Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs

Qizhen Zhang[1], Yifan Cai[1,2], Xinyi Chen[1], Sebastian Angel[1]
Ang Chen[3], Vincent Liu[1], Boon Thau Loo[1]

[1]University of Pennsylvania, [2]Shanghai Jiao Tong University, [3]Rice University
[1]{qizhen, caiyifan, cxinyic, sga001, liuv, boonloo}@seas.upenn.edu
[2]fyc1007261@sjtu.edu.cn, [3]angchen@rice.edu

ABSTRACT

Resource disaggregation is a new architecture for data centers in which resources like memory and storage are decoupled from the CPU, managed independently, and connected through a high-speed network. Recent work has shown that although disaggregated data centers (DDCs) provide operational benefits, applications running on DDCs experience degraded performance due to extra network latency between the CPU and their working sets in main memory. DBMSs are an interesting case study for DDCs for two main reasons: (1) DBMSs normally process data-intensive workloads and require data movement between different resource components; and (2) disaggregation drastically changes the assumption that DBMSs can rely on their own internal resource management.

We take the first step to thoroughly evaluate the query execution performance of production DBMSs in disaggregated data centers. We evaluate two popular open-source DBMSs (MonetDB and PostgreSQL) and test their performance with the TPC-H benchmark in a recently released operating system for resource disaggregation. We evaluate these DBMSs with various configurations and compare their performance with that of single-machine Linux with the same hardware resources. Our results confirm that significant performance degradation does occur, but, perhaps surprisingly, we also find settings in which the degradation is minor or where DDCs actually improve performance.

PVLDB Reference Format:
Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *PVLDB*, 13(9): 1568-1581, 2020.
DOI: https://doi.org/10.14778/3397230.3397249

1. INTRODUCTION

An emerging trend in data centers is the physical disaggregation of resources. In a fully (resource) disaggregated data center (DDC), servers are no longer built as standalone machines equipped with sufficient compute, memory, and storage to process a single job. Instead, each resource node in a DDC is kept physically separate, with some nodes specialized for processing, others for memory, and others for storage. To complete a single task, a processing node will need to 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 CPUs or GPUs.

Disaggregating resources in this way provides substantial benefits to data center operators. It allows them to upgrade and expand each resource independently, e.g., if a new processor technology becomes available or if the workload changes require additional CPUs. It also allows them to prevent fragmentation and over-provisioning, e.g., if a customer requests an unusual balance between CPU cores, RAM, and GPUs that does not fit neatly into an existing machine. Finally, to users, disaggregation creates the illusion of a near-infinite pool of any resource for any program.

Disaggregation has fundamental implications on the performance of data-intensive applications, not all of which are positive. For example, our recent work [8, 38] highlights the potential performance degradation that stems from moving the storage and the bulk of memory to a remote machine. While extrapolating the results of these previous studies would lead one to conclude that DBMSs would also fare very poorly, doing so would be speculative: these prior works consider only synthetic workloads and simple applications. In contrast, DBMSs have complex software stacks; having a thorough understanding of their end-to-end performance in a DDC is therefore critical for the design, implementation, and optimization of DBMSs in future cloud architectures.

Further, to DDCs, database systems provide an interesting case study of the effects of disaggregation. At a basic level, query executions in DBMSs are typically data-intensive, involving frequent and repeated movement of large quantities of data between disk and memory (loading data from storage to main memory and spilling intermediate data to disk when memory is limited), memory and CPU (moving data between compute units and working sets in main memory), CPU and CPU (data shuffling between workers). In a DDC, each of these steps requires network communication, which can impact query performance. Even so, queries, each having unique access patterns, exhibit a great deal of diversity in their reaction to disaggregation. The case study also presents an opportunity to examine modern DBMS design in a different light. Specifically, decades of optimization and tuning on top of traditional servers and operating systems have resulted in a series of baked-in assumptions about memory access latency, buffer management, and paging strategies. Disaggregation exposes many of these fundamental assumptions.

Similarly, to DBMSs, disaggregation presents a unique set of challenges even when compared to the extensive literature on production DBMS performance in new architectures, e.g., disaggregated storage [7, 25, 6] and remote memory [14, 20, 10]. First,

Proceedings of the VLDB Endowment, Vol. 13, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3397230.3397249

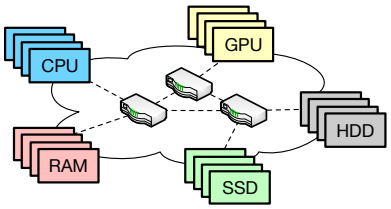


Figure 1: The architecture of disaggregated data centers (DDCs). Hardware resources are split into independently managed pools that are connected by a high-performance network fabric.

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 present the first characterization and analysis of modern production database systems running on a DDC. Enabling our study is a combination of recent hardware, network, and operating system advances that, for the first time, provide a complete disaggregated operating environment. This environment allows us to investigate the interactions between each layer in detail.

More specifically, we evaluate queries from the TPC-H benchmark in MonetDB [26] and PostgreSQL [31] in a variety of disaggregation settings. We find that PostgreSQL is less sensitive to disaggregation than MonetDB, but PostgreSQL is also incapable of adapting to varying levels of local memory since it delegates disk caching to the underlying OS (i.e., PostgreSQL achieves similar performance when the compute nodes' cache is very large and when it is small). We also observe that without modifications to either MonetDB or PostgreSQL, DDCs can enable these production databases to scale up and achieve *high* and *stable* performance. This is in contrast to traditional architectures that spill to disk and introduce significant performance variability. While RDMA-based DBMSs [20, 10] may achieve similar benefits, it comes at the cost of an extensive redesign of these DBMSs.

In summary, this paper makes the following contributions. First, we use the complete TPC-H benchmark to validate that DDC degrades the performance of DBMSs due to expensive remote memory accesses (data movement between compute and memory components). Second, we identify several scenarios where DDCs can be a better alternative for DBMSs. Last, we analyze the bottlenecks of executing DBMSs on DDCs and shed light on different ways to optimize the execution of future DBMSs in this new architecture.

2. BACKGROUND

This section introduces the key architectural elements of recent DDC proposals, with a focus on their effect on DBMS operation.

2.1 Disaggregated Data Centers

The key idea behind resource disaggregation is to break up monolithic servers that traditionally keep all of their resources on the motherboard connected by high-speed buses (e.g., GPUs, memory, storage) into separate "disaggregated" pools of resources that are physically distinct. The machines housing each of the disaggregated resources are connected using a fast network fabric such as RDMA over Infiniband (although special-purpose connectors have also been proposed [34]). Figure 1 depicts this high-level architecture. DDCs bring significant operational benefits over traditional architectures. These benefits include:

- **Independent expansion.** The hardware resources can be expanded and upgraded independently. For example, if a DDC is running low on memory, the operator can just hot-plug more memory in the memory pool. This is more flexible and cost-efficient than traditional data centers where an operator would need to add large servers with more resources.
- **Independent failures.** Since resources are decoupled, the failure of one resource does not signify the failure of all others. For example, it is possible for a memory node to fail, while the associated CPU remains alive. Prior work suggests ways to recover from these types of failures in DDCs [8].
- **Independent allocation.** For cloud operators, resource allocation becomes a simpler task: packing virtual machines to DDCs simply requires identifying the appropriate resource pools and creating the appropriate forwarding rules in the network fabric. In comparison, packing VMs to monolithic servers while maximizing utilization and minimizing resource fragmentation is an NP-hard knapsack problem.

In exchange for the above benefits, DDCs convert a subset of what used to be local memory and device accesses to remote accesses. Though the networks in these proposals are designed to be very fast, they are nevertheless higher latency than accessing resources on the same motherboard. This results in expensive data movement between hardware components (e.g., CPU and memory), particularly for applications where prefetching and pipelining are hard to do. Indeed, previous work hypothesizes that these data movements are likely to degrade performance in data-intensive applications by orders of magnitude [38]. Our work confirms this hypothesis with a thorough experimental evaluation and proposes several ways to reduce the degradation.

2.2 Disaggregated Operating Systems

A critical piece of the above architecture is the disaggregated operating system. Fundamentally, the migration of memory away from compute means that, while CPU nodes may have a nominal amount of memory to store a kernel, it may not have enough for the code segment, data segment, heap, and/or stack. In the same way, memory nodes may have enough compute to perform address translation and basic access control, but will not have enough to execute queries. Thus, the operation of a DDC will likely need to be mediated through a specialized operating system.

A state-of-the-art disaggregated OS is LegoOS [33], which takes a splitkernel approach to dividing kernel responsibilities over resource-disaggregated nodes. In LegoOS, the local kernel on a computation node, where DBMS instructions are expected to run, is in charge of configuring and negotiating access to external resources, and of managing a small amount of local memory that is attached to the CPU. This memory hosts the local kernel and serves as a cache for applications. LegoOS supports the Linux system call interface as well as an unmodified Linux ABI, allowing users—in principle—to run unmodified Linux applications.

In this work, we take advantage of LegoOS's interface. Unfortunately, while LegoOS is a working research prototype that highlights the complexities of building a distributed operating system that coordinates and manages disaggregated resources, it is not sufficiently complete to run a real production DBMS. One of the contributions of our work is, therefore, to extend LegoOS's codebase with several system calls and additional functionality that is needed to run these DBMSs. We discuss these efforts in Section 3.

2.3 DBMSs in DDCs

How do modern DBMSs fare in a disaggregated environment? In this section, we sketch the operation of these systems on DDCs, before measuring and analyzing them in subsequent sections. Our discussion here focuses on three types of hardware used by a DBMS: CPU, random access memory, and disk storage. Like prior work, we assume that processing nodes have a limited amount of memory and that memory/storage nodes have a limited amount of compute. Otherwise, resources are decoupled and connected via a low-latency, high-bandwidth network.

Figure 2 depicts the typical execution of DBMSs when running in a DDC. A pool of *storage nodes* holds the database data in persistent storage, a pool of *memory nodes* holds the buffer pool of the DBMS in random access memory, and a pool of *processing nodes* runs the actual DBMS processes, with the local memory of the processing nodes serving as a cache of the buffer pool. The original copies of each process's virtual memory, therefore, reside entirely remotely, either in the remote memory pool, or paged onto remote disk. To execute a query, the database tables are scanned and loaded into the buffer pool; in-memory data will then be transferred to and from the processing and the memory pools during execution. The processing and storage nodes do not exchange data directly.

The OS chooses which pages to maintain in the local memory of processing nodes using well-known page eviction policies like LRU or FIFO—data is fetched from remote memory on a local memory cache miss and fetched from storage on a remote memory cache miss. We term the former *remote memory accesses* and the latter *disk page faults* to differentiate the two in this paper. In both cases, if a query requires data beyond what is cached in its local memory, a kernel trap will block the execution of the query until the memory can be fetched from the memory pool.

The overall performance cost of this additional layer in the memory hierarchy depends on several factors. For instance, the relative size of local memory compared to the buffer pool will determine the frequency of accesses. The interplay between the buffer pool management strategy and the OS local memory eviction policy can also have a significant effect on performance, as can the interaction between remote memory accesses and disk page faults, and the pattern of accesses and the architecture of the DBMS. To illustrate one example of the complexities of this space, consider an LRU buffer pool on top of an LRU local memory eviction policy. When the DBMS evicts an item from the buffer pool, it might:

1. Bring the new item into a memory node from storage.
2. Bring the new item into local memory, evicting others.
3. Bring the LRU item from memory into local memory.
4. Finally, copy from local memory to the buffer pool.

Step 3 is due to the DBMS's replacement algorithm running in the processing node. This highlights how two in-memory buffers result in two sets of replacement policies whose interaction may be suboptimal, suggesting the need for the buffer pool to be aware of "cheaper" local memory and more expensive remote memory.

3. EXPERIMENT SETUP AND METHODS

To explore the implications of this paradigm shift, this paper presents an in-depth characterization and analysis of the performance of production DBMSs running on DDCs. This section details the setup of our performance measurements.

3.1 Testbed Setup

Our DDC testbed consists of three bare-metal servers in CloudLab [15]: the processing node has a single Xeon E5-2450 CPU (8 cores, 2.1 Ghz), the memory node has 16 GB of DDR3-1600 MHz DRAM, and the storage node has four 500 GB hard disk drives in a RAID-5 configuration. Each node runs LegoOS [33] and is equipped with an RDMA-enabled Mellanox MX354A NIC and connected over a 56 Gbps Infiniband network with a Mellanox SX6036G switch. We are currently restricted to this hardware configuration due to LegoOS's limited driver support, and the low availability of compatible servers in CloudLab. To provide a fair baseline, we compare to a single Ubuntu Linux 3.11 server with the same compute, memory, and storage resources as our DDC testbed.

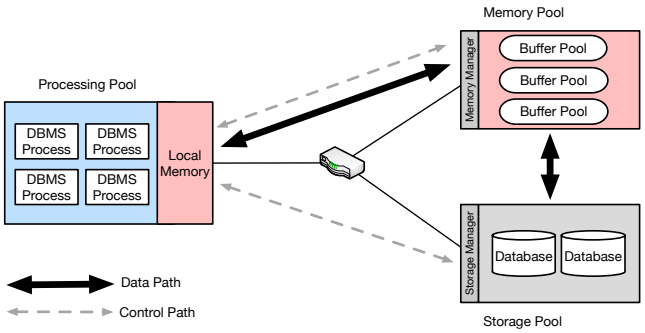


Figure 2: DBMS execution in DDCs. DBMS workers are spawned on processing nodes with their small local memory acting as a cache. Buffer pools live in a remote memory pool; a storage pool stores and manages the database files. Workers send control messages to allocate and manage resources, and the data is transferred between memory and storage pool (loading and spilling) and the processing and memory pool (fetching and eviction).

	MonetDB	PostgreSQL
Execution	In-memory	Out-of-core
Storage	Column-based	Row-based
Architecture	Client/Server	Client/Server
Buffer pool size	$\min(S_{\text{Capacity}}, S_{\text{Demand}})$	Customizable

Figure 3: Summary of parameters in MonetDB and PostgreSQL.

Local memory configuration. Vendors and cloud providers have not yet settled on the size of local memory in DDCs, but we expect the most cost-efficient nominal (i.e., per-CPU) sizes to be smaller than typical DBMS buffer pools. We evaluate DDC performance on a variety of local memory sizes ranging from low (64 MB) to high (4 to 6 GB) capacity to emulate different degrees of disaggregation.

Storage. Due to hardware availability, our testbed uses hard disk drives for storage. While SSDs would improve performance, we expect that the general trend of the disk being a bottleneck in some of our experiments would still hold as our Infiniband network significantly outperforms SSDs in both latency and throughput.

3.2 System Selection and Adaptation

We select two popular open-source DBMSs: MonetDB [26] (Version 11.33.11) and PostgreSQL [31] (Version 11.5)—both are the latest versions at the time of this evaluation. We select these two systems to represent different types of DBMSs: MonetDB is a column store, designed to be executed in-memory; PostgreSQL is a row-based system and it adopts an out-of-core execution model. We summarize and compare the technical parameters of MonetDB and PostgreSQL in Figure 3. One parameter of interest is the buffer pool size. In MonetDB, the system consumes as much memory as needed to match application demand (S_{Demand}) as long as it does not

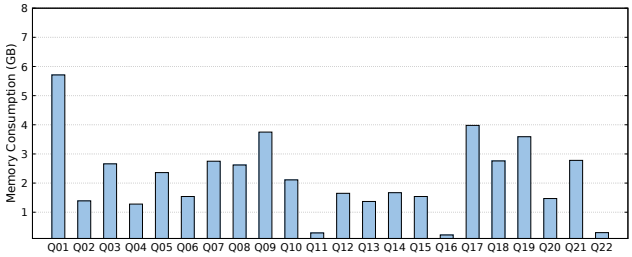


Figure 4: Peak memory usage of TPC-H queries in MonetDB.

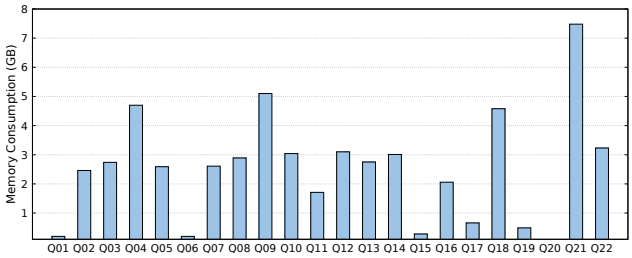


Figure 5: Peak memory usage of TPC-H queries in PostgreSQL.

exceed the amount of physical memory (S_{Capacity}). In PostgreSQL, the buffer pool size is customizable. For both Linux and LegoOS, we tune the PostgreSQL buffer pool size to maximize performance.

We note that LegoOS currently supports only a subset of Linux system calls. Thus, to execute PostgreSQL and MonetDB, we spent significant effort adapting these two DBMSs to LegoOS (for reference, PostgreSQL has ∼1.3M lines of C code, MonetDB has ∼400K lines of C and MAL code, and the LegoOS kernel consists of ∼300K lines of C code). We highlight three examples:

Socket support. LegoOS, which relies solely on RDMA for communication between nodes, currently does not support sockets, but the client and the server communications of both PostgreSQL and MonetDB are based on sockets. Thus, we bypass the client and directly start the server to execute the SQL queries and benchmark the query execution performance on the server.

Read system call. Another example is a slight difference between the implementation of the `read` system call in LegoOS and Linux. When the application calls `read` to read N bytes from a file, due to disaggregation, LegoOS allocates N bytes of memory in kernel space in the processing node to receive the data that is finally returned from the memory node (refer to the data paths in Figure 2). If N is large, the processing node can run out of memory, leaving other components of the system hanging. We added additional functionality to address this issue.

Relative paths. The original version of LegoOS could only support absolute paths while relative paths are extensively used in the selected DBMSs; we implemented two system calls (`getcwd` and `chdir`) in order to run MonetDB and PostgreSQL.

Additionally, we fixed several inconsistent behaviors in the way that LegoOS performs file system operations. For example, in LegoOS, `rename` always unlinks the old file on the storage node without detecting the existence of the new file, so if the new file does not exist, then the old file is deleted, while in Linux, the old file is still safe. We note that these issues are due to the immaturity of the current LegoOS codebase, rather than its higher-level design.

3.3 Workload Selection and Characterization

To study the implications of disaggregation on the end-to-end query execution performance of real-world complex queries, we select the TPC-H benchmark and use all of its 22 queries, which represent a wide range of execution patterns. Unless otherwise specified, we use a scale factor of 10.

As discussed, memory disaggregation makes memory accesses a bottleneck for many applications in DDCs, so overall memory consumption is an important factor in this study. To that end, we provide a characterization of the memory demands of all 22 TPC-H queries. The exact demands of each query, of course, vary as each DBMS will select its own execution plan for each query depending on a number of factors; different plans will result in different memory usage patterns. Thus, we run the queries with the aforementioned scale factor in MonetDB and PostgreSQL in Linux and measure the memory consumption of each query. We note that the memory used by the OS for disk caching is not included here because it is determined by the OS, and the OS (for example, Linux) can aggressively use available memory for caching disk data as long as it does not affect the memory usage of the applications.

Figure 4 and Figure 5 show the measurement results for MonetDB and PostgreSQL respectively. Note that in the case of PostgreSQL, we configured the maximum buffer pool as 8 GB to allow for sufficient OS and disk cache space, and we excluded Q20 because it could not finish execution [29]. The memory consumption of different queries running on a single DBMS can vary substantially, as can the consumption of a single query on two different DBMSs. Even so, we can summarize a few patterns: (1) all queries consume more than 200 MB of memory; (2) most queries use around the average amount of memory (2.2 GB in MonetDB and 2.8 GB in PostgreSQL); (3) a few queries use significantly higher memory (Query 1, 9, 17, 19 in MonetDB, Query 4, 9, 18, 21 in PostgreSQL) than others; and (4) a few queries use significantly lower memory (Query 11, 16, 22 in MonetDB, Query 1, 6, 15, 17, 19 in PostgreSQL) than others. We will refer back to these two figures when we analyze experimental results in the next sections.

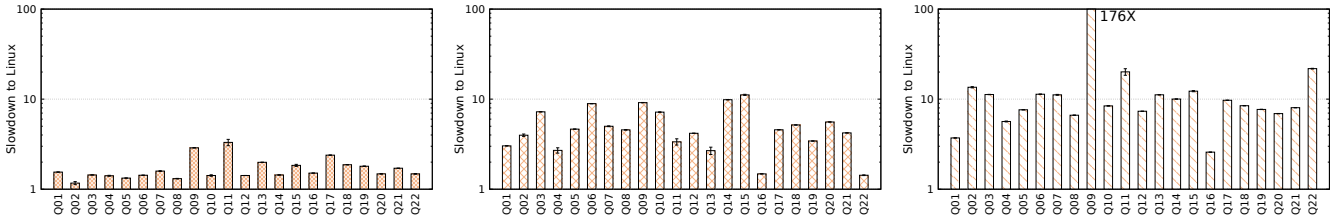
4. THE COST OF DISAGGREGATION

We evaluate the overhead of disaggregation by running both production DBMSs on LegoOS and a traditional standalone Linux server. We equalize the amount of compute, memory, and storage resources between LegoOS and Linux to ensure a fair comparison.

4.1 In-memory Execution

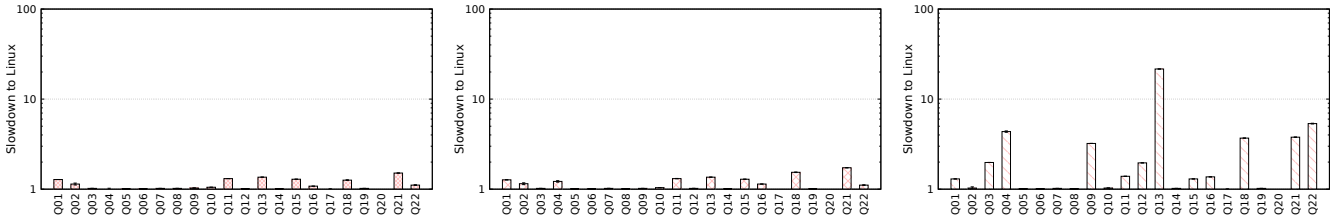
We first evaluate MonetDB under three different local memory sizes (4 GB, 1 GB, and 64 MB). Before running each TPC-H query, we warm up the DB buffer pool, to remove the effects of disk paging. For each graph, we show the slowdown relative to Linux for all 22 TPC-H queries, where a slowdown ratio of 1 means performance is on par with Linux. In all figures, all bars are augmented with 95% confidence intervals, which show that the results are stable with low variance. We summarize our findings as follows:

4 GB local memory (Figure 6a). The slowdown is moderate: an average of 1.7× and a median of 1.5×. The slowdown stems from fetching data from the remote buffer pool in the memory pool to the local memory. However, LegoOS's optimizations on memory prefetching and lazy memory allocation keep the slowdown moderate. Moreover, because CPU is the primary bottleneck, even though the working set of Q1 (Figure 4) does not fit into 4 GB, the slowdown is only 1.6×. Q9 and Q17 experience higher slowdowns (2.9× and 2.4× respectively) because their actual working sets (once kernel, stack, and instruction cache are included) well exceed 4 GB, resulting in thrashing of local memory. Q11 has the highest slowdown given that it is very short (it only runs 0.07 s) and a handful of memory stalls incur a high relative slowdown.



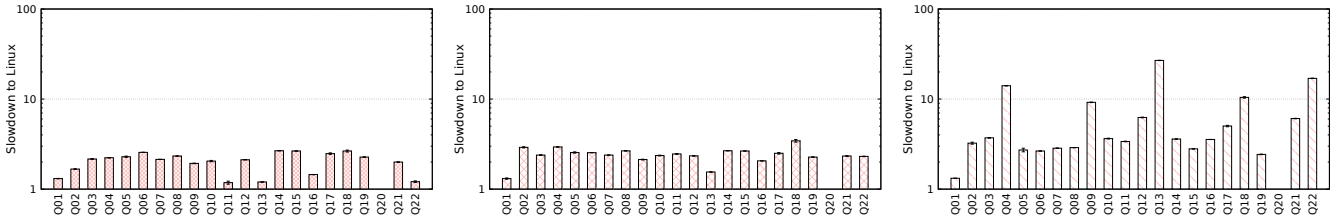
(a) LegoOS with 4 GB local memory. **(b)** LegoOS with 1 GB local memory. **(c)** LegoOS with 64 MB local memory.

Figure 6: MonetDB query execution time slowdowns with different degrees of disaggregation. Baseline: Linux server.



(a) LegoOS with 4 GB local memory. **(b)** LegoOS with 1 GB local memory. **(c)** LegoOS with 64 MB local memory.

Figure 7: PostgreSQL cold execution time slowdowns with different degrees of disaggregation (Q20 excluded). Baseline: Linux.



(a) LegoOS with 4 GB local memory. **(b)** LegoOS with 1 GB local memory. **(c)** LegoOS with 64 MB local memory.

Figure 8: PostgreSQL hot execution time slowdowns with different degrees of disaggregation (Q20 excluded). Baseline: Linux.

1 GB local memory (Figure 6b). As local memory decreases, the average slowdown increases because most queries utilize more than 1 GB of memory. Queries with small memory footprint (Q16, Q22) are not affected by the local memory reduction, and Q11 is also less sensitive because, although it does spill data to remote memory, going from 4 GB to 1 GB does not exacerbate the effect.

64 MB local memory (Figure 6c). The final configuration reduces local memory to only 64 MB. All queries are more than 2.5× slower than their non-disaggregated executions and ten of them have performance degradation larger than 10×. Q9 has the most extreme slowdown of 176×. This is because Q9 adopts nested loop joins for six tables, and together with an expression calculation, they result in frequent random accesses to the buffer pool. Those random accesses cause extreme inefficiency when the local memory is constrained. We analyze the slowdowns in greater detail by relating them to remote memory accesses in Section 6.

4.2 Out-of-Core Execution

We next evaluate PostgresSQL to understand the impact of out-of-core execution under two settings: (1) execution in a cold hardware/software cache scenario (*cold execution*); and (2) execution after the buffer pool and caches are warmed up by running the same query multiple times (*hot execution*). We differentiate between those two scenarios because PostgreSQL heavily relies on OS mechanisms to cache recent data.

Cold execution. Figure 7 shows the cold execution performance in LegoOS with different sizes of local memory. In cold execution given 4 GB and 1 GB local memory (Figures 7a and 7b), most queries have negligible slowdowns since disk I/O overshadows the additional network latency. Consider the plan of Q3 (Figure 9) which consists of a right-deep tree of a 3-way join, the tree is executed in a pipelined fashion: every time a tuple of `lineitem` is scanned, it is used to join with the rest of the tree. Given the size of the `lineitem` table, significant disk I/O incurred during the scan dominates the execution. This is still true when we migrate to a disaggregated environment—disk I/O takes a longer time than the memory stalls that fetch data from remote memory. This disk bottleneck closes the gap between LegoOS and Linux. In the 64 MB setting (Figure 7c), the majority of queries continue to achieve similar performance to their non-disaggregated executions, as shown in Figure 7c. The queries that experience higher than 2× slowdowns (e.g., Q13), do so because of unmasked memory stalls (e.g., executions that are not pipelined or that perform many random accesses).

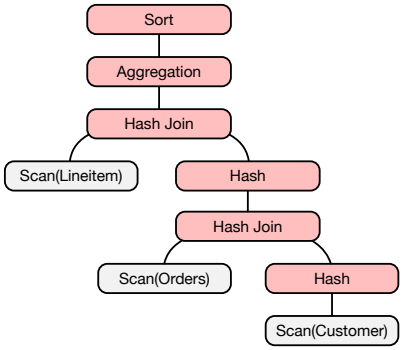


Figure 9: The simplified execution plan for Q3 in PostgreSQL. Grey operators involve disk I/O and red operators are in memory.

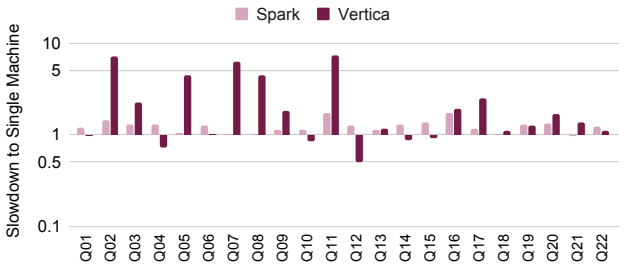


Figure 10: The slowdowns of running distributed DBMSs in a cluster compared to a single machine of the same hardware.

Hot execution. Figure 8a shows that given 4 GB local memory, average and median hot execution slowdowns are 2× and 2.1×, respectively. At 1 GB memory (Figure 8b), the average slowdown increases only slightly to 2.4×, indicating that the performance is still largely bottlenecked by I/O.

These two results show an interesting effect. Although in-memory and hot out-of-core execution both bypass disk I/O, they perform very differently in DDCs when local memory is sufficient: the latter still suffers from I/O bottlenecks while the former does not. The reason is a gap between the efficacy of application-based disk cache management (as done by MonetDB) and LegoOS's disk management (as outsourced by PostgreSQL). The difference is that while the application data can be cached locally in the processing pool, LegoOS stores its disk cache remotely in the memory and storage pool. Consequently, MonetDB's manual management of the disk cache results in much better data reuse and pipelining.

A further reduction of local memory to 64 MB results in a significant slowdown for a subset of queries (Figure 8c). Memory intensive queries (Q4, Q9, Q18, and Q22, cf. Figure 5) experience 10× slowdowns, and Q13 has a worst-case 27× slowdown. The slowdown is larger in hot executions because they eliminate the disk I/O that masks the performance degradation in cold executions.

4.3 Distributed Baseline

Next, we study how scale-out setups affect the performance of traditional, distributed DBMSs, relying on these results to put DDC performance slowdowns into perspective. We have chosen two highly-optimized DBMSs: Apache Spark SQL [1] v2.4.5 and Vertica [2] v9.3.0. We first ran these systems using TPC-H (scale factor 10) in a single Linux machine, and then set up a distributed environment using three "smaller" machines that collectively provide equivalent hardware resources, including CPU cores, memory, and storage. We configured Spark SQL to use NFS to access a remote storage server, ensuring the same disk I/O performance. Vertica, however, does not support NFS, so we configured it to use the local storage on each machine. We note that this gives the distributed setup of Vertica a slight advantage in aggregate disk I/O throughput. Nevertheless, this setup paints a useful picture of the contrast of DDC and distributed DBMS slowdowns.

Figure 10 shows the results for performance slowdowns of these DBMSs due to distributed execution. Overall, the distributed setup led to an average slowdown of 1.2× in Spark and 2.3× in Vertica. Spark performs better because it has a higher sensitivity to computation than network communication [28]. Vertica, on the other hand, has more performance variance. It is more sensitive to network communication in some queries; for example, in Q2, Q7, and Q11, the execution incurs heavy communication between workers. In Q12, the distributed setup is even better than the single-machine setting because of good partitioning and higher aggregate disk bandwidth. Comparing these results with DDC slowdowns (Figures 6c, 7c, and 8c), we see that the overhead of scaling out is more significant in DDCs, highlighting the need for optimizations.

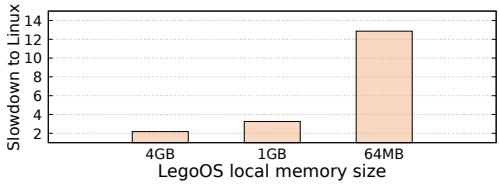


Figure 11: The slowdowns of LegoOS in the TPC-H throughput benchmark. The high-level trends are similar to the observations for single query performance.

4.4 Query Throughput

So far, we have focused on quantifying the slowdown of query completion time; we have similar findings in query throughput. As discussed, the main bottleneck in the DDC setting stems from memory stalls not compute parallelism—in fact, DDCs can spawn as many compute workers as the resource pools allow. We, therefore, observe similar trends for query throughput as we did for individual query completion times.

We evaluate the impact on TPC-H throughput by feeding two streams of TPC-H queries to MonetDB and compare the respective throughput of Linux and LegoOS. Figure 11 shows slowdowns in LegoOS with different local memory sizes, with the highest-level takeaway that the trends match those in Figure 6 for individual queries. The DDC setting, of course, provides new opportunities for rethinking how parallel/concurrent executions can be further optimized [38]. This may require redesigning the underlying OS abstractions and compute models, which we leave for future work.

4.5 Summary

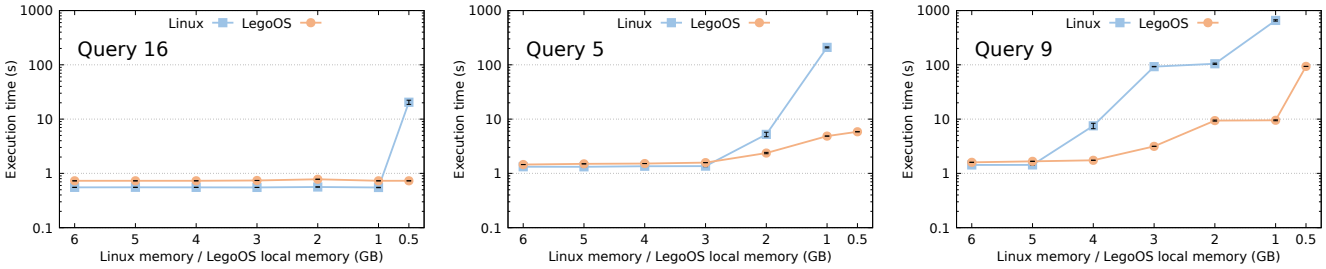
The overhead of DDCs is moderate for in-memory query executions if each query's working set fits into the processing pool's local memory. However, as query memory requirements exceed the local memory, the communication overhead can result in a significant degradation in query execution times. The degradation is even worse under frequent random accesses. In both cases, the interaction between the OS and DBMS-level memory access patterns can heavily influence the effect of disaggregation.

There are significant differences in disaggregation slowdowns in out-of-core vs in-memory systems. Even within out-of-core systems, hot and cold executions vary in slowdowns as well. Cold executions are dominated by disk I/O and hence less sensitive to the network overheads introduced by disaggregation. Hot executions rely too heavily on default LegoOS disk cache management, which stores the cache in remote memory. Overall, out-of-core executions are generally less sensitive to the degree of disaggregation than in-memory executions because they are bottlenecked by other factors, though we note that significant slowdowns can still occur when the degree of disaggregation is extreme (for instance, Q13 in LegoOS with 64 MB local memory).

Moreover, distributed DBMSs set a good baseline for DDCs the cost of scaling out, and highlight the need for codesigning DDCs and DBMSs to avoid redundancy and mismatched policies.

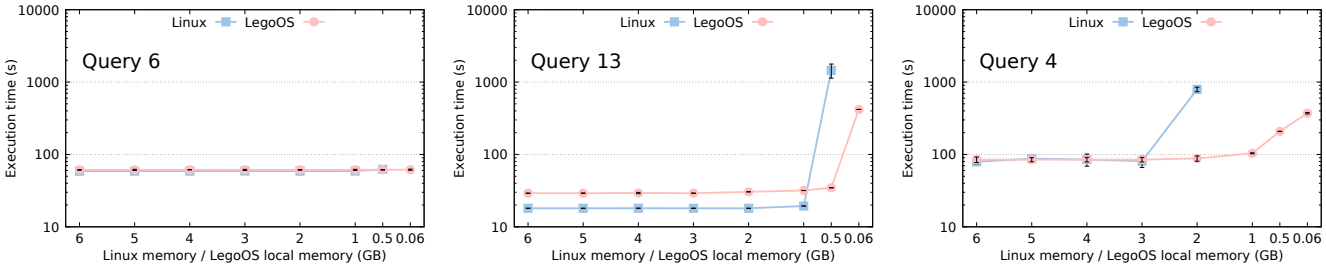
5. THE ELASTICITY OF DDCS

While disaggregation can introduce new overheads, a key advantage of DDCs is their elasticity—a DDC can provision an almost



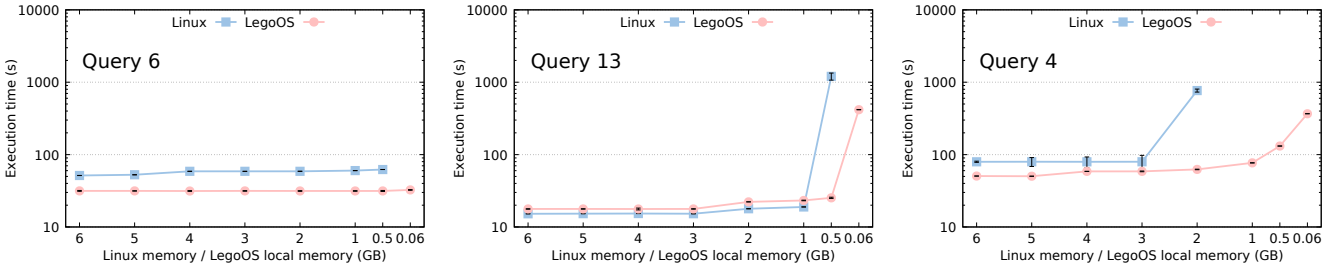
(a) Query 16 performance with memory changes. **(b)** Query 5 performance with memory changes. **(c)** Query 9 performance with memory changes.

Figure 12: Query execution performance of MonetDB when varying memory size in Linux and local memory size in LegoOS.



(a) Query 6 performance with memory changes. **(b)** Query 13 performance with memory changes. **(c)** Query 4 performance with memory changes.

Figure 13: Query execution (cold) performance of PostgreSQL when varying memory size in Linux and local memory size in LegoOS.



(a) Query 6 performance with memory changes. **(b)** Query 13 performance with memory changes. **(c)** Query 4 performance with memory changes.

Figure 14: Query execution (hot) performance of PostgreSQL when varying memory size in Linux and local memory size in LegoOS.

arbitrary amount of resources to each process, and this provisioning can expand beyond the resources contained in any one server. This elasticity can have concrete performance benefits, preventing the DBMS from needing to spill data to disk when it overwhelms a single machine's capacity.

To evaluate these effects, we compare LegoOS's efficiency to that of a monolithic server across varying local memory capacities and working set sizes. For some of the more constrained local memory sizes, we note that it is unlikely that monolithic servers will be built with such limited memory; instead, the goal of the experiments is to isolate the implications of having a pool of remote memory that is orders of magnitude larger than local memory.

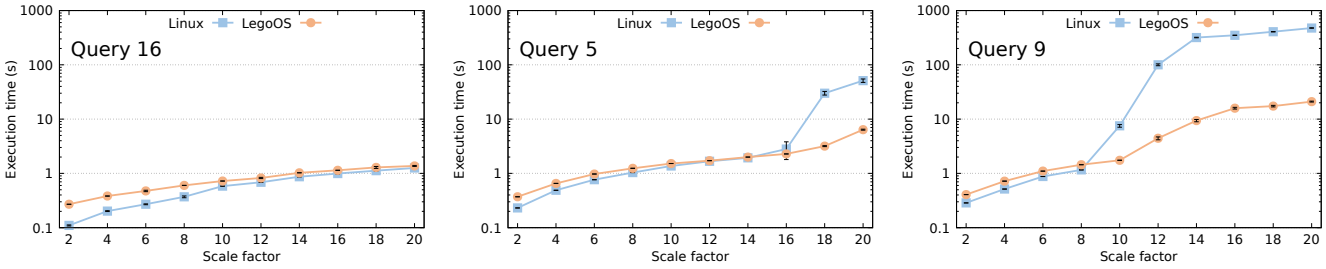
5.1 Versus a Constrained Monolithic Server

We begin by matching and scaling down the local memories of both the LegoOS processing node and a monolithic server in order to emulate a case where today's monolithic servers are augmented with a large pool of remote memory. This is in contrast with the previous section in which we matched the total amount of *remote* memory in the DDC to the memory of the monolithic server. In some ways, the latter represents a lower bound on the relative performance of DDCs. This subsection represents an upper bound.

As before, we fix the scale factor of the TPC-H workload at 10 and set the memory pool capacity to 16 GB, large enough for this particular workload.

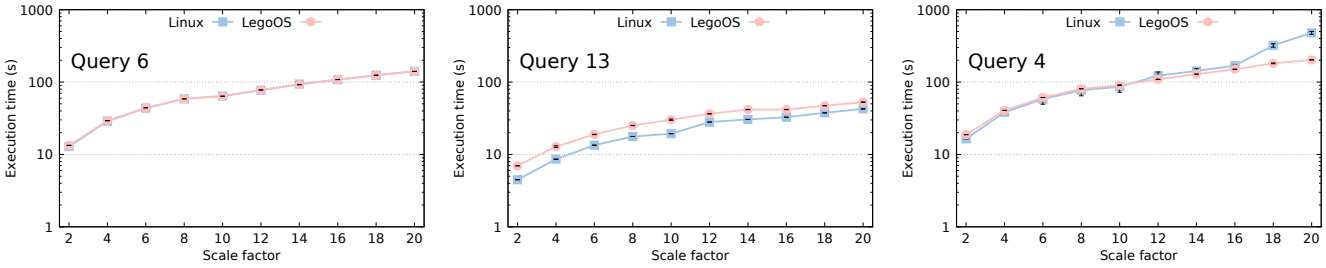
In-memory execution. We select three representative queries with which to explore these effects: Q16, Q5, and Q9. These three queries represent three different levels of sensitivity to local memory capacity: low, medium, and high, respectively (cf. Figure 6c). Other queries with similar sensitivity exhibit similar results. Figures 12a, 12b, and 12c show the execution times of all three queries against local memory capacity in both a single server and a DDC.

As expected, the low-sensitivity query, Q16, maintains its performance across different local memory capacities in the DDC. The monolithic server also retains its (slightly better) performance across most memory capacities; however, when memory is very constrained, performance suffers greatly as data is spilled to disk, with a ∼37× slowdown when memory is constrained to 512 MB. LegoOS is 28× faster than the monolithic server in this scenario. The two more sensitive queries, Q5 and Q9, exhibit similar effects except that the monolithic server slows down much earlier. In fact, for both queries, the 512 MB case fails in the monolithic server with an out-of-memory error. The DDC is still able to execute the queries with a more graceful degradation in performance (but with similar scalability trends). Execution time does rise as memory becomes very constrained, but even in that case, the DDC is consistently 1–2 orders of magnitude faster as data is spilled to remote memory rather than disk. For instance, the most sensitive query in the monolithic server at the smallest capacity that completes, 1 GB, experiences a 460× slowdown compared to LegoOS on the DDC.



(a) Query 16 performance. **(b)** Query 5 performance. **(c)** Query 9 performance.

Figure 15: Query execution performance of MonetDB when varying data set size in Linux and LegoOS.



(a) Query 6 performance. **(b)** Query 13 performance. **(c)** Query 4 performance.

Figure 16: Query execution performance (cold) of PostgreSQL when varying data set size in Linux and LegoOS.

Out-of-core execution. In evaluating PostgreSQL, we selected another three representative queries: Q6, Q13, and Q4 for low, medium, and high sensitivity, respectively. These three queries are different from the three queries chosen for MonetDB as the two DBMSs generate different plans with different sensitivities.

Figure 13 shows the results of cold executions in PostgreSQL for the three queries. Unsurprisingly, the DDC performance on the low-sensitivity query is again very stable across local memory capacities. Also like the in-memory case, the monolithic server begins to fail in low-memory situations. For both environments, these graphs provide a fine-grained record of performance degradation versus local memory size, showing exactly where local memory becomes the bottleneck of the execution.

Overall, LegoOS performance is significantly more stable across local memory sizes. There are two main reasons for this. The first is related to how query planning is done in a DDC versus a traditional server. One of the key inputs to a query planner is the size of memory—different memory sizes can result in significantly different plans and performance, and a wrong choice in a plan can have bad consequences. When creating a plan for a DDC, LegoOS presents to the DBMS the size of remote memory, rather than local memory. Second is the aforementioned conversion of disk spills to remote memory spills. Disaggregation thus provides an easy to understand scaling model: *When disk I/O dominates the time of a pipelined execution, disaggregation causes no harm; when memory becomes stringent in a single server, disaggregation provides better performance and more graceful degradation.*

Figure 14 presents results for hot executions, which show similar trends to the cold executions. One notable difference is that LegoOS benefits from hot executions of all three queries due to its use of the OS disk cache for repeated loading of the same data. In contrast, the monolithic deployment fails to show a similar improvement because there is insufficient memory in the monolithic server to cache the largest tables used in each query.

5.2 The Impact of Dataset Size

Next, we compare how monolithic servers and DDCs scale with their workload. To do this, we fix the memory capacity of both the monolithic server and the DDC processing node to 4 GB, and we vary the scale factor (SF) of TPC-H from 2 to 20 with a step of 2.

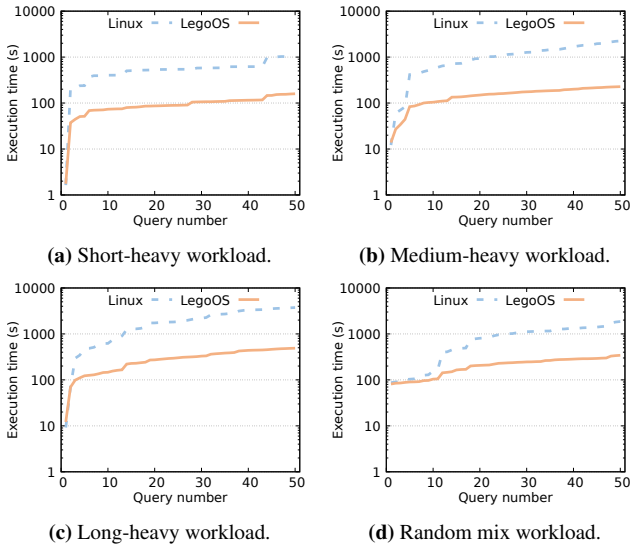
Figure 15 shows the query execution times for Q16, Q5, and Q9, the same three queries used in the previous subsection for MonetDB. As MonetDB does not require much memory to execute Q16, which joins three small tables (`part`, `part supp`, and `supplier`), 4 GB memory is enough for even SF 20. Execution time, therefore, grows slowly with the size of the data set in both disaggregated and non-disaggregated environments.

For the queries with higher memory sensitivity, LegoOS significantly outperforms the monolithic server on large data sets, just as it did when we decreased local memory in Section 5.1. For example, in Q5, when the SF is 16 or above, i.e., when the input size exceeds physical memory on the monolithic server, MonetDB runs faster in the DDC. At SF 20, the speedup is 6.8×. This effect occurs much earlier for Q9, where SF 12 already results in DDCs having a 27× speedup compared to the monolithic server.

PostgreSQL's cold execution performance, shown in Figure 16, also reflects the results of Section 5.1. We omit hot executions as the trends are similar. In each case, LegoOS demonstrates comparable performance to the monolithic machine, except for memory-sensitive workloads and large data sets. In these cases, monolithic server performance degrades quickly as soon as local memory is insufficient for the working set. Finally, we note that, in both deployments, PostgreSQL has better overall scaling effects than MonetDB, partially due to the role of disk I/O as the bottleneck.

5.3 Large, Compound Workloads

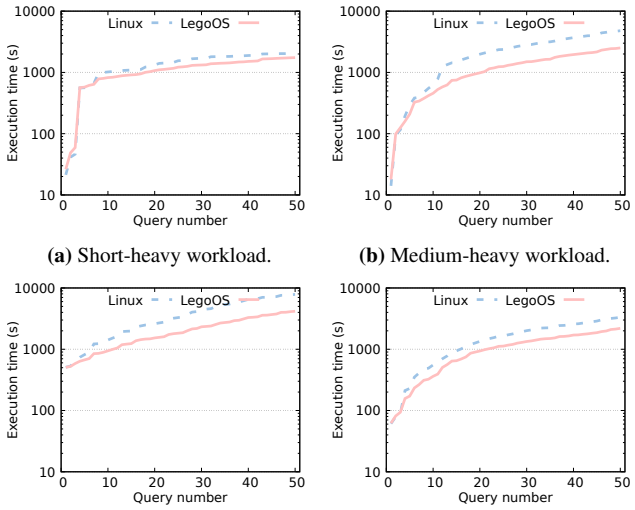
Finally, we extend the above experiments to cases where the workload is large and involves multiple query workloads. Specifically, we fix the physical memory of the monolithic server and local memory of the DDC processing nodes to 4 GB, and we fix the TPC-H scale factor to 10. In this environment, we randomly draw 50 queries from the set of all 22 TPC-H queries. We classify the queries into three categories by their execution times: short, medium, and long queries. We control the portions of short queries (S), medium queries (M), and long queries (L) to create four configurations: (1) *short-heavy workload:* 80% S, 10% M, and 10% L;



(a) Short-heavy workload. **(b)** Medium-heavy workload.

(c) Long-heavy workload. **(d)** Random mix workload.

Figure 17: MonetDB query execution performance in Linux and LegoOS with mixed workloads starting with cold memory.



(a) Short-heavy workload. **(b)** Medium-heavy workload.

(c) Long-heavy workload. **(d)** Random mix workload.

Figure 18: PostgreSQL performance with mixed workloads.

(2) *medium-heavy workload:* 10% S, 80% M, and 10% L; (3) *long-heavy workload:* 10% S, 10% M, and 80% L; and (4) *random mix:* each query has equal probability. After the queries are drawn, we permute and evaluate them sequentially in MonetDB and PostgreSQL starting from cold buffer pools, i.e., no prior cached data.

The results are presented in Figure 17 and 18. The x-axis denotes the query progress within the 50-query trace. The y-axis is the cumulative execution time up to and including that query. In the monolithic server, due to limited memory, MonetDB selects more memory-constrained and less efficient execution plans, while in the DDC, it can take advantage of enough memory in the memory pool to execute the queries more efficiently. For the first few queries, MonetDB has similar performance in both deployments because of the data loading, which is dominated by disk I/O. When more queries have been executed and the buffer pool warms, the DDC becomes increasingly effective compared to today's systems due to its additional remote memory.

The effect is most pronounced in MonetDB, where the short-, medium-, long-heavy, and mixed workloads exhibit total speedups

Figure 19: The effect of prefetching in PostgreSQL.

of 6.7×, 9.9×, 7.7×, and 5.4×, respectively. These speedups manifest quickly. For long-heavy workloads, for instance, the speedup is 3.1× at only 5 queries and 6.1× at 25. The relative speedup of PostgreSQL is lower because it uses out-of-core execution. The speedups range from 1.2×–1.9× across all workload mixes.

5.4 The Effect of Prefetching

Prefetching the data to be used in the execution from disk can mitigate the I/O bottleneck for out-of-core systems. We evaluate this effect in PostgreSQL through the `pg_prewarm` module, which allows the user to preload specified tables into either the OS cache or the buffer pool. Figure 19 shows the results of applying this module in LegoOS with 4 GB local memory. To ease the comparison, we normalized the times of prefetching, execution after prefetching, and hot execution to the cold execution time, and we stacked the first two to show the overhead of prefetching. There are a few interesting findings. Overall, prefetching can effectively cache necessary data: the performance of executions after prefetching matches the performance of hot executions. Although the total times of prefetching and the following execution are generally higher than cold execution times, we can leverage a large memory pool in a DDC to make the prefetching a one-time overhead: prefetching all tables in the memory pool for arbitrary queries.

5.5 Summary

For both in-memory and out-of-core executions, the resource consolidation of DDCs can provide applications with much more memory than a single server. For that reason, resource disaggregation provides better and more stable performance than a single server for DBMSs when the execution reaches the memory limit in the server and they have to spill data to disk.

The experiments of mixed workloads further validate the advantage of disaggregation in having more resources to provision for a single program. This advantage potentially enables DBMSs to scale to much larger workloads than when in a single server.

The disaggregated deployment can also cache additional data in the memory pool, through either historical queries or prefetching. Better caching leads to higher performance.

In addition, we note that the advantage of DDCs does not require any changes in DBMSs. DBMSs can be directly run in a DDC to utilize more resources, while alternative approaches to acquiring more memory, e.g., distributed [13, 30] or RDMA-based [14, 20, 10] DBMSs typically require drastic architectural changes.

6. ANALYSIS AND TUNING

Section 4 shows that with the same resource capacity, DBMS executions are slower in a DDC than in a single server due to higher memory access latency. It also shows that this overhead depends on both the DBMS workload and the degree of disaggregation. In this

section, we analyze this overhead through the profiling statistics we acquire in LegoOS and consider how we might tune DBMS performance in DDCs based on this analysis. The goal here is to gain insight into potential ways to modify the behavior of DBMSs to reduce (or mask) the overhead of disaggregation.

6.1 Remote Memory Access Analysis

We measure the hardware counters for page faults and InfiniBand communication volume between local memory and remote memory in LegoOS to estimate N_{RM}—the amount of remote memory data transferred (in bytes) during an execution. We first present the results for in-memory executions in MonetDB.

6.1.1 In-memory execution

Figure 20 shows N_{RM} for the experiments in Section 4.1, where we configured different local memory sizes. Figure 20a shows the statistics for the setting where local memory is enough for most queries. We make two key observations: (1) all queries have non-zero N_{RM}; and (2) most queries have N_{RM} smaller than 1 GB.

The first observation suggests that the overhead of disaggregation is inevitable: there are remote memory accesses even when the local memory is larger than what an application demands. Those remote memory accesses include program data and the initial transfer of data into processing nodes' local memory.

The second observation (combined with the $< 2\times$ slowdowns in Section 4.1) suggests that, for most queries, the extra latency due to this overhead is smaller than the execution time in a single server. There are, however, a few exceptions: Q17 transfers almost 10 GB data and its execution time is inflated by $2\times$ (shown in Figure 6a); the N_{RM} of Q9 is 500 MB, which incurs a $2.9\times$ slowdown. For those queries, remote memory accesses dominate the execution times. In fact, the impact of memory stalls on the execution time is highly dependent on queries. As examples, Q1 and Q7 transfer 10 GB and 2.3 GB data, respectively, but both of them cause a $1.6\times$ slowdown because computation dominates the execution time. In comparison, Q11 transfers only 8 MB data, but it has a $3.3\times$ slowdown—this suggests that low-latency queries are more sensitive to memory stalls.

Figure 20b shows the results when LegoOS has 1 GB local memory. The average and median N_{RM} has increased to 3.3 GB and 1.7 GB respectively, significantly higher compared to the case with 4 GB local memory. This is because most TPC-H queries consume more than 1 GB memory, as shown in Figure 4, and therefore across the execution, virtually all cached data has to be transferred from the memory pool. This shows the mismatch between DBMS execution and the current OS caching mechanism that uses LRU or its variants (e.g., FIFO). This mismatch results in serious performance degradation: $\sim 5\times$ on average as shown in Figure 6b. When the local memory size is as low as 64 MB (Figure 20c), the average N_{RM} increases to 9.1 GB and the median to 3.8 GB, showing that the buffer pool data is accessed multiple times. Those multiple rounds of data transfers cause an order of magnitude performance degradation (Figure 6c). As an extreme case, Query 9 transfers 89 GB data, causing two orders of magnitude degradation.

6.1.2 Out-of-core execution

For out-of-core executions in PostgreSQL, although the performance slowdowns of cold and hot executions are very different (Figures 7 and 8), the remote memory accesses are similar. This is because PostgreSQL relies on the OS to cache the raw input data when it reads from files, rather than loading/storing it to virtual memory or its buffer pool directly. LegoOS caches this disk data in memory nodes and storage nodes and not in local memory. As a result, even for hot executions, the DBMS needs to access remote memory for the cached data.

Figure 21a plots the results for hot executions in PostgreSQL in a DDC processing node with 4 GB local memory. The average N_{RM} is 9 GB, which is much higher than the peak memory usage derived in Figure 5. The reason we did not observe an associated slowdown in the cold executions of Section 4.2 is pipelined execution, which hides these accesses by the time spent in disk I/O. The absence of good pipelining is why we do see a slowdown of $\sim 2.2\times$ in the hot executions of Section 4.2. Moving to limited local memory sizes (Figures 21b and 21c) increases the average N_{RM}, resulting in a $1.2\times$ and $6.3\times$ increase in the average slowdown, respectively. In the latter case, the PostgreSQL buffer pool is frequently evicted. The queries with the largest N_{RM} (Queries 4, 9, 13, 18, 21, and 22) also have the worst performance slowdowns (cf. Figures 7 and 8).

6.1.3 Summary

This set of experiments quantitatively evaluates the overhead of resource disaggregation that comes from remote memory accesses, which, being synchronous, stall the DBMS execution. The results also reveal two mismatches between the current OS design and the DBMS data access patterns: (1) LRU-like local memory eviction policies are a poor fit for DBMSs when the local memory in the processing pool is too small to cache the working set; and (2) out-of-core executions that rely on the OS to cache raw input data from the disk cannot take advantage of processing units' local memory since the cached data is still remote to the processing pool.

We note that the absolute numbers in this analysis are from running existing DBMSs directly in LegoOS—neither the DBMS nor LegoOS is aware of the other side. More flexible data access granularity that leverages the patterns of DBMS workloads can improve data transfer efficiency in LegoOS for DBMS executions, but we leave this optimization as future work.

6.2 Plan Optimality

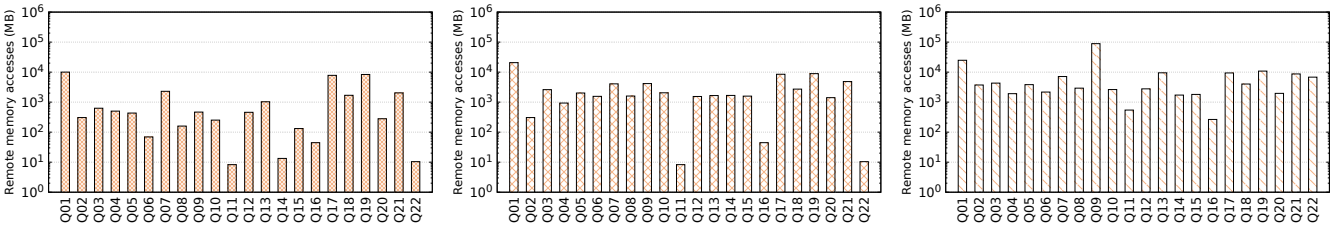
Plan selection is an important function of the query planner and optimizer. We measure the impact of different execution plans on performance. In particular, we focus on two aspects: a) the size of the buffer pool, and b) the join algorithm.

6.2.1 Buffer Pool Size

The size of the buffer pool is a key determining factor for the execution plan chosen by the DBMS. To measure the impact of this choice, we focus on MonetDB for two reasons: (1) as an in-memory DBMS, it is more sensitive to memory size, and (2) the main bottleneck of PostgreSQL is either disk I/O (in cold executions) or network communication overhead incurred by fetching cached data from remote memory (in hot executions), so the size of the buffer pool is not a dominant factor. We study three representative queries: 16, 5, and 9, and evaluate them with the workload configured to a scale factor of 10. We vary the buffer pool size between 16 GB (enough memory), 4 GB (reasonably large) and 1 GB (small), and test two LegoOS configurations: one with 4 GB local memory and one with 64 MB of memory in each processing node. The baseline is a monolithic server with 16 GB memory.

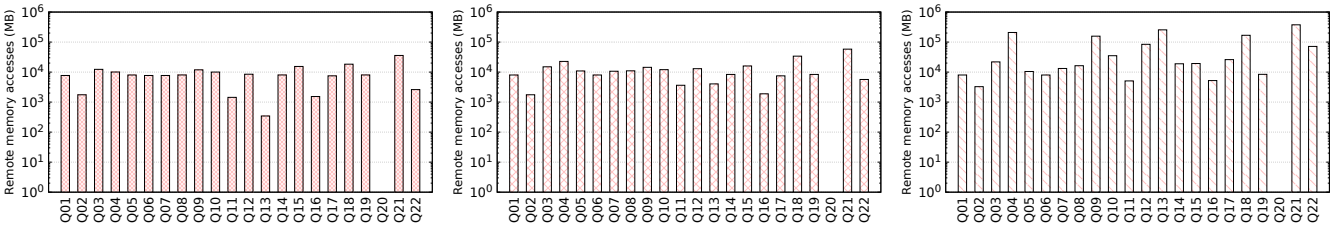
Figure 22 shows the results. In the monolithic server, a larger buffer pool results in better performance. The DDC results are similar with one exception: Q16 performs marginally better with 1 GB memory than it does with 4 GB (0.7 s vs. 0.73 s), attributed to noise. We also observed that MonetDB's query planner had some difficulty in planning for intermediate buffer pool sizes in Q16.

A somewhat surprising result is that, when moving to smaller buffer pool sizes compared to available memory, the penalty to Le-



(a) LegoOS with 4 GB local memory. **(b)** LegoOS with 1 GB local memory. **(c)** LegoOS with 64 MB local memory.

Figure 20: Remote memory accesses (in MB) in MonetDB executions with different levels of disaggregation.



(a) LegoOS with 4 GB local memory. **(b)** LegoOS with 1 GB local memory. **(c)** LegoOS with 64 MB local memory.

Figure 21: Remote memory accesses (in MB) in hot PostgreSQL with different levels of disaggregation. Cold results are almost identical.

	Buffer Pool Size	16 GB	4 GB	1 GB
Query 16	Linux	**0.5 s**	0.75 s	0.53 s
	LegoOS (4 GB)	0.73 s	10.78 s	**0.7 s**
	LegoOS (64 MB)	**1.29 s**	11.13 s	1.37 s
Query 5	Linux	**1.14 s**	1.14 s	5.08 s
	LegoOS (4 GB)	**1.44 s**	1.44 s	18.11 s
	LegoOS (64 MB)	**8.72 s**	8.9 s	52.74 s
Query 9	Linux	**1.11 s**	2.2 s	9.65 s
	LegoOS (4 GB)	**1.7 s**	10.55 s	40.81 s
	LegoOS (64 MB)	**178.55 s**	190.8 s	257.53 s

Figure 22: MonetDB buffer pool size tuning in Linux and LegoOS

	Join Algorithm	Nested-Loop	Merge	Hash
Query 13	Linux	>1 h	**14.45 s**	15.69 s*
	LegoOS (4 GB)	>1 h	**18.34 s**	18.51 s*
	LegoOS (64 MB)	>1 h	**122.84 s**	373.49 s*
Query 4	Linux	**3.97 s**	30.55 s	26.5 s*
	LegoOS (4 GB)	**15.95 s**	59.34 s	57.42 s*
	LegoOS (64 MB)	**18.82 s**	351.78 s	348.58 s*

Figure 23: PostgreSQL join algorithm tuning in Linux and LegoOS. Algorithms marked with * are what PostgreSQL selected.

goOS is outsized. When data needs to be spilled out of the buffer pool, one might think that the monolithic server would need to spill to disk and that LegoOS, spilling to the remote memory pool would gain an advantage. In fact, it is the opposite. If the buffer pool is less than the total memory of the system, the monolithic server can spill data to memory. That memory is local, unlike the memory to which LegoOS spills data, which is on a remote machine. Thus, spilling data out of the buffer pool comes at a significantly higher cost in a DDC than a traditional system.

6.2.2 Join Algorithm

We next evaluate the performance characteristics of join algorithms. We use PostgreSQL because unlike MonetDB, it allows the user to select from three different join algorithms: `Nested-Loop Join`, `Merge Join`, and `Hash Join`. We disable two of them to ensure that PostgreSQL selects the remaining one. For example, to use `Hash Join`, we set `enable_nestloop` and `enable_mergejoin` to `off`. Since Q6 does not involve joins, we evaluate only Q4 and Q13. We use Linux with 16 GB memory as the baseline, configure LegoOS to use a 16 GB memory pool and vary the local memory size between 4 GB and 64 MB.

Figure 23 shows the results. For Q13, nested-loop join cannot finish within 1 hour in both Linux and any LegoOS setting; merge join has slightly better performance than hash join in this query. Since merge joins incur less random accesses, they are much more efficient than hash joins when local memory is small: merge join incurs 62 GB of remote memory accesses when the local memory is 64 MB, but hash join incurs 250 GB. This is an interesting observation because PostgreSQL suboptimally selects hash joins when all join algorithms are enabled. Q4 is different: nested-loop join is the best algorithm (i.e., uses the least amount of memory) in both Linux and LegoOS. When running in LegoOS with 64 MB of local memory, nested-loop join, hash join, and merge join incur 8 GB, 134 GB, and 149 GB of remote memory accesses, respectively. Unlike in Q13, hash join performs slightly better than merge join in this query. Again, PostgreSQL suboptimally chooses hash join (over nested-loop join). These experiments show that join algorithms with small memory footprint work better in DDCs, but current DBMSs (PostgreSQL) do not make this choice. It might be possible (and even profitable) to design join algorithms that are tailored for DDCs.

6.3 OS Configuration

We now examine how changing different parameters in the underlying disaggregated OS can affect the performance of DBMS. In the previous sections, we observe that the bulk of the overhead comes from accessing remote memory, which could in principle be improved with better caching. We, therefore, experiment with two key choices: the cache eviction and the cache placement policies.

6.3.1 Cache Eviction Policy

LegoOS supports two eviction policies: FIFO and LRU. Neither one favors DBMSs workloads. We evaluate how these policies affect the execution of MonetDB and PostgreSQL and find that there is little difference in terms of the number and size of remote memory accesses for both eviction policies. As one example,

Set Associativity	1-way	256-way	8K-way
MonetDB	22,763,177	22,762,688	22,721,776
PostgreSQL	35,227,319	35,216,064	35,108,257

Figure 24: Page faults in different set associativity configurations.

consider the setting where the local memory size is 64 MB (where eviction frequently happens) and where we use 256-way set associativity using the most memory-intensive queries in both MonetDB (Q9) and PostgreSQL (Q4). In this setting, MonetDB with LRU incurs 22,763,745 page faults that trigger remote memory accesses (each page has 4 KB size) while FIFO incurs 22,763,053 page faults. Similarly, in PostgreSQL, LRU is roughly the same as FIFO (35,216,064 vs. 35,216,712) and this is consistent between cold and hot executions. However, as described in LegoOS [33], LRU introduces lock contentions on the LRU list; Query 9 in MonetDB finishes in 179.16 s with FIFO and 181.76 s with LRU.

6.3.2 Cache Placement Policy

To evaluate the effect of cache placement policies, we vary the set associativity for the local memory. We study 1-way, 256-way, and half of fully associative (8K-way for 64 MB local memory, the highest associativity that LegoOS supports). As in the previous experiment, we use the most memory-intensive queries in MonetDB and PostgreSQL and a 64 MB local memory in LegoOS to magnify the effect of caching mechanisms. The results are in Figure 24.

We find that increasing the set associativity improves the hit rate of the local memory so that the remote memory accesses are reduced in both MonetDB and PostgreSQL. However, the reduction on the remote memory accesses is not significant from the lowest to the highest associativity, and this benefit is offset by the cost of maintaining high set associativity so we do not observe a significant difference between the execution times of different configurations.

In conclusion, switching between current configurations without resource capacity change has little to no effect on data-intensive executions. We leave the codesign of the OS and the DBMSs, which we believe can improve this state of affairs, as future work.

7. RELATED WORK

Resource disaggregation offers great benefits for cloud operators, but as we speculate in our position papers [38, 8], significant changes to disaggregated OSes and applications are crucial to achieve reasonable performance. The present work provides the first empirical backing for such claims, with a comprehensive evaluation of production DBMSs on a disaggregated data center setup. In the process, we highlight different sources of overhead and uncover many subtleties that were not obvious a priori, such as the effect of the disk caching policy (manual or delegated to the OS) and the placement of a DBMS buffer pool. In addition, we evaluate the benefits of DDC elasticity for data-intensive DBMS workloads.

We now discuss other related work. A key distinction is our focus on DBMSs, our experimental setup, and results. We remark, however, that our general call to action—that codesign is not only desirable but necessary in this case—shares a similar ethos to decades-old proposals to codesign OSes and DBMSs [35, 18].

Operating systems for disaggregated data centers. With the advent of DDC hardware, the first wave of software innovation has focused on operating systems support. For example, LegoOS [33] introduces an operating system that decouples hardware resources and connects them with a fast network, while still providing the abstraction of a single machine to applications, which can be run without modifications. Other proposals include new architectures [21, 22], network architectures [17, 34, 12], and data structures for remote memory [5]. Our work is the first to explore application-level optimizations, in this case how DBMS performance is impacted and can be improved by disaggregation.

Remote memory and distributed shared memory. Prior work has revisited the idea of remote memory (RM) in fast data center networks [4], proposing a standard API for RM access with exported files [3]; implementing generic data structures like vector, map, and queue for RM by customizing hardware primitives [5]; and designing a new paging system to avoid application modification [19]. Previous work has proposed novel memory management for DBMSs to utilize remote memories [20, 10, 16] and even provided distributed shared memory (DSM) abstraction [11, 32, 14]. While RM and DSM, and disaggregation overlap in spirit, the ideas differ in that previous work assumes that significant resources remain coupled with the compute components while disaggregated data centers target at completely separating computation and memory. This fundamental difference has implications for buffer management, cost estimation, physical executions, and other important components in DBMSs.

Hierarchical storage management. DDCs have a richer memory and storage hierarchy than traditional distributed environments. Existing work has investigated improving DBMSs on hierarchical storage, such as cache-aware DBMS execution [24, 27], caching [23, 37], and memory management for new hardware [36, 9]. We believe that the existing work would serve as further inspiration for DDC optimizations while noting that the separation between compute and memory represents a radical change unique to DDCs.

8. CONCLUSION AND FUTURE WORK

This paper conducts a detailed study of two production DBMSs, PostgreSQL and MonetDB, to understand the performance implications of deploying them on a disaggregated data center. We find that a wide variety of factors come into play, including in-memory vs. out-of-core execution, degree of disaggregation, the choice of join algorithms, and buffer pool sizes. One interesting finding relates to the entity that manages memory and, in particular, the disk cache. If it is the application, as is the case with MonetDB, some of the data can be kept in local memory. If it is the OS, as is the case with PostgreSQL, the disk cache is kept in remote memory, which hurts performance. We also find that DDCs can actually be beneficial (in some settings) to DBMSs!

These results shed light on exciting research opportunities, including designing DDC-aware buffer pool policies that differentiate between local memory and remote memory, and leverage the qualities of each; looking at relational operators (including joins) that are better fit for DDCs (e.g., operators that have a smaller memory footprint but that perhaps require additional computation); developing new cost models for query planning and optimization; and optimizing parallel DBMSs by avoiding unnecessary data shuffling in the network (since the data of multiple processes lives in the same memory pool). Finally, DDCs are potentially more amenable to mechanisms that ensure performance isolation, and we are interested in exploring DBMS designs for multi-tenant settings.

Acknowledgments

We thank Zack Ives for conversations that improved this work. We also thank the anonymous reviewers for their thoughtful feedback. This work was supported in part by NSF grants CNS-1513679, CNS-1801884, CNS-1845749, CNS-1942219, DARPA Contract No. HR001117C0047, and ONR N00014-18-1-2618.

9. REFERENCES

[1] Apache spark - unified analytics engine for big data. https://spark.apache.org.

[2] Big data analytics on-premises, in the cloud, or on hadoop — vertica. https://www.vertica.com.

[3] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018.

[4] 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.

[5] 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.

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

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

[8] S. Angel, M. Nanavati, and S. Sen. Disaggregation and the application. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, July 2020.

[9] J. Arulraj and A. Pavlo. How to build a non-volatile memory database management system. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1753–1758. ACM, 2017.

[10] 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.

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

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

[13] Citus-Data. Citus data: Worry-free postgres. built to scale out. https://www.citusdata.com/, 2019.

[14] 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.

[15] 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)*, July 2019.

[16] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server database architectures. In L. Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, pages 596–609. Morgan Kaufmann, 1992.

[17] 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.

[18] J. Giceva, G. Zellweger, G. Alonso, and T. Roscoe. Customized OS support for data-processing. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 2:1–2:6. ACM, 2016.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] X. Liu, A. Aboulnaga, K. Salem, and X. Li. CLIC: client-informed caching for storage servers. In M. I. Seltzer and R. Wheeler, editors, *7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA. Proceedings*, pages 297–310. USENIX, 2009.

[24] S. Manegold, P. A. Boncz, and N. Nes. Cache-conscious radix-decluster projections. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 684–695. Morgan Kaufmann, 2004.

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

[26] MonetDB. Monetdb - the column-store pioneer. https://www.monetdb.org/Home, 2019.

[27] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber. Cache-efficient aggregation: Hashing is sorting. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1123–1136. ACM, 2015.

[28] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[29] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *PVLDB*,

11(6):663–676, 2018.

[30] Postgres-XL. Postgres-xl: Open source scalable sql database cluster. https://www.postgres-xl.org/, 2019.

[31] PostgreSQL. PostgreSQL: The world's most advanced open source relational database. https://www.postgresql.org/, 2019.

[32] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojevic, D. Narayanan, and M. Castro. Fast general distributed transactions with opacity. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 433–448. ACM, 2019.

[33] 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.

[34] 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.

[35] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7), June 1981.

[36] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. Managing non-volatile memory in database systems. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1541–1555. ACM, 2018.

[37] G. Yadgar, M. Factor, K. Li, and A. Schuster. Management of multilevel, multiclient cache hierarchies with application hints. *ACM Trans. Comput. Syst.*, 29(2):5:1–5:51, 2011.

[38] Q. Zhang, Y. Cai, S. Angel, A. Chen, V. Liu, and B. T. Loo. Rethinking data management systems for disaggregated data centers. In *Proceedings of Conference on Innovative Data Systems Research (CIDR)*, Jan. 2020.