Data Flow Architectures for Data Processing on Modern Hardware

Alberto Lerner
eXascale Infolab
University of Fribourg, Switzerland
alberto.lerner@unifr.ch

Gustavo Alonso
Systems Group, Department of Computer Science
ETH Zurich, Switzerland
alonso@inf.ethz.ch

Abstract—**The requirements arising from ever growing amounts of data and tight performance constraints as well as the limitations encountered in improving conventional CPU performance have led to a proliferation of specialized architectures involving a wide variety of processor types (GPU, TPU, DPU, etc.) with processing becoming distributed across all points of the computing fabric (smart storage, smart memory, smart NICs, programmable switches, etc.). Examples abound both in industry and academia of new architectural configurations and hardware accelerators improving different aspects of a system. These developments raise an important question that is still open but has not attracted sufficient attention: how to design data processing engines systems over such highly heterogeneous and distributed architectures. In this paper we argue that data management engines on modern hardware will necessarily be based on data flow designs where processing happens in a streaming and pipelined fashion across the entire architecture, a radical departure from existing engines. In the paper we argue why this will be the case, the advantages of such designs, and outline a research program to allow data processing engines take advantage of hardware developments.**

1. Introduction

The IT industry is experiencing fundamental changes. The cloud is the dominant computing platform,with a scale and service model that represent a major departure form conventional hardware and software. Emerging applications from data science such as data analytics, machine learning, or large language models, are extremely demanding in terms of the volumes of data involved, the computing resources needed, and the tight performance constraints. At the same time, the compute fabric is quickly evolving to cope with the increasing scale of the cloud and the inadequacy of CPU based systems for data driven applications [1].

A very visible aspect of these trends is the increasing amount of specialization at the hardware level [2], specialization that is driven by the need to cope with the so called *data center tax* [3], to accelerate operations that do not work well on conventional CPUs (e.g., quantized ML with unconventional number representations [4], [5]), and to increase the efficiency of data movement in large scale, largely imbalanced, and highly disaggregated systems.

Specialization today can be seen in the increasing amount of computing not done on a CPU but on GPUs, TPU (Tensor Processing Units), or DPUs (Data Processing Units) [6], [7]. It is also visible in the way processing is spreading along the entire compute fabric: smart storage [8], [9], [10], [11], smart memory [12], [13], [14], [15], smart NICs [16], [17], programmable switches [18], [19], [20], [21], and specialized control modules and devices to perform relevant operations such as compression, encryption, or network virtualization [22], [23], [24], [25]. The resulting picture is one in which processing no longer takes place on the CPU but in other places of the architecture with a multitude of active components along the data path.

Existing database architectures are ill suited to these changes. Relational engines already face many issues in a cloud setting because of, e.g., virtualization and disaggregation [26]. Pre-cloud architectures run in the cloud under a *legacy mode*, often explicitly created for them using systems like disaggregated block storage instead of the cloud standard object store [27] or directly hosting the entire hardware/software stack of the database [28]. These systems have limitations in terms of migration of VMs due to their very large state, they are not elastic, and are cumbersome to start—which is not surprising since these engines were built for the stand-alone server model that goes against the design principles of the cloud.

As a result, the last decade has seen a flurry of new database and data processing engines—so called *cloud native*—that adapt their architecture in a variety of ways: either through a redesign that eschews many common database components (e.g., discarding conventional indexes because of the way disaggregated storage works [29]) or simply by focusing on distributed processing over files instead of on the data management aspects of a typical database engine.

In this paper we argue that neither the cloud native relational engines nor distributed data processing systems in use today will remain competitive in the highly disaggregated, heterogeneous hardware landscape that is already commercially available. The main reason is the substantial *data movement* these systems produce and the disproportionate impact it has on performance and efficiency both at the individual application as well as the data center level.

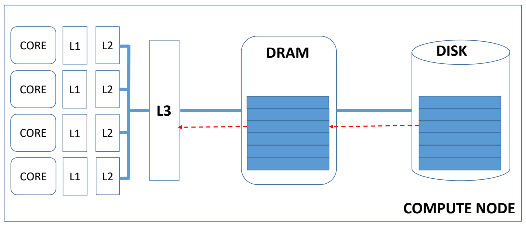

Figure 1. Database engines are still designed for the conventional data path in von Neumann architectures: disk ↔ memory ↔ caches ↔ registers.

Put differently, conventional database engines are largely oblivious to data movement and focus instead on optimizing I/O and keeping as much data as possible in main memory. These goals are at odds with good cloud architecture practices, where storage is disaggregated and DRAM is the most expensive component.

To move forward in the new hardware landscape, data processing has to radically change. In a nutshell, engines will have to become data flow engines processing data in a pipeline, streaming fashion, over a fabric of distributed, heterogeneous processors. This may entail efforts such as:

- Traditional and new operators will have to be redesigned to work on data as it flows from the storage to the network, coming from the network though the NIC, or from memory to a processing element.
- Query plans will need to include a multitude of operations that are now standard in the cloud (e.g., compression, encryption, format transformations), and accommodate execution models that are very different from the pull-based Volcano model [30] or its modern variations.
- Query optimizers will have to consider many more plan options to include the alternatives for offloading of operations along the data path as well as processing on heterogeneous elements.
- The optimizers will need to consider data movement cost in a disaggregated setting as a first-class concern when ranking query plans.
- As the computing unit moves away from the conventional server towards a larger element (e.g., the rack as being proposed by developments such as CXL [31]), the availability of large scale shared memory and massive amounts of parallelism in the form of processors rather than threads will force a complete redesign of almost all aspects of data processing.

2. The Emerging Hardware Landscape

In this section we describe how specialization and disaggregation affect database engines. Later we discuss data processing along the data movement path using different elements of a heterogeneous architecture.

2.1. Conventional Hardware

The conventional von Neumann architecture is based on a processing unit (a CPU as processing and control unit), memory for the data and instructions, external mass storage, and interfaces for I/O (Figure 1). Over the years, different bottlenecks in the architecture have been addressed by adding new elements such as caches to bridge the performance gap between CPU and memories. The technologies for each component have also evolved, e.g., from slow, magnetic HDDs to modern, Flash-based SSDs devices. Database engines have reacted to these changes by making some adjustments without deviating from the central tenets of trying to keep as much data as possible in main memory.

These databases still run as if they accessed local storage, but in the cloud they request data from other systems. Fulfilling these requests requires the storage systems to locate, fetch, decode (for error checking), perhaps decompress, and then ship data across systems. Clearly, the cost of doing so is much larger than if the database system was running on a stand-alone machine requesting data from local storage.

It is fair to say that database engines have been oblivious to the cost changes that moving data in the clouds entail; typical database systems still move data over "large distances" prior to process it. In their defense, though, they try to reduce the amount of data movement by, for instance, using indexes in conventional engines or zone maps in cloud native engines to fetch as little data as possible.

Unfortunately, the issues go beyond data movement. If keeping all data in memory led to simpler architectures, that would be understandable. However, to achieve high I/O bandwidth, the database must partition the data. It is a bit of a contradiction to keep as much data as possible centralized in memory but operate on it as if the data were distributed (i.e., in a partitioned fashion). This approach leads to complex architectures that must deal with separate caches and creates problems with replication, consistency, data locality, resource utilization, etc.

The point we make is that databases should and have evolved to accommodate the changes to what conventional hardware has become. However, the architecture necessary to adapt to the heterogeneous character of the cloud hardware are more substantial. Let us look into what these hardware characteristics are next.

2.2. The Trend towards Disaggregation

What we call the cloud today was initially built on the conventional hardware described above. However, this platform's limitations very soon emerged because certain concepts that are important for cloud processing were not in the picture before. For instance, multi-tenancy and elasticity—arguably the economic pillars of the cloud—require separating storage and computing in different layers connected through the network. This separation leads to disaggregated storage, which, in turn, makes I/O much more expensive. To minimize I/O, users over-provision their VMs with more memory than needed, leading to *stranded memory* [32], [33].

Stranded memory unleashes its own snow ball of issues. It leads, for instance, to adopting disaggregated memory solutions [34] in an attempt to better remaneuver memory among users, which makes not only storage remote but also main memory. Moreover, the need for efficiency and security to deal with remote memory adds significant overhead in terms of data serialization, compression, encryption, etc., all steps needed in a cloud setting [3].

In short, the trend in the cloud is towards increasingly disaggregated systems so as to be able to independently scale each component of the architecture (processors, accelerators, NICs, memory, storage) without being bound by any of them. We see a number of recent technologies to emerge, such as the unrelenting increases in networking speed—the only technology whose speed is doubling consistently every other year–, whose main aim is to support disaggregation. (Why else would a server chassis need 800 Gbps NICs, or even the upcoming 1.6 Tbps ones?). Many such signs indicate that disaggregation is here to stay.

Alas, disaggregation exacerbates the data movement problem and affects data processing applications in fundamental ways. As mentioned above, accessing storage across subsystems is a factor that many database systems ignore. Accessing memory across subsystems is also poorly understood. The cloud providers overwhelmingly use it (e.g., [35]), but there has been documented subtle bugs when database systems adopt it [36]. The reason, we speculate, is that many database systems still work on a legacy data path (disk-memory-cache-CPU) and optimize it for hardware that no longer exists in such form. We argue that database systems need to embrace specialization, a recommendation made by others as well [37].

2.3. The Trend towards Specialization

The specialization that the cloud enables creates the economies of scale that justify large investments in new hardware replacing or complementing the CPU [2], [22], [25], [38], [39], [40]. But, not only the CPU is losing its central place; even the GPU is being replaced or complemented with new types of processors [39], [41], [42].

The demands of AI applications notwithstanding, it is intriguing to see that data management systems are not adopting the specialized solutions that the cloud is adopting for problems that both have. The risk is that future systems will provide very efficient solutions to what are intrinsic problems with data movement and distribution in the cloud but none of them suitable to a data management context.

2.4. Data Processing on Modern Hardware

In view of the growing specialization and disaggregation at all levels of the architecture, a big gap is appearing compared to what is happening with data management. Part of the inertia can be explained by the fact that existing engines are large pieces of code that are slow to develop, tune, and deploy. While more and more cloud native engines are appearing, the market seem to be still dominated by conventional relational products. And quite a few of these cloud engines are, to a very large extent, just variations of well known architectures for distributed databases, e.g., [43].

In the remainder of this paper we argue that a significant research effort is needed to move data processing to be on par with current hardware/cloud developments and trends. The effort is probably similar in scope to the one invested in making the relational model an efficient basis for data processing as it requires to completely change the engine's internal architecture to reflect the reality of modern hardware. In what follows we identify four system components in the architecture (storage, network, memory, and interconnects) that should be involved in data processing and outline a number of research questions to address in each one of them.

3. Processing Near Storage

In the cloud, data resides on the storage layer. Can such layer do something more than just storing the data?

3.1. Conventional Storage

The first layer to consider when trying to improve the situation regarding data management in distributed architectures is the storage. Database engines use indexes to quickly locate data in the storage layer and, thus, reduce the amount of data that needs to be brought all way to the CPU for processing. The block interface of disks makes indexes efficient only to certain extent but they are better than having to read an entire table to discard most of it after it has passed through the I/O bus, the memory, the caches, and finally reaches the CPU registers.

A great deal of an engine's infrastructure (and considerable research) has gone into making the process more efficient through better cache replacement policies, better indexing, different data and page representations, sizing of the buffer pool, etc. But these mechanisms simply do not scale to very large data collections (indexing is actually expensive and competes with the actual data for bandwidth and main memory) and are ill suited to a cloud environment with multiple layers of caching, virtualization, limited network bandwidth, etc.

3.2. Storage in the Cloud and at Scale

In the cloud, the nature of the storage medium is no longer visible. File systems and block devices are offered mostly to support legacy applications. Real cloud storage entail object stores with a very different interface, data formats, and performance characteristics. True cloud native databases (not those who are modifications of conventional engines) already discard things like indexes as they are of not much use in this context. They also work on different data representations, introducing the need for reformatting the data, as it can be seen in Query-As-A-Service systems like Google Bigtable or Amazon Athena which work directly on object storage and do not assume the data is in a

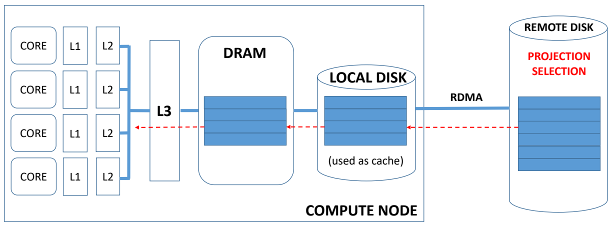

Figure 2. Offloading projection and selection to the remote storage as a way reduce data movement and optimize network utilization.

given format. Proof that data movement here is of paramount importance is the fact that these systems charge for the amount of data read from storage rather than for the actual computation.

An obvious way to reduce the amount of data that needs to move from the storage layer to the compute layer is to move all the filtering stages (projection, selection) to the storage (Figure 2). Pushing down part of the query processing to storage is not exclusive to the cloud. It has been explored in research [9], [11], [44], [45] and systems like Oracle Exadata use it extensively [28]. However, in the cloud it can be far more efficient since the processing infrastructure needed can be shared by all those accessing the storage layer. At the same time, the economies of scale make it feasible to develop specialized processors more efficient than the small CPU cores used in storage servers.

3.3. Research on Smart Storage

A first set of open research questions include identifying the SQL operators that make sense to push down to the storage layer. Even for simple selection, for what data types does it make sense to filter them at the storage rather than at the compute layer? Amazon AQUA [22], for instance, pushed down the LIKE predicate to process regular expressions as that has been proven to be more efficient on accelerators than on a CPU [46]. Keeping in mind that the processing in the storage layer has to be done in a streaming fashion to avoid adding latency and copying data, and that probably has to be mostly stateless to avoid requiring additional memory, there is also room for designing new non-blocking and stateless algorithms for standard operators. Similarly, a certain amount of pre-processing can also be efficiently done in storage: pre-aggregation, pre-sorting, hashing, etc. although probably only to parts of the data rather than to the entire data set. At what granularity does that make sense and how would operators on the compute layer side change given these pre-processing stages?

In addition to these algorithmic and data structure design questions, there are a number of important systems issues to explore. On the one hand, the architecture of the processor on the storage is still a wide open questions that needs further study (see [47] for an example). Disaggregated storage is attached to the network and there will be a NIC involved in sending the data (see below). Should the processing be combined? Similarly, the support for multi-tenancy and cost efficiency dictate that the processing capacity might be limited. What operators make more sense to push down to obtain the bigger gains? And what would be the nature of such processor to support as many operators as possible? Some initial examples exist, e.g., [48] but only for very specific architectures and much more work is needed to explore all possibilities.

Lastly, from the engine perspective, the query optimizer needs to take into account the option of executing part of the query plan in storage. Moreover, the access methods are likely to work in a very different way than in conventional engines since they will be operating on the storage directly.

4. Processing on the Network

Data in the cloud moves across the network. From the NICs in the machines (compute or storage nodes) through switches, to the NICs of the receiving nodes. As with the storage, the question to ask is whether the network can do more than just move the data.

4.1. Conventional and Cloud Networking

Distributed databases have never paid much attention to the network. Certainly not as much as domains such as High-Performance Computing, where additional abstractions have been built to make it easier to optimize network communication with systems like the Message Passing Interface (MPI). In the past, the assumption was that the network would be a TCP/IP stack. Recently, the attention has shifted to RDMA [49], [50], [51], [52] since, in the cloud, most of the traffic is RDMA even if it is not yet available to the cloud user due to security concerns. But even RDMA is changing since RoCE/RDMA is a translation of the concepts in Infiniband RDMA which was designed for supercomputers and not for data centers. An example is Google's Falcon [40], intended to replace the underlying mechanisms of RoCE for an implementation better suited to data centers and directly supported in hardware.

4.2. Smart NICs

Central to these changes in the network are two aspects. One is the need of cloud providers to have better support for network virtualization. This immediately leads to smart NICs where this functionality can be offloaded instead of using CPUs for it [25]. The other aspect is the growing use of accelerators and the bottleneck that having to go through the CPU represents. For instance, when moving data from the storage layer to the GPU, conventional network stacks require to go through the CPU with copies of the data being made along the way and blocking CPU resources. This has led to ways to bypass the CPU [53] and also to smart NICs that can not only communicate directly with the GPU but also perform processing on the network data stream on the fly, e.g., NVIDIA's Bluefield series which terms the devices as Data Processing Units (DPUs) [54]. Their use in database engines is yet to be explored.

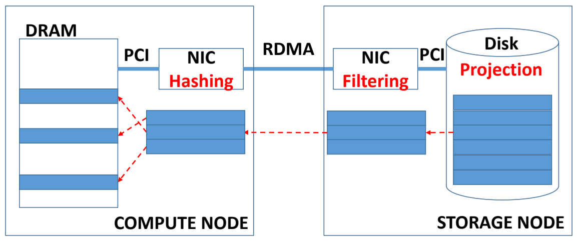

Figure 3. Example of a streaming pipeline between NICs with projection being done directly on storage and hashing done by the receiving NIC.

4.3. Bump-on-the-Wire Accelerators

The widespread use of smart NICs and their growing availability opens up many opportunities for processing on the network using the NIC as a *bump-in-the-wire* accelerator. For instance, Microsoft's Brainwave project [4] uses a smart NIC for a variety of purposes including accelerating key-value stores [55]. Smart NICs can operate on the data as it flows from/to the network to another system component (CPU, memory, or an accelerator). However, while the smart NIC as well as the processing capacity on the storage node are potentially shared by many applications, there will be limitations to what can be done there. The smart NIC on the receiving side (the compute node) does not have such tight limitations and could complement the initial steps of processing with more tailored operations on the data.

4.4. Research on Smart Networking

A first question regarding using smart NICs for data processing is how to distribute the work between the smart storage, the smart NIC on the storage layer and the smart NIC on the compute node. This requires to look at query plans and operators as a finer degree of granularity and, potentially, as a set of stages improving on the previous one. For instance, pre-aggregation could be done first at the storage layer, once more on the sending NIC, and then again on the receiving NIC, thereby creating a pipeline of group-by stages that can achieve more than a single accelerator and significantly cut down the amount of work needed at the final stage of processing.

A more contrived but nevertheless interesting idea would be to perform a join in stages by partitioning one table into blocks spread across the pipeline and streaming the other through the three stages. Similar ideas apply to sorting, hashing, etc. (Figure 3). These questions need to be explored first at an algorithmic and system level but also from the point of view of the optimizer that will need to decide where to perform each operation.

Smart NICs also offer the possibility to implement *exchange* operators that do more than just send data. Smart NICs can be used to partition the data on the fly, perform collective communication (scatter-gather, broadcast), and orchestrate distributed query execution without involvement of the CPU (Figure 4). This opens up many possibilities by delegating the execution of parts of a query plan to the smart NIC that could potentially complete entire queries without even involving the CPU or transferring data to the host memory. For instance, a query returning only a *COUNT* can be executed directly on the NIC that simply counts the data as it arrives and discards it, providing the final results at the end. Depending on the size of the result, the same could be done with, e.g., aggregation queries or joins involving a small table, performing the entire query on the smart NIC.

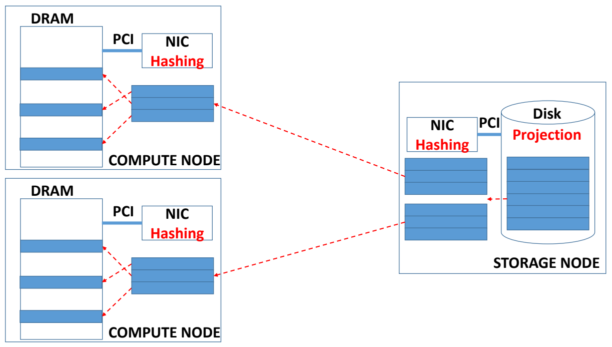

Figure 4. Example of a scattering pipeline to support a distributed, partitioned hash join.

5. Processing Near Memory

The relationship between traditional database engines and main memory is perhaps the most outdated among all the resources. Because of this over-reliance on data movement, database VMs have little flexibility to have quick start-up time [56], present poor VM relocation agility [57], and deal with elastic memory only indirectly [58], to name a few. Having a dataflow approach where the data is processed along the path it moves rather than at the end of it would alleviate all these issues. To understand how, we start by revisiting the state of affairs between CPUs and DRAM.

5.1. Conventional Memory

The relative performance of CPUs and memory have evolved quite irregularly. On certain metrics, the evolution is aligned. Server DRAM capacity, an example of one of these metrics, has been growing steadily with the addition of memory controllers at every new CPU socket.

The issue is that CPUs have not been able to seize these benefits and, in fact, there are stagnant trends when it comes to moving the data. For instance, the access latency has not improved between DDR4 and DDR5, having been at roughly the same levels during the last years. In addition, a CPU core's ability to sustain memory bandwidth has never reached 100% of what a controller is capable. Historically, the best rate that a single thread can achieve on a read workload is 75-85% of the controller's bandwidth and has remained constant for a long time [59], [60].

To make matters worse, some important metrics are even decreasing. CPUs' computational power has grown faster than memory bandwidth: CPUs gained 5x power in GFlops but only 2x bandwidth improvement from 2010 to 2023 [59]. Moreover, the controllers in a typical CPU are oversubscribed w.r.t. the number of cores. Even though no single core can saturate a controller, heavy contention would ensue if even a moderate number of cores on a typical CPU attempt to issue such a memory-bound workload as a database system's.

For these reasons, CPUs have relied on an increasing number of fast cache layers that attenuate the discrepancy to DRAM's speed, and the sizes of these layers have also been increasing over the years. Three cache layers have been the norm, but recently chips reached the market that can use HBM as a fourth layer [61]. To benefit from this type of architecture, traditional database systems have been forced to adopt cache-friendly memory access patterns. This approach puts tremendous constraints on the system design.

Regardless of the use of caches and many optimizations, typical database tasks can and often do suffer from cache faults—not only data but also TLBs faults, which happen when the physical addresses of too many memory pages are required during a short period. Cache faults cause CPUs to stall, waiting a relatively long time (number of cycles) for data to arrive. Moreover, if the data requested during one of these faults is not stored in the local DRAM but on a memory attached to a neighbor CPU socket, there are additional penalties for higher access latency. The phenomenon, called Non-Uniform Memory Access (NUMA), is unavoidable in servers that use two or more CPU sockets—anecdotally, the large majority of servers available in the cloud.

5.2. Processing In- vs. Near-Memory

Conceptually, the solution is simple; we ought to be able to move more of the computing capacity to where the data lies. The rationale is that, by doing so, (a) we alleviate the CPU by performing some of its tasks in other areas of the platform, and (b) if the computing that we perform is reductive, i.e., if the output is smaller than the input, there would be less data to move along the processing pipeline. This is the essence of approach we suggest here.

As it turns out, there is a robust body of knowledge that allows some computations to be executed away from the CPU and closer to memory. To understand the options, it helps to have a conceptual view of a typical memory controller. It is connected to the CPU on one side and to some memory packages assembled in some form factor, e.g., DRAM DIMMs, on the other side. If we change the nature of the memory packages so that, besides storing data, they could also perform some operations over them, we call these methods *processing in-memory*. If, instead, we change the controller so that it can operate on the data that it is moving to and from DRAM, we call these methods *processing near-memory*.

In-memory processing typically leverages the very fabric that interconnects the elemental transistors so that the fabric

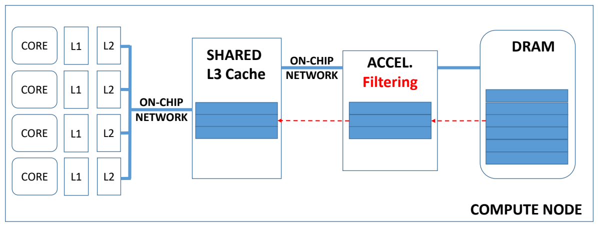

Figure 5. Example of filtering data along the data path from memory to caches.

itself can perform computations. Describing how this is done is beyond our scope, but we should mention that some techniques can embed matrix multiplications [62] or bitwise operations [63] into the fabric. As fascinating as these methods are, they provide a very narrow computing capability and their implementation can require special fabrication, sometimes involving memory types that are yet to reach commercial maturity, such as Resistive RAMs.

In-memory computing can also take a slightly different form. It can couple a specialized accelerator unit with the memory fabric without changing the latter. This approach gives the accelerator unit a privileged access to memory, e.g, with a much larger bandwidth than it would be possible via the data path the CPU uses. Examples of such an approach exist for HBM memory [64] and DDR4 DRAM DIMMs [65], the latter being a commercial product. The accelerator unit attached to memory can offer broader computation capabilities than the techniques that changes the fabric described above.

This type of in-memory computing has been shown to benefit analytic database workloads [66], [67]. The drawback with this approach is that the accelerator unit may not know all ongoing memory operations at a given point. Applications can emit conflicting instructions to the accelerator and to memory, leaving the memory in an inconsistent state. For this reason, when using such accelerators, one should refrain from touching parts of the memory being operated by the unit, which is a confusing programming model.

In contrast to the previous approaches, processing near-memory can also perform computation over data that is in flight but it does so by interposing an accelerator between the memory controller and the CPU. This approach has many successful implementations. The Oracle-Sparc M7 was a CPU that housed one such data accelerator together with the memory controller [68]. The M7 accelerator supported a handful of database operations such as filters that could be customized by the database system via software.

Near-memory accelerators can provide a more powerful computational model because they are not constrained by the inner workings of the memory packages and do not require to modify the CPU since they are external to it (Figure 5).

5.3. Disaggregated Memory

Main memory is the second most precious resource in a data center after computing resources [32]. As such, it is imperative in cloud settings that memory does not get *stranded* in any way because of its physical location [32]. Many attempts have been made to allow an application running on a given server to somehow read and write to external memory—memory placed remotely with respect to that server. Currently, RDMA is the most efficient way to perform such remote accesses, as discussed above. (However, we will mention shortly that a new means is emerging that allows remote memory access with additional benefits.) Because of this remote placement vis-a-vis a server accessing it, this technology is known by *disaggregated memory*.

Disaggregated memory creates an ideal scenario for processing near memory because it naturally separates the CPU on the accessing server and the memory on the external server. The added path between the two allows many different placements for near-memory accelerators that are naturally decoupled from the CPU. For example, an accelerator can very well be coupled with one or both the source and target NICs. An example of such an approach exists that shows how to offload query operators on the bottom part of query plans to NIC-based accelerators [13] . By starting to execute a query plan near memory, the portion of the plan and the remaining that that needs to be processed by the CPU is greatly reduced. Moreover, the accelerator executes in an efficient, specialized hardware what the CPU would need to execute via software. This combination of data reduction and hardware specialization provides performance gains while reducing resource consumption.

5.4. Research on Processing Near memory

Once again, one of the main open questions is to divide the work within a data flow among the available accelerators in the pipeline, as discussed above. We discuss instead a different research question: What kind of hardware functional units should a near-memory accelerator carry?

Like in the Oracle-Sparc M7 chip, we need an array of options to filter data, such as by value, range, or via a provided filtering function. Moreover, it would be interesting if the possibility existed of keeping data in memory compressed and having the accelerator decompress on demand. Such a set of functional units would allow the rest of the pipeline (the cores, aided by the caches) to see only filtered and uncompressed data [68].

The opportunities do not stop there. Another functional unit that can be useful for memory access patterns is pointer chasing. In a CPU-centric architecture, a block of data containing pointers must reach the CPU before one can decide which next data block to request. The processing of the pointer indirection on the CPU requires extensive data movements, which are inefficient. A pointer dereferencing functional unit on the memory controller can exist that, given a data block format and a key (or range), could traverse a hierarchical structure and only send leaf data block up the pipeline. Put differently, let the memory controller perform hierarchical data traversals.

Another functional unit that could support critical operation is one that could perform data transposition. Modern HTAP engines strive to keep data in a recent or historical format for processing and make the conversion from the former to the latter only once. A data transposition functional unit on the memory controller could help in this conversion. It could also virtually reverse it by presenting data in a different format than that in storage. This flexiblity can, to the very least, allow an HTAP engine more leeway of when and how to perform data conversions.

Lastly, several database engines perform background operations that are memory-centric such as garbage collection. A functional unit with fast list primitives could perform some of these maintenance operations near memory.

6. Processing on Interconnects

Disaggregate memory opens tremendous opportunities, but by decoupling the CPU and memory controllers, e.g., via generic network protocols such as RDMA, two great benefits can be lost: bandwidth and cache coherency. Fortunately, there are commercial efforts such as CXL [31], NVLink [69], and InfinityFabric [70] that can support disaggregation without these disadvantages. To understand how, we start by briefly introducing PCIe, an interconnect protocol that is at the heart of systems integration.

6.1. Conventional Architectures

PCIe is, for all practical purposes, a networking protocol [71]. It is the *de facto* standard to connect a server's CPU and memory to peripheral devices, such NICs, SSDs, GPUs, and all sorts of accelerator cards. Unbeknown to most, PCIe can also be used to interconnect two different servers [72]—more on that shortly. Like memory, PCIe evolution has also been somewhat irregular. It quickly reached its third generation in 2010, PCIe 3, which saw tremendous adoption, but it somewhat stagnated then.

Not surprisingly, competing interconnect technologies have surfaced, pushed by the affected parties. NVidia, one of the companies driving the widespread GPU adoption, created its own, faster interconnect, NVLink. NVLink is a closed protocol and remains used exclusively by NVidia hardware. AMD also pushed a variation called InfinityFabric. That protocol, too, remains closed.

The bandwidth limitations of PCIe fragmented the market but allowed faster interconnects to appear, which are fundamental to the data flow architectures we propose here, but another important element was still missing.

6.2. Beyond PCIe

We mentioned above that PCIe is a networking protocol. The statement is correct, but it tells little about the type of networking that PCIe supports. It is, in fact, a memory-centric networking protocol. In practice, this means that PCIe transport layer packets, called TLP, are typed, i.e., each packet type corresponds to a memory operation. In other words, PCIe is a protocol that allows two entities to access each other's memory. These entities are typically a server and peripheral card. The addresses on both entities are unified by a hardware unit called IOMMU.

The important aspect missing in PCIe is that these memory operations are not *coherent*. When a peripheral reads a memory area that is controlled by a server, it effectively creates a copy of that area. If the server updates any values in that area, the copy held by the peripheral would be outdated. Protocols like RDMA and NVMe [73], which use PCIe underneath to have the server communicate with, respectively, a NIC or an SSD, do have shared data structures, but these structures are designed so that servers and peripherals never manipulate the same region simultaneously.

Similarly, the lack of coherence has not prevented database systems from running on disaggregated scenarios. Consider examples of indices maintained on disaggregated memory, such as [74] and [75]. They allow several servers to concurrently access remote memory through a combination of PCIe and RDMA, where the coherence is maintained, once again, via software—but not without its pitfalls [36].

Some important entities in the Industry were attuned to the need to operate on remote memory coherently and, while at it, be able to do so with lower latencies and wider bandwidths. In 2019, a consortium led by Intel was formed, whose goal was to draft the evolution of PCIe [76]. The first specification of a new protocol, called Compute Express Link (CXL), was soon ratified [77]. The protocol has evolved since and has swallowed many competing efforts, such as OpenCAPI, CCIX, and Gen-Z.

CXL brought tremendous advantages for our data flow purposes in at least two ways. First, it forced the evolution of PCI to its 5th and 6th generations, doubling the bandwidth twice to 64 GB/s (or 128 GB/s bidirectional). The PCIe 7 protocol should be ratified in 2025 and will double bandwidth again. It does not seem we will lack bandwidth improvements for the foreseeable future for disaggregated processing schemes, removing one of the main concerns about this type of architecture for data processing.

The second advantage of CXL in data flow architectures is perhaps the most impactful: cache coherency. Applications do not need to coordinate MRd's and MRw's PCIe operations, nor built software-based coherence techniques using RDMA—although they still can since CXL keeps compatibility with PCIe in a protocol called `cxl.io`. In fact, CXL added brought two additional protocols: `cxl.mem` and `cxl.cache`. With `cxl.mem`, the memory on peripherals and hosts can be unified. The recent addition of *Global Integrated Memory* to the protocol allows the memory of multiple hosts and peripherals to be federated into a single memory space. With `cxl.cache`, any device can *coherently* cache any portion of the memory space.

The coherency need not be maintained by software or the application; *cache coherency is the responsibility of the hardware*. In practice, coherency allows a near-memory accelerator to operate on the data at the same time as a CPU core. Consider Figure 5. If the accelerator updates the memory, e.g., by writing a new value to a database tuple, any cache holding the modified address will be invalidated through a series of *cxl.cache* messages that the caches and the memory controller would automatically send. Ultimately, cache coherency expands the design space of data flow architectures because it allows many active agents to cache and operate on the latest version of the memory's contents simultaneously.

6.3. Beyond Memory Controllers

Another subtle but powerful consequence of implementing the coherency protocol in hardware is that applications can use it without the need for an API. In the same way that an application issues a `load` or a `store` instruction to access values from local memory, it can do exactly the same to access remote memory with CXL. As long as the remote controller participates in a CXL *coherency domain*, and as long as the interconnect can carry the cache invalidation traffic, the application cannot tell (other than a slightly higher latency) whether a given portion of the memory is local or remote.

CXL is not exclusively available to memory controllers. Any device wanting to cache memory from another device or offer its memory to the collective can emit and process `cxl.cache` and `cxl.mem` messages, respectively, and join a CXL coherency domain. There is a growing number of NICs and SSDs that can support alternative interfaces this way [78], [79] .

In summary, CXL can blur the server boundaries from an application point of view with several ramifications we quickly discuss next.

6.4. Research on Processing on Interconnects

If memory and storage can be disaggregated and applications can access them through CXL, we can question whether the way to build a large platform is still through connecting servers that bundle CPUs, memory, storage, and networking under the same chassis. A much more flexible way is to think of computers in terms of racks and populate the rack with more carefully apportioned resources, i.e., build fully disaggregated platforms.

The research agenda involving these platforms are plentiful. We discuss the motivation and alternatives to structure them towards serving data-intensive systems in much more detail elsewhere [80]. For now, we note that laying out a dataflow pipeline over such a disaggregated platform can be done quite naturally (Figure 6). The advantages of doing so via CXL are having the hardware take care of coherency and not requiring special APIs to access external memory. Both of these benefits greatly reduce the software complexity necessary to implement the data flows.

7. A New Query Processing Model

Given the above, a design exploiting all possible processing steps along the data path will involve the following pipeline of processing elements: near storage processing,

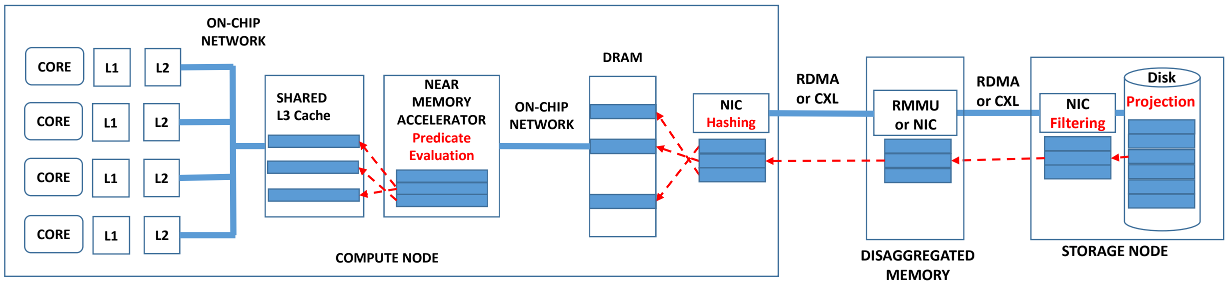

Figure 6. Example of a full pipeline of processing stages along the data path from storage to cores.

processing on the network (sending and/or receiving NIC), processing on interconnects, processing near memory (Figure 6). Intuitively, if designed correctly, such an architecture will optimize data movement and perform operations much faster than the CPU. However, the key question is what does it mean to design such a pipeline correctly? How does a query plan look like in such a system?

7.1. Data Movement on Query Plans

As we shift away from a CPU-centric architecture, it should not be surprising that the data flow model we propose here does not rely on having the CPU be the sole responsible for pulling data. A growing body of work advocates using the DMA engines along the data path to perform customized data movement on an application's behalf since these engines already exist in, e.g., NICs, SSDs, or CPUs.

What we envisage for data movement is a sequence of queues placed strategically in the pipeline that are connected via DMA engines. Data is processed in one stage and sent to the next depending on that stage's queue availability. This flow control method is called *credit-based* and is used, for instance, in the PCIe stack [71]. Credit-based flow control requires a counter stream of messages from one stage into the previous, informing about a credit budget (in terms of space). This type of control flow is easy to implement and it is low traffic.

Naturally, the queue elements will eventually reach queues in the compute nodes, and that data will be handled by software. Whether the software pulls one or more data elements at a time is relevant to the performance of that stage only—and is completely orthogonal to data flow architecture. The point here is that moving data within the last stage of the plan (the compute node) should represent only a fraction of the work done by the query plan.

7.2. Query Interpretation vs. Compilation

A typical database engine will run a query plan as a CPU process or thread set. The program can be obtained on a per-query basis through a compilation process, or it can be a generic query plan interpreter that takes the query plan as input. Implicit in a typical database query execution process is that all hardware access is done through an ISA, i.e., via software instructions. This is a fair assumption when all the hardware a plan uses amounts to a CPU.

Some accelerators, however, are programmed directly—they lack an ISA—, simply by filling a small set of memory-mapped registers. For example, turning compression/decompression on and off on a per-flow basis can be done this way. Such an accelerator needs to provide a flow identifier and a value for the compression option. The accelerator can have specific registers for such information.

Other accelerators can be programmed through more sophisticated ways, depending on how they decide to expose (or layer abstractions on top of) their functional units. These accelerators may require a combination of register activations, as above, in addition to the installation of some logic—still through other means than an ISA. For instance, when it comes to operations that require finding tuples on database pages and perhaps filtering those tuples, registers can be used to characterize the filter, but parsing logic is necessary to find where the tuples and relevant attributes are within a page.

The literature refers to the operational information passed on to accelerators as *kernels*. Kernels can be explicitly coded, as in CUDA kernels for GPU platforms, but can also be derived through a transformation process, e.g., via program synthesis [81]. For what we propose here, whether this process consists of compilation or interpretation is immaterial. The debate about the merits of each approach is centered around how CPU efficient the plans generated by each different methods are. Efficiency in our approach comes from engaging the accelerators available along the data path.

7.3. Query Scheduling

The enemy of sustained performance in this environment is *interference*. The issue with interference is that when two or more query plans wish to access a given limited resource, performance loss can ensue because of the additional work that (a) the arbitration for the resource may entail and (b) the overhead of repeating acquisition and relinquishing of the resource. This is the reason why scheduling is particularly important on this platform. It is responsible for co-locating plans on the platform while avoiding interference.

For a scheduling subsystem to effectively guard against interference, query plans in this architecture should adopt at least two aspects. First, they should contain several data path alternatives. The natural ones are a plan that uses every available accelerator on the data path and a plan entirely executed on a compute node (a traditional CPU-centric plan). Ideally, there would be some variations in between. Given a set of options, a scheduler may decide which plan variation to activate at runtime.

The second characteristic of a data plan is the ability to adjust its resource consumption during runtime. The fundamental resource in this architecture is bandwidth for data flow. If DMA engines push the data through a large portion of query plans, the scheduler should be able to rate limit the bandwidth used. Rate-limiting DMA engines is a commonplace feature and can take place dynamically.

7.4. No More Buffer Pools

Since the early days of the relational model, an obsession in data processing engines is keeping data in memory for fast access. This is the purpose of the buffer pool, which acts as a main memory cache for the query processing threads. Such an architecture anchors the engine to a given machine and makes it difficult to move the workload around and scale it. In distributed databases, complex protocols implement a form of shared memory across multiple buffer pools requiring to optimize *data-* and *function-shipping* and keeping track of where the data is located. The approach distributes the engine but limits its elasticity. In the cloud, the problem is addressed for OLTP engines by introducing limitations such as read-only copies and primary-copy approaches [43] long known form the literature [82]. For OLAP engines, the solution is to rely on caching layers using main memory or even customized hardware [22]. Both solutions are highly inefficient in terms of the memory required, which is the most expensive component of the data center.

The architecture we propose is the antidote to the main memory addiction of database designs. As storage and network speeds increase and are complemented with processing along the data path, engines can potentially completely operate without a buffer pool and working directly over the stored data. The idea is not as radical as it might appear as this is how data-as-a-service systems such as Amazon's Athena or Google Bigtable operate (i.e., without a buffer pool) but lacking an explicit processing pipeline along the data path. Our approach would minimize the amount of main memory required for data processing and make it fully elastic as the compute layer would be stateless.

7.5. No More Data Caches

A similar argument applies to the heavy use of main memory caches in the cloud to minimize the impact of disaggregation. Cloud object storage is implemented on slow disks for cost reasons. And it is accessible through the network. This all adds latency and it also requires to use many parallel disks to get a reasonable bandwidth. Caching is used to shorten the data path and to make access faster by putting the data on a faster medium (SSDs or memory). However, this is predicated on the model that the data has to be brought all the way to the CPU to determine whether it is actually needed, making the approach highly inefficient. Our active pipeline model offers a different trade-off: eliminate caches but fully optimize the data that has to be moved and process it where the maximum performance advantage can be obtained (since many operators are faster on the type of streaming operators we have discussed that on the CPU). Caching of results would still make sense but there would be no caching of base tables or raw data.

8. Conclusion

In summary, current trends in specialization and disaggregation require to completely redesign our approach to data processing. Our approach is a design based on the emergence of multiple processing opportunities along the data path using processors and accelerators that are being introduced for other purposes but that could as well hep with data processing, the key element of the most important workloads these days. We have motivated such a design and listed numerous research questions that, when addressed, will provide key insights for future designs and also help inform the development and evolution of future hardware. These research questions constitute a first step towards an agenda that should lead to innovative engines aligned with the trends in the IT world.

Acknowledgments

This work received funding from the the Swiss State Secretariat for Education (SERI) in the context of the Smart-Edge EU project (grant agreement No. 101092908).

References

[1] B. Dally, "Power, programmability, and granularity: The challenges of exascale computing," in *2011 IEEE International Test Conference*. IEEE Computer Society, 2011, pp. 12–12.

[2] N. C. Thompson and S. Spanuth, "The decline of computers as a general purpose technology," *Communications of the ACM*, vol. 64, no. 3, feb 2021.

[3] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," *ISCA'15*, vol. 43, 2015.

[4] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, 2018.

[5] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale dnn processor for real-time AI," in *ISCA*, 2018.

[6] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Communications of the ACM*, vol. 63, no. 7, pp. 48–57, 2020.

[7] W. Noureddine, "The Fungible DPU: A New Category of Microprocessor for the Data-Centric Era," in *2020 IEEE Hot Chips 32 Symposium (HCS)*, 2020.

[8] Amazon, "Filtering and retrieving data using Amazon S3 Select," https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html.

[9] L. Woods, Z. István, and G. Alonso, "Ibex: an intelligent storage engine with support for advanced SQL offloading," *VLDB*, 2014.

[10] S. Kang, J. Kim, G. Lee, J. Lee, J. Seo, H. Jung, Y. H. Song, and Y. Park, "ISP agent: A generalized in-storage-processing workload offloading framework by providing multiple optimization opportunities," *ACM Transactions on Architecture and Code Optimization*, jan 2024.

[11] K. Lee, I. Jo, J. Ahn, H. Lee, H. Lee, W. Sul, and H. Jung, "Deploying computational storage for htap dbmss takes more than just computation offloading," *VLDB*, 2023.

[12] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access, High-Performance memory disaggregation with DirectCXL," in *USENIX ATC'22'*, 2022. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/gouk

[13] D. Korolija, D. Koutsoukos, K. Keeton, K. Taranov, D. S. Milojicic, and G. Alonso, "Farview: Disaggregated memory with operator off-loading for database engines," in *CIDR*, 2022.

[14] M. K. Aguilera, E. Amaro, N. Amit, E. Hunhoff, A. Yelam, and G. Zellweger, "Memory disaggregation: why now and what are the challenges," *ACM SIGOPS Oper. Syst. Rev.*, vol. 57, no. 1, 2023.

[15] R. Wang, J. Wang, S. Idreos, M. T. Özsu, and W. G. Aref, "The case for distributed shared-memory databases with RDMA-enabled memory disaggregation," *VLDB*, 2022.

[16] Z. Guo, H. Zhang, C. Zhao, Y. Bai, M. Swift, and M. Liu, "LEED: A low-power, fast persistent key-value store on SmartNIC JBOFs," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023.

[17] Z. István, D. Sidler, and G. Alonso, "Caribou: Intelligent distributed storage," *VLDB*, 2017.

[18] J. Li, Y. Lu, Y. Zhang, Q. Wang, Z. Cheng, K. Huang, and J. Shu, "Switchtx: scalable in-network coordination for distributed transaction processing," *VLDB*, 2022.

[19] T. Jepsen, A. Lerner, F. Pedone, R. Soulé, and P. Cudré-Mauroux, "In-network support for transaction triaging," *VLDB*, 2021.

[20] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. K. Ports, I. Stoica, and X. Jin, "Harmonia: near-linear scalability for replicated storage with in-network conflict detection," *VLDB*, 2019.

[21] R. Hussein, A. Lerner, A. Ryser, L. Bürgi, A. Blarer, and P. Cudré-Mauroux, "GraphINC: Graph pattern mining at network speed," in *SIGMOD*, 2023.

[22] J. Barr. (2021) Aqua (advanced query accelerator) – a speed boost for your amazon redshift queries. [Online]. Available: https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/

[23] M. Chiosa, F. Maschi, I. Müller, G. Alonso, and N. May, "Hardware acceleration of compression and encryption in SAP HANA," *VLDB*, 2022.

[24] Y. Fang, C. Zou, and A. A. Chien, "Accelerating raw data analysis with the ACCORDA software and hardware architecture," *VLDB*, 2019.

[25] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: Smartnics in the public cloud," in *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, 2018.

[26] M. Stonebraker, A. Pavlo, R. Taft, and M. L. Brodie, "Enterprise database applications and the cloud: A difficult road ahead," in *2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, March 11-14, 2014*, 2014, pp. 1–6.

[27] Amazon, "Managed SQL Databases," https://aws.amazon.com/rds/.

[28] Oracle, "Why Oracle Exadata platforms are the best for Oracle Database," https://www.oracle.com/engineered-systems/exadata/.

[29] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner, "The Snowflake Elastic Data Warehouse," in *SIGMOD*, 2016.

[30] G. Graefe and W. McKenna, "The volcano optimizer generator: extensibility and efficient search," in *ICDE*, 1993.

[31] D. D. Sharma, "Compute express link (CXL): Enabling heterogeneous data-centric computing with heterogeneous memory hierarchy," *IEEE Micro*, vol. 43, no. 2, pp. 99–109, 2023.

[32] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: CXL-based memory pooling systems for cloud platforms," in *ASPLOS 2023*, 2023.

[33] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "TPP: Transparent page placement for CXL-enabled tiered-memory," in *ASPLOS 2023*, 2023.

[34] Samsung, "Samsung electronics introduces industry's first 512gb CXL memory module," https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module.

[35] W. Bai, S. S. Abdeen, A. Agrawal, K. K. Attre, P. Bahl, A. Bhagat, G. Bhaskara, T. Brokhman, L. Cao, A. Cheema, R. Chow, J. Cohen, M. Elhaddad, V. Ette, I. Figlin, D. Firestone, M. George, I. German, L. Ghai, E. Green, A. Greenberg, M. Gupta, R. Haagens, M. Hendel, R. Howlader, N. John, J. Johnstone, T. Jolly, G. Kramer, D. Kruse, A. Kumar, E. Lan, I. Lee, A. Levy, M. Lipshteyn, X. Liu, C. Liu, G. Lu, Y. Lu, X. Lu, V. Makhervaks, U. Malashanka, D. A. Maltz, I. Marinos, R. Mehta, S. Murthi, A. Namdhari, A. Ogus, J. Padhye, M. Pandya, D. Phillips, A. Power, S. Puri, S. Raindel, J. Rhee, A. Russo, M. Sah, A. Sheriff, C. Sparacino, A. Srivastava, W. Sun, N. Swanson, F. Tian, L. Tomczyk, V. Vadlamuri, A. Wolman, Y. Xie, J. Yom, L. Yuan, Y. Zhang, and B. Zill, "Empowering azure storage with RDMA," in *NSDI 23*, 2023.

[36] T. Ziegler, J. Nelson-Slivon, V. Leis, and C. Binnig, "Design guidelines for correct, efficient, and scalable synchronization using one-sided RDMA," *SIGMOD*, 2023.

[37] J. Hennessy and D. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," in *ISCA*, 2018.

[38] Microsoft, "Improved cloud service performance through ASIC acceleration," https://azure.microsoft.com/en-us/blog/improved-cloud-service-performance-through-asic-acceleration/.

[39] Google, "Accelerate AI development with Google Cloud TPUs," https://cloud.google.com/tpu/.

[40] Google, "Google opens Falcon, a reliable low-latency hardware transport, to the ecosystem," https://cloud.google.com/blog/topics/systems/introducing-falcon-a-reliable-low-latency-hardware-transport.

[41] Microsoft, "With a systems approach to chips, Microsoft aims to tailor everything 'from silicon to service' to meet AI demand," https://news.microsoft.com/source/features/ai/in-house-chips-silicon-to-service-to-meet-ai-demand/.

[42] AWS, "Amazon web services high-performance, low-cost ML infrastructure is accelerating innovation in the cloud," https://www.technologyreview.com/2021/11/01/1038962/high-performance-low-cost-machine-learning-infrastructure-is-accelerating-innovation-in-the-cloud/.

[43] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon Aurora: Design considerations for high throughput cloud-native relational databases," in *SIGMOD'17*, 2017.

[44] S. Lee, A. Lerner, P. Bonnet, and P. Cudré-Mauroux, "Database kernels: Seamless integration of database systems and fast storage via CXL," in *CIDR*, 2024.

[45] A. Lerner and P. Bonnet, *Principles of Database and Solid-State Drive Co-Design*, ser. Synthesis Lectures on Data Management. Springer, 2024.

[46] D. Sidler, Z. István, M. Owaida, and G. Alonso, "Accelerating pattern matching queries in hybrid CPU-FPGA architectures," in *SIGMOD*, 2017.

[47] S. Lee, A. Lerner, A. Ryser, K. Park, C. Jeon, J. Park, Y. H. Song, and P. Cudré-Mauroux, "X-SSD: A storage system with native support for database logging and replication," in *SIGMOD*, 2022.

[48] H. Caminal, Y. Chronis, T. Wu, J. M. Patel, and J. F. Martínez, "Accelerating database analytic query workloads using an associative processor," in *ISCA'22*.

[49] E. Zamanian, X. Yu, M. Stonebraker, and T. Kraska, "Rethinking database high availability with rdma networks," *VLDB*, 2019.

[50] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes, "Tailwind: Fast and atomic rdma-based replication," in *USENIX ATC'18*, 2018. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/taleb

[51] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler, "Distributed join algorithms on thousands of cores," *VLDB*, 2017.

[52] C. Barthels, I. Müller, K. Taranov, G. Alonso, and T. Hoefler, "Strong consistency is not hard to get: Two-phase locking and two-phase commit on thousands of cores," *VLDB*, 2019.

[53] NVIDIAG, "Nvidia opens nvlink for custom silicon integration," https://nvidianews.nvidia.com/news/nvidia-opens-nvlink-for-custom-silicon-integration.

[54] NVidia, "NVIDIA BlueField Networking Platform," https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[55] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-Direct: High-performance in-memory key-value store with programmable NIC," in *OSDI, 2017.*

[56] O. Poppe, Q. Guo, W. Lang, P. Arora, M. Oslake, S. Xu, and A. Kalhan, "Moneyball: proactive auto-scaling in Microsoft Azure SQL database serverless," *VLDB*, 2022.

[57] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: live migration in shared nothing databases for elastic cloud platforms," in *SIGMOD*, 2011.

[58] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application level ballooning for efficient server consolidation," in *EuroSys*, 2013.

[59] J. D. McCalpin, "The evolution of single-core bandwidth in multicore processors," https://sites.utexas.edu/jdm4372/2023/04/25/the-evolution-of-single-core-bandwidth-in-multicore-processors/.

[60] ——, "The evolution of single-core bandwidth in multicore systems — update," https://sites.utexas.edu/jdm4372/2023/12/19/the-evolution-of-single-core-bandwidth-in-multicore-systems-update/.

[61] Intel, "Intel xeon cpu max series – product brief," https://www.intel.com/content/www/us/en/content-details/765366/intel-xeon-cpu-max-series-product-brief.html.

[62] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ISCA*, 2016.

[63] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: in-memory accelerator for bulk bitwise operations using commodity dram technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[64] T. R. Kepe, E. C. de Almeida, and M. A. Z. Alves, "Database processing-in-memory: an experimental study," *VLDB*, 2019.

[65] UpMEM, https://www.upmem.com.

[66] C. Lim, S. Lee, J. Choi, J. Lee, S. Park, H. Kim, J. Lee, and Y. Kim, "Design and analysis of a Processing-in-DIMM join algorithm: A case study with UPMEM DIMMs," *SIGMOD*, 2023.

[67] A. Baumstark, M. A. Jibril, and K.-U. Sattler, "Accelerating large table scan using processing-in-memory technology," *Datenbank-Spektrum*, vol. 23, no. 3, pp. 199–209, 2023.

[68] K. Aingaran, S. Jairath, G. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, and T. Wicki, "M7: Oracle's next-generation sparc processor," *IEEE Micro*, vol. 35, no. 2, pp. 36–45, 2015.

[69] NVidia, "NVLink and NVSwitch: The building blocks of advanced multi-GPU communication—within and between servers," https://www.nvidia.com/en-us/data-center/nvlink/.

[70] AMD, "Infinity architecture: A new era in accelerated system connectivity," https://www.amd.com/en/technologies/infinity-architecture.

[71] M. Jackson, R. Budruk, J. Winkles, and D. Anderson, *PCI Express Technology 3.0*. Mindshare Press, 2012.

[72] Linux, "Non-transparent bridge driver," https://www.kernel.org/doc/Documentation/ntb.txt.

[73] NVMe, "NVMe specifications overview," https://nvmexpress.org/specifications/.

[74] T. Ziegler, S. T. Vani, C. Binnig, R. Fonseca, and T. Kraska, "Designing distributed tree-based index structures for fast rdma-capable networks," in *SIGMOD*, 2019.

[75] Q. Wang, Y. Lu, and J. Shu, "Sherman: A write-optimized distributed b+tree index on disaggregated memory," in *SIGMOD*, 2022.

[76] CXL, "CXL consortium members," https://computeexpresslink.org/our-members/.

[77] ——, "CXL specification," https://computeexpresslink.org/cxl-specification/.

[78] E. Amaro, Z. Luo, A. Ousterhout, A. Krishnamurthy, A. Panda, S. Ratnasamy, and S. Shenker, "Remote memory calls," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, 2020.

[79] M. Jung, "Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd)," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022.

[80] A. Lerner and G. Alonso, "CXL and the return of scale-up database engines," *CoRR*, vol. abs/2401.01150, 2024.

[81] S. Bhatia, S. Kohli, S. A. Seshia, and A. Cheung, "Building Code Transpilers for Domain-Specific Languages Using Program Synthesis," in *ECOOP*, 2023.

[82] C. Plattner and G. Alonso, "Ganymed: Scalable replication for transactional web applications," in *Middleware 2004, ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada, October 18-20, 2004, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3231. Springer, 2004, pp. 155–174.