Optimizing Data-intensive Systems in Disaggregated Data Centers with TELEPORT

Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen[§], Vincent Liu, Boon Thau Loo
University of Pennsylvania, [§]Rice University
{qizhen, cxinyic, sankhe, zhileiz, kezhong, sga001, liuv, boonloo}@seas.upenn.edu, [§]angchen@rice.edu

ABSTRACT

Recent proposals for the disaggregation of compute, memory, storage, and accelerators in data centers promise substantial operational benefits. Unfortunately, for resources like memory, this comes at the cost of performance overhead due to the potential insertion of network latency into every load and store operation. This effect is particularly felt by data-intensive systems due to the size of their working sets, the frequency at which they need to access memory, and the relatively low computation per access. This performance impairment offsets the elasticity benefit of disaggregated memory.

This paper presents TELEPORT, a *compute pushdown* framework for data-intensive systems that run on disaggregated architectures; compared to prior work on compute pushdown, TELEPORT is unique in its efficiency and flexibility. We have developed optimization principles for several popular systems including a columnar in-memory DBMS, a graph processing system, and a MapReduce system. The evaluation results show that using TELEPORT to push down simple operators improves the performance of these systems on state-of-the-art disaggregated OSes by an order of magnitude, thus fully exploiting the elasticity of disaggregated data centers.

CCS CONCEPTS

• **Information systems → Data management systems**; **Parallel and distributed DBMSs**; • **Networks → Data center networks**.

KEYWORDS

Memory disaggregation, Data processing, Compute pushdown

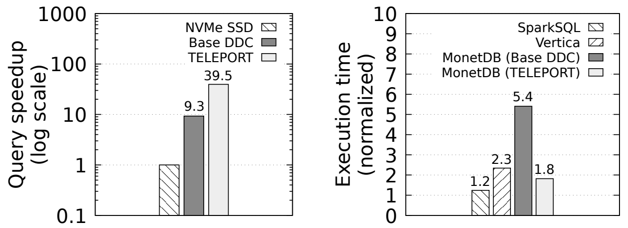
ACM Reference Format:
Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen[§], Vincent Liu, Boon Thau Loo. 2022. Optimizing Data-intensive Systems in Disaggregated Data Centers with TELEPORT. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3517856

1 INTRODUCTION

Resource disaggregation promises to fundamentally change the way in which we design and operate cloud infrastructure. Unlike today's data centers where each server contains enough compute, memory, and storage to execute tasks autonomously, the hardware resources of a disaggregated data center (DDC) are partitioned into physically distinct resource *pools* (e.g., a pool of blades that houses the majority of CPUs, a pool that houses the majority of DRAM/NVM, etc.) all connected via a fast network fabric. This distribution is not only beneficial to the operational and cost efficiency of data centers [49], it also enables more elastic provisioning of resources that expand beyond a single machine [56]. This, in particular, is attractive to data-intensive systems in which the presence of a large memory pool can reduce the amount of data that is spilled to secondary storage, hence improving overall performance. Figure 1a demonstrates this benefit empirically (using memory-intensive TPC-H queries): the ability to spill an in-memory query execution to remote memory rather than to a local SSD results in an order of magnitude of performance improvement when memory is constrained.

There have been a number of recent proposals for resource disaggregation [25, 32, 44]. Some of these propose the complete redesign of applications using novel programming models or custom DBMSs [9, 35, 40, 41, 57]. While these potentially provide good performance in the face of disaggregation, they also typically require radical modifications that block the use of legacy data, applications, and libraries. In contrast, proposals for disaggregated operating systems (OSes) distribute traditional OS responsibilities while emulating the same API/ABI. Applications can, therefore, run with minimal modification. While this, in principle, enables the reuse of existing data-intensive systems like DBMSs and graph processing systems, unfortunately, the performance effects of running these systems unmodified can be significant, offsetting the operation, efficiency, and elasticity benefits of disaggregation.

To demonstrate this issue, Figure 1b evaluates the *cost of scaling* incurred by DDCs. Specifically, it shows the average execution time of TPC-H queries on several data center configurations compared to a purely local execution that uses same resources (i.e., the same amount of CPU, memory, and disks but all in a single



(a) The benefits of DDCs. **(b) The cost of scaling.**

Figure 1: The benefits and cost of running DBMSs in DDCs.

high-end server). For DDCs, we executed MonetDB [4], a single-machine in-memory DBMS on two different disaggregated platforms: LegoOS [43], the current state-of-the-art disaggregated OS, and Teleport, our proposed platform. Both were configured with compute-local memory as 10% of the entire working set. As a reference, we also show the 'cost-of-scaling' for two distributed in-memory DBMSs—SparkSQL [12] and Vertica [1]—running on a more traditional configuration that uses monolithic servers.

The cost of scaling in the above experiments is a result of the insertion of network communication into execution—in the form of paging to/from remote memory in the case of DDCs, and in the form of message passing in the case of distributed execution. Distributed data processing systems—having been thoroughly optimized over decades—successfully achieve a reasonable 'cost of scaling' (average costs are 1.2× and 2.3× in SparkSQL and Vertica, respectively). The cost of the unmodified execution in a state-of-the-art DDC is, unfortunately, significantly higher: 5.4× on average. As we show later in this paper, this cost can, in the worst case, balloon to 52.4× for some common data analytics tasks. This is despite OS-level optimizations in existing DDC platforms such as caching and prefetching which, on their own, are insufficient.

How can we enable all of the operation, efficiency, and usability benefits of DDCs while ensuring a comparable 'cost-of-scaling' to traditional distributed architectures? Our answer is Teleport, a novel OS kernel primitive for DDCs that enables—with a single system call, minimal overhead, and no other application changes—data-intensive systems to choose where to execute their application logic. Conceptually, Teleport's primitive resembles that of compute pushdown: applications can choose to ship complete function calls to remote memory where the functions can execute using local data. For memory-bound tasks, proximity can improve performance by orders of magnitude. For many such operations, minimal computation is required, maintaining the disaggregation of compute and data in the memory pool. As a preview of Teleport's benefits, Figure 1b shows that Teleport can significantly lower the cost of scaling with DDCs and, as a result, can truly unlock the benefits of DDCs (Figure 1a).

Teleport differs from prior work on compute pushdown [19, 20, 22, 30, 36, 38, 46] in its focus on the novel environment of memory disaggregation, in which a process's entire address space resides in the remote memory pool, including the text segment, heap, stack, and full page table—compute-local memory is nothing more than a cache. Assuming a consistent instruction set architecture (ISA) across the compute and the memory pools (but not necessarily homogeneous hardware), applying Teleport to offload a piece of computation to the memory pool is as straightforward as pointing a process running in the memory pool to the correct program counter, stack, and page table residing in the cache of the compute pool. Not only is this more efficient than traditional pushdown mechanisms, it allows for the use of pointers, complex data structures, and open files—the capabilities of a local function—without additional user effort. Teleport's target level of flexibility and ease of use also leads to new challenges unaddressed in prior compute pushdown proposals. For instance, in order to achieve good performance and correctness, updates must be propagated lazily, yet correctly, so as to ensure memory consistency in the presence of distributed execution over a shared process context.

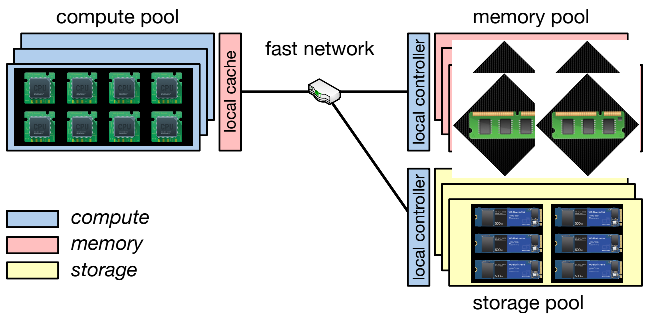


Figure 2: An illustration of resource disaggregation. Same type of resources are centralized in a *resource pool*. Resource pools are disaggregated and connected by a fast network.

In summary, this paper makes the following contributions:

- We introduce the design and implementation of Teleport, a compute pushdown primitive in the OS kernel designed for optimizing data-intensive systems for resource disaggregation. It presents a uniquely flexible and usable abstraction for mitigating overheads from excessive remote memory accesses.
- To handle parallel threads, we describe a set of specialized synchronization primitives (inspired by prior work on MESI cache coherence [37]) that guarantees memory coherence of a logical process context shared across resource pools and multiple concurrent threads within each place.
- Finally, we present a set of pushdown-optimized data-intensive systems (DBMS, graph processing, and MapReduce). Applying Teleport only involved the selective wrapping of existing function calls. These optimized systems are an order of magnitude faster than a state-of-the-art disaggregated OS, even when the memory pool has limited CPU capacity.

2 BACKGROUND AND MOTIVATION

Resource disaggregation is an architectural style in which the resources of a data center, traditionally spread across every server, are instead partitioned into physically distinct pools of resources connected with a fast network fabric such as RDMA over InfiniBand, as illustrated in Figure 2. While today's data centers already disaggregate storage, a defining feature of DDCs is the more complete disaggregation of resources including of memory. As mentioned in prior work, these changes enable substantial operational benefits including independent expansion, allocation, and failures as well as increased density [10, 15, 18, 21, 43, 44, 49, 55, 56]. While pools hosting each type of resource may also contain a small amount of other resources (e.g., low-frequency CPUs in the memory/storage pools that manage local resources and process accesses, or a modest amount of DRAM in the compute pool that caches data), the expectation is that any computation of sufficient size will require coordination across pools spanning different resource types.

In exchange for those benefits, DDCs convert a subset of what used to be local memory and device accesses to remote accesses. While the latest InfiniBand networks are undoubtedly very fast (sub-600 ns latency at 200 Gb/s [5]) and some proposals have advocated for new network substrates [44], both are, nevertheless, much slower than accessing resources on the same motherboard.

2.1 Disaggregated Operating Systems

A disaggregated OS inherits all traditional OS concepts (program contexts, resource allocation, file systems, and isolation) and the original API/ABI. Underneath, the OS implements these functionalities using disaggregated hardware resources. It hides the complexity of infrastructure changes from the data center applications, hence ensuring backward compatibility for big software systems like DBMSs [4, 6, 12] and graph processing systems [17, 23], which have been developed over many years and consist of up to million lines of code [2, 4, 6]. The OS approach is thus more appealing compared to alternatives that either require new programming models [9, 35, 40, 41] or only share subsets of memory [25, 32].

Regardless of the specific architecture, disaggregated OSes allow the complete decoupling of compute and data. Application data in virtual memory spaces resides in the memory pool. The compute pool schedules and executes worker threads/processes with its local memory caching data from the memory pool. This clean separation enables a great benefit—*independent elasticity*, where programs can use an arbitrary number of CPU cores and, independently, allocate arbitrary amounts of memory and storage. For example, DBMSs can create a database of any size in the storage pool, allocate a buffer pool of any size to hold the working set in the memory pool, and spawn any number of query execution workers in the compute pool. Thus, to read a new piece of data from persistent storage, the user-level process in the compute pool will trigger a page fault on its local cache. This page fault is forwarded to the controller in the memory pool, which checks the process's full page table and triggers a recursive page fault that forwards the request to the storage pool. Finally, the requested page will flow back to the CPU node in the reverse direction: the memory controller will page in the data and update the process's page table, and the CPU node will bring that page into its local cache. The whole process is mediated by the disaggregated OS. Traditional OSes would execute these operations in a single machine.

An example of this approach is LegoOS [43], which proposes a *splitkernel* OS. It 'splits' kernel responsibilities across resource-disaggregated nodes, e.g., the piece of the kernel on each compute node manages the process and scheduling of a traditional Linux server, while the pieces on each memory node, focus primarily on memory management. While our Teleport prototype is implemented on top of LegoOS, its core ideas can apply to any disaggregated OS that provides complete compute and data decoupling.

2.2 System Performance in DDCs

Figure 1 shows the benefits of DDCs' large disaggregated memory pools but also their 'cost of scaling.' This cost is particularly pronounced for data-intensive systems due to frequent memory accesses and the fact that local DRAM accesses are an order of magnitude faster than network communications such as RDMA. Consider the following examples:

Database systems. DBMSs are designed to execute SQL queries with low latency and high throughput. Data that is actively used is kept in an in-memory buffer pool to avoid slow disk I/O. In DDCs, however, query execution happens in the compute pool while the buffer pool data lives in remote memory. This arrangement can be expensive. For example, aggregate queries require *all* of the data

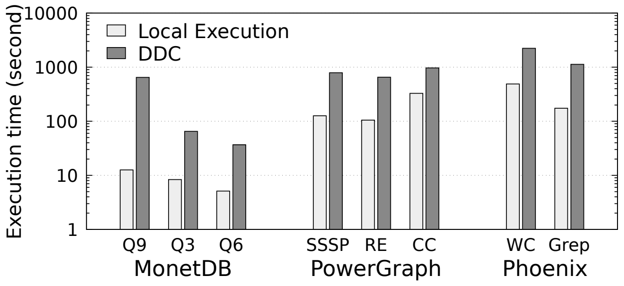


Figure 3: DDC performance overhead compared to a monolithic server. For DBMS (MonetDB), we show three TPC-H queries that have the highest disaggregation cost; for graph processing (PowerGraph) and MapReduce (Phoenix), we show popular benchmark queries (details are in Section 7).

to be brought to the compute-local cache; selections suffer from a similar bottleneck. A more complex example is a binary hash join, which (1) scans the tuples in the outer table, (2) probes the hash index of the inner table, and (3) generates the join results. The random accesses in step (2) can result in substantial cache misses, while step (1) and (3) are a poor fit for typical LRU-based caching strategies [47]. Previous studies [14, 55, 56] show that queries can take up to two orders of magnitude longer to complete (compared to a purely local-memory deployment) for precisely these reasons when the degree of compute-memory disaggregation is high.

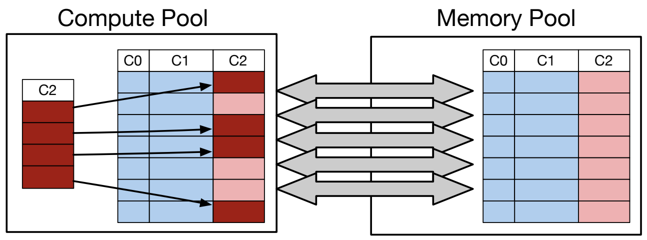
Graph processing. Systems like Pregel [31] and PowerGraph [24] process structured pointer-based graph datasets that lead to unpredictable memory access to different parts of the input graphs depending on the query and data characteristics. In PowerGraph, for example, every gather-apply-scatter iteration requires a vertex to communicate with its neighboring vertices to exchange local data for the next round. In a traditional server, this is simply a set of local accesses; in a DDC, each iteration requires expensive remote memory accesses for large graph states.

Data-parallel frameworks. MapReduce-like systems such as Phoenix [28] have interleaved stages of memory-intensive operations. After each processing stage, workers exchange their results with the next set of workers. When co-located on the same server, the communication is fast; however, in a DDC, intermediate results must all be written just to be fetched back for the next iteration [10].

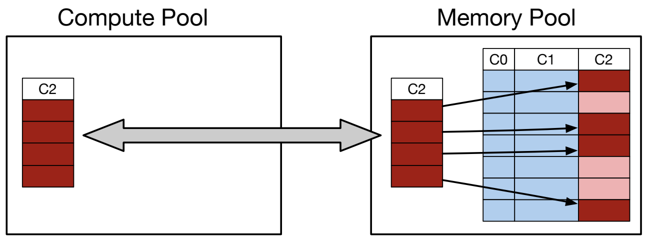
Summary. In short, data-intensive systems have a set of core processing primitives that are computationally lightweight but involve a high degree of memory accesses. Figure 3 shows the results of running typical data-intensive queries in a DDC testbed managed by a state-of-the-art disaggregated OS. Slowdowns range from 5× up to 52.4×. Similar to prior work [56], we find that remote memory accesses dominate the slowdowns of these systems, and argue that *the slowdowns are unavoidable with a constrained cache size in the compute pool.* Our position is that, by optimizing the placement of computation, Teleport can dramatically improve performance.

2.3 Benefits of Compute Pushdown

In this paper, we focus on alleviating the memory bottleneck by selectively performing operator pushdown from compute to memory



(a) Executing a selection in a DDC.



(b) Saving data movement by offloading the selection.

Figure 4: Example of running a data-intensive relational operator, selection, in DDCs. Pushing this simple operator to the memory pool speeds up overall query execution.

pools. To understand the potential benefits, consider the 'selection' operator. Using MonetDB [4], a production in-memory database system as an example, the implementation of selection takes as input (1) a table, (2) the filter to be applied on the tuples in the table, and (3) an optional candidate list that is the result of previous selections. It performs a scan of the table and applies the filter. Every tuple that passes the filter is then materialized to a temporary table. Figure 4a depicts how this process would unfold in a DDC: assuming that the working set does not fit in compute-local cache, the selection process needs to bring all tuples in the original table from the buffer pool in the memory pool, resulting in massive data migration and thus significant execution time increase.

If, instead, we were to migrate this simple, but data-intensive compute operation to the memory pool (Figure 4b), the accesses to the original table are all *in situ*, resulting in minimal communication. Even if the computational power of the memory pool is low, most selection filters are computationally inexpensive to run. Hence, the pushdown of the operator would ensure a performance benefit.

3 DESIGN OF TELEPORT

Teleport introduces a new system call for applications to push down arbitrary functions at runtime in a memory-disaggregated architecture. This avoids expensive data movements.

The key observation behind Teleport is that the memory pool (as the backing store for the process context in a disaggregated OS) already has the majority of the data and metadata necessary for executing the user process—the compute pool is merely a cache and forwards all new memory allocations, page faults, and file I/O through the memory pool. In principle, pushdown is, thus, as simple as launching a new thread in the memory pool and reusing the existing page table. In practice, inconsistencies between the data in the compute/memory pools before and after pushdown, memory accesses by concurrent threads, and the overhead of creating process contexts all introduce significant technical challenges to realizing this goal. We now describe how to overcome these challenges.

3.1 The TELEPORT Abstraction

Using Teleport, user applications running in the compute pool in a DDC can push arbitrary functions to the memory pool. While prior work has explored, extensively, the concept of compute pushdown in various contexts (see Section 8), Teleport is unique in its ability to provide, to pushdown code, unfettered access to the process context of the original program in the memory pool, including the program stack, page table mappings, and code pages. Among other benefits, this allows pushdown code the ability to use arbitrary function pointers and leverage large, complex data structures freely.

In order to migrate execution from the compute pool to the memory pool, we introduce a new system call:

```
pushdown(fn, arg, flags)
```

With a C-library wrapper, the call takes three parameters: `fn` is a pointer to the function to be executed on the memory pool controller. `arg` is a pointer to an argument vector that is to be passed to `fn`, which can be implemented as an array of values or a structure of arbitrary type. In both cases, all pointers and contained pointers can be left in terms of the current virtual address space. Also included in the parameters to the syscall is an optional `flags` parameter that activates or deactivates features of the syscall, as appropriate.

Semantically, a pushdown function works just like a local function. The thread that calls `pushdown` blocks until the function completes, but other threads can continue their execution. When the pushed function runs in the memory pool, Teleport guarantees that all data involved is up to date, even in the presence of concurrent threads in the compute pool.

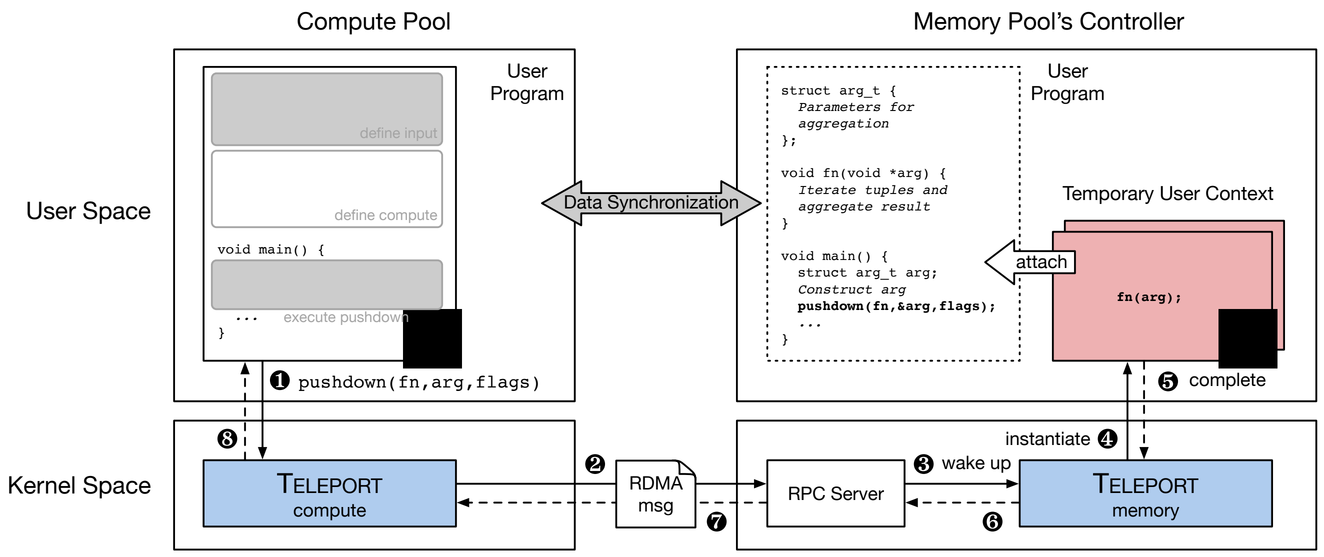
3.2 TELEPORTing the Computation

In this subsection, we describe the operation of Teleport assuming perfect synchronization of memory stores between the compute pool and memory pool. Later in Section 4, we describe how synchronization is implemented in Teleport.

Figure 5 shows the process of migrating a function. When the application calls the `pushdown` syscall (❶), the application thread stalls and both pointers (`fn` and `arg`) are passed to the compute-pool instance of Teleport in the kernel space. The instance then packs the parameters into a pushdown request and sends it to the memory pool's controller using an RDMA write operation that implements a low-latency RPC mechanism (❷).

The RPC server on the memory controller waits for incoming messages and, upon receiving one, enqueues it to the workqueue of the memory-pool instance of Teleport and wakes up the thread if it is sleeping (❸) (when its workqueue is empty, the instance sleeps to save the scarce compute resource in the memory pool). The Teleport instance dequeues a request and instantiates a temporary user context with a new kernel thread (❹).

Teleport *attaches* the temporary user context to the virtual memory space of the caller application by borrowing the page table of the caller and setting it as the table of the newly created user context. This procedure is akin to the POSIX `vfork` function in that it creates a new process but the virtual address space, file descriptor table, and other parts of the process image are not cloned—rather,



```
void main() {
  ...
}
```

```
struct arg_t {
  Parameters for
  aggregation
};

void fn(void *arg) {
  Iterate tuples and
  aggregate result
}

void main() {
  struct arg_t arg;
  Construct arg
  pushdown(fn,&arg,flags);
  ...
}
```

```
fn(arg);
```

Figure 5: Teleport architecture.

the new process shares the resources of the original. Compared to a traditional fork, this procedure is more efficient as the memory pages are neither copied nor set to read-only. Furthermore, memory modifications are supported through the techniques in Section 4 and returning from a function simply returns execution to the Teleport stub. The end result is that *the temporary context is able to access any code and data of the caller*, specifically `fn`, `arg`, and any data processed by `fn`. Inside the context, `fn` is called with `arg` as the input. Internally, the function dereferences the parameters from the argument pointer and starts the execution.

After `fn` returns, the temporary context is recycled (❺). The memory-pool instance of Teleport notifies the RPC server of the completion (❻), which then either processes the next request in its workqueue or sleeps to free compute resources. Finally, the RPC server responds to the request with the completion (❼) so that the compute instance of Teleport returns back to the application (❽), which continues execution.

Handling concurrent pushdown requests. Depending on the computation capabilities of the memory pool, multiple pushdown requests can potentially execute in parallel. Teleport implements this by maintaining a pool of instances that each polls the request queue managed by the RPC server. Note that if multiple requests arrive from the same process (two or more threads in the process called `pushdown` concurrently), these memory-side threads share the same page table and context. If the compute resource is limited in the memory pool and only one Teleport instance is allowed, then the concurrent requests are serialized in the instance's workqueue and processed one after another.

Exception and fault handling. Teleport must handle several types of exceptions and failure scenarios. Teleported functions are allowed to throw and catch C++ exceptions. The stub function that wraps the call to `fn` in the temporary user context contains an exception handler that the C++ runtime will detect during the stack unwinding phase. The handler catches the exception structure and passes it back to be rethrown by the compute pool context. General protection faults (e.g., segfaults) are also handled this way.

In Teleport, the `pushdown` function is blocking and does not time out by default. However, applications can specify a timeout. In the event of a timeout, Teleport issues a `try_cancel` request to the memory pool. If the request succeeds, the application is free to execute `fn` directly in the compute pool, re-execute the call to `pushdown`, or call some other function. Cancellation is easy if the memory pool has not yet started working on the computation, as the request can simply be removed from the workqueue. However, if the pushed function is already running, cancellation requires care. In particular, the process's memory pages need to be flushed back to the cache in the compute pool, and the instruction pointer needs to be set accordingly. In our implementation, however, the memory pool declines to cancel requests that are running, and instead forces the application to wait until they complete.

Pushdown code that is buggy and fails to complete in the memory pool within a conservative timeout is killed by Teleport to avoid indefinitely blocking other pushdown requests. The corresponding `pushdown` function in the compute pool triggers an abort signal. Finally, Teleport detects when the memory pool becomes unreachable due to a network or memory hardware failure with a background thread that runs in the compute pool and issues heartbeats. In the event of such failures, Teleport triggers a kernel panic since the main memory is lost. We leave the handling of partial resource failures that are introduced in DDCs to future work.

4 DATA SYNCHRONIZATION

A critical challenge in Teleport is keeping the cache in the compute pool and the main memory in the memory pool synchronized. There are a few points at which the two may diverge.

(1) *Before pushdown*, where the compute pool may have modifications in its local memory that have not yet been flushed to the memory pool. Changes must be synchronized to ensure that pushdown operates on fresh data.

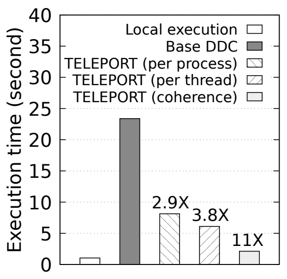


Figure 6: Application performance in different systems and with different data sync approaches in Teleport.

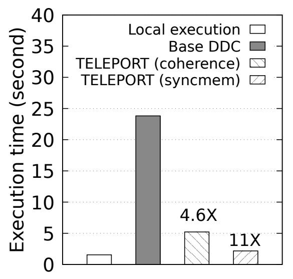


Figure 7: The benefit of a manual data sync with `syncmem` when false sharing occurs in the application.

(2) *After pushdown*, where the compute pool's cache may be stale. When execution returns to the compute pool, modified pages should be synchronized back as well.

(3) *During pushdown*, concurrent threads may continue to modify pages in the compute pool; these need to be kept coherent with the memory pool.

Without synchronization, two distinct threads T_{comp} and T_{push} running in the compute and memory pools, respectively, may access the same memory pages (because the compute pool caches pages in the main memory) without observing each other's updates (at least until a natural page fault). This can happen even if the threads utilize atomic operations, memory fences, and proper lock discipline.

We note that a naïve approach to guaranteeing consistency for all threads is to migrate the entire process and clear all memory in the compute node. While correct, this may be a substantial overkill. For multi-threaded applications, this may result in too much computation pushed to the memory pool, particularly if the threads handle unrelated requests. Even for single-threaded applications, it still requires, before pushdown, the synchronous transfer of all dirty pages from the compute node back to the memory pool and, after pushdown, the page-by-page re-fetching of every piece of data to the compute pool (as it now contains no cached pages).

Teleport instead minimizes the amount of data transmitted before, during, and after pushdown. By default, Teleport does not transfer any pages when initiating a pushdown. Instead, consistency is kept between the compute and temporary-context page tables with a write-invalidate coherence protocol inspired by MESI [37]. Applications can also instruct Teleport to use weaker memory consistency models via optional flag parameters.

To illustrate the importance of Teleport's techniques, we consider a microbenchmark involving an application with two threads: a compute-intensive thread performing arithmetic calculations (e.g., expression evaluation in a database query) and a memory-intensive thread randomly accessing a 50 GB memory space (e.g., probing a hash table). The results of the ablation study are in Figure 6. When the application runs locally in Linux, each thread finishes in 1s. In the baseline DDC, however, execution slows to 23s because of the memory-intensive thread. Pushdown using the above naïve, full-process approach can speed this up by 2.9×. Separating the two threads and only pushing the memory-intensive thread (and only evicting its memory) does slightly better with a 3.8× speedup over

```
   // Runs in the memory pool
1  Function Invalidate(pte, write):
2      if write then
3          pte.present ← False
4      else
5          pte.writable ← False
6  Function MemorySetup(tmp_context, compute_pgs):
7      t_mm ← Clone of the caller's full page table
8      foreach pte in t_mm do
9          c_pte ← compute_pgs[pte.address]
10         if not c_pte.present then
11             Continue
12         Invalidate(pte, c_pte.writable)
13     end
14     Set t_mm as the active page table of tmp_context
```

Figure 8: Preparation of the page tables before pushdown execution in the memory pool. *compute_pgs* **is the transmitted list of pages from the compute pool.**

the baseline DDC. With Teleport's default coherence, however, the synchronization overhead and the gap between local execution and DDCs are minimized, resulting in a jump up to an 11× speedup.

In this section, we describe the default protocol in detail and then expand on Teleport's memory consistency and its relaxations.

4.1 On-demand Memory Synchronization

Teleport's protocol for synchronizing data between the compute and memory pools draws inspiration from MESI cache coherence protocols—a classic approach in write-back caches. Rather than processors and cache lines, however, Teleport implements the MESI-style protocol through careful management of the page tables in both the original compute process and the temporary context such that, at any point in time, if there is a writable copy of the page between the two contexts, then it is the only such copy.

Temporary-context page table construction. When the `pushdown` function is called, the compute pool begins by building a list of memory pages that are either currently in local memory or that have an outstanding page fault request. This list of memory pages and their write permissions are sent to the memory pool as parameters to the pushdown RPC call.

When Teleport instantiates the temporary context in the memory pool, it uses the list in the procedure outlined in Figure 8. Specifically, when building the context, Teleport clones the page table of the caller thread. This cloned page table is identical to the process's page table, except that any writable page in the compute pool is excluded (Figure 8, line 3) and any read-only page in the compute pool is also set to read-only locally (Figure 8, line 5). In effect, this guarantees that the system begins with the invariant stated at the beginning of this subsection: for each page, (a) the page is writable and only in the compute pool, (b) the page is writable and only in the temporary context in the memory pool, or (c) the page is read-only and can exist in any context.

Online data synchronization. Teleport maintains the above invariant throughout pushdown execution even as the compute pool process and the temporary context execute concurrently. When either side tries to read or write to a memory page without proper permissions, a page fault is triggered to obtain the permissions.

```
// Compute-pool page faults
Function ComputeOnPageFault(address, write):
    Send request to the memory pool
Function MemoryOnPageRequest(address, write):
    mm ← process's full page table
    t_mm ← temporary context's page table
    if not mm[address].present
      or (write and not mm[address].writable) then
        Page fault to storage, copy to mm and t_mm
    pte ← t_mm[address]
    Invalidate(pte, write)
    Send *pte to the compute pool

// Memory-pool page faults
Function MemoryOnPageFault(address, write):
    mm ← process's full page table
    t_mm ← temporary context's page table
    if not mm[address].present
      or (write and not mm[address].writable) then
        Page fault to storage, copy to mm and t_mm
    else
        Send request to the compute pool
Function ComputeOnPageRequest(address, write):
    c_mm ← local page table of the caller
    pte ← c_mm[address]
    if write then
        Evict pte
    else
        pte.writable ← False
        Notify the memory pool
```

Figure 9: Handling of page faults during pushdown in order to guarantee write atomicity.

On a compute-pool page fault, the fault is forwarded immediately to the memory pool as normal; however, the corresponding page fault handler in the memory controller changes slightly during pushdown (Figure 9, lines 3–10). Specifically, after ensuring that the page is in the temporary context page table, the controller executes an operation similar to Figure 8, removing the page from the temporary context if the compute pool requested write permissions, or setting it to read-only if it requested only read permissions.

Temporary-context page faults are handled similarly, except that we must distinguish between a 'true' page fault, which should be forwarded to the storage pool and a pushdown-related page fault, which invalidates the cached pages in the compute pool. TELEPORT distinguishes this by checking the full page table and the temporary context's page table, both stored locally in the memory pool. Evictions from the memory pool to the storage preserve the correct page table entry (*pte*) dirty bits.

When pushdown completes, the dirty bits of the temporary context's page table should be merged back into the full page table but no external communication is necessary.

Concurrent page faults. One key difference between TELEPORT's protocol and that of traditional MESI implementations is the lack of either a common directory or a bus between the members of the system, removing those components as serialization points for permission requests. TELEPORT addresses concurrent page faults by taking advantage of the fact that for a two-side protocol (compute and memory), each side can deduce the current state of the system locally, so a global coordinator is unnecessary.

Consider the possible states of the system. For every page, each side can have one of three permissions: $\varnothing$ for an absent page, R for read-only, and W for writable. Let system state be denoted by the tuple of pool permissions: $(\text{compute}, \text{memory})$. Note that we can disregard any state with W in either position as there will never be concurrent faults as long as RPC messages are received and handled in FIFO order (enforced using reliable RDMA connections). We can further disregard any state with $\varnothing$. In $(\varnothing, \varnothing)$ or $(\varnothing, R)$, the memory pool does not need to contact the compute pool—both are true page faults and any request from the compute pool will be handled after the page fault is complete. $(R, \varnothing)$ does not exist in our protocol.

The only state in which concurrent faults are possible is, therefore, (R, R) where both the compute and memory pools try to acquire exclusive write access. In this situation, both sides will note that there is an outstanding request and break the tie by favoring the memory pool. Specifically, the memory pool, upon receiving a new page fault request before a response to its own request arrives, will simply ignore the request. The compute pool, on the other hand, will satisfy the memory pool's request, wait for a predetermined amount of time t, and then reissue the request. We favor the memory pool in order to complete the pushdown execution as soon as possible, and we wait for t time to allow some amount of progress on the memory pool before taking write access back. Note that in the case of thrashing when the compute and memory pools contend on memory pages, additional backoff mechanisms would ensure progress. However, applications should avoid data contention between the two pools for pushdown performance.

Correctness. The correctness of our protocol follows directly from our adherence to the Single-Writer-Multiple-Reader (SWMR) invariant [34]. Just like MESI, writes in TELEPORT are serialized as there is ever only a single writer in the system, and writes are propagated when the other node explicitly invalidates the writer's exclusive 'lock.' Cache coherence makes our system transparently compatible with existing architectures and their memory consistency models.

4.2 Alternative Coherence Mechanisms

In addition to the above cache coherence protocol, TELEPORT provides support for certain user-applied optimizations. An important optimization is an additional `syncmem` syscall that manually and preemptively flushes dirty pages from the compute pool. This mechanism can be triggered before or during pushdown and is useful if the user already knows which pages will be accessed by `fn`.

TELEPORT also provides options (specified via `flags`) for coherence protocols that support weaker memory consistency models. These improve performance, but should be used carefully by programmers to ensure correctness. One simple relaxation is to disable the coherence entirely. This might be useful, for example, if the user wants to manually synchronize pages. An example use is to handle false sharing, which occurs when threads in the compute and memory pool access data (either variables on the stack or allocated memory on the heap) that are not shared but that reside on the same page. Although false sharing is uncommon, Figure 7 shows that when it occurs, it can negatively affect the pushdown performance. In this case, users can disable the coherence protocol and manually synchronize the data with `syncmem` at a finer granularity.

Another relaxation follows the default coherence mechanism. Rather than removing pages when the other pool requests write permissions, TELEPORT sets them as read-only. Effectively, this maintains write serialization for individual memory locations, but relaxes the guarantees of write propagation. Combined with typical processor memory consistency models, this relaxation amounts to an implementation of Partial Store Ordering (PSO) [26]. Again, for applications that can take advantage of this relaxed consistency model (e.g., by converting important reads to RMW instructions or memory fences to explicit synchronization of modified page lists), it may provide better performance. Section 7.6 provides a more detailed evaluation of the coherence protocol and benefits of the relaxations when the data contention rate is high.

5 APPLYING TELEPORT

In this section, we present three case studies to demonstrate the benefits of Teleport for data processing: an in-memory database, a graph processing system, and MapReduce. For each use case, we will describe how we identify functionalities to be pushed down to memory. We focus here on identifying the general rules of thumb to determine the pushdown functionalities. We find that these heuristics work well in practice, although cost-based approaches can automate the decision-making; we leave this to future work.

5.1 In-memory Database

To evaluate database workloads with Teleport, we select MonetDB [4], a columnar in-memory DBMS that provides high performance processing for analytical queries.

Filtering/summarization operators. Several commonly used operators such as *projection*, *aggregation*, and *selection* require simple computation but process a large number of tuples. Further, the result set is typically much smaller than the input (projected column in *projection*, matching tuples in *selection*, and sub-aggregates in *aggregation*). Hence, users should push these operators down, particularly when they are highly selective; similar observations were made in the context of disaggregated storage [33, 53]. In DDCs, however, they can be pushed to the memory pool so that the compute pool only receives the summaries for further processing.

For example, consider the following database query, Q_{filter}, which consists of a *selection*, an *aggregation*, and *projections*:

```
SELECT SUM(quantity) FROM Lineitem WHERE shipdate < $DATE
```

By pushing down the predicate `shipdate < $DATE` as well as the projections of `shipdate` and `quantity` attributes, we avoid transferring the entire `Lineitem` table. Note that it is still required to transfer the matching tuples to the compute node for the `SUM` aggregation. Thus, in the extreme, one could imagine Teleporting all operators of this query. Offloading all operators to the memory pool would provide additional bandwidth savings, but at the potential cost of pushdown overhead. In general, the final decision should depend on the amount of data to be synchronized, the selectivity, and the computational complexity of the operators.

Complex queries. A more complex case is Query 9, the most expensive query in the TPC-H benchmark in Figure 3. Figure 10 breaks down its execution time in disaggregated execution (compute-local memory is configured to be 1 GB) with a scale factor of 50 into its constituent operator types. We observe that in addition to the memory-intensive projection operator, hash join also incurs significant remote memory accesses and bottlenecks the overall performance. While hash join tends to have relatively high computational requirements when run on a traditional OS, in a DDC, it becomes severely memory-bound due to random accesses to the hash index. As such, it is a strong candidate for pushdown, as the results in Section 7 will later verify. Other operations such as merge join and expressions also experienced degradation in disaggregation, but they are not blockers to end-to-end query performance.

Code modification. Finally, an important criterion for pushdown is the complexity of application changes. Figure 11 summarizes the amount of changes required to support each operator pushdown in MonetDB, as well as the size of the specific pushdown function. We observe that modifications across all operators are negligible relative to MonetDB's code base (~400K LoC), and the amount of code executed in the memory pool is restricted under 100 lines.

Automatic query optimization. Automating the porting process is achievable via static analysis and code transformation, given the structured nature of relational operators. An interesting and challenging task is to automatically decide which operators should be pushed down at runtime. There are general trade-offs in applying compute pushdown in DDCs: offloading an operator close to data can reduce the cost of data movement between pools, but it can also incur pushdown overhead, including shipping the operator, potential data synchronization, and degraded computation power. Section 7.4 evaluates the impact of these trade-offs and a potential metric for determining the viability of an operator for pushdown. We note, however, that the optimal plan of pushdown is determined by various factors: operator characteristics, workloads, and the DDC configuration. A potential solution is a DDC-aware query optimizer that captures the resource constraints in different resource pools and finds the optimal plan for operator placement. This paper focuses on the Teleport mechanism and leaves a full investigation of DDC query optimizer design to future work.

5.2 Graph Processing

To showcase another challenging data-intensive workload, we look at PowerGraph [2], a high-performance in-memory graph processing system. Similar to prior DDC settings [43, 51], we run PowerGraph in the compute pool and utilize multiple threads as compute workers. The main graph state is in the memory pool.

To execute a graph query, PowerGraph first loads the input graph to the main memory, runs a *finalize* phase to partition and shuffle the graph among multiple workers, and then iteratively executes *gather*, *apply*, and *scatter* in sequence until the graph algorithm terminates. We observe that the *finalize*, *gather*, and *scatter* phases are data-intensive because the vertex and edge states in the working set are frequently (and potentially randomly) accessed. Therefore these three phases are often a bottleneck in our setting. Using single-source shortest path (SSSP) as an example, the scatter phase combines and sends the messages, which contain the distances to the source, to vertices in their adjacency list for the next round of execution. This scatter process is expensive when the working set is larger than the local cache of the compute pool.

Figure 10 shows the time breakdown of this execution on a real-world social network graph [52]. *finalize* and *scatter* account for

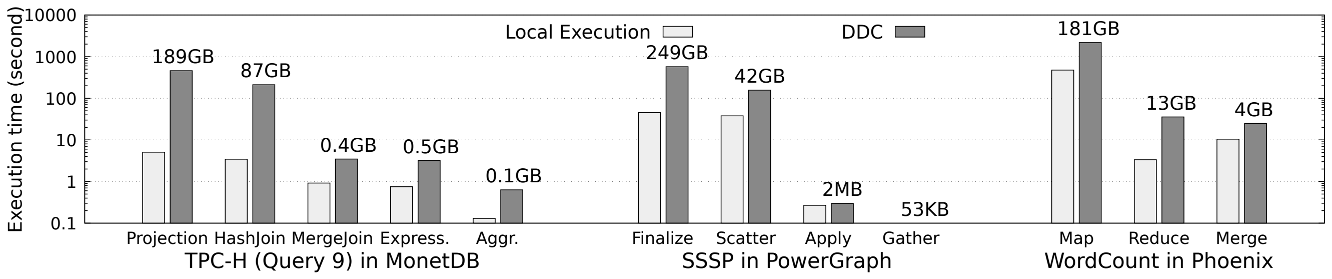


Figure 10: Performance breakdown of the query with the greatest cost of scaling in DDCs in each system. For every operator/phase, we show the times in both local and DDC executions and the remote memory accesses in the DDC. A common pattern is that there are one or two arbitrary operators/components dominating the overall query execution time.

System	Operator	Functionality	Code Change	Pushed Code
MonetDB (400K LoC)	*Projection*	Get a subset of columns from a list of records.	117	51
	Aggregation	Apply an aggregate function over tuples.	214	60
	Selection	Select tuples with filters from the input table to a temporary table.	302	58
	HashJoin	Scan the outer table, probe hash index, and generate join results.	75	42
PowerGraph (150K LoC)	*Finalize*	Partition and shuffle input graph among the worker threads.	77	52
	Scatter	Exchange and combine messages between vertices.	82	39
	Gather	Aggregate messages and apply a user-defined function.	82	39
Phoenix (2K LoC)	*MapShuffle*	Shuffle map results (key-values) to the buffers of reduce tasks.	173	28

Figure 11: The flexibility of Teleport enables the pushdown of various memory-intensive operators in existing data processing systems with minimal modification.

most of the overhead, although the *gather* phase can also bottleneck other applications, e.g., PageRank. All three components can be Teleported with fewer than 100 lines of code each (see Table 11).

5.3 MapReduce

Our third use case is Phoenix [28], a native, shared-memory MapReduce system. In Phoenix, there are *map*, *reduce*, and *merge* phases. The *map* phase performs the actual map computation, generates key-value records, and shuffles the records to the reduce workers. We observe that the *map* phase is normally the bottleneck in a DDC because of the shuffle operation, a data-intensive sub-component.

Revisiting Figure 10 (the last group), we can examine the performance breakdown of `WordCount` in Phoenix. In this figure, as a point of comparison, we include *reduce* and *merge* execution times as well. We observe that the *map* phase experienced much greater remote memory accesses compared to other phases. The *map* phase, however, is computationally expensive as a whole. To push down only data-intensive operations at a finer granularity, we further divide the *map* phase into *map-compute*, which applies the user-defined map function and generates key-value records, and *map-shuffle*, which shuffles the records among reduce tasks, sub-phases. The *map-shuffle* dominates the running time in DDC execution—95% of *map* time. This suggests moving the *map-shuffle* phase close to the data, which we achieve with minimal code changes (see Figure 11); the pushdown function requires only 28 lines of code.

6 IMPLEMENTATION

We have implemented Teleport[1] on top of LegoOS [3]. Similar to LegoOS, Teleport uses the Mellanox mlx4 InfiniBand driver for fast network accesses and assumes the x86-64 architecture. It consists of 6,500 lines of C code, split across the kernels for the compute and memory pools, focusing on memory disaggregation.

Our implementation utilizes the RDMA RPC messaging framework that is built atop LITE [48], a two-sided RDMA kernel module implemented by the one-sided *write* verb. We pre-allocate and register physical memory to the network card as RPC buffers, which are kept separate from the LegoOS buffers to provide a degree of isolation. We describe the details of each kernel as follows.

Teleport compute kernel supports the `pushdown` system call, sends the request to the memory pool, and handles synchronization. As part of the latter, we needed to add functionality to enable the compute kernel to serve incoming page faults, invalidating the page and flushing the TLB as necessary.

To reduce network cost, our implementation adds an additional optimization to the protocol of Section 4. Specifically, it compresses the list of resident pages sent at the beginning of pushdown using *run-length encoding*, which provides 20× reductions in the message size, making it feasible to pack the list of pages and their permissions along with necessary metadata into a single RDMA message.

Teleport memory kernel handles incoming RPC requests and cache coherence. It runs a number of parallel RPC handlers to perform these tasks—each on its own a kernel thread. This number is configurable to reflect the compute power limitation in the memory pool. Upon receiving a pushdown request, the server enqueues it into the workqueue of the memory-pool instance of Teleport and eventually processes it in the manner described in Sections 3 and 4.

7 EVALUATION

We conduct a comprehensive set of experiments to evaluate the benefits of Teleport based on the use cases presented in Section 5, the trade-offs in compute pushdown, and the efficiency of Teleport designs. We compare Teleport with two baselines: (1) a disaggregated baseline, LegoOS, which incurs cost of scaling due to remote

[1]Teleport source code is available at https://github.com/eniac/TELEPORT.

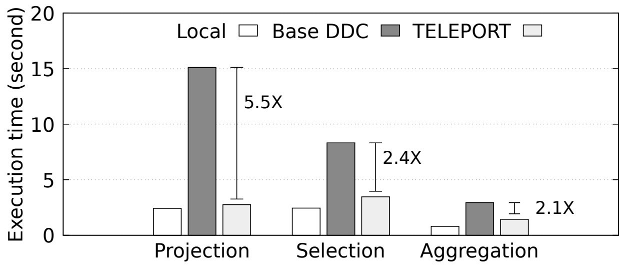


Figure 12: The performance improvement of pushing the operators in Q_{filter} to the memory pool with Teleport.

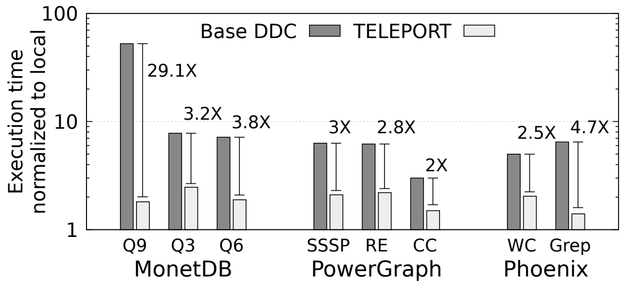


Figure 13: The performance improvement of applying Teleport in a wide range of data-intensive applications. By removing expensive data movement, Teleport significantly reduces the overhead of memory disaggregation, up to an order of magnitude speedup compared to the baseline DDC.

memory accesses, and (2) a single-machine baseline, Linux, where the cost is either low when local memory is sufficient for the workload or high when local memory is constrained and data is spilled to secondary storage, e.g., SSDs. In all experiments, the applications, datasets, and the number of CPUs used by the applications are consistent across all platforms to ensure a fair comparison.

Experimental setup. The baremetal machines in our testbed have Intel Xeon E5-2630L CPUs and 64 GB DDR4 RAM, and run either Linux, LegoOS, or Teleport. The emulated DDC cluster consists of all three types of pools: compute, memory, and storage. Machines are connected with an InfiniBand network of NVIDIA Mellanox Connect-X3 NICs and an EDR switch, with 56 Gbps throughput and 1.2μs latency. The compute pool consists of a single physical machine and has access to 1 GB of local DDR4 memory; the memory pool consolidates 128 GB DDR4 memory with a single controller, and the storage pool has a 1 TB NVMe SSD. We chose 1 GB of compute-local memory per application since many of the benefits of DDCs come from high density configurations where many CPUs reside in the same pool, resulting in a modest amount of local memory per CPU.

7.1 The Effectiveness of Teleport

We first quantify the performance improvement achieved by operator pushdown for data-intensive systems over baseline DDC. Hence, this section first focuses on the *cost of disaggregation*. We use a default setup: the CPU cores in the compute and memory pools have the same clock speed, but numbers of cores are different—the memory pool is limited to a single thread; concurrent pushdown requests are serialized. Beyond the cost of disaggregation, Section 7.2 showcases the elasticity of DDCs. Section 7.3 further investigates the impact of the degree of disaggregation by varying CPU clock speed and the number of threads in the memory pool.

Database microbenchmark. Our first experiment involves the synthetic Q_{filter} query presented in Section 5.1. Recall that this query involves a selection operator followed by projections and an aggregation. During setup, we supply as input a `Lineitem` table with 300 million tuples. Figure 12 summarizes our main findings for these operators, where the Y-axis shows the query execution times in Linux (local execution), LegoOS (baseline DDC), and Teleport. We make the following observations. First, compared with the local execution, baseline DDC adds significant overhead, experiencing 3–6× slowdowns, primarily due to paging data from remote memory. With Teleport, the slowdowns are drastically reduced to less than 2×. In fact, Teleport is faster than LegoOS by 2.1–5.5×. The improvements are most visible for projection, which would otherwise have to ship many tuples from the remote memory pool just to identify attributes of interest and apply filters.

Database TPC-H benchmark. Figure 13 (left figure) compares the performance of MonetDB in Linux, LegoOS, and Teleport with the three TPC-H queries (scale factor 50) with the longest execution times, namely Query 9 (Q_9), Query 3 (Q_3), and Query 6 (Q_6). These queries, as described in Section 5, consist of a mix of relational operators involving selections, projections, aggregations, hash and merge joins, and expression calculation.

We make the following observations. In all three queries, the significant slowdowns, higher than 50× in the worst case, render the baseline DDC prohibitively costly for database query processing in scaling out the hardware resources. Using Teleport, we pushed down a subset of the most bandwidth-intensive operators that bottleneck the DDC performance to the memory pool. The speedup improvements over LegoOS range from 3–29×. Teleport, with a compute-local memory that is ~2% of the database size (1 GB versus 50 GB), is only slightly slower than local execution. The cost of scaling for DBMSs in DDCs with Teleport is comparable to the cost in distributed DBMSs as we see in Figure 1b.

Graph processing. Our next experiment is on graph processing. Figure 13 (center figure) summarizes the results obtained on PowerGraph for three graph queries: SSSP (single-source shortest path), RE (single-source reachability), and CC (connected components). We use as input a real-world social-network graph [52]. Our results show that the cost of scaling in baseline DDC is 5×. In comparison, Teleport closes the gap between DDC and local execution quite noticeably, achieving 2–3× speedup over LegoOS. The primary benefits obtained are in pushing down the scatter-gather and finalize stages, as described in Section 5.2.

MapReduce. Our final use case is MapReduce using the WC (WordCount) and Grep applications on a real-world NLP dataset consisting of 15 million Reddit comments[2]. Our observations in Figure 13 (right figure) are consistent with the other two systems. Teleport achieves 2.5× and 4.7× performance improvements over LegoOS, significantly narrowing the gap from local execution in Linux.

[2]https://www.kaggle.com/reddit/reddit-comments-may-2015

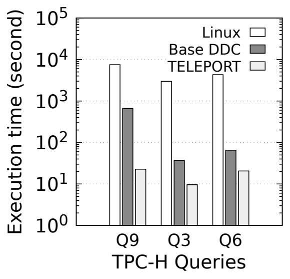


Figure 14: Query speedups by using disaggregated and large memory pools compared to NVMe SSDs.

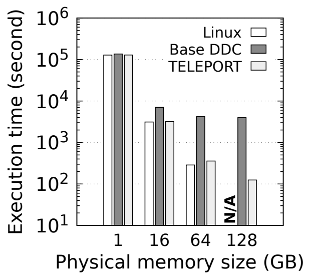


Figure 15: The performance benefits of increasing physical memory for large workloads in different systems.

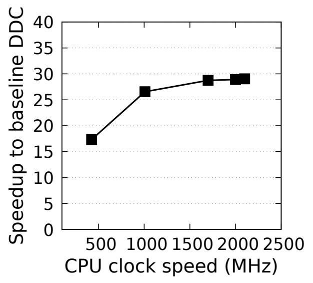


Figure 16: Pushdown performance (Q_9) with different computation power settings in the memory pool.

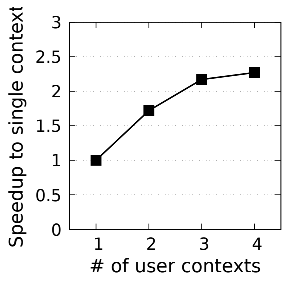


Figure 17: The benefits of parallelizing the processing of concurrent pushdown requests in the memory pool.

The takeaway is that TELEPORT results in up to an order of magnitude performance improvement over the baseline disaggregated OS and minimizes the gap between disaggregated and traditional environments. We note that the goal is not to surpass a local execution where resources are all centralized in a single place, but rather to narrow the performance gap to achieve a low cost of scaling while reaping the benefits of DDCs [21, 43, 55].

7.2 The Benefits of Memory Disaggregation

We next compare the performance of data-intensive applications in both monolithic and DDC deployments with varying levels of memory. We first fix the amount of local memory available to all systems to 1 GB to emulate the effects that occur when processing large-scale workloads—namely, the effects of being able to access a large remote memory pool in DDCs instead of needing to spill data to disks in Linux. To ensure efficient disk I/O, we use an NVMe SSD that supports 3 GB/s (sequential) and 600K IOPS (random) I/O.

Figure 14 shows the results of processing the three most expensive queries (scale factor 50) in TPC-H in MonetDB. Unsurprisingly, when local memory is insufficient, LegoOS is 10×, 65×, and 80× faster than Linux with SSDs for Q_9, Q_3, and Q_6, respectively. However, with TELEPORT, this benefit increases to *two orders of magnitude*: 330×, 210×, and 310×, respectively. In sum, TELEPORT, by offloading a small set of operations, enables memory-intensive workloads to more efficiently take advantage of the large memory pools envisioned by proposals for memory disaggregation.

We also evaluated the effect of TELEPORT when varying the amount of memory available in the memory pool (local memory is kept at 1 GB). For this experiment, we used Q_9 and increased the workload size to scale factor 200 (a 200 GB database). In addition to the baseline DDC, we again show a monolithic Linux configuration for comparison—all versions are provided a consistent amount of memory before they need to spill to disks until 128 GB, which exceeds the memory capacity of the Linux server.

Figure 15 shows that with 1 GB total memory [3], all platforms perform poorly. In principle, provisioning more total memory will spill less data to disk; however, at 64 GB, the disaggregation cost in LegoOS begins to dominate the execution time, which is significantly longer than the time in Linux. TELEPORT instead effectively minimizes this cost and achieves a similar performance to Linux until 128 GB where Linux cannot match the amount of resources. TELEPORT provides 2.3× higher performance than the best Linux execution, and 31.7× higher performance than LegoOS with the same memory size. These benefits will grow with larger workloads.

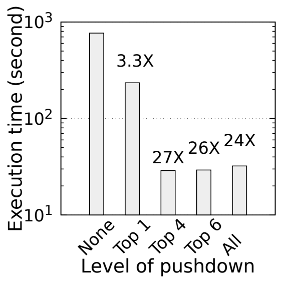
7.3 Varying the Degree of Disaggregation

Our next set of experiments performs a sensitivity analysis on the degree of disaggregation along two dimensions: (1) where the memory pool has lower CPU clock speed compared to CPUs in the compute pool, and (2) degree of parallelism—the number of threads—in the memory pool. We emulate these effects by throttling CPU clock rate in the memory pool and varying the number of threads that are used to process parallel pushdown requests. Our results also help future DDC and hardware designers understand sweetspots in cost-performance ratios when determining the relative hardware costs for the disaggregated compute and memory pools.

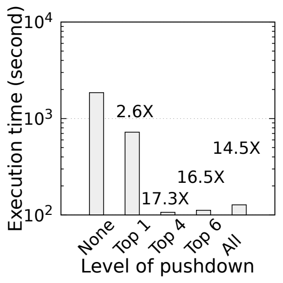
Figure 16 shows the effect that computation power in the memory pool has on pushdown execution time. We control the CPU clock speed with throttling for Q_9 in the MonetDB TPC-H benchmark (scale factor 50). We observe that as the CPU capabilities in the memory pool increases from 20% (0.4 GHz) to 100% (2.1 GHz) of the compute pool, query speedup of Q_9 relative to the baseline DDC increases as expected. Our results suggest that even at very modest CPU speeds (0.4 GHz), emulating a memory pool with very limited compute resource and thus a high degree of disaggregation, TELEPORT is still able to achieve a 17 × speedup over the baseline. Moreover, at clock speeds above 1.7 GHz, the speedups level off at 29×, suggesting there is no need to match the fastest CPU speed to reap the performance benefits of TELEPORT.

Figure 17 shows the effect of memory-pool parallelism on the performance of processing concurrent pushdown calls. We evaluate a parallel aggregation query on TPC-H `Lineitem` table. We maintain the same CPU speed (2.1 GHz) in both the compute and memory pools. The application uses eight threads in the compute pool, one on each physical core. The memory pool uses two physical cores for the user contexts, emulating a scenario where the disaggregated memory pool does not have significant compute resource dedicated for pushdown. The Y-axis in the figure shows the speedup over a single user context, as we vary the number of parallel user contexts in the memory pool on the X-axis. We find that as the parallelism

[3]Total memory allocated in the application. The 1 GB compute-local cache in the DDC is not allocatable by applications.



(a) 50% lower CPU clock rate.



(b) 75% lower CPU clock rate.

Figure 18: The performance of different levels of pushdown.

increases, it takes less time to process the eight concurrent requests as expected. However, we see diminishing returns in speedup, primarily due to context switching overheads when scheduling more threads than there are physical cores.

7.4 Varying the Level of Pushdown

Recall from Section 5.1 that there are trade-offs in compute pushdown. We now evaluate the impact of these trade-offs by using a metric we call *memory intensity*. To compute memory intensity, we first execute a profiling run in the baseline DDC; memory intensity is then the total remote memory accesses divided by the execution time (i.e., remote memory accesses per second, RM/s). We compute memory intensity for Q_9 in MonetDB and order its eight operators by this metric. For reference, *Projection* has the highest intensity (110K RM/s) and *Group* has the lowest (45K RM/s).

Figure 18a shows the performance of different levels of pushdown. We constrain the computation power in the memory pool to be 50% of that in the compute pool. Compared to no pushdown, offloading the most expensive operator to the memory pool brings 3× performance speedup. The speedup increases to 27× when we apply TELEPORT to the top four operators. *Being too aggressive, however, backfires*: the speedup decreases to 26× and 24× when we push down the top six and all operators, respectively. This is because, for these operators, the benefit of saving network communications does not compensate the overhead of pushdown and a lower CPU clock rate. These effects are magnified when the computation power in the memory pool is more constrained (Figure 18b).

We found that 80K RM/s is a good split for pushdown decisions in our DDC testbed. However, the optimal level of compute pushdown is determined by the operators, the workload, and the DDC configuration. Applying TELEPORT automatically while accounting for these parameters is a promising future direction.

7.5 TELEPORT Execution Breakdown

We next quantify the benefits of on-demand synchronization methods, and provide a comprehensive look at the costs associated with them. Figure 19 presents a factor analysis on the execution time in TELEPORT for processing a pushdown call. This time consists of six parts: (1) pre-pushdown synchronization, (2) pushdown request transfer from the compute pool to the memory pool in the RDMA network, (3) user context setup, (4) pushdown function execution and synchronization during the execution, (5) pushdown response transfer from the memory pool to the compute pool via RDMA, and

#	Component	Determined by
1	Pre-pushdown sync time	Synchronization method, cache size
2	Request transfer time	Message size, the network
3	Context setup time	Synchronization method, cache size
4	Function execution	User function
	Online sync time	Synchronization method, cache size
5	Response transfer time	Message size, the network
6	Post-pushdown sync time	Synchronization method, cache size

Figure 19: The components in executing a pushdown request. Grayed are factors on what TELEPORT has no control.

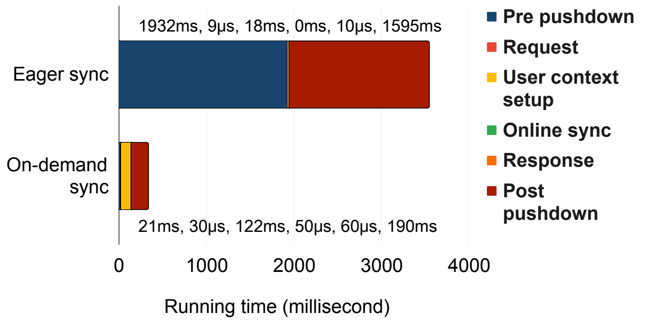


Figure 20: The breakdown of TELEPORT performance in different data sync methods with 1 GB compute-local cache.

(6) post-pushdown synchronization. While all parts have factors that are not controlled by TELEPORT, specifically the cache size in the compute pool, the network, and the user function, the data synchronization method in use is important for every part. Message sizes for (2) and (4) also vary across different methods.

Figure 20 summarizes our cost breakdown results for a 1 GB local memory in the compute pool (user function time was excluded so that the result can be generalized). In the figure, *eager* memory synchronization is a strawman that synchronizes all pages at the beginning and end of pushdown function execution. The on-demand memory synchronization is our default technique presented in Section 4.1. We make the following observations. In both techniques, pre/post pushdown and user context setup are the dominant costs, though at varying degrees. Overall, TELEPORT's on-demand synchronization is significantly faster than eager synchronization (0.3s vs. 3.5s for one pushdown call), since data is only fetched on demand as required by the user function. Although synchronizing data on-demand requires extra time in setting up the user context (yellow region in figure) because of page table entry checking described in Section 4.1, its substantial savings in parts (1) (blue) and (6) (red) reduce overall execution time by an order of magnitude. Our results suggest that a careful data-synchronization approach does impact performance significantly, compared to sunk costs that are tied to the underlying network speed.

7.6 Coherence Protocol Efficiency

Finally, we evaluate the efficiency of TELEPORT's coherence protocol. We do this by extending the microbenchmark in Section 4. We add shared memory between the compute-intensive thread and the memory-intensive thread, and vary the contention (where both

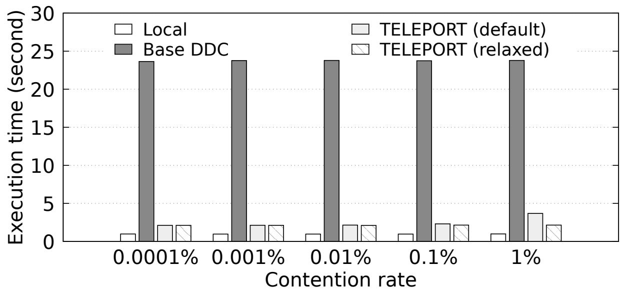


Figure 21: Application performance in different systems with varying levels of contention between threads.

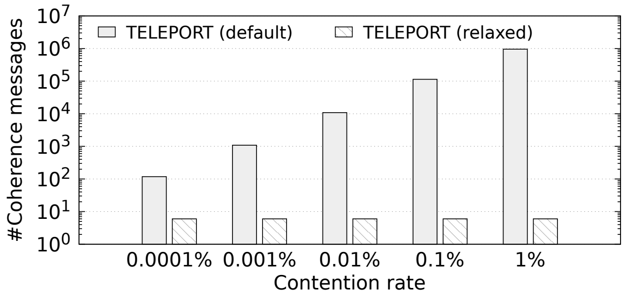


Figure 22: The number of coherence messages in Teleport.

threads request write permissions) rate from low (0.0001%; one in a million operations) to high (1%; one in a hundred operations).

Figure 21 shows the application performance in different systems when the contention between the threads increases. As the contentions in both local execution and base DDC are local to the threads (within the same NUMA node), increasing the contention rate barely affects the performance. In Teleport with the default coherence protocol, the contention leads to network communication. At low contention, the application completes in 2.1s. There are observable performance changes when the contention rate reaches 0.1% (2.3s) and 1% (3.7s). Figure 22 shows the number of network messages incurred by the protocol. The average messaging latency in our coherence protocol (1.6μs) is close to the raw network latency (1.2μs). In contentions, favoring the memory thread in tiebreaking completes the pushdown faster: 15% improvement at 1% contention rate. Adding more threads increases the contention correspondingly. For example, when we fix the contention rate at 0.1% per thread, increasing the number of compute-intensive threads to four brings the execution time up to 2.9s.

A Weak Ordering [8] relaxation avoids contention between writers. Figures 21 and 22 show that the performance and the number of coherence messages no longer change with the contention rate when the application adopts the relaxation. Other relaxations work similarly for other types of contentions, e.g., the PSO relaxation (Section 4.2) for contentions between writers and readers.

In summary, the default coherence protocol of Teleport achieves low latency when exchanging messages between the compute and memory pools. Hence, it can tolerate a moderate amount of data contention without observable performance degradation. Applications can also leverage the relaxations that Teleport supports for weaker memory models to avoid/manage contention directly.

8 RELATED WORK

Teleport is related to the classic idea of pushing computation closer to the data [19, 20, 22, 30, 36, 38, 46]. Pushing down selection predicates is also a well-studied technique in databases, including distributed databases [13, 16, 42] and sensor networks [29]. Teleport's instantiation of these prior ideas is unique owing to the memory disaggregation setting. Teleport has access to a process's entire memory address space, can compute arbitrary functions, can modify the memory at will, and can dereference pointers.

Today's cloud DBMSs already leverage storage disaggregation [7, 11, 50], which decouples processing and data so they can scale independently; however, workers in these systems are still constrained by their local memory—a limitation that memory disaggregation addresses. Operator pushdown has been applied to these storage-disaggregated environments [27, 33, 39, 53]. Unfortunately, they are typically limited by the operations supported by the storage service and, thus, relegated to simple tasks like scanning tuples. Combining Teleport and storage pushdown may yield further improvements in a fully disaggregated environment.

Within the DDC and remote memory context, Semeru [51] pushes down part of the JVM garbage collector to remote memory, while the work of Aguilera et al.[9] pushes down pointer chasing, and StRoM [45] pushes down checksum computations to remote SmartNICs. Finally, CompuCache [54] supports compute offloading for remote memory caching with spot VMs. Teleport is distinct in its generality—it can be used to push down arbitrary functions. This is possible because of disaggregated memory and OSes.

Improving database systems in DDCs is a timely topic [14, 55–57]. Recent studies [55, 56] investigate the performance overhead of DDCs for DBMSs and sketch the opportunities for optimizations. Redesigned DBMSs [14, 57] can significantly lower the overhead. DDC architectures are continuously evolving. Keeping up with the hardware by redesigning DBMSs requires expensive investment. Teleport provides a simpler and more portable alternative for DBMSs to harvest many benefits of DDCs. Teleport can also be applied to other data-intensive systems for the same advantage.

9 CONCLUSION

Disaggregated data centers (DDCs) are an emerging trend for organizing hardware resources, and disaggregated OSes have risen to help manage them. Unfortunately, data-intensive systems suffer from high performance overhead in DDCs. We propose Teleport, a framework that can flexibly transport a piece of computation to the memory pool for saving expensive data movement and thus improving overall execution. Our design challenges center around ensuring consistent views on the memory space, synchronization, and temporary context creation in a pushdown call. By applying Teleport to three popular data-intensive systems, we showcase the significant performance benefit of Teleport for DDCs.

ACKNOWLEDGMENTS

We thank the SIGMOD reviewers for their thoughtful feedback. We also thank Yizhou Shan for helping us set up LegoOS. This work was supported in part by Google, Facebook, VMWare, NSF grants CNS-2107147, CNS-2124184, CNS-2045861, CNS-2106388, CNS-2104882, and DARPA Contract No. HR001117C0047.

REFERENCES

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

[2] GraphLab PowerGraph. https://github.com/jegonzal/PowerGraph.

[3] LegoOS. https://github.com/WukLab/LegoOS.

[4] MonetDB. https://www.monetdb.org/.

[5] Connectx-6 single/dual-port adapter supporting 200Gb/s with VPI. https://www.mellanox.com/products/infiniband-adapters/connectx-6, 2020.

[6] Postgresql: The world's most advanced open source relational database. https://www.postgresql.org/, 2020.

[7] Bigquery: Cloud data warehouse. https://cloud.google.com/bigquery, 2021.

[8] Sarita V. Adve and Mark D. Hill. Weak ordering - A new definition. In Jean-Loup Baer, Larry Snyder, and James R. Goodman, editors, *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, June 1990*, pages 2–14. ACM, 1990.

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

[10] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2020.

[11] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. Socrates: The new SQL server in the cloud. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1743–1756. ACM, 2019.

[12] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in spark. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394. ACM, 2015.

[13] David F. Bacon, Nathan Bales, Nicolas Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. Spanner: Becoming a SQL system. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 331–343. ACM, 2017.

[14] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. Polardb serverless: A cloud native database for disaggregated data centers. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2477–2489. ACM, 2021.

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

[16] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. The memsql query optimizer: A modern optimizer for real-time analytics in a distributed database. *Proc. VLDB Endow.*, 9(13):1401–1412, 2016.

[17] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, 2015.

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

[19] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. Woo Kang, I. Kim, and G. Daglikoca. The architecture of the DIVA processing-in-memory chip. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2002.

[20] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. Mckenzie. Computational RAM: implementing processors in memory. *IEEE Design & Test of Computers*, 16(1), 1999.

[21] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[22] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: The Terasys massively parallel PIM array. *IEEE Computer*, 28(4), 1995.

[23] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 17–30. USENIX Association, 2012.

[24] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[25] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with INFINISWAP. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[26] SPARC International Inc and David L Weaver. *The SPARC architecture manual*. Prentice-Hall, 1994.

[27] Xiaowei Jiang, Yuejun Hu, Yu Xiang, Guangran Jiang, Xiaojun Jin, Chen Xia, Weihua Jiang, Jun Yu, Haitao Wang, Yuan Jiang, Jihong Ma, Li Su, and Kai Zeng. Alibaba hologres: A cloud-native service for hybrid serving/analytical processing. *Proc. VLDB Endow.*, 13(12):3272–3284, 2020.

[28] Christos Kozyrakis. Phoenix. https://github.com/kozyraki/phoenix.

[29] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[30] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2000.

[31] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD Conference*, 2010.

[32] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, July 2020.

[33] Ingo Müller, Renato Marroquin, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the ACM SIGMOD Conference*, 2020.

[34] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. A primer on memory consistency and cache coherence, second edition. *Synthesis Lectures on Computer Architecture*, 15(1):1–294, 2020.

[35] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, July 2015.

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

[37] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1984.

[38] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2), 1997.

[39] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the ACM SIGMOD Conference*, 2020.

[40] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, USA, 2010.

[41] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: high-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 315–332. USENIX Association, 2020.

[42] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shivakumar Venkataraman. F1 query: Declarative querying at scale. *Proc. VLDB Endow.*, 11(12):1835–1848, 2018.

[43] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[44] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A Network Architecture for Disaggregated Racks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[45] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: Smart remote memory. In *Proceedings of the ACM European Conference on*

Computer Systems (EuroSys), 2020.

[46] H. S. Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, C-19(1), 1970.

[47] Michael Stonebraker and Akhil Kumar. Operating system support for data management. *IEEE Database Eng. Bull.*, 9(3):43–50, 1986.

[48] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[49] Amin Vahdat. Coming of age in the fifth epoch of distributed computing: The power of sustained exponential growth. SIGCOMM 2020 Keynote, 2020.

[50] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1041–1052. ACM, 2017.

[51] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[52] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the IEEE International Conference on Data Mining series (ICDM)*, 2012.

[53] Xiangyao Yu, Matt Youill, Matthew E. Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. Pushdowndb: Accelerating a DBMS using S3 computation. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2020.

[54] Qizhen Zhang, Philip Bernstein, Daniel Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. Compucache: Remote computable caching using spot vms. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022, Online Proceedings*. www.cidrdb.org, 2022.

[55] Qizhen Zhang, Yifan Cai, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Rethinking data management systems for disaggregated data centers. In *Proceedings of Conference on Innovative Data Systems Research (CIDR)*, January 2020.

[56] 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. *Proceedings of the VLDB Endowment*, 13(9):1568–1581, May 2020.

[57] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proc. VLDB Endow.*, 14(10):1900–1912, 2021.