oriented database system prototype that uses the framework's memory abstraction for allocating buffer frames and evicts pages to local and remote background storage. Our evaluation shows that the buffer manager performs best when fully utilizing the distributed buffer pool first before swapping out pages locally.

## 2. RELATED WORK

For the underlying distributed data storage, we rely on memory caching mechanisms of main memory as well as of background storage. For main memory, the memory-coherence protocol *GAM* [3] has been presented that manages main memory of distributed machines connected via InfiniBand. We build up on this idea by allowing flexibly sized pages for main memory—later used by our buffer manager—and extend the memory coherence protocol to manage background storage.

**RDMA is not persistent memory aware.** Other recent developments for larger than memory databases include the use of non-volatile main memory (NVM) [1, 6, 25, 26]. In general, remote access to non-volatile memory has many synergizing effects [14]. However, RDMA does not directly guarantee persistence in NVM due to NIC write caches [27, 11]. The currently proposed workarounds all induce high overhead and NVM is both, much more expensive and much more limited in capacity than state-of-the-art SSDs.

**In-memory capabilities of RDMA.** When the memory of a single machine is exceeded, RDMA helps to accelerate query processing of distributed database systems [7, 12]. Ziegler et. al. [29] build tree-based index structures for network-attached-memory (NAM), where servers, providing RDMA, either host the data or do the computations. In the context of join processing, research has been conducted on parallel in-memory hash joins over RDMA-based distributed database systems [2] where remote data is directly mapped using RDMA network primitives similar to our mapping. When remote data is cached or pages are accessed, locking mechanisms to avoid the reading of dirty data are essential.

*DSLR* [28] is an example of decentralized lock management on RDMA-based networks to avoid starvation.

## 3. CACHE COHERENCE PROTOCOL

The distributed buffer manager relies on an efficient memory cache coherence protocol that offers an abstraction for distributed memory. The basic idea is to create memory primitives for RDMA-based distributed systems with each entity called *node*. Although the system is split into multiple nodes, a partitioned global address space (PGAS) provides a logically unified view to identify memory. This simplifies the development of multi threaded distributed applications. We base our cache coherence model on GAM [3], but extend the protocol to manage persistent storage as well. In the following, we will explain the architecture of the cache coherence protocol for main memory as well as persistent storage with its interfaces.

### 3.1 Node

*Nodes* form the entities of the distributed system and provide an interface for remote reads and writes. On every machine of the system, a node is initialized in the beginning. Every node has its own individual identifier, which is used to identify where data is stored, to lock data and to detect which node is responsible for the data at a specific address.

### 3.2 Global Address

Each address, on main memory as well as on background storage, is represented as a *global address* (see Listing 1). Technically, this is a struct that identifies a global address with the following information:

- the size that is stored at the address,
- the pointer to the node which hosts the data,
- its node ID,
- and a flag that indicates whether the data is stored on main memory or in a file on background storage.

In the latter case, the global address defines the location of a file and the pointer contains the filename instead of a pointer.

```
struct __attribute__ ((packed)) GlobalAddress {
  size_t size; char *ptr; uint16_t id; bool isFile; }
```

Listing 1: The struct `GlobalAddress` to identify distributed main memory and files globally.

### 3.3 Distributed Main Memory

The cache coherence protocol manages distributed main memory uniformly, therefore each node provides access to its RAM and caches data of remote nodes. The cache coherence protocol for main memory is similarly implemented to the original proposal but allowing flexible page sizes. Each node acts as a server to answer read and write requests. Every task is bound to one server node, to which we will refer as the *home node*. This node routes all data requests for the specific task . The protocol includes the four basic operations `malloc`, `free`, `read` and `write` with additional locking mechanisms. For local accesses, the *home node* checks whether the chosen access method is compatible with the lock of the requested area. Requests for remote access are redirected to a remote node and the result is cached locally.
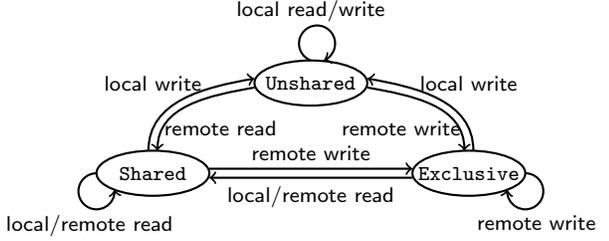
Figure 2: Transition between the different cache states.



(a) `shared`/`unshared`

(b) `exclusive`

Figure 3: Remote reads request the data from another node.



(a) Local write (`shared`).



(b) Remote write (`shared`/`exclusive`).

Figure 4: Writing: The home node requests sharing nodes to invalidate cached data.

### 3.3.1 Cache

Even though the throughput of networks using RDMA is comparably high and the latency is comparably low, remote memory access is ten times slower compared to local memory access [3]. This software cache aims to reduce the remote memory accesses as it reduces calls to lower memory layers and remote accesses. As a result, a cache coherence protocol is used for better performance and minimizing the necessary connections.

Depending on the data item, there are five roles a node can be assigned with. The *home node* is the node the data is physically stored on. All other nodes are *remote nodes*. The *request node* is the node that requests shared (read) or exclusive (write) access to the data. If this access is granted, the request node will become the *sharing node* (read) or *exclusive node* (write). In the beginning, the *home node* is responsible for the data. When a request node requires access to the data, which the *home node* grants, it becomes a *sharing* or an *exclusive node*. There can be multiple sharing nodes for one data item, but there at most one exclusive node.

The state of a cache item can be *unshared*, *shared* or *exclusive* (see Figure 2). *Unshared* means that the data resides in the home node. The state is *shared* when one or more nodes have read permission on that data. *Exclusive* means that one remote node has write permission on that data.

### 3.3.2 Read

Read is used to access data that is stored in the distributed system. The data's address is defined in a *global address* that is initially created when `malloc` is called. There are two read scenarios depending on where the data is stored. A read request that addresses data, which is stored in the request node itself, is referred to as *local read*, whereas a read request to a remote address is called a *remote read*.

*Local Read.* The data a local read wants to access can be either *unshared*, *shared* or *exclusive*. In all three cases, the data can be accessed right away. The status will remain the same, since we do not change the sharing nodes.

*Remote Read.* In the scenario of a remote read, the request node differs from the node where the data is stored on. If the cache of the request node already contains the required data, the cached copy will be simply returned without further inter-node communication.

Otherwise, the request node sends a read request to the associated remote node (see Figure 3a). In the case of the data status being *unshared*, the requested data is accessed on the remote node using the pointer stored in the global address. The status is set to *shared* and the data is returned to the initial request node. The request node stores the data in its cache in case it is needed again.
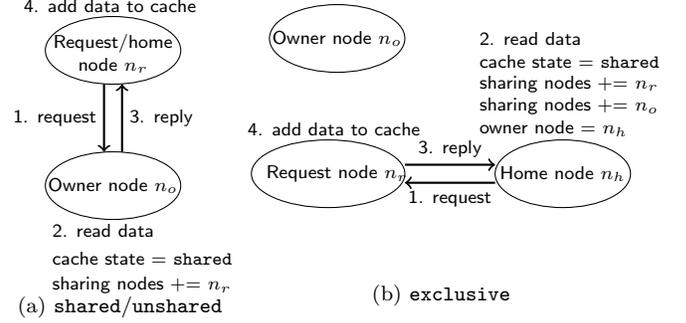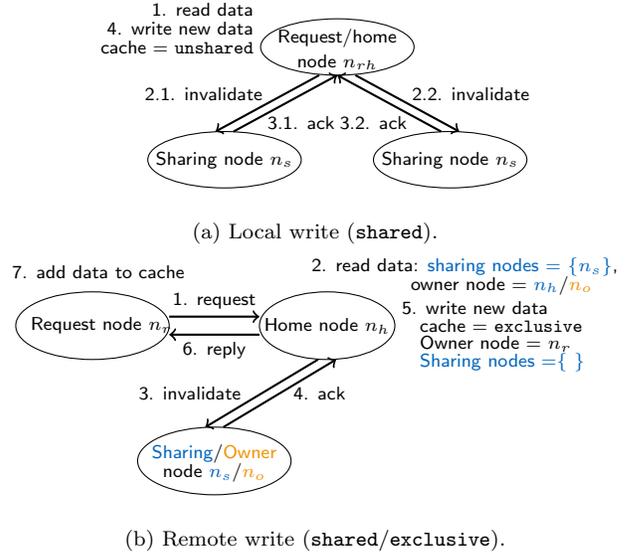
If the status of the data is shared, the data will be just returned to the request node. Other sharing nodes do not need to interact and the status remains the same. If the status is exclusive, the node will change the status to shared (see Figure 3b). Then the data will be returned to the request node. The node saves the data in its cache to ease further read requests.

### 3.3.3 Write

Write is used to save data to memory that was allocated before or to alternate existing data. It writes to a predefined address. Similar to *read*, there are two possible scenarios: *local write* (if the memory to access is located at the request node) and *remote write* (if the request node differs from the node where the address is at).

*Local Write.* Local write means that the addressed memory space is in the request node. In this case, it is necessary to check if the data is already cached by another node (see Figure 4a). If not, we can simply write the data and return the updated address. If other nodes have accessed the data and saved them in their cache, it is necessary to invalidate their cache. Therefore, the node sends invalidate requests to all sharing nodes. The remote node removes the correspond-

ing cached item and sends an acknowledgment as response. After all cached items are invalidated, the request node can perform the write in its memory and alter the data.

*Remote Write.* In case of a *remote write*, the memory is placed in a different node (see Figure 4b). There are three scenarios depending on the cache state of the accessed data. In all of the three, scenarios the request node connects to the home node and sends a write request. Furthermore, the new data is stored in the cache of the request node in case it needs to access it soon. Again, we have to distinguish between *unshared*, *shared* or *exclusive*.

In the *unshared* case, the request node sends a request to the *home node*. It checks the state for information and recognizes it as *unshared*. The data is stored at the corresponding address and the state is set to *exclusive*. Afterwards, the request node also saves the data in its cache.

The *shared* case is similar to the *unshared* scenario. The request node sends a request to the home node and checks the state. Indicating it as *shared*, the next step is to send a request to invalidate the cached item on each sharing node. After receiving an acknowledgment from each sharing node, the home node alters the data. It returns the updated *global address* to the request node, which also stores the changed data now as well in its cache.

In the third, *exclusive*, case, the data cannot be changed and the request will be denied.

### 3.3.4    LRU-based Cache

We use a least recently used (LRU) policy to manage the software cache, which caches data from remote nodes for faster access. Each cacheline is stored in a hash-table indexed by its global address. The size of a cacheline as well as the overall size are limited and depend on the hardware capabilities as not the whole main memory can be used for caching. If the data is too large for the cache in general, no data item will be removed from the cache and the data has to be loaded from the remote node on each new access.

In case of a `read` request, the cache of the *home node* is checked first. If the requested data is already stored, its content will be returned. Otherwise, we perform a *remote read* and store the returned data in the software cache. Also on a `write` request, when the global address points to a remote node, the data will be cached locally as well for future `read` requests.

## 3.4    Interface

For remote read or write requests, our cache coherence protocol provides an application interface (API) for data access out of the six functions `malloc`, `free`, `read`, `write`, `freadf` and `fprintf` (see Figure 5) and additional locking mechanisms.

### 3.4.1    Malloc

Similarly to the function of the standard library, *malloc* reserves memory and returns a *global address* as follows:

First the *home node*, on which the current program is running, our access point, tries to allocate the requested memory plus memory for caching and ownership on its own local storage. On success, when enough memory is available, we return the address of the allocated memory packed in a global address. Otherwise, the node aims to reserve memory in the storage of a remote node. Therefore, it sends a
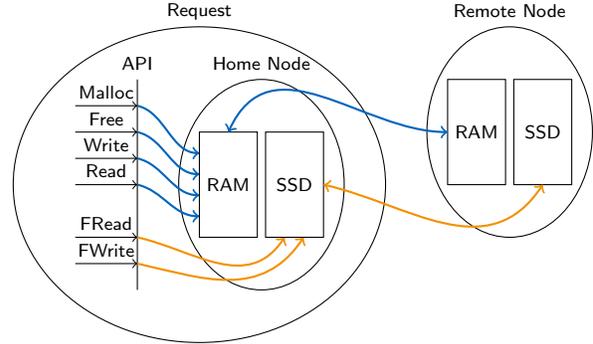


Figure 5: The implemented memory input/output primitives implemented by our cache coherence protocol.

request to another node with remaining memory capabilities. This remote node allocates the requested memory plus space for additional data. When the system does not have any available memory in the requested size, an exception is thrown, which has to be caught in a higher level application.

### 3.4.2    Free

This operation releases memory on a given *global address*, either managed by the node itself or a remote node.

If the address belongs to the *home node*, the saved data is examined to get information about the owner node and the sharing nodes. If the owner node is not the current one or sharing nodes exist, it is necessary to send invalidation requests to the sharing nodes and/or the owner node. When the invalidation is over, the initially sent pointer is freed.

If the given pointer is not located in the *home node*'s address space, the remote node, where the addressed memory resides, is identified. The *home node* requests the corresponding remote node to release the corresponding memory and to notify each sharing node to invalidate their cache entry.

### 3.4.3    Locks

To avoid race conditions on parallel read or write requests, the cache coherence protocol provides a distributed locking mechanism as part of its application interface. A lock is responsible for a hashed global address together with a state (either *shared*, *exclusive* or *unshared*) and the node identifier of the node holding the lock. One pre-defined *lock node* is responsible for locks across the distributed systems.

In case of a *read* request, a *shared lock* is created, in case of a `write` request, an *exclusive lock*. Multiple *shared locks* are allowed, as they define read-only accesses and are not compatible with an *exclusive lock* request. An *exclusive lock* can only be held by one single thread, which has exclusive rights to write on the locked data.

## 3.5    Distributed Persistent Storage

Additionally to the main memory, the project is expanded by the usage of a SSD storage. If main memory exceeds, this approach will allow to swap out pages to SSD.

### 3.5.1    FPrintF

The API function `FPrintF` is used to write data into a file. The function is called with four parameters: the new data, the global address, the size of the data and the offset in the file. First, it is checked if the address is situated on

the access node. If this is the case, the corresponding file is opened. If the size of the file is smaller than the size of the new data, the file is resized and the size saved in the global address is adjusted. The new data is written to the open file. Afterwards, the global address is returned.

If the file is not in the file system of the *access node*, the *access node* connects with the node storing the file and sends the data. The remote node writes the data to the file and returns the global address on success.

### 3.5.2  FReadF

The function `FReadF` is the second part of the implementation of the distributed file system. It enables to read a file. The function expects the global address of the file, the size of the data that should be read, the offset in the file and the pointer to a return buffer where the data should be stored, as parameters. Since the pointer of the return buffer is given as a parameter, the function has no return type.

When the global address points to a file on the *access node*, the data is read from the file and stored in the return buffer. Otherwise, the *access node* needs to connect to the associated *remote node* and send a request containing the order to read from the specified file. The remote node reads the file and copies the data into the send buffer. The access node copies the received data into the return buffer.

## 4.  DISTRIBUTED HASH TABLE

The first application is a simple key-value store. The design of a hash table was used for the implementation of this storage. A hash table is a data structure that can map keys to values. The data structure is split into a fixed amount of buckets. It uses a hash function on the key to compute an index used to identify the bucket in which the value is stored. In contrast to the implementation of a distributed hash table in *GAM* [3], which divided the key space and assigned each part to a different node, this study uses the provided API functions for allocation, storing and accessing data in the distributed hash table implementation.

The entries are stored in a vector in the access node that contains the global addresses of the values. If a new insert is executed, the hash table calls `Malloc` and gets the global address in return. This address is stored in the associated bucket, which is determined by hashing the key. The value is then written to the specified global address with the use of `Write`. On look-ups the hashed key identifies the associated global address and `Read` returns the data that is positioned at the address. If an item should be erased, the function `Free` releases the memory that is specified in the global address.

## 5.  DISTRIBUTED BUFFER MANAGER

The extended cache coherence protocol provides an abstraction to adapt higher-level applications such as a disk-based database system for distributed systems. For disk-based database systems, buffer management is essential to evict pages to background storage and to load pages into main memory. A buffer manager has a limited amount of frames in main memory and the number of pages depends on the size of background storage. When reading or writing a page, the page has to be loaded necessarily to main memory, thus fixed into a frame, and can be written out to disk when the amount of available frames is exceeded.
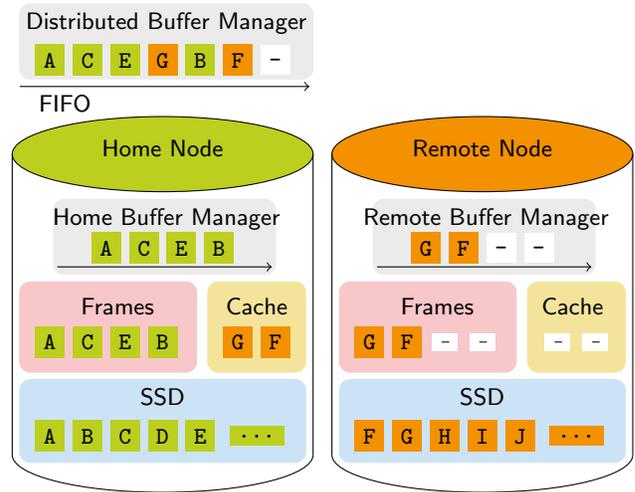


(a) One local buffer manager per node.
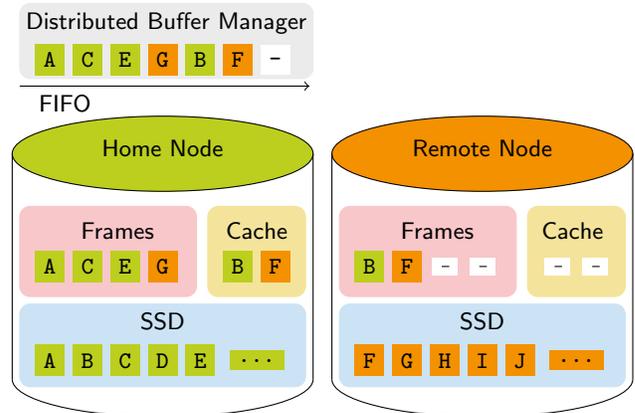


(b) One global buffer manager.

Figure 6: Design decision between (a) one local buffer manager per node and (b) one global buffer manager over two nodes with up to four frames in each node: When using local buffer managers, each node (un-)fixes pages locally. In contrast with RDMA `write`, it is possible to directly map remote background storage to local main memory.

When building a distributed buffer manager, traditionally, every node provides a local buffer manager and evicts pages to the local background storage (see Figure 6a). Hereby, the *global buffer manager* corresponds to the amount of local buffer managers.

As RDMA allows to bypass the CPU to access remote memory directly, evicting and loading pages from remote nodes is possible. So we have to decide between local buffer management per node, which replaces frames locally, or to implement a global buffer management. As our cache coherence provides an interface for abstractions of memory input/output primitives over a distributed network, we adjust a buffer manager to deal with frames on a distributed file system. As a result, every node has access to files on remote nodes and can load pages into local frames. Hereby, the *global buffer manager* corresponds implicitly to the amount of loaded frames (see Figure 6b), but no node has to manage pages locally.

The basic operations of the buffer manager are:

- `FIX(page_number, shared)` for loading the page and

- `UNFIX(page_number, dirty)`.

Pages can only be altered if they are fixed. One entity of a page is represented by a buffer frame, which contains the state of a specific page.

In this paper, the buffer is implemented in the form of a hash table. The previously described distributed hash table was used to enable a distributed buffer storage. A whole buffer frame is saved as the value with its page number as the key.

If the buffer is full, some pages need to be evicted to create space for new pages. If a page is clean, it can be discarded immediately. If it is dirty, the new data needs to be written back before the page can be replaced. To decide which page should be replaced first, different methods can be used. Two strategies are combined, hence two queues are used: First-In-First-Out (FIFO) and Least-Recently-Used (LRU). FIFO is a simple replacement strategy that uses a linked list. New items are added to the end and removal is done from the head, hence old items are removed first. LRU is similar to FIFO, but uses a double-linked list to store the items. The items are also removed from the head, but when an item is reused, it is moved to the end of the buffer.

The slots for buffer frames in the buffer manager are limited. As a result the buffer manager can become full and pages need to be evicted before other pages are loaded. In this implementation, a two queue replacement strategy is used. The pages are stored in the FIFO queue. If another reference to the page is made, the page is moved to the LRU queue. When a page gets evicted and is dirty, it is necessary to write the page's data to an associated file. These files are also stored in the distributed system. Therefore, the API function `FPrintF` is used to write to local and remote files. The same principle is used for loading pages: the function `FReadF` reads the requested data from the file defined in the global memory.

The locking, which enables multi-threading in the original buffer manager, had to be adjusted. This paper provides its own locking mechanism as described previously. This mechanism is used in the API functions.

## 6. EVALUATION

This section evaluates and discusses the distributed system. First, micro-benchmarks were performed on the different API functions. Afterwards, observations regarding the implementation of the applications are described and the performance is measured and discussed. For measuring, PerfEvent[1] was used, a C++ wrapper for Linux' performance events API. Two Ubuntu 18.04.03 LTS servers with two sockets and twenty cores of Intel Xeon E5-2660 v2 processors (supporting hyperthreading) were used. Each server has 256GiB of main memory and 1TiB of SSD as background storage, provides a Mellanox ConnectX–3 VPI NIC supporting FDR InfiniBand with 56 GBit/s, and is connected via a Mellanox SX6005 switch.

## 6.1 Runtime Analysis

First, `Malloc`, `Free`, `Read` and `Write` as functions for main memory are measured. Second, the functions `FPrintF` and `FReadF`, which are handling the file system, are benchmarked.
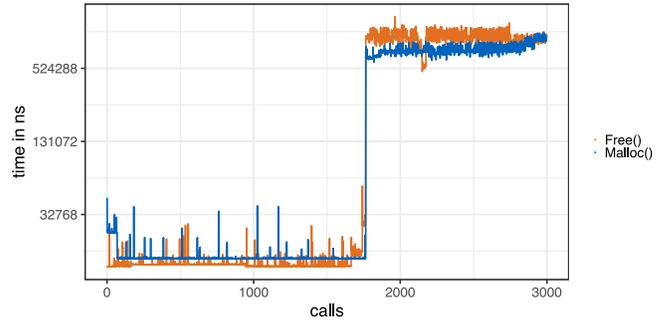
---
[1] `https://github.com/viktorleis/perfevent`



Figure 7: Time of `Malloc` and `Free` with a $log_2$ scale.

In this study, the time for each call is examined. The sent data size in this benchmark is set to the maximum block size, estimated during the implementation. For the main memory functions, three thousand calls were made. For the file system functions, one thousand calls were made. The time is measured in nanoseconds. Furthermore other measured values, e.g. cache-misses, were used to explain peculiarities. The tests were conducted on the side of the access node (client side), since the remote node (server side) is not calling the functions explicitly. The remote node only reacts to the messages received from the access node.

### 6.1.1 Micro-benchmark of Allocation and Freeing

To measure the allocation without filling the complete memory of the server, a limitation for the occupied memory on each node was set. It is also good practice to limit the available memory for the node in general, since the main memory also includes the software cache. In this measurement, the reason for the limitation is that the access node should be forced to allocate memory on a remote node.

The time of a `Malloc()` and `Free()` is shown in Figure 7. 3000 times the maximum block size (912B) was allocated in this benchmark. The duration of each allocation was measured. Afterwards, the three thousand addresses returned by the allocation process were freed again. Every `Free` call was measured as well.

Observing both graphs, there is a big jump shown at around 1800 calls. Examining `Malloc`, the reason for this jump is that the local memory is fully occupied. Hence, memory on a remote node is allocated and remote requests need to take place. Even though RDMA is used and the data can be written directly into the remote node, the data has to be transferred. In case of `Free`, the addresses previously allocated were partially on the access node and partially on the remote node. As a result, the same amount of remote requests as used in the allocation process need to take place. Although RDMA is much faster than standard TCP/IP, since it works directly over the NIC, remote accesses are still much slower than local accesses.

Furthermore, there are many peaks when looking at the local accesses separately. The reason can be seen in Figure 8a and Figure 8b. There are peaks at the same positions as in Figure 7. The calls that take significantly more time produce cache misses. Figure 8a shows misses on cache level one. Figure 8b shows the last level cache (LLC). In comparison to `Malloc`, `Free` takes more time on remote requests, but less time when it is called locally. This shows that `Free`

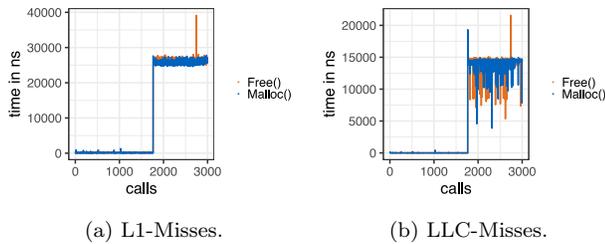(a) L1-Misses.  (b) LLC-Misses.

Figure 8: Cache misses on `Malloc` and `Free` operations.
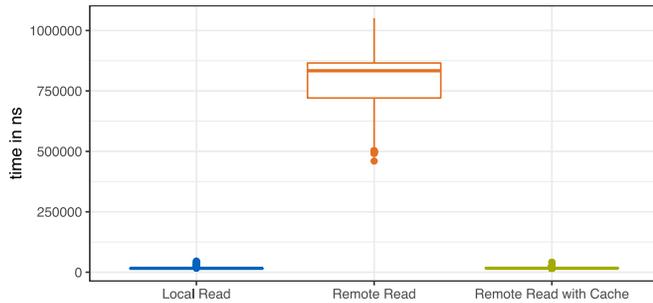


Figure 9: Time of `Read`.

has more irregularities when operating locally than `Malloc`.

### 6.1.2 Read

Similar to `Malloc` and `Free`, the function `Read` was benchmarked. Three different types of read were measured: local read, remote read and remote read with caching. The different results can be seen in Figure 9. The lock node was set to the access node in all three measurements. The mean and median of the three measurements are shown in Table 1 together with the maximum and minimum values. In every `Read` operation, a locking of the address is done before the memory look-up. Local read means that the address of the requested data points to memory situated in the access node. Remote read means the address points to a remote node. All remote reads include locking, connecting to the remote node, checking the data, sending back the data to the request node, releasing the lock and returning the data. In comparison, a *remote read* using the cache gets the data on the first request from the remote node. In all other reads, the access node already has cached the data. Therefore, it can access the software cache on a read request.

It can be observed that the idea of the software cache worked out. The `Read` using the software cache takes much less time than the basic *remote read*. This is useful especially for applications that use more look-ups than updates.

Table 1: Measured values of the different `Read` operations.

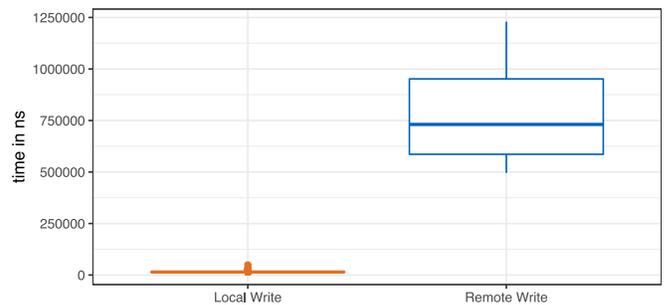|  | Local Read | Remote Read | |
|---|---|---|---|
|  |  | no Cache | with Cache |
| min | 15μs | 459μs | 16μs |
| mean | 16μs | 783μs | 16μs |
| median | 16μs | 833μs | 16μs |
| max | 46μs | 1051μs | 41μs |



Figure 10: Time of `Write`.

Another remarkable observation is that *local reads* take less time than reading from cache. One reason is that the remote read needs at least one "real" *remote read*, which adds time to the average duration. Another reason might be the implementation of the cache. In this study, a hash table, containing the different cache items, was used. The calculation of the hash to look-up the item can be time consuming.

Overall, `Read` is fast when performed locally or with the usage of a cache. In comparison, it would be much slower to do a `remote read` on every call. Since new data is cached when a remote write is performed, the remote reads can be minimized. Another aspect is that normally, only the access node is responsible for the data, since most applications run on one access node and do not share addresses of their memory with other nodes. Hence, the probability that the cache is often invalidated is low. A problem occurs when the size of frequently accessed data exceeds the cache. For example, if five items would fit into the cache but six different items were accessed subsequently, then the specific cache item would be replaced from the cache before accessed again. LRU would remove each corresponding item because of the oldest `last-used` timestamp and every data item would have to be accessed via *remote read* instead of using the cache. If this is the case in a desired application, the size of the cache, the size of a cache line or the cache replacement strategy have to be adjusted according to the application requirements.

### 6.1.3 Write

The operation `Write` was benchmarked in the same way as `Read`. Hence, *local write* and *remote write* were measured. The third benchmark using the cache is unnecessary since remote writes are always remote and the cache is only used to store data for further reads. The impact is very small and it is useful for `Reads`, so it was only measured with cache inserts. The results of the two different measurements are shown in Figure 10.

Similar to `Read`, the *local write* execution is much faster than the *remote write* execution. Furthermore, the variation of the duration of *remote writes* is huge. The minimum, maximum, mean and median values of the `Write` measurements are shown in table 2. In every `Write` operation, the address is locked first. Then the data is written. The measured *local writes* perform data changes on an address situated in the access node, whereas *remote writes* aim to write data that is located on a remote node.

It can be observed that a *remote write* needs much more

Table 2: Measured values of the different `Write` operations.

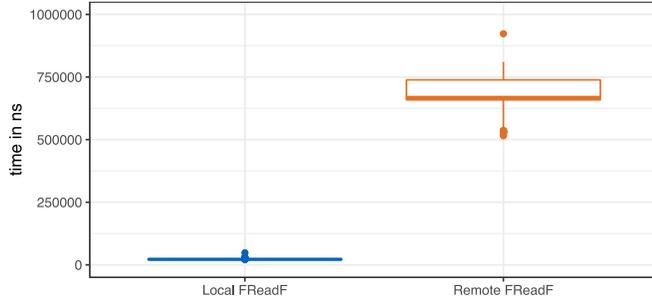|        | Local Write | Remote Write |
| ------ | ----------- | ------------ |
| min    | 14µs        | 495µs        |
| mean   | 14µs        | 771µs        |
| median | 14µs        | 730µs        |
| max    | 49µs        | 1229µs       |



Figure 11: Time of `FReadF`.

time to be executed than a *local write*. Furthermore, the *remote write* has a bigger variety of duration. In comparison to `Read`, the `Write` operation has a smaller mean and median. On local nodes, writes have a smaller minimum and maximum duration than reads. Though on remote nodes, writes in general need more time, which is reflected in larger minimum and maximum values.

### 6.1.4 FReadF and FPrintF

In addition to the main memory functions, the methods operating in the file system were measured as well. First, the benchmark of `FReadF` is analyzed. A local and a remote address are created and data is written to the specified files. Second, the fixed block size is read from the two files and every read is measured. The results of the `FReadF` benchmark are shown in Figure 11.

In the benchmark of `FPrintF`, a local address and a remote address are created as well. Then some data with the maximum packet size is saved in the files. Each save operation is measured. The results of this measurement can be seen in Figure 12.

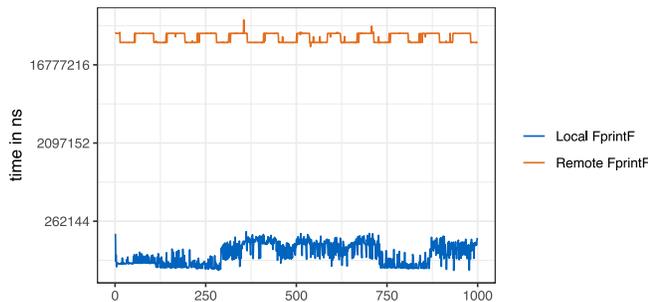The mean, median, minimum and maximum values of both benchmarks are shown in Table 3. Similar to the main



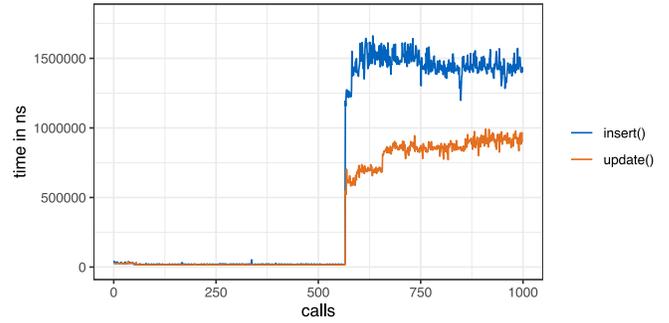Figure 12: Time of `FPrintF` with a $log_2$ scale.



Figure 13: Duration of inserts and updates.

memory API functions, the duration for remote operations is significantly larger than for the local ones. The duration received in the `FReadF` are similar to the `Read` results. The `FReadF` runtimes for the remote nodes are even slightly faster than the *remote reads*. This is quite surprising since the file is situated on a SSD and not in the RAM, which is normally faster. A reason could be that the caching mechanism of a *remote read* takes more time than just sending the data. In the file system no second level of cache is implemented. Hence, no further storing of the data, which has been remote read, is needed. Furthermore, the OS normally buffers the file I/O which leads to faster duration for the reading of newly written data.

In comparison, `FPrintF` is 50 times slower compared to the main memory `Writes`. As shown in Figure 12, the duration of remote `FPrintF` is mostly around two different values.

Overall, it can be observed that the remote `FReadF` could be a helpful add-on, whereas file data should be stored preferably in local storage. It makes sense to store data in a local file that can be accessed from a *remote node*. The storing of data in remote files is not recommended, since it takes a lot more time. One possible solution to reduce the duration of `FPrintF` could be to load data from a remote file and write adjusted data to the remote main memory. Then the *remote node* can write it back to the file. This results in a local `FPrintF` on the remote node, much faster than writing to a remote file.

## 6.2 Applications

Additionally to the measurements of the API functions, the implemented applications were measured, analyzed and discussed. First, the performance hash table is measured and analyzed. Second, the performance of the buffer manager is measured and discussed.

### 6.2.1 Hash Table

In this benchmark, thousand values were inserted into the hash table. Afterwards, every value was updated. The result of the `insert` and `update` operations are shown in Figure 13. The graph shows similar characteristics as the test of `Malloc` and `Free` in subsection 6.1.1. A big performance drop can be observed. In case of the `insert` operation, the drop can be explained by the fact that the predefined storage of the *access node* is exceeded. Hence, the following inserts allocate memory on a *remote node* and write to the specified address on the *remote node*.

`Update` has the same jump, but with a smaller amplitude.

Table 3: Measured values of local (left) and remote (right) `FReadF` and `FPrintF` operations.

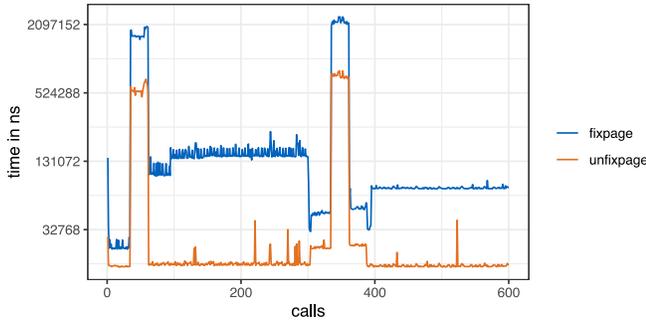| | Local FReadF | Remote FReadF | Local FPrintF | Remote FPrintF |
|---|---|---|---|---|
| min | 21µs | 459µs | 70µs | 27192µs |
| mean | 22µs | 495µs | 114µs | 34801µs |
| median | 22µs | 668µs | 106µs | 38531µs |
| max | 47µs | 922µs | 198µs | 55404µs |



Figure 14: Duration of the buffer manager with a $log_2$ scale.

The reason for this is that the `insert` function uses two separate remote requests. First, the needed memory is allocated by `Malloc`, then the value is written to the memory using `Write`. In case of `update`, the global address associated with the key is saved in a local storage structure. Then this address is used to simply write the new value to the previously allocated memory on a remote node. As a result, only one remote request is needed.

One aspect taking more time is that a queue pair is newly connected for each API call. The reason for this is that if the developed system contains more nodes than one, each API call could point to another *remote node*. Every call is directed by the global address, which is handed over by the function call. This address can point to every node in the system. Hence, every call pointing to a remote address needs to establish a connected queue pair with the node that is specified by the global address. Thus, the remote `update` takes approximately half the time of the remote `insert`.

Although this is not so obvious, the duration of local `inserts` is nearly two times the duration of local `updates`. This is also caused by the previously described reason. Even though local operations do not need to connect the queue pair, the duration of the local inserts is longer than the duration of the local updates. Furthermore, the amplitude of the `inserts` is bigger than the amplitude of the `updates`.

### 6.2.2 Buffer manager

A test for persistent restart was performed for the buffer manager. In this scenario, five segments with 60 pages (page size 912B) each were fixed and unfixed. Afterwards, the buffer manager gets destroyed and the five segments with 60 pages each were fixed and unfixed again. The time of each fix and unfix was measured (see Figure 14).

Two significant peaks can be seen. They identify the remote accesses in the hash table. The files where data is written back are stored on the access node, so they are all local. The size limits for local memory were set so that half of the pages are stored in the remote node. As a result,

it can be observed that half of the 60 first pages, which are fixed, are stored locally and the other half is stored remotely.

While each call of `unfix page`, except for the remote calls, takes mainly the same time, the method `fix page` has different stages. `Fix page` takes longer than `unfix page`, which can be explained by the fact that unfix only sets two variables of the page and stores it. `Fix page`, however, loads the page from file, evicts pages if needed or creates a new page. It can be observed that `fix page` does not return to the initial time (calls one to 30) before storing remotely (calls 31 to 60). At this point, the main storage for loaded pages of the buffer manager is completely occupied and pages need to get evicted and written to file. Opening a file and writing to it takes more time than simply creating a file or loading data from main memory.

At 300 calls, the buffer manager is reset and all pages must be reloaded from file. In this part, the data is loaded from the files. As shown above, `FReadF` has a similar duration as `Read`. Nevertheless, in the first run, the local creation of the pages takes less time than when the buffer manager is restarted. On eviction of the pages, the duration is smaller than before the restart. The reason for this could be that the files were already opened and the data in the file is only updated and not created.

One interesting aspect is that only the first inserts in the hash table initialize remote calls. This could be caused by the eviction algorithm, which only replaces local pages.

## 7. CONCLUSION

This paper has implemented and analyzed a distributed memory coherence protocol using RDMA. The developed distributed system implemented a unified global address space. The implementation includes an API that allows to operate on distributed main memory and on a distributed file system. The functions operating on main memory included allocation and freeing of addresses in the global address space, as well as a write and a read function. Furthermore, a function to write to files and a function to read from files are provided. The unified memory model is obtained by an abstraction from the RDMA network. A second layer of cache was added on top of the API. This aimed to reduce remote requests by storing the data locally for further reads.

It has been shown that remote operations using RDMA are slower than local operations. Hence, remote request should be reduced to a minimum. The usage of a cache reduced remote reads, which is a helpful step in reducing latency. The unified global address space can be used in the same way as normal local addresses. This is shown in two developed applications. The first application was a distributed hash table. This hash table uses the provided API functions to distribute the values in the system. The second application was a buffer manager, which reused the previously implemented hash table and combined it with the implemented API functions operating on the file system.

## 8. REFERENCES

[1] M. Banikazemi and B. Abali. Eucalyptus: Support for effective use of persistent memory. In *IPDPS Workshops*, pages 1152–1159. IEEE Computer Society, 2012.

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

[3] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K. Tan, Y. M. Teo, and S. Wang. Efficient distributed memory management with RDMA and caching. *PVLDB*, 11(11):1604–1617, 2018.

[4] Z. Chen, Y. Zhang, Y. Zhou, H. Scott, and B. Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *SIGMETRICS*, pages 145–156. ACM, 2005.

[5] W. Effelsberg and T. Härder. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, 1984.

[6] A. F. Farahani, D. Roberts, and N. Jayasena. Analytical study on bandwidth efficiency of heterogeneous memory systems. In *MEMSYS*, pages 104–118. ACM, 2016.

[7] P. Fent, A. van Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper. Low-latency communication for fast DBMS using RDMA and shared memory. In *ICDE*, pages 1477–1488. IEEE, 2020.

[8] N. Hubig, L. Passing, M. E. Schüle, D. Vorona, A. Kemper, and T. Neumann. Hyperinsight: Data exploration deep inside hyper. In *CIKM*, pages 2467–2470. ACM, 2017.

[9] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur-Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached design on high performance RDMA capable interconnects. In *ICPP*, pages 743–752. IEEE Computer Society, 2011.

[10] L. Karnowski, M. E. Schüle, A. Kemper, and T. Neumann. Umbra as a time machine. In *BTW*, volume P-311 of *LNI*, pages 123–132. Gesellschaft für Informatik, Bonn, 2021.

[11] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *SIGCOMM*, pages 297–312. ACM, 2018.

[12] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA. In *SIGMOD Conference*, pages 355–370. ACM, 2016.

[13] F. Liu, L. Yin, and S. Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. *ACM Trans. Database Syst.*, 44(4):17:1–17:45, 2019.

[14] X. Liu, Y. Hua, X. Li, and Q. Liu. Write-optimized and consistent rdma-based NVM systems. *CoRR*, abs/1906.08173, 2019.

[15] P. Memarzia, S. Ray, and V. C. Bhavsar. The art of efficient in-memory query processing on NUMA systems: a systematic approach. In *ICDE*, pages 781–792. IEEE, 2020.

[16] S. T. On, Y. Li, B. He, M. Wu, Q. Luo, and J. Xu. Fd-buffer: a buffer manager for databases on flash disks. In *CIKM*, pages 1297–1300. ACM, 2010.

[17] I. L. Picoli, N. Hedam, P. Bonnet, and P. Tözün. Open-channel SSD (what is it good for). In *CIDR*. www.cidrdb.org, 2020.

[18] D. Porobic, P. Tözün, R. Appuswamy, and A. Ailamaki. More than a network: distributed OLTP on clusters of hardware islands. In *DaMoN*, pages 6:1–6:8. ACM, 2016.

[19] J. Schmeißer, M. E. Schüle, V. Leis, T. Neumann, and A. Kemper. B$^2$-tree: Cache-friendly string indexing within b-trees. In *BTW*, volume P-311 of *LNI*, pages 39–58. Gesellschaft für Informatik, Bonn, 2021.

[20] M. E. Schüle, M. Bungeroth, D. Vorona, A. Kemper, S. Günnemann, and T. Neumann. ML2SQL - compiling a declarative machine learning language to SQL and python. In *EDBT*, pages 562–565. OpenProceedings.org, 2019.

[21] M. E. Schüle, J. Huber, A. Kemper, and T. Neumann. Freedom for the sql-lambda: Just-in-time-compiling user-injected functions in postgresql. In *SSDBM*, pages 6:1–6:12. ACM, 2020.

[22] M. E. Schüle, L. Karnowski, J. Schmeißer, B. Kleiner, A. Kemper, and T. Neumann. Versioning in main-memory database systems: From musaeusdb to tardisdb. In *SSDBM*, pages 169–180. ACM, 2019.

[23] M. E. Schüle, P. Schliski, T. Hutzelmann, T. Rosenberger, V. Leis, D. Vorona, A. Kemper, and T. Neumann. Monopedia: Staying single is good enough - the hyper way for web scale applications. *PVLDB*, 10(12):1921–1924, 2017.

[24] M. E. Schüle, D. Vorona, L. Passing, H. Lang, A. Kemper, S. Günnemann, and T. Neumann. The power of SQL lambda functions. In *EDBT*, pages 534–537. OpenProceedings.org, 2019.

[25] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. Managing non-volatile memory in database systems. In *SIGMOD Conference*, pages 1541–1555. ACM, 2018.

[26] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. Building blocks for persistent memory. *VLDB J.*, 29(6):1223–1241, 2020.

[27] J. Yang, J. Izraelevitz, and S. Swanson. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *FAST*, pages 221–234. USENIX Association, 2019.

[28] D. Y. Yoon, M. Chowdhury, and B. Mozafari. Distributed lock management with RDMA: decentralization without starvation. In *SIGMOD Conference*, pages 1571–1586. ACM, 2018.

[29] T. Ziegler, S. T. Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. In *SIGMOD Conference*, pages 741–758. ACM, 2019.