Rethinking Data Management Systems for Disaggregated Data Centers

Qizhen Zhang, Yifan Cai*, Sebastian Angel, Ang Chen†, Vincent Liu and Boon Thau Loo
*University of Pennsylvania, *Shanghai Jiao Tong University, †Rice University*

ABSTRACT

One recent trend of cloud data center design is *resource disaggregation*. Instead of having server units with "converged" compute, memory, and storage resources, a disaggregated data center (DDC) has pools of resources of each type connected via a network. While the systems community has been investigating the research challenges of DDC by designing new OS and network stacks, the implications of DDC for next-generation database systems remain unclear. In this paper, we take a first step towards understanding how DDCs might affect the design of relational databases, discuss the potential advantages and drawbacks in the context of data processing, and outline research challenges in addressing them.

1. INTRODUCTION

Over the past few decades, we have witnessed a number of hardware inflection points that required rethinking the design of databases. An early example was the transition of relational database systems (RDBMSs) from mainframes to networks of workstations [11, 17]. Since then, we have seen the rise of multicore machines, GPUs and FPGAs that augment existing compute resources, and recent interest in non-volatile memory.

In each of the these cases, the hardware enabled RDBMSs to improve their performance, scalability, and/or reliability. We believe that we are approaching a new inflection point. One that is fundamentally different from past ones because the change in hardware is likely to *harm*—rather than improve—performance for RDBMSs. This is the case with the *disaggregation* of cloud data center resources [27, 13, 25, 29, 30, 26, 38].

Traditionally, data center resources are arranged in the form of monolithic "converged" servers, each of which contains a small amount of compute, memory, and storage that can be used to process independent jobs or slices of jobs. In contrast, each resource node in a disaggregated data center (DDC) is physically distinct: some nodes are dedicated to processing, others to memory or storage. All of these nodes, regardless of type, might include a small amount of processing and memory to run simple control software, but these resources are ancillary. Instead, processing nodes will continually "page" memory from remote nodes into and out of its small on-board working set, write chunks to remote disks, or farm out tasks to remote GPUs. To each program running in this environment, the system provides the illusion of a near-infinite pool of any resource.

The operational benefits of disaggregation to data center operators are vast. First, DDCs allow operators to upgrade and expand each resource independently. For instance, if a new processor technology becomes available or if the workload changes to require additional CPUs, the operator can deploy additional compute nodes without needing to upgrade memory nodes or worrying about compatibility between different components. Second, DDCs promote efficient resource utilization and prevents fragmentation. For example, if a customer requests an unusual hardware configuration—such as 7 cores, 100 GB RAM, 3 GPUs—the operator can allocate those resources without committing an overprovisioned machine; to fulfill this request today, the closest option in AWS is a p3.8xlarge instance with 32 cores, 244 GB of RAM, and 4 GPUs.

Despite the above advantages, disaggregation is not without its challenges. What used to be a local memory access is now network communication. How do we handle the high latencies and the need to move large volumes of data between compute and memory nodes? How do we deal with a likely trend toward relatively smaller and smaller working set memory? How do we address the fact that memory can fail independently of the CPU or that different parts of memory can fail independently of each other?

While disaggregated storage [10, 33, 9] and remote memory [18, 28, 14, 37] are well-studied fields, the complete disaggregation of resources presents a novel set of questions and concerns. One distinction is that, unlike traditional remote memory systems where the remote memory is treated as extra cache, disaggregation is typically accompanied by a corresponding decrease in local memory—remote access becomes a necessity rather than an optimization. Second and related, in DDCs, these accesses are mediated by the operating system and network infrastructure rather than controlled by the application. This means that the interactions between each layer of the stack are critical to the system's overall performance.

In this paper, we lay out a series of challenges to database design that are unique to DDC architectures, and present some preliminary benchmark results that urge the redesign of RDBMSs in DDCs: naïve query execution on DDCs results in order-of-magnitude worse performance compared to running the same query on one of today's monolithic servers!

Many within the systems community have already started redesigning OSes [38], data structures [8], and network stacks [39]. We argue that more is required: we need the help of databases and data processing frameworks to fill in many of the performance gaps. To that end, we outline ways in which disaggregation affects database systems and take a first step toward rethinking RDBMS

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2020.
10th Biennial Conference on Innovative Data Systems Research (CIDR '20) January 12-15, 2020 , Amsterdam, Netherlands.

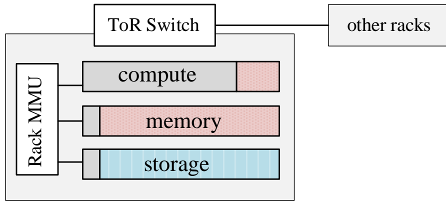

Figure 1: A proposed DDC architecture. Racks consist of blades with compute, memory, and storage elements connected through a Rack MMU. Compute elements have some local memory and caches, while memory and storage elements are fronted by a CPU that mediates accesses. Network communication between compute elements within or across racks uses the Top-of-Rack switch.

design. We focus on the use of traditional RDBMSs before expanding the discussion to other big data applications.

Specifically, we make the following key contributions:

- **RDBMSs in DDCs.** We provide an introduction to disaggregated architectures, and explore how existing RDBMSs might be deployed in a DDC. We consider single and parallel RDBMSs.
- **Needs and challenges.** RDBMSs have high data transfer bandwidth requirements from storage to memory, and from memory to compute. Using a case study of a single and a parallel join operator, we demonstrate how a naive implementation can result in multiple redundant round-trips of memory to memory copies. We validate the actual performance degradation by running query execution on LegoOS [38], which is an OS designed for disaggregated environments.
- **Performance optimizations.** Based on our preliminary findings, we propose new hardware and OS primitives that RDBMSs can use to perform memory copies more efficiently. These primitives are inspired by decades of work in *near data processing* [35, 34, 41, 23] where the memory has some small computational ability that can be leveraged for significant gains. For example, we propose new mechanisms that bypass the compute nodes entirely when partitioning data in preparation for a hash join. We show that these primitives (in conjunction with modifications to the database execution engine) apply to different parallel relational operators and can reduce the overheads introduced by DDCs.
- **Fault tolerance.** We explore how RDBMSs can benefit from the resource decoupling of DDCs to improve fault tolerance.

2. THE FUTURE IS DISAGGREGATED

Disaggregated data centers (DDCs) [27, 13, 25, 29, 30, 26, 38] propose splitting up compute (CPUs, GPUs, FPGAs), memory, storage, and other resources into *resource pools* that are stitched together by a fast interconnect. This is in contrast to existing architectures in which all of these resources are housed within a single monolithic system (a "server"). A key driver for DDCs is that they afford the provider significantly more flexibility by (1) allowing the provider to evolve each resource pool independently and (2) achieving better utilization and lower fragmentation.

While there are several competing proposals for how to architect a DDC, most have trended towards *rack-level disaggregation*, i.e., the placement of associated compute and memory in the same physical rack for lower latency and easier management. Storage and other resources can be placed either in the same rack or other racks for fault tolerance in the case of a rack-level failure. We inherit these general assumptions in this work, although our concerns (and many of our solutions) apply to other architectures as well.

Figure 1 depicts one example of a typical proposed DDC architecture. Again, slight variants exist. There are several core components to this architecture: individual blades with *compute elements*, *memory elements*, and *storage elements* connected over a low-latency *resource interconnect*. For ease of reference we call this interconnect the 'Rack MMU'. Each rack also has a Top-of-Rack (ToR) switch that allows compute elements in the same rack to communicate with each other and elements in other racks. We describe each of these components in more detail below.

Compute, memory, and storage elements. Compute elements consist of commodity processors with their associated memory hierarchy (including private and shared caches) and a small amount of local memory. This local memory is used primarily for the OS and as another cache to improve performance [20, 38].

Memory elements are composed of a dense array of DRAM or NVRAM chips, which are typically accompanied by a small computing element (processor, RNIC, FPGA, ASIC, etc.) that proxies communication with the Rack MMU and converts network requests into memory reads and writes. This processor interacts with memory through a standard MMU, and is responsible for addressing and access control. Storage elements are structured similarly.

OSes and processes. Traditionally, the OS is responsible for resource partitioning, isolation, and sharing as it is the component with global visibility across all resources. In a disaggregated environment, it no longer has that visibility; instead, the responsibility for this functionality is split between the Rack MMU and the local OSes. The OS at compute elements continues to be responsible for managing the underlying hardware, providing local scheduling and isolation, and presenting a standard programming interface to applications. The OS is also responsible for transparently fetching and writing back data from remote memory elements and managing the contents and coherence of the local memory [24, 38].

We assume the same resource sharing policy as LegoOS [38]: processes may share the same memory element, but not the same memory (so there is no shared memory). Furthermore, a compute element can host multiple processes, but a process can only run in a single compute element. This makes caching easier; otherwise, shared memory would require coherence across the local memories of different compute elements.

Resource interconnect. In Figure 1, the Rack MMU is abstracted as a single box. In reality, it is a complex system that may consist of a network of switches. There are a few proposals for its design:

(1) Packet switching: Compute elements interact with memory and storage by sending packets over a network of switches. The primary benefit of this approach is that all of its components—the Ethernet switches, RNICS, and Ethernet links—are readily available and commoditized. Compared to the other proposals, packet switches also typically have lower latency for small individual memory requests, and higher utilization efficiency. Further, recent advances in programmable switches are an easy platform on top of which to implement the Rack MMU logic.

(2) Circuit switching: In circuit switching, compute elements communicate with memory and storage via dedicated circuits [26, 39]. Researchers have argued that for the large port counts and throughput requirements of rack disaggregation, packet switches will eventually become too demanding for typical rack-level power budgets. These physical requirements have led to the exploration of simpler circuit switches, which transmit optical or electrical signals at the physical layer rather than parsing, processing, and buffering

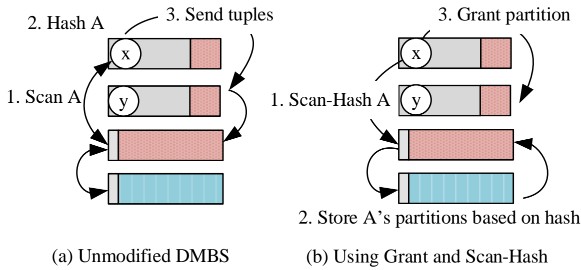

Figure 2: Figure (a) depicts a hash partitioning when the DBMS is running on LegoOS. Compute node x scans table A from remote storage by first attempting to fetch A from remote memory. When that fetch fails, the memory forwards that request to the remote storage. A is then scanned into the remote memory, and then into x's local memory. At this point, x can compute a hash on the join key, and determine the partitions. x then sends some of the partitions over the network (via ToR) to y, who is then responsible for those partitions. This forces y to copy those tuples from its local memory to its remote memory. Figure (b) shows the same operation but with additional primitives (§4).

packets. Scheduling, setting up, and tearing down circuits impose a performance cost to circuit switching, but systems like Shoal [39] propose potential solutions to compensate.

(3) Direct connect: Finally, compute elements can be connected directly to memory and storage nodes (e.g., using a 3D Torus network [16]), eliminating switches entirely. Direct connect topologies are cheap, and in some cases, also efficient and low-latency. Unfortunately, these properties depend on the provided workload as some messages may need to traverse multiple other nodes before reaching their destination. In this proposal, the Rack MMU may be a logically centralized—yet physically distributed—component.

Regardless of how the Rack MMU is instantiated, we assume that any compute element can access any memory element, and that accesses can be reconfigured at runtime at fine granularity.

3. ARE WE UP TO THE CHALLENGE?

To demonstrate the challenges brought by disaggregation, we describe the result of running an unmodified RDBMS on top of LegoOS [38], which is a state-of-the-art OS for DDCs.

The query execution engines would run on the powerful processors of the compute nodes. The code of the query optimizer and execution engine, as well as the execution plan and buffer pools, would preferably be placed on the compute nodes as well. Unfortunately, since the local memory of the compute nodes are limited, much of this would need to spill out into the remote memory nodes, with a preference for keeping the execution engine and execution plan local. Storage nodes keep any persistent data, and fetching data from them is recursive: compute nodes will attempt to read from memory, and memory nodes will fetch from storage on a miss.

We note that, conceptually, this architecture resembles that of a shared-nothing parallel DBMS. The primary difference is that the memory—and particularly the buffer pools—while logically independent, are physically hosted on the same memory nodes.

3.1 Single Join Operation

Now consider a simple single-pass, single-machine join on the above architecture. Assume two tables, A and B, are joined using a traditional hash join. Also assume that A is the bigger of the two tables and that we have an existing hash index of B built using the join key. In a traditional RDBMS, we would scan table A, compute a hash on the join attribute for each tuple in A, and probe the hash index of B for matches, returning the matching tuples.

In a DDC, this operation would proceed as follows.

- **Initial scan of A.** To perform the initial scan, a query engine on a compute node will have to first request table A's blocks from storage. As mentioned, this process is recursive, involving multiple rounds of communication until the requested blocks are transferred over to the local memory of the compute node. If local memory is scarce, the compute node may need to fetch a single block at a time; this is inefficient, but correct, as the algorithm requires only a single block of A as its working set.
- **Probing B's hash index.** When a block of A is in the compute node, the query processor iterates through every A-tuple to probe B's hash index. Again, a portion of B's hash index needs to be fetched from disk to remote memory, and then brought into the local memory of the query processor. Recent work [8] shows that fetching entries from a hash table stored in "far memory" is particularly expensive because of hash collisions that require multiple round trips between the compute node and remote memory/storage to traverse the hash table's buckets. While such collisions also exist in today's systems, the overhead in is typically dominated by disk I/O.

The join operation leads to more inefficiencies if the tables of A or B are too large to even fit in remote memory, requiring multiple passes over the data with either sort-merge or grace hash join.

3.2 Exchange and Symmetric Hash Join

We next consider the case of a parallel join operation. Since the local join operation is similar to the single-machine hash join described above, we focus on the *exchange operator* [4, 22] followed by a parallel pipelined symmetric hash join. Recall that the hash exchange operator repartitions the data by hashing the join attributes. Consider a join of tables A and B executed in parallel on a number of compute nodes, each running a query processor with its own local memory. Assume that each compute node is responsible for a subset of the blocks in each table.

- **Initial scan.** In the initial scan, each compute node does a scan of its assigned A and B blocks. As above, data needs to be transferred from storage to remote memory, then from remote to local, potentially over multiple iterations if cache is limited.
- **Parallel hash partitioning.** Each compute node iterates through A and B tuples in its local memory and applies a hash function on the join attribute value to determine the *destination* node performing the join. The repartitioned tuples (by join key) are then pushed to the destination node, who stores them in local memory. If needed (e.g., due to insufficient local memory), the destination node may need to copy these tuples *out* to remote memory before it can receive more tuples. This process is bandwidth intensive, as shown in Figure 2(a).
- **Pipelined symmetric hash join.** Finally, each destination compute node performs a pipelined symmetric hash join in which a local-memory hash table is built for each of the partitions, and incoming A and B tuples are probed against the hash tables. If the local memory is insufficient, the in-memory hash tables may also need to be continually fetched from and evicted to remote memory. In this case, there is another round trip to construct the hash tables in local memory (from the rehashed tuples that arrive in the previous step) and then transfer them to remote memory.

In each of the above steps, data is repeatedly transferred between storage, remote, and local memory—all to end up storing the data back in remote memory anyway! In Section 4 we show that with

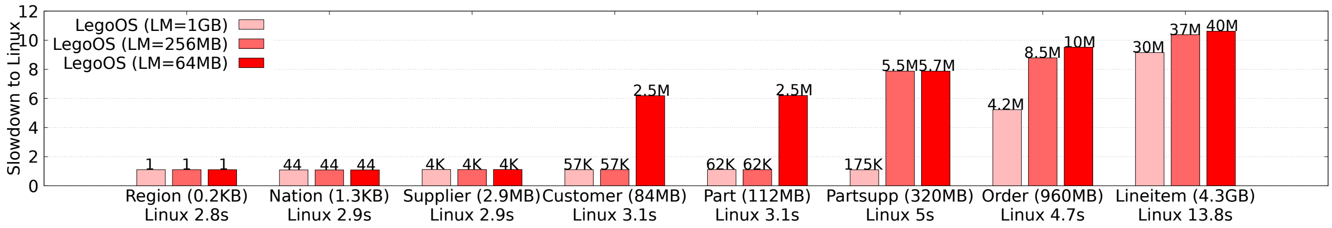

Figure 3: Query performance of hash tables that are built to index TPC-H 10GB tables by their primary keys. We run 100 million random accesses to each hash table and compare the running times of LegoOS with different sizes of local memory (LM=1GB, 256MB and 64MB) and Linux. The x-axis shows each TPC-H table along with its hash index size and the performance in Linux. The y-axis shows the slowdown in LegoOS compared to that in Linux. Each bar is labeled (above) with the number of remote memory accesses in LegoOS. When data is larger than local memory, the performance of accessing hash tables in LegoOS is an order of magnitude worse than in Linux.

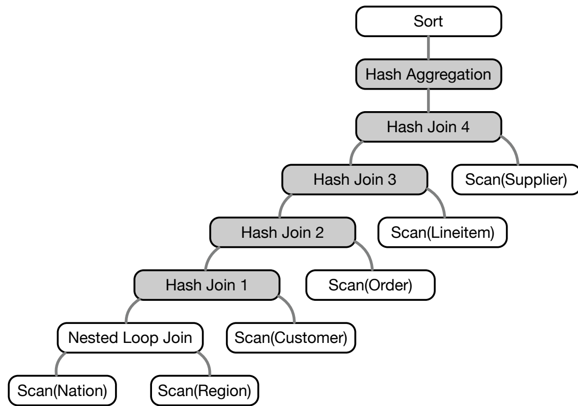

Figure 4: An optimized physical plan for TPC-H Query 5. Shaded are the operators that use hash tables.

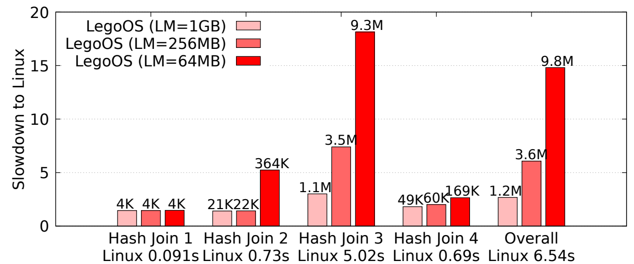

Figure 5: Execution performance of hash joins and end-to-end TPC-H Query 5. X-axis shows the execution of four hash join operators in Figure 4, the end-to-end query execution, and their performance in Linux. Y-axis shows the slowdown of the performance in LegoOS compared to that in Linux. Each bar is labeled with the number of remote memory accesses in LegoOS.

small modifications to the OS and the RDBMS, many of these data transfers can be avoided. We note that the above issues (and our proposed solutions) apply to other parallel join algorithms as well.

3.3 Performance Challenges

To demonstrate the impact of running RDBMSs over DDCs, we evaluate the performance of hash operators in Linux and LegoOS using the TPC-H benchmark. Our testbed consists of three RDMA-enabled CloudLab r320 machines [19] that emulate one compute node, one memory node, and one storage node running LegoOS. All of these nodes are connected via a 56 Gbps Infiniband network using Mellanox MX354A NICs and a Mellanox SX6036G switch. Compute nodes have access to a Xeon E5-2450 (8-cores, 2.1 Ghz) and memory nodes have 16 GB of RAM. As the amount of local memory on compute nodes is currently undetermined, we test a range of possible values: 64 MB, 256 MB, and 1 GB. We expect the eventual value, as a ratio to remote memory and dataset sizes, will trend lower in the spirit of disaggregation.

LegoOS currently supports a subset of the Linux system calls, so we extended the codebase as needed to implement a query execution engine for RDBMS operations. For comparison, we also make our query engine compatible with Linux 3.11 and run it on a single machine with the same compute, memory, and storage resources as the disaggregated testbed. The engine currently supports hash join, hash aggregation, nested loop, filter, project, and sort operations, all of which are needed for the TPC-H queries.

Hash table performance. Hash table performance is crucial to the execution of many RDBMS operators. Our first experiment evaluates the performance of querying hash tables in DDCs. Figure 3 describes the details of the experiment and shows the results. We observed that, when the size of the hash table is within the local memory capacity, the whole table can be cached locally and the performance slowdown of LegoOS (relative to Linux) is within 2×. However, when the table is larger than local memory, the corresponding slowdown ranges from 5× to 11×, showing severe performance degradation.

Query execution performance. We next evaluate the effect of the above slowdowns on an end-to-end TPC-H query—Query 5—the optimized plan of which (shown in Figure 4[1]) involves multiple hash joins. While we present results based on this specific query and plan, we believe that the insights are general to any other RDBMS query execution and leave this validation as future work.

Figure 5 compares the query execution performance in LegoOS with different sizes of local memory and Linux. The results show that when the working set of an operator can fit in local memory, the performance slowdown can be controlled around 2×. As expected, the worst degradation is observed at Hash Join 3 where the two largest tables are joined: the slowdown relative to Linux is 3×, 7× and 18× for LegoOS with 1 GB, 256 MB and 64 MB local memory, respectively. *This degradation results in the slowdown of 2.7×, 6×, and 14.8×, respectively, in overall query execution.* We expect greater slowdowns in larger-scale executions.

Remote memory access. To confirm our hypothesis that the majority of the overhead in LegoOS (over Linux) comes from remote

[1] Adopted from the optimized plan in Microsoft SQL Server [5].

memory requests, we measure the number of remote memory accesses in LegoOS. As both Figure 3 and Figure 5 depict (in the label above each bar), LegoOS needs to constantly page remote memory in and out for queries that require large working sets.

While limited in scope, the above experiments show that a naïve RDBMS implementation on a DDC would suffer severe performance degradation. We leave a more extensive exploration of the performance challenges to future work. For now, however, our results highlight the need to develop new techniques to make the performance of RDBMSs palatable in this brave new world.

3.4 Reliability Challenges

The implications of DDCs to RDBMSs extend beyond performance. Of particular note are the novel failure modes introduced by a DDC architecture [15]. For instance, while a process is running, some or all the associated memory nodes can fail, leaving the process in a state that most applications do not consider. It is even the case that nodes holding the instruction cache of the program are prone to failure, making application-level recovery techniques significantly more complex. Likewise, a process can fail independently of memory, leaving the memory stranded indefinitely—a data-center-level memory leak. Just as in traditional memory leaks, this can degrade the data center's resources over time.

We note that a key challenge is detecting and locating failures, and differentiating failures from message drops and slow machines. Simple timeouts on processes and messages are insufficient for this purpose as they would significantly increase the tail latency of applications. Even if timeouts were possible, failures may be due to other components, such as the network, which may result in non-uniform failure detection. Fast failure detection and localization at data center scale are known to be challenging [42, 40].

4. PAVING A WAY FORWARD

DDCs prompt us to rethink many aspects of RDBMS design, from concurrency control to caching and buffer pool architectures. In this paper, we focus on a small subset of these issues that affect performance/reliability.

4.1 Improving Performance

We start with the example in Figure 2(a). Recall the two main issues: (1) process x's scan of table A leads to two Rack MMU round-trips: one between x and its memory, and one between the memory and storage. (2) process x then sends the corresponding partition tuples to process y via the ToR switch, who must then store them into its own memory. This is particularly problematic because communication over the ToR switch might not be as fast as through the Rack MMU, and because all the data ends up in the same memory blade anyway! We make the following observation: if x could somehow push the partitioning task to the storage node, so that the storage node could directly place the partitions in the appropriate memory elements, we could cut most data movement.

To achieve this, we draw inspiration from decades of work on *near data processing* (NDP) [35, 34, 41, 23]. At a high-level, NDP enhances memory (and in our particular case also storage) with the ability to perform simple operations on the data. In our disaggregated rack architecture (Figure 1) this is possible thanks to the small processing unit (CPU, FPGA, ASIC, or even programmable NIC) attached to storage and memory blades that mediates all accesses. Given this functionality, we propose a new operation that runs at the storage node called Scan-Hash.

Scan-Hash. Scan-Hash streams through a particular table while computing a hash function on the records' join-key. Note that this is simple enough that it can be performed by the CPU or ASIC on the memory and storage blades at high speed. Given this operator, process x would issue a Scan-Hash request to its remote memory, which then does two things: (1) forwards the request to the storage node; (2) places the results provided by the storage node in a memory location based on the returned hash.

At the end of the Scan-Hash operation (Figure 2(b), Step 2), the corresponding partitions are stored in memory, all under the ownership of process x (who initiated the Scan-Hash). Note that x is not given the data: when x sends a Scan-Hash to the memory node, the memory node simply stores the result in the appropriate memory locations, and notifies x when this task is done.

At this point, x can use the *memory grant* operation proposed in prior work [12]. A memory grant is essentially a way for process x to "gift" some of its remote memory pages to some other process. It consists of x telling the Rack MMU to change the permissions at the memory node to allow some other process (in this case y) to access those pages. The Rack MMU then notifies y's OS that new pages have been added to its address space, and the OS propagates this information to y via a signal. The result is that the data is moved exactly once during a partitioning (instead of 4 times): from the storage node into the memory node. This technique generalizes to multiple storage nodes, multiple memory nodes, and other partition strategies. If range partition is preferred over hash partitioning, we can generalize Scan-Hash into a Scan-Partition operation where the application can pass in a partition function.

Collision-avoidance. At the end of the above partitioning protocol, the remote memory contains partitions of records that can be used to build corresponding hash indexes. These indexes can then be probed with records from another table to compute the join. However, as we mention in Section 3, accessing hash indexes in remote memory is expensive due to the possibility of collisions. In particular, every time there is a collision, the query processor (x in the above example) must traverse the corresponding bucket by issuing a series of requests over the Rack MMU. To avoid these costly collisions, we adopt two techniques recently introduced by Aguilera et al. [8]: *indirect addressing* and *HT-Trees*.

Indirect addressing allows the remote memory element to automatically dereference a pointer to determine the corresponding address to load or store, saving one Rack MMU round trip in our context. Without indirect addressing, a query processor would first have to fetch the memory address stored at the pointer's address, and then fetch the data (leading to two RTTs). In an HT-Tree, leaf nodes store base pointers to hash tables (but not the hash tables themselves). A query processor can cache the (small) HT-Tree locally, and leave the (large) hash indexes in remote memory. The query processor can then: (1) retrieve a key by traversing the local HT-Tree to obtain the base pointer of the target hash index; (2) apply the hash function to calculate the bucket number; (3) fetch the appropriate value from the target hash and bucket using indirect addressing to follow the pointer in the bucket.

Remote-memory aggregation. Like joins, aggregation can be potentially expensive due to global reshuffling. In a DDC architecture, there are opportunities for in-memory aggregation given that data is consolidated within memory blades. For example, the Scan-Hash mechanism above can be enhanced to hash tuples into buckets based on group-by keys, and a reduction phase applied to each bucket to generate aggregation results. This avoids expensive round-trip times to compute nodes, and in fact, allows us to avoid extensive data shuffling within a rack. If data resides across memory blades on different racks, one can compute intermediate aggregates at the rack level before combining across racks.

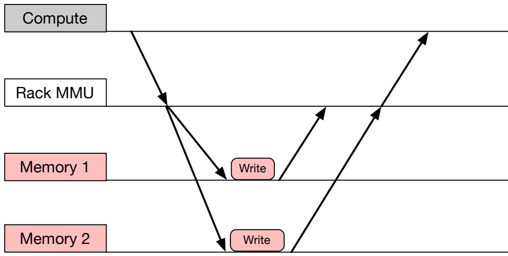

Figure 6: The Rack MMU can multicast write operations to a set of memory replicas to reduce the probability of data loss in the event of memory failures. Note that we are discussing volatile memory here; data loss means that the application cannot make progress without recreating the current state from persistent storage.

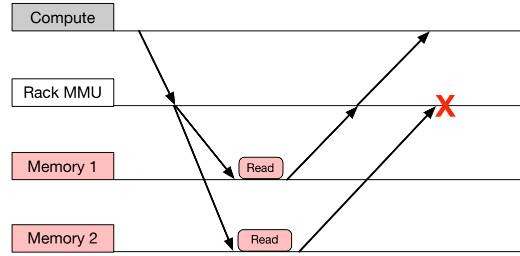

Figure 7: To reduce the effect of stragglers and memory failures, read operations can also be replicated. Only the first response is returned. Duplicates are removed at the Rack MMU.

4.2 Improving Reliability

The primary difference between DDCs and traditional data centers is that resource elements may fail independently. Following a recent proposal [15], we assume that resources within a single node (e.g., DRAM chips) can fail independently, and that different nodes may also fail independently. A node failure is equivalent to a failure of its resource components. We present a few initial proposals for tolerating compute and memory failures below.

Compute node failures. Unlike today's systems, when compute nodes fail, the decoupling of processing and memory allows us to recover without recomputing the job or invoking long-term storage. We propose an approach with a centralized job manager not unlike those used in most modern Big Data processing frameworks. The manager can detect processing node failures through heartbeat messages. If it detects a failure, it can determine the progress made by the previous node before it crashed. For this, we can assign each individual operation or set of operations a designated space in memory, each annotated using a "done" bit. A simple binary search through the memory locations will allow the manager or the new processing node to determine the last completed operation.

Read/write replication. Borrowing from RAID-style disk replication [36] and the work of Carbonari and Beschastnikh [15] in the context of DDCs, we propose replicating all memory writes, and some memory reads as well. With replication, the choice to store data in memory or disk will be based primarily on anticipated access frequency ("hot" vs "cold"), rather than persistence. The Rack MMU provides an attractive point for implementing the replication protocol since it serves as a serialization point for all memory requests. Although the Rack MMU is a single point of failure, like the ToR switch or server NICs, fate sharing ensures that the Rack MMU fails if and only if dependent services have failed as well. Using this approach, when a processing node wishes to write a block of memory, it will, with the help of the Rack MMU, multicast a set of writes to a predetermined set of replicas (as shown in Figure 6). Read requests can also be multicast if they are "high-priority," in which case, multiple replicas are queried and only the first result is taken. Duplicates can be detected using a sequence number, and potentially eliminated early in the Rack MMU to avoid unnecessary network overhead (as shown in Figure 7).

Memory failure modes. Given the above replication protocol, the following failure modes can be tolerated:

Failure before a read. For a high-priority read, one of the requests will return a response as long as at least one replica remains active. For a regular read, the processing node's memory paging kernel procedure will set a short timeout and retry if the response is late. In both cases, data loss can be detected if all requests timeout.

Failure before a write. The mechanism is similar to that of a high-priority read request. The writes will be multicasted to the designated set of replicas through the Rack MMU. Writes can be done asynchronously, although if the process wishes to ensure that the write was successful it must wait for an acknowledgment.

Failure after a write. These failures are silent from the perspective of the writer. Instead, we can view them as failures before a read. Replication should ensure that the data is tolerant to multiple failures; if all the replicas fail, the data loss will be detected by the next reader, who should perform application-level recovery.

5. RELATED WORK

Resource disaggregation offers great benefits for cloud data centers, but requires the redesign of applications for good performance. This paper is the first work to discuss the changes, challenges, and opportunities that disaggregation brings to data management systems. Here we survey systems that relate to resource disaggregation as well as data management systems in related contexts.

Disaggregated Data Centers. Both industry and academia are beginning to explore OSes, architectures, and networks for DDCs. LegoOS [38] introduces an OS that decouples hardware resources and connects them with a fast network, while still providing a virtual machine abstraction for applications. Other proposals include new architectures [31, 32], network systems [21, 39, 15], and data structures [8] for DDCs. Our work is the first to explore how RDBMS performance is impacted by disaggregation.

Distributed Remote Memory. Prior work has revisited the idea of remote memory in fast data center networks [6], proposing a standard API for remote memory access with exported files [7]; implementing generic data structures like vector, map, and queue on top of remote memory by customizing hardware primitives [8]; and designing a new paging system to avoid application modification [24]. While remote memory and disaggregation overlap in spirit, the two ideas differ in that remote memory assumes that significant resources remain coupled with the compute components—the remote memory is an optimization, not a necessity. This fundamental difference presents significantly different implications to upper-layer applications, as we show in Section 3.

Database systems on RDMA. Finally, researchers have explored the use of RDMA as an efficient channel for data movement between database instances in traditional data centers. Examples include industrial databases that use RDMA to speed up data transfer between storage and memory [1, 2, 3], and database systems that use RDMA for larger-scale in-memory processing [28, 14, 37]. As mentioned above, these remote memory proposals are fundamentally different from disaggregation.

6. CONCLUSIONS AND FUTURE WORK

This paper advocates rethinking how we should design data management systems under the paradigm shift of *resource disaggregation*. Researchers have identified unique challenges in DDCs for the OS, network, and architecture design. We believe that it is also crucial to understand the implications of DDCs for data management. We have described why a naïve adaptation of RDBMSs would lead to performance inefficiencies, and outlined potential ways to address these challenges in DDCs. We have also performed an initial set of experiments for validation.

Looking forward, we plan to perform studies on a more comprehensive set of relational operators and investigate the challenges required in a DDC setting. Transaction processing and concurrency control are also interesting directions, in light of the fault tolerance discussion in this paper. We also plan to generalize our investigation to big data processing systems beyond RDBMSs, e.g., to MapReduce, machine learning, or graph processing systems. We hope that our paper provides an initial step to these investigations and promotes more discussions in the database community.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. This work was supported in part by VMWare, Facebook, the National Science Foundation (CNS-1845749, CNS-1801884, CNS-1513679, CNS-1703936, and CCF-1763514), DARPA (contracts No. HR0011-17-C-0047 and No. HR0011-16-C-0056), and ONR (N00014-18-1-2618).

7. REFERENCES

[1] IBM DB2 pure scale. http://www.ibm.com/developerworks/data/library/dmmag/DBMag_2010_Issue1/DBMag_Issue109_pureScale/.

[2] Microsoft analytics platform system. http://www.microsoft.com/en-us/server-cloud/products/analytics-platform-system/.

[3] Oracle RAC over InfiniBand. http://www.cisco.com/c/en/us/products/collateral/cloud-systems-management/vframe-ib-software/prod_white_paper0900aecd8056d64c.html.

[4] The parallelism operator (aka exchange). https://blogs.msdn.microsoft.com/craigfr/2006/10/25/the-parallelism-operator-aka-exchange/.

[5] TPC-H SF100 non-parallel plans, SQL Server 2008. http://www.qdpma.com/tpch/TPCH100_Query_plans.html.

[6] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2017.

[7] M. K. Aguilera and et al. Remote regions: a simple abstraction for remote memory. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018.

[8] M. K. Aguilera, K. Keeton, S. Novakovic, and S. Singhal. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.

[9] Alibaba. ApsaraDB for POLARDB: A next-generation relational database - Alibaba cloud. https://www.alibabacloud.com/products/apsaradb-for-polardb, 2019.

[10] Amazon-Aurora. Amazon aurora - Relational database built for the cloud - AWS. https://aws.amazon.com/rds/aurora/, 2019.

[11] T. E. Anderson, D. E. Culler, and D. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, Feb 1995.

[12] S. Angel, M. Nanavati, and S. Sen. Disaggregation and the application. arXiv:1910/13056, Oct. 2019. http://arxiv.org/abs/1910.13056.

[13] K. Asanović. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[14] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *Proceedings of the ACM SIGMOD Conference*, 2015.

[15] A. Carbonari and I. Beschastnikh. Tolerating Faults in Disaggregated Datacenters. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2017.

[16] P. Costa, H. Ballani, and D. Narayanan. Rethinking the network stack for rack-scale computers. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2014.

[17] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, Mar. 1990.

[18] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[19] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[20] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[21] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[22] G. Graefe and et al. Extensible query optimization and parallel execution in volcano. In *Query Processing for Advanced Database Systems, June 1991*.

[23] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J.-U. K. Jonghyun Yoon, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[24] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[25] K. Katrinis and et al. Rack-scale Disaggregated Cloud Data Centers: The dReDBox Project Vision. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016.

[26] K. Keeton. The Machine: An Architecture for Memory-centric Computing. In *Proceedings of the Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2015.

[27] J. Kyathsandra and E. Dahlen. Intel Rack Scale Architecture Overview. http://presentations.interop.com/events/las-vegas/2013/free-sessions---keynote-presentations/download/463, 2013.

[28] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA. In *Proceedings of the ACM SIGMOD Conference*, 2016.

[29] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009.

[30] K. Lim, Y. Turnet, J. Chang, J. Renato Santos, and P. Ranganathan. Disaggregated Memory Benefits for Server Consolidation. Technical Report HPL-2011-31, HP Laboratories, 2011.

[31] K. T. Lim, J. Chang, T. N. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009.

[32] K. T. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2012.

[33] Microsoft-SQL-Database. Sql database – cloud database as a service | Microsoft Azure. https://azure.microsoft.com/en-us/services/sql-database/, 2019.

[34] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: A computation model for intelligent memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1998.

[35] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM: IRAM. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1997.

[36] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD Conference*, 1988.

[37] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2015.

[38] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[39] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon. Shoal: A Network Architecture for Disaggregated Racks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[40] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang. Netbouncer: Active device and link failure localization in data center networks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[41] S. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos. Beyond the wall: Near-data processing for databases. In *Proceedings of the International Workshop on Data Management on New Hardware*, 2015.

[42] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. E. Anderson. Deepview: Virtual disk failure diagnosis and pattern detection for azure. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.