The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation

Ruihong Wang
Purdue University
wang4996@purdue.edu

Jianguo Wang
Purdue University
csjgwang@purdue.edu

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

M. Tamer Özsu
University of Waterloo
tamer.ozsu@uwaterloo.ca

Walid G. Aref
Purdue University
aref@purdue.edu

ABSTRACT

Memory disaggregation (MD) allows for scalable and elastic data center design by separating compute (CPU) from memory. With MD, compute and memory are no longer coupled into the same server box. Instead, they are connected to each other via ultra-fast networking such as RDMA. MD can bring many advantages, e.g., higher memory utilization, better independent scaling (of compute and memory), and lower cost of ownership. This paper makes the case that MD can fuel the next wave of innovation on database systems. We observe that MD revives the great debate of "shared what" in the database community. We envision that *distributed shared-memory databases (DSM-DB, for short)* – that have not received much attention before – can be promising in the future with MD. We present a list of challenges and opportunities that can inspire next steps in system design making the case for DSM-DB.

PVLDB Reference Format:
Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. PVLDB, 16(1): 15-22, 2022.
doi:10.14778/3561261.3561263

1 INTRODUCTION

Memory Disaggregation, MD for short, is emerging as a promising architecture in modern data centers, especially in the cloud [17, 71, 72, 74]. MD enables data center design based on independent pools of *compute nodes* and *memory nodes* that are physically separated but are connected via ultra-fast RDMA networks [17, 71, 72, 74]. A critical enabler for MD is hardware advances, especially networking technology. RDMA, e.g., Mellanox Connectx-6 [6], achieves 0.8 μsec latency and 200Gb/s throughput – close to local memory performance though there is still a gap.

Overall, this is in contrast to traditional data centers that consist of a collection of monolithic "converged" servers, where compute and memory are tightly coupled in the same physical servers. With MD, compute nodes focus on computation while memory nodes are dedicated for provisioning memory.[1] The compute nodes may include small amounts of memory and memory nodes may have some compute capability to run simple control software. Thus, a compute node has strong computing power (e.g., 100s of CPU cores) but limited local memory (e.g., a few GBs) while a memory node has weak computing capability (e.g., a few CPU cores) but abundant memory (e.g., 100s of GBs) to store data [71, 72].

MD brings many advantages for data centers [1, 17, 71, 72, 74]. (1) MD results in higher memory utilization with less fragmentation due to memory pooling. This translates into lower memory consumption and lower total cost of ownership (TCO) as memory is still an expensive resource. (2) MD provides independent elastic scaling of compute and memory, which is very useful in the cloud. It allows users to request instances with arbitrary combinations of compute and memory that existing monolithic servers fail to provide. Also, compute or memory can be elastically adjusted with workload changes. (3) MD achieves better reliability and lower operational cost because compute and memory failures and upgrades are independent, and do not affect each other. (4) MD provides users virtually a near-infinite pool of memory for applications.

For all the above reasons, MD is receiving increasingly more attention from industry. For example, PolarDB, a cloud-native DBMS, relies on MD to improve memory utilization, and supports independent scaling [17, 74]. Microsoft Azure has started to build an MD system in the cloud to improve memory utilization [39]. IBM Cloud provides MD in cloud data centers to significantly reduce cost, and achieve better reliability [1]. Intel RSD (Rack Scale Design) [5] and HP "The Machine" [32] also support MD at rack scale.

Position. We argue that the next wave in database system innovation should be shared-memory designs enabled by RDMA-based MD. Similar to previous design evolution in decoupling system components in favor of scalability and elasticity, we believe that MD is the key to move database design to the next frontier. We present the challenges and opportunities in realizing distributed shared-memory databases (DSM-DB) with MD. This vision paper focuses on OLTP main-memory databases for high performance.

2 VISION AND CONTRIBUTIONS

Novelty and Architectural Evolution. DSM-DB architectures have been proposed in the 1980s [55]. However, slow networking at the time has made these architectures infeasible due to the slow access of remote memory. Thus, existing distributed DBMSs are

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:10.14778/3561261.3561263

[1]With storage disaggregation, there are also dedicated storage nodes, but this paper focuses on memory disaggregation.

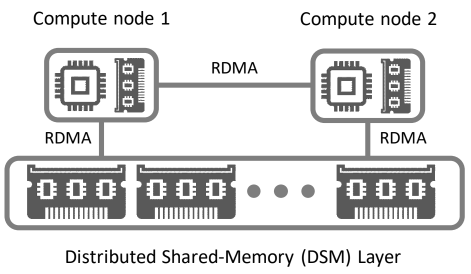

Figure 1: DSM-DB with memory disaggregation (MD)

mostly shared-nothing, or recently, shared-storage architectures. For decades, the shared-nothing architecture has been regarded as the "gold standard" in distributed DBMSs due to their high performance especially in supporting single-shard queries [55, 56]. Examples include MySQL Cluster, PostgreSQL Citus [20], Teradata, MemSQL [18], VoltDB [8], SQL Server PDW, and Greenplum. As the cloud becomes prevalent, shared-storage DBMSs start to gain attention because they can best leverage the cloud infrastructure of storage disaggregation. Examples include Amazon Aurora [61], Google AlloyDB [2], Alibaba PolarDB [16], Alibaba AnalyticDB [68], Microsoft Socrates [11], Microsoft Polaris [10], Huawei Taurus [23], and Snowflake [21]. The shared-storage architecture is compelling in the cloud because it can support independent scaling of compute and storage, better elasticity, and fast crash recovery – all are important design considerations for the cloud. However, the shared-storage architecture still suffers from high memory consumption and high cost because compute and memory remain tightly coupled. This vision paper is a natural step forward arguing for a DSM-DB architecture driven by MD and ultra-fast RDMA networking.

Distributed Shared-Memory Database Architecture (DSM-DB) . Figure 1 shows the DSM-DB architecture that separates compute and memory nodes. Memory nodes form a distributed shared-memory (DSM) layer that is shared by compute nodes via an ultra-fast RDMA network. Compute nodes have high computing power with limited local memory while memory nodes have large memory capacity with limited computing power. Compute nodes can communicate with each other via RDMA. DSM-DB is an OLTP distributed main-memory DBMS that stores data in the DSM layer with hot data being cached in the compute nodes' local memories.

Expected Benefits. DSM-DB has many advantages over distributed shared-storage or shared-nothing databases. (1) DSM-DB has independent elasticity of compute and memory due to MD. (2) DSM-DB incurs lower total cost of ownership by increasing memory utilization as the majority of data is stored in the distributed shared-memory while each compute node only keeps small amounts of local private memory. (3) DSM-DB supports independent failure of compute and memory nodes to achieve high availability. (4) DSM-DB is more robust to query and data skew that are difficult to handle in distributed shared-nothing databases as data can be easily resharded in DSM. (5) DSM-DB can achieve better scalability with multi-masters (Sec. 4) that existing distributed shared-storage databases would not have. Note that the performance issues in DSM-DB due to MD can be mitigated (see Sec. 7).

Contributions. This paper presents a list of challenges in realizing DSM-DB. (1) *DSM Layer* (Sec. 3): This layer needs to provide high durability and availability at a low cost. Also, it needs to expose a rich API to the DBMS. (2) *Concurrency Control* (Sec. 4): Multi-node concurrency control is challenging due to the lack of cache-coherence across compute nodes. (3) *Buffer Management* (Sec. 5): Managing data movement between local and remote memory requires rethinking how to leverage fast RDMA networking; otherwise, due to the relatively high speeds of these layers in contrast to disk, the software layer can become the new bottleneck. (4) *Index Design* (Sec. 6): It is non-trivial to develop indexes that fully exploit RDMA, and that support highly-concurrent accesses.

3 DISTRIBUTED SHARED-MEMORY

The goal for having distributed shared-memory (DSM) in DSM-DB is to manage a cluster of memory nodes (each provisioning large memory) and provide a unified memory space with the necessary APIs for DBMSs to build on. Compute nodes access DSM via these APIs. DSM-DB introduces a DSM layer to encapsulate all memory management details and hide them from compute nodes for two reasons. First, it enables independent elasticity of compute vs. memory. Second, it simplifies system design because each compute node sees a unified infinite memory space and only focuses on query processing logic inside a compute node without worrying about the complicated memory management underneath.

Challenge #1: Exposing Abstract APIs. What APIs best support DBMSs? The APIs must include not only basic memory access APIs, e.g., memory allocation, deallocation, read, and write, but also database functions for near-data computing. Thus, DSM will provide the following APIs to support database applications.

Memory Allocation APIs. DSM will provide memory allocation APIs similar to those in a single memory node, e.g., memory allocation, deallocation, and reallocation. However, **memory address representation** is challenging as it cannot be the physical address allocated via a programming language, e.g., `malloc` in C++. If a memory node crashes then recovers, the memory space changes and the old address cannot refer to the new memory. Thus, the memory address must be a logical address, e.g., virtual node ID and offset. To allocate memory efficiently and reduce memory fragmentation, DSM-DB can allocate a giant continuous memory space and keep track of memory usage in user space [58].

Data Transmission APIs. These APIs provide one- and two-sided RDMA, memory access (read/write), and atomic operations for concurrency.

Function Offloading APIs. These APIs will push down certain database functions (e.g., [73]) to memory nodes that leverage the computing resources in DSM to reduce data transfer.

Challenge #2: Durability. A single memory node is volatile. DSM must be durable to ensure that committed data is not lost. At a minimum, a crash in a single memory node must not cause data loss. Upon transaction commit, logs must be written to persistent storage. However, log persistence needs to be highly-performant at low cost so that it is not the bottleneck in main-memory databases. The following are possible directions to be explored.

Approach #1. One possibility is to write logs to durable storage as in main-memory databases [27]. DSM-DB can choose cloud storage, e.g., AWS EBS and S3 are highly reliable with low cost, and can achieve 99.999% and 99.999999999% durability, respectively [3, 4].

Cloud storage can be viewed as distributed shared storage that is accessible by all compute and memory nodes. Crash recovery is similar to that in main-memory databases [27]. However, writing to cloud storage is relatively slow and is on the critical path for transaction commit. The same problem arises in main-memory databases [27]. Thus, similar optimizations need to be revisited for DSM-DB, e.g., group commit [24, 28], command logging [42], logging only base data but not indexes [25]. For instance, command logging in DSM-DB cannot rebuild the same states upon crash because with multi-master, the system may not be able to determine the global transaction order in advance.

Approach #2. Another possibility is to follow RAMCloud [50] that uses memory replication to emulate durable storage. It writes a log synchronously to k different memory nodes ($k = 3$ in RAMCloud) and a log write is considered "persistent" if all k memory nodes successfully write the log to their own main-memories. If one node crashes, a new node may be identified and is restored from the $(k-1)$ replicated nodes. Compared to Approach #1, log persistence is fast as it does not involve disk. But it may not guarantee 100% durability as the probability of all k memory nodes crashing is not zero, and this could lead to data loss. Remedies could be battery-backed memory [28], persistent memory [12], or SSDs [76]. More research is needed to investigate the interaction with RDMA.

Challenge #3: Availability. Main-memory is volatile and DSM-DB can become unusable upon crash. The goal is to achieve reasonable availability to minimize downtime while taking a low (monetary) cost. A simple solution is to replicate data in different memory nodes to support high availability. This consumes memory, and hence is expensive. Another solution is to use erasure code [35, 53] but the recovery process is long if there is a crash. The third solution is to follow the RAMCloud approach [50] that stores data pages in main-memory only once to reduce memory consumption. To improve availability, RAMCloud periodically checkpoints data pages from memory nodes to persistent store (this can be cloud storage in DSM-DB). If a memory node crashes, its content can be recovered by accessing the persistent store and possibly replaying some of the logs. More research is required to speedup crash recovery since accessing cloud-based persistent storage is relatively slow.

Existing Research. Early works on DSM, e.g., [33, 40, 46, 52], do not target MD. In those works, all nodes are homogeneous and compute and memory nodes are not differentiated. Also, they do not use RDMA and do not support durability and availability. Existing distributed memory systems, e.g., GAM [15], NAM [14, 67], FarM [26], Redy [69], and Memcached [7] do not provide durability/availability, or do not provide database-specific functions.

RAMCloud [50] is memory-based and offers durability and availability but may not be used as the DSM layer in DSM-DB. First, RAMCloud has no memory APIs (e.g., memory allocation and memory read/write). Instead, it has key-value APIs [49]. Second, RAMCloud does not provide database-specific functions, e.g., offloading. Third, RAMCloud assumes TCP/IP rather than RDMA networking.

4 CONCURRENCY CONTROL

Since the compute nodes share access to the memory nodes in DSM-DB, their concurrent accesses to the memory pool need to be protected through a scalable concurrency control (CC) protocol. There are a few new challenges when compared to the CC protocols in the multi-core architecture, e.g., [13, 57, 64, 66].

Challenge #4: The Cache Coherence Challenge. In DSM-DB, there is no hardware-level cache coherence among the compute nodes. If a compute node updates a data item, another compute node may not see the update immediately. This is different from the conventional single-server multi-core architecture (with different CPU cores sharing the memory) because hardware-level cache coherence among different CPU cores is natively provided.

Cache coherence is important as it affects the design decision on whether or not to use the local buffer memory in compute nodes. If local buffers are used, then cache coherence needs to be addressed at the software-level, which adds performance overhead. Otherwise, the cache coherence problem is bypassed at the expense of more remote accesses as the compute nodes will have to always access data from remote memory. A related design decision is whether or not to allow different compute nodes to access disjoint data partitions, i.e., sharding. With sharding, the cache coherence issue can be avoided when every data page is accessed by a single compute node. The following three approaches to address the cache coherence challenge need to be systematically evaluated.

Approach #1: No Cache, No Sharding (Figure 2a). Recent RDMA-optimized CC techniques [62, 63, 67, 77] follow this approach. A compute node reads and writes data remotely and does not store any data in local memory. Data is stored in DSM with a lock per data item. Compute nodes use RDMA Compare & Swap (CAS) to acquire a lock before accessing data. There is no cache coherence issue as no local data exists in a compute node. Because a compute node always accesses data remotely, this incurs performance overhead.

Approach #2: Cache, No Sharding (Figure 2b). Compute nodes leverage local memory to perform reads and writes. This may create cache coherence issues when two compute nodes update the same data locally. To resolve conflicts, a software-level cache coherence protocol, e.g., [15, 17], is needed to broadcast changes made by a compute node. However, the effect of software overhead is unclear as many implementation details can affect performance, e.g., invalidation- vs. update-based, one- vs. two-sided RDMA, fetching missed data from neighboring compute nodes or from the shared-memory layer. To reduce cache coherence overhead, a lazy cache coherence protocol can trade mutual consistency for performance.

Approach #3: Cache, Sharding (Figure 2c). In this approach, we perform *logical sharding*, where each compute node maintains sharding information (e.g., range information) of the data it is responsible for. CC is similar to the case in distributed shared-nothing databases, e.g., [29]. However, the difference is that a compute node does not store the entire data shard because of the limited local memory although it can cache some hot data. The advantage is that there is no cache coherence issue due to sharding. Also, this approach can best leverage local memory. Another advantage is that elasticity can be supported very well, e.g., if a new compute node is added, only the metadata (e.g., range information) is copied into the new node without physically moving data immediately (due to logical sharding), and the obsolete data from the old compute nodes can be recycled asynchronously. The downside is in cross-shard transactions. However, this can be alleviated via dynamic resharding [9, 41] that is efficient in DSM-DB since the DSM layer can transfer data quickly.

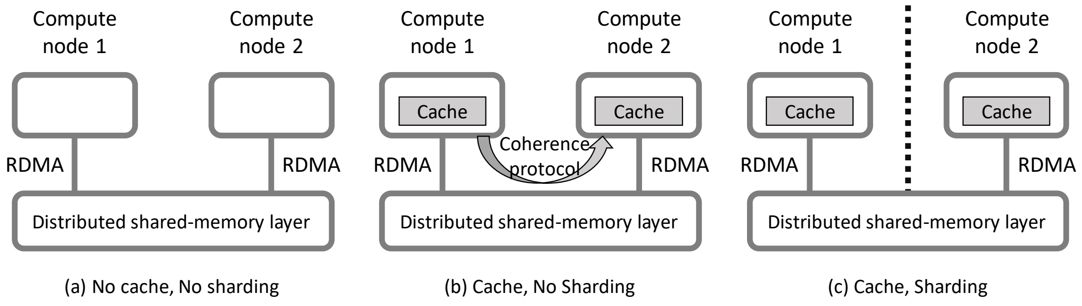

Figure 2: Concurrency control design tradeoffs in DSM-DB

Challenge #5: Rethinking Distributed Commit. Are the distributed commit protocols (e.g., 2PC) [51] – widely used in distributed shared-nothing databases – still applicable in DSM-DB? Observe that distributed commit may not always be relevant in DSM-DB depending on which architecture in Figure 2 is adopted. If DSM-DB uses a no-sharding architecture (Figure 2a or Figure 2b), there is no need for distributed commit. If there is no sharding between different compute nodes, every compute node can read/write all the data. Thus, every transaction could be executed by a single compute node, which does not require distributed commit.

In contrast, if DSM-DB uses sharding (Figure 2c), distributed commit may become relevant. Each compute node is responsible for a shard of data pages and a transaction may access pages belonging to different compute nodes. More research is needed to leverage RDMA primitives in distributed commit protocols. Notice that sharding data among memory nodes does not necessarily mandate using 2PC. If a compute node uses one-sided RDMA to access memory nodes, it knows whether a write is successful or not.

Challenge #6: Revisiting Existing CC Protocols. In DSM-DB, compute nodes access remote memory via RDMA. It is costly to implement locks over RDMA (compared with the conventional local locks). This has implications to both lock-based and non-lock-based CC protocols that need to be revisited in DSM-DB.

Lock-based CC. Lock-based concurrency control protocols, e.g., 2PL, rely on fine-grained locks to improve the level of concurrency, e.g., shared-exclusive locks and intention locks. It is challenging to implement these lock variants on RDMA and the implementation overhead varies. RDMA can only implement a simple exclusive spinlock *within a single round trip* through the CAS atomic primitive. Advanced lock types require more RDMA round trips, e.g., an RDMA shared-exclusive lock needs at least 2 round trips. Thus, it is unclear whether advanced lock types could be used for lock-based CC algorithms. It remains open if the allowed extra concurrency can offset the performance overhead of the advanced locks.

Non-lock-based CC. RDMA can also impact non-lock-based concurrency control protocols, e.g., timestamp-based CC, MVCC, and optimistic CC though in a mild way. These protocols need latches using RDMA to exclusively access some shared states, e.g., global timestamps. Another related optimization is how to generate timestamps. One-sided RDMA (RDMA Fetch & Add) is more preferable than two-sided RDMA in case that the centralized timestamp generator becomes a bottleneck. It is interesting to investigate other approaches (e.g., vector timestamp [67] and clock synchronization [62]). A systematic evaluation of different concurrency control protocols over RDMA is necessary.

Challenge #7: Supporting Massive Concurrency. Prior work has examined concurrency control for 1000s of cores per compute node [13, 57, 64, 66]. DSM-DB can enable ultra-high number of cores (e.g., millions) accessing the same shared-memory by supporting many compute nodes (e.g., 10s to 1000s). It would be interesting to re-evaluate and re-think CC protocols to support extreme concurrency, e.g., millions of cores. This may require distinguishing the local concurrency control (within the same compute node) and global concurrency control (across different compute nodes).

Existing Research. There are a number of existing works in concurrency control that are relevant to DSM-DB.

Concurrency Control in Multi-Core Databases. Most CC work, e.g., 2PL, MVCC, and OCC are designed for multi-core setups [13, 57, 64, 66]. They share the implicit assumption of having hardware-supported cache coherence, which does not exist in DSM-DB.

Concurrency Control in Distributed Shared-Storage Databases (DSS-DB). DSS-DBs, e.g., Aurora [61], PolarDB [16], Socrates [11], and Taurus [23] do not support concurrent transactions among multiple compute nodes in order to avoid conflicts. Instead, only the primary node can support writes (aka single-writer) while all the other nodes are replicas for read-only transactions. However, DSM-DB can support multi-writers where every compute node supports writes to improve write scalability.

Concurrency Control in Distributed Shared-Nothing Databases (DSN-DB). Concurrency control is extensively studied in DSN-DBs [19, 29, 59]. While it is possible to adapt these techniques into DSM-DB by sharding the data among different compute nodes, there are two differences in DSM-DB that may inspire innovation. (1) A compute node does not physically store the entire data shard due to the limited local memory. (2) Data/state can be moved quickly among compute nodes via the DSM layer.

Other Works. PolarDB considers MD [17, 74] but only the primary node can write data. There are concurrency control protocols optimized for RDMA, e.g., [62, 63, 67, 77], but they do not leverage local memory in order to bypass the cache coherence issue.

5 BUFFER MANAGEMENT

Accessing local memory in a compute node is still faster than accessing RDMA-enabled remote memory in DSM. Thus, it makes sense to cache hot data in the limited local memory (of compute nodes) to minimize remote memory accesses. But there are unique challenges for the buffer management in DSM-DB.

Challenge #8: Designing Light-weight Buffer Management. Existing buffer management is optimized for the hierarchy where there exists a huge performance gap between a cache hit and miss, e.g., the latency gap between main-memory and disk can be 100,000×. Thus, the goal of existing buffer management is to improve the cache hit rates by developing optimized techniques, e.g., optimizing buffer replacement policies [30, 44, 47, 54] and storing compressed pages in the buffer [45]. In DSM-DB, we need to rethink buffer management because the performance gap between local and remote memory is significantly narrowed, e.g., down to 10× or less, due to fast RDMA networking. Thus, we need to focus on the actual running time instead of just cache hit rates. That is because, software overhead, e.g., lookup cost, maintenance cost to reorganize buffer contents (in, say LRU), and synchronization cost due to multi-threaded access may become the performance bottlenecks for fast RDMA. These have traditionally not been major concerns for slower devices, e.g., SSDs or HDDs. Thus, research is needed to evaluate the overhead of popular buffer management policies, e.g., LRU, LRU-K [47], 2Q [31], CLOCK, and ARC [44]. New buffer management policies (e.g., [76]) must consider actual running time instead of purely optimizing cache hit rates.

Buffer management in DSM-DB is different from the buffer management optimized for the hierarchy of local memory and local persistent memory (PM) [60, 75], because CPU can directly operate on data stored on the local PM if there is a cache miss. However, in DSM-DB, data must be transferred from remote memory to local memory first before being accessed if a cache miss happens.

Another direction is to evaluate the effectiveness of caching compressed pages. Depending on different data types and compression techniques, decompression overhead might even be higher than directly fetching uncompressed data from remote memory. Thus, light-weight compression is important for DSM-DB.

Challenge #9: Caching vs. Offloading. Caching and offloading are two popular techniques to improve the performance in disaggregated databases. A recent study [65] shows that caching and offloading are not orthogonal to each other on storage-disaggregated databases with conventional TCP/IP networking [65]. It is important to re-investigate their interaction within DSM-DB for the following reasons: (1) DSM-DB uses RDMA that may make caching more favorable. Intuitively, if network latency is zero, it is favorable to bring data from remote memory to local memory upon a cache miss because compute nodes have better compute power. (2) The analytical model in [65] has high software overhead to decide when to cache or offload, while DSM-DB requires a light-weight model. (3) DSM-DB targets OLTP applications that involve updates, which can incur inconsistencies between the cached data and the underlying data (due to offloading).

Existing Research. Existing work on MD has not focused on the challenges of buffer management mentioned above because they still use disk-based buffer management for MD, e.g., [17, 71, 72, 74]. There are other works on using remote memory as a buffer of disk, e.g., [38, 70]. However, this work focuses on the buffer management for the two-level hierarchy with local memory and remote memory (without involving disk) where all the data is stored in remote memory with hot data being cached in local memory.

6 INDEX DESIGN

RDMA-based MD also has profound implications on index design for DSM-DB that we highlight in this section.

Challenge #10: RDMA-Conscious Index Design. Index design needs to be hardware conscious to truly achieve good performance. All state of the art indexes used in modern systems heavily rely on hardware conscious design, e.g., Bw-tree [37], Masstree [43], ART [36], and LSM-tree [22, 48]. In DSM-DB, compute nodes access remote memory, i.e., the DSM layer, via RDMA. The intrinsic properties of RDMA networking need to be at the core of index design. There are numerous new hardware related design factors to explore that do not always have an equivalent context with past hardware properties. These factors include: (1) Which RDMA primitive to use, e.g., one- or two-sided RDMA, synchronous or asynchronous RDMA; (2) How to best utilize the limited buffer memory in each compute node; (3) How to reduce software overhead to best leverage high-performance RDMA; and (4) How to exploit the computing capability of memory nodes to reduce data transfer.

These design choices are not independent, e.g., having bigger local memory gives preference to using one-sided RDMA in favor of fewer round trips. Also, using near-data computing impacts the way of using buffers. Another direction is to design indexes that adaptively balance available compute, memory, and RDMA resources to prevent imbalanced resource utilization, e.g., a compute node's CPU is fully saturated while a memory node's CPU or RDMA bandwidth are largely idle.

While we may able to utilize and build on past research in many cases, in other cases it might be necessary to reconsider and drop design choices that are considered state of the art. For example, if we can access remote nodes extremely fast, then approaches that rely on sketches and summaries to filter data remotely may prove less impactful in this context.

Challenge #11: Concurrent Index Operations. How to handle concurrent accesses (reads/writes) from a very large number of compute nodes? We expect a stronger requirement for concurrency with MD because more nodes (and thus more queries) will have access to the same memory on shared data copies. Although existing indexes support multi-threaded access via lock-based or lock-free designs, e.g., Bw-tree [37], it is unclear how they perform in DSM-DB due to (1) Expensive RDMA locking overhead; and (2) No hardware-supported cache coherence. Also, for lock-based approaches, e.g., ART [36], it is unclear what lock types to use. Thus, we need to design new concurrent indexes that optimize for multiple compute nodes accessing DSM via RDMA.

Besides that, LSM-based indexing [22, 48] can be worth investigating because it naturally fits the local memory and remote memory hierarchy. For example, LSM-trees can hold filters and fence pointers in compute nodes as they help protect from unnecessary round trips. More research is needed to reduce software overhead and leverage the compute capabilities of memory nodes, e.g., offloading LSM compaction to memory nodes.

Existing Research. Sherman [63] is an optimized B-tree for MD. It uses one-sided RDMA to access remote memory, and addresses concurrent accesses using RDMA-based exclusive locks and version validation. To reduce network round trips (due to one-sided RDMA), Sherman [63] caches all internal nodes into local memory, which consumes more memory. Moreover, Sherman does not leverage the compute resources in memory nodes. Ziegler et al. propose an RDMA-optimized B-tree structure that spans multiple memory nodes [77], but it does not target MD and does not use the compute nodes' local memory. RACE is a hash index for MD [78] but it only uses one-sided RDMA. It implements a lock-free multi-node CC protocol for the hash buckets. Overall, while there are a few prior works targeting MD, there is still big room for improvement because existing works have not fully leveraged RDMA characteristics.

7 DISCUSSION

Performance. The performance of DSM-DB due to MD can be mitigated by several approaches. (1) Use more local memory in each compute node. As demonstrated in [74], caching 50% data in local memory achieves almost no performance drop. Obviously, there is a tradeoff between more local memory capacity and memory utilization. But MD introduces the flexibility in controlling local memory size and can break the memory capacity limit of a single server. (2) Optimize buffer management as in Sec. 5. (3) Send only logs (i.e., log-as-the-database) as in Aurora [61]. (4) Leverage near-data computing in the shared-memory layer to reduce data movement [73].

Distributed Shared-Nothing vs. DSM. For the main-memory DBMSs considered in this paper, there are the DSN-DBs and DSM-DBs choices, so what is the difference between the two, really?

It is true that DSN-DBs can benefit from fast RDMA to reduce network communication cost, but maybe not too much because DSN-DBs are purposely designed for slow networks by carefully developing techniques to localize transactions and query processing to minimize network data accesses. With RDMA, one can relax the network accesses by allowing every node in DSN-DBs to access the main-memory of every node, which somehow mimics DSM accesses. However, that is not the DSM-DB this paper is advocating for because that does not support memory elasticity and independent memory scaling. In order to do so, we need to have asymmetric architecture of "compute nodes" and "memory nodes" with different compute and memory capabilities. Also, the memory nodes form a DSM layer, which becomes DSM-DB's proposal. A benchmark that systematically compares the DSN-DBs and DSM-DBs is required to best understand the two designs. We believe that DSM-DBs can better leverage fast RDMA networking than DSN-DBs.

Observe that DSN-DBs and DSM-DBs are not incompatible. For large-scale applications that require cross data-center deployment, DSM-DBs alone would not work because RDMA is not applicable due to the long latency dominated by speed-of-light delays among data-centers. Thus, a hybrid design that combines shared-memory and shared-nothing is required with shared-memory within the same data center and shared-nothing across data centers.

8 RELATED WORK

Distributed Shared-Storage Databases (DSS-DB). DSM-DB has the potential to address several issues that are found challenging in DSS-DBs, e.g., Aurora [61], Socrates [11], PolarDB [16], and Taurus [23]. (1) DSM-DB can address the multi-master issue – a challenging issue in DSS-DBs – by leveraging fast RDMA and the DSM layer to quickly synchronize the states between compute nodes. Note that although PolarDB uses RDMA [16], it does not efficiently support multi-master because a compute node cannot use one-sided RDMA to access the storage nodes. (2) DSM-DB can easily support memory elasticity, independent scaling and failure of compute and memory that are not possible in DSS-DBs.

Distributed Shared-Nothing Databases (DSN-DB). In addition to supporting memory elasticity and independent scaling, DSM-DB has the potential to efficiently address the distributed transaction issues in DSN-DBs (e.g., VoltDB [8], MemSQL [18], and Hekaton [25]), because the RDMA-connected DSM layer in DSM-DB provides a fast way to reshard data among compute nodes. This makes DSM-DB more resilient to skew due to fast resharding.

Distributed Main-Memory Databases. DSM-DB is different from existing distributed main-memory databases (e.g., VoltDB [8], MemSQL [18], and Hekaton [25]) because DSM-DB is shared-memory while these databases are shared-nothing. Simply adding RDMA networking to these shared-nothing main-memory databases does not solve the memory elasticity and independent scaling problems. It is required to have separated notions of "compute nodes" and "memory nodes" and allow compute nodes to access memory nodes via abstract APIs, which is DSM-DB's proposal.

NAM (Network-Attached Memory) [14, 67]. NAM is an innovative architecture that allows compute nodes to access a shared-memory pool of memory nodes. However, NAM is not designed for MD. The compute and memory nodes in NAM are *logically decoupled*, while DSM-DB emphasizes *physical decoupling* due to MD. Although logical decoupling has the potential to co-locate compute and memory nodes (where each "node" is a process) to reduce network access, physical decoupling enables additional benefits, e.g., independent failure and crash handling of compute and memory nodes, and better resource utilization and elasticity. Besides that, this vision paper has a wider scope that also includes durability, availability, DSM APIs, buffer management, and index design.

Impact of MD on Databases. Recent research investigates the impact of MD on databases, e.g., [34, 71–73]. These works target single-node databases (with a single compute node and a single memory node) instead of a distributed DBMS as in DSM-DB. Also, they focus on OLAP databases while DSM-DB focuses on OLTP databases. PolarDB incorporates MD [17, 74], but is still a disk-based DBMS with a single master node. In contrast, DSM-DB is main-memory-based that supports multi-masters, where every compute node can process read/write requests to improve scalability.

9 CONCLUSION

This paper presents our vision on the impact of MD to distributed databases, in particular OLTP main-memory databases. We envision that the distributed shared-memory (DSM) architecture that has been under-appreciated in the past can be promising in the future due to MD. This paper highlights new problems and challenges in realizing DSM-DB with memory disaggregation over RDMA.

ACKNOWLEDGMENTS

Walid G. Aref acknowledges the support of the U.S. National Science Foundation under Grant Numbers III-1815796 and IIS-1910216. M. Tamer Özsu's research is supported by a grant from NSERC Canada.

REFERENCES

[1] [n.d.]. Advancing Cloud with Memory Disaggregation, https://www.ibm.com/blogs/research/2018/01/advancing-cloud-memory-disaggregation/.

[2] [n.d.]. AlloyDB for PostgreSQL, https://cloud.google.com/alloydb.

[3] [n.d.]. Amazon EBS, https://aws.amazon.com/ebs/features/.

[4] [n.d.]. Amazon S3, https://aws.amazon.com/pm/serv-s3/.

[5] [n.d.]. Intel RSD. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html

[6] [n.d.]. Mellanox Connectx-6, https://www.nvidia.com/en-us/networking/ethernet/connectx-6/.

[7] [n.d.]. Memcached, https://memcached.org/.

[8] [n.d.]. VoltDB, https://www.voltdb.com/.

[9] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. DynaMast: Adaptive Dynamic Mastering for Replicated Systems. In *International Conference on Data Engineering (ICDE)*. 1381–1392.

[10] Josep Aguilar-Saborit and Raghu Ramakrishnan. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *Proceedings of the VLDB Endowment (PVLDB)* 13, 12 (2020), 3204–3216.

[11] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1743–1756.

[12] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *Proceedings of the VLDB Endowment (PVLDB)* 10, 4 (2016), 337–348.

[13] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. 2020. The Tale of 1000 Cores: An Evaluation of Concurrency Control on Real(ly) Large Multi-socket Hardware. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*. 3:1–3:9.

[14] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proceedings of the VLDB Endowment (PVLDB)* 9, 7 (2016), 528–539.

[15] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proceedings of the VLDB Endowment (PVLDB)* 11, 11 (2018), 1604–1617.

[16] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-Low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proceedings of the VLDB Endowment (PVLDB)* 11, 12 (2018), 1849–1862.

[17] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 2477–2489.

[18] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. 2016. The MemSQL Query Optimizer: A Modern Optimizer for Real-time Analytics in a Distributed Database. *Proceedings of the VLDB Endowment (PVLDB)* 9, 13 (2016), 1401–1412.

[19] James A. Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In *USENIX Annual Technical Conference (ATC)*. 223–235.

[20] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed PostgreSQL for Data-Intensive Applications. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 2490–2502.

[21] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 215–226.

[22] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 79–94.

[23] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1463–1478.

[24] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1–8.

[25] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1243–1254.

[26] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 401–414.

[27] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin J. Levandoski, Thomas Neumann, and Andrew Pavlo. 2017. Main Memory Database Systems. *Foundations and Trends in Databases* 8, 1-2 (2017), 1–130.

[28] Hector Garcia-Molina and Kenneth Salem. 1992. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 4, 6 (1992), 509–516.

[29] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proceedings of the VLDB Endowment (PVLDB)* 10, 5 (2017), 553–564.

[30] Song Jiang and Xiaodong Zhang. 2002. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 31–42.

[31] Theodore Johnson and Dennis E. Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *International Conference on Very Large Data Bases (VLDB)*. 439–450.

[32] Kimberly Keeton. 2017. Memory-Driven Computing. In *USENIX Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/sites/default/files/conference/protected-files/fast17_slides_keeton.pdf

[33] Peter J. Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX Winter Technical Conference*. 115–132.

[34] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *Conference on Innovative Data Systems Research (CIDR)*.

[35] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. 2022. Hydra: Resilient and Highly Available Remote Memory. In *USENIX Conference on File and Storage Technologies (FAST)*. 181–197.

[36] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *International Conference on Data Engineering (ICDE)*. 38–49.

[37] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *International Conference on Data Engineering (ICDE)*. 302–313.

[38] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 355–370.

[39] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2022. First-generation Memory Disaggregation for Cloud Platforms. *CoRR* abs/2203.00241 (2022).

[40] Kai Li and Paul Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems (TOCS)* 7, 4 (1989), 321–359.

[41] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1659–1674.

[42] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking Main Memory OLTP Recovery. In *International Conference on Data Engineering (ICDE)*. 604–615.

[43] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *European Conference on Computer Systems (EuroSys)*. 183–196.

[44] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *USENIX Conference on File and Storage Technologies (FAST)*. 115 – 130.

[45] Sanjay Mishra. 2009. Data Compression: Strategy, Capacity Planning and Best Practices, https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008/dd894051(v=sql.100).

[46] Bill Nitzberg and Virginia Mary Lo. 1991. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer* 24, 8 (1991), 52–60.

[47] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 297–306.

[48] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[49] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John K. Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 29–41.

[50] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2009. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review* 43, 4 (2009), 92–105.

[51] M. Tamer Özsu and Patrick Valduriez. 2014. *Distributed and Parallel Database Systems, Third Edition.* CRC Press.

[52] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. 1996. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel & Distributed Technology: Systems & Applications* 4, 2 (1996), 63–71.

[53] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris S. Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment (PVLDB)* 6, 5 (2013), 325–336.

[54] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In *Proceedings of the International Symposium on Microarchitecture (MICRO).* 413–425.

[55] Michael Stonebraker. 1986. The Case for Shared Nothing. *IEEE Data Engineering Bulletin* 9, 1 (1986), 4–9.

[56] Michael Stonebraker. 2011. Shared-nothing vs Shared-disk, https://www.youtube.com/watch?v=G-o2bFd91Sw. In *Extremely Large Databases Workshop (XLDB).*

[57] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. 2020. An Analysis of Concurrency Control Protocols for In-Memory Database with CCBench. *Proceedings of the VLDB Endowment (PVLDB)* 13, 13 (2020), 3531–3544.

[58] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. 2021. CoRM: Compactable Remote Memory over RDMA. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* 1811–1824.

[59] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* 1–12.

[60] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* 1541–1555.

[61] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* 1041–1052.

[62] Chao Wang and Xuehai Qian. 2021. RDMA-enabled Concurrency Control Protocols for Transactions in the Cloud Era. *IEEE Transactions on Cloud Computing* (2021).

[63] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* 1033–1048.

[64] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *Proceedings of the VLDB Endowment (PVLDB)* 10, 2 (2016), 49–60.

[65] Yifei Yang, Matt Youill, Matthew E. Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *Proceedings of the VLDB Endowment (PVLDB)* 14, 11 (2021), 2101–2113.

[66] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proceedings of the VLDB Endowment (PVLDB)* 8, 3 (2014), 209–220.

[67] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2017. The End of a Myth: Distributed Transaction Can Scale. *Proceedings of the VLDB Endowment (PVLDB)* 10, 6 (2017), 685–696.

[68] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: Real-time OLAP Database System at Alibaba Cloud. *Proceedings of the VLDB Endowment (PVLDB)* 12, 12 (2019), 2059–2070.

[69] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. 2022. Redy: Remote Dynamic Memory Cache. *Proceedings of the VLDB Endowment (PVLDB)* 15, 4 (2022), 766 – 779.

[70] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. 2022. CompuCache: Remote Computable Caching using Spot VMs. In *Conference on Innovative Data Systems Research (CIDR).*

[71] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers. In *Conference on Innovative Data Systems Research (CIDR).*

[72] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proceedings of the VLDB Endowment (PVLDB)* 13, 9 (2020), 1568–1581.

[73] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2022. Optimizing Data-intensive Systems in Disaggregated Data Centers with TELEPORT. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* 1345–1359.

[74] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proceedings of the VLDB Endowment (PVLDB)* 14, 10 (2021), 1900–1912.

[75] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* 2195–2207.

[76] 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 ACM International Conference on Management of Data (SIGMOD).* 685–699.

[77] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* 741–758.

[78] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *USENIX Annual Technical Conference (ATC).* 15–29.