DISS. ETH NO. 30184

ADAPTING DATABASE COMPONENTS TO HETEROGENEOUS ENVIRONMENTS

A thesis accepted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zurich)

presented by

DIMITRIOS KOUTSOUKOS

MSc in Data Science, ETH

born on 20.10.1992

accepted on the recommendation of

PROF. DR. GUSTAVO ALONSO

PROF. DR. ANA KLIMOVIC

DR. THEODOROS REKATSINAS

PROF. DR. THOMAS NEUMANN

2024

IKS-Lab
Institute for Computing Platforms
ETH Department of Computer Science

Dimitrios Koutsoukos

A dissertation submitted to
ETH Zurich
for the degree of Doctor of Sciences

DISS. ETH NO. 30184

examiner:
Prof. Dr. Gustavo Alonso
co-examiners:
Prof. Dr. Ana Klimovic
Dr. Theodoros Rekatsinas
Prof. Dr. Thomas Neumann

Examination date: April 11, 2024

Adapting database components to heterogeneous environments

Abstract

Data management has seen rapid evolution during the last years, influenced by factors such as data explosion, the prevalence of machine and deep learning, the slowdown of Moore's law and the popularity of hardware accelerators. Data processing systems are trying to adapt to all these trends by building monolithic and highly specialized systems, which are blazingly fast but quickly become obsolete.

In this thesis, we show how to adapt data processing components that easily and quickly adapt to the underlying technology and new applications and deployments. We first explore software abstractions and how we can use them for general data processing. We find that a more granular implementation of traditional relational operators, called sub-operators, can be used to port a query engine to different platforms. Then, we investigate tensors as a core processing element for graph and relational algorithms. We utilize tensor computing runtimes and we demonstrate tensors to be a suitable representation to accelerate execution and adapt to the underlying hardware automatically.

Afterwards, we shift our attention to the I/O bottleneck and data movement problem, one of the biggest challenges databases face today. We perform an extensive experimental data analysis on Intel Optane Persistent Memory, the first public implementation of Non-volatile memory. Our analysis provides insights on how to best use it and showcases the weak points of the hardware. Data movement is also a difficult problem for end users in the cloud, given that cloud providers have conflicting profit and performance goals. Therefore, we introduce serverless cracking, a modern form of database cracking adapted to QaaS that can lower both the execution time and cost for infrequent workloads based on semi-structured data. Both approaches demonstrate that, although we have made steps towards improving I/O performance, there is still much ground to cover in this area.

Our findings show how to make data management systems more robust to software and hardware evolution and how to adapt and adopt them to the expanding data demands. We also underscore

the critical need for continued innovation in reducing the data movement bottleneck and we make a step towards developing responsive and more efficient data processing systems.

Keywords: data management systems, software abstractions, adaptability, tensors, hardware evolution, modular operators, I/O bottleneck, non-volatile memory, serverless cracking, query-as-a-service systems

ZUSAMMENFASSUNG

Die Datenverwaltung hat in den letzten Jahren eine rasante Entwicklung erlebt, beeinflusst durch Faktoren wie die Datenexplosion, die Verbreitung von Machine- und Deeplearning, die Verlangsamung des Mooreschen Gesetzes und die Popularität von Hardware-Beschleunigern. Datenverarbeitungssysteme versuchen, sich an all diese Trends anzupassen, indem sie monolithische und hochspezialisierte Systeme entwickeln, die blitzschnell sind, aber schnell veralten.

In dieser Doktorarbeit untersuchen wir, wie man Datenverarbeitungssysteme anpassen kann, die sich leicht und schnell an die zugrundeliegende Technologie sowie an neue Anwendungen und Implementierungen anpassen können. Zuerst erforschen wir Software-Abstraktionen und wie wir sie für die allgemeine Datenverarbeitung nutzen können. Wir zeigen, dass eine detailliertere Version traditioneller relationaler Operatoren, sogenannte Sub-Operatoren, dafür verwendet werden können, um eine Abfrage-Engine an verschiedene Plattformen anzupassen. Darüber hinaus verwenden wir Tensoren als ein Kernelement der Verarbeitung für Graph- und relationale Algorithmen und nutzen Tensor-Computing-Laufzeiten, um die Ausführung zu beschleunigen und automatisch an die zugrundeliegende Hardware anzupassen.

Dann verlagern wir unsere Aufmerksamkeit auf das I/O-Flaschenhals- und Datenbewegungsproblem, eine der größten Herausforderungen, mit denen Datenbanken heute konfrontiert sind. Wir führen eine umfangreiche experimentelle Datenanalyse auf Intel Optane Persistent Memory durch. Intel Optane Persistent Memory ist die erste öffentliche Implementierung von nichtflüchtigem Speicher. Unsere Analyse liefert Einblicke, wie man diesen Speicher am besten nutzen kann, und zeigt die Schwachstellen der Hardware auf. Zusätzlich führen wir serverloses Cracking ein, eine moderne Form des Datenbank-Crackings, angepasst an QaaS, das sowohl die Ausführungszeit als auch die Kosten für gelegentliche Workloads auf Basis von semi-strukturierten Daten senken kann.Unsere Analysen zeigen, dass in diesem Bereich noch viel zu tun bleibt. Unsere beiden Ansätze machen erste Schritte zur Verbesserung der I/O-Leistung.

Unsere Ergebnisse zeigen, wie man Datenmanagementsysteme robuster gegenüber Software-

und Hardware-Evolution machen und wie man sie an die wachsenden Datenanforderungen anpassen und adoptieren kann. Wir betonen auch die kritische Notwendigkeit kontinuierlicher Innovationen, um das Datenbewegungsproblem zu reduzieren, und machen einen Schritt in der Entwicklung reaktionsfähiger und effizienterer Datenverarbeitungssysteme.

Schlüsselwörter: Datenmanagementsysteme, Software-Abstraktionen, Anpassungsfähigkeit, Tensoren, Hardware-Evolution, modulare Operatoren, I/O-Flaschenhals, nichtflüchtiger Speicher, serverloses Cracking, Query-as-a-Service-Systeme

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Gustavo Alonso, for his unwavering support, guidance, and mentorship throughout the course of my PhD journey. His expertise and insights have been invaluable, not only to this thesis but also to my growth as a researcher.

Equally, I am very thankful to my co-advisor, Ana Klimovic, whose discussions, questions and criticism have significantly enriched my research approach. I would also like to extend my thanks to Theo Rekatsinas and Thomas Neumann for serving in my committee. Especially for Theo, his advice have been crucial to my professional development and thinking. A special thank you goes to my colleagues and peers in the Systems Group for creating a stimulating and supportive research environment.

On a personal note, I would like to thank all my friends that are spread throughout the globe for their emotional support and love. Thank you Stratos, Antonis, Yannis, Panagiotis, Marcel, Vaggelis, Nikos, Vasilis, Tania, Monica, Andrea and Renato. The list is not exhaustive so please forgive me if I haven't written some of you.

Finally, I must acknowledge the unwavering support and love from my parents Yannis and Anastasia, my brother Vasilis, and my grandma Maria. Their belief in me, even in the face of challenges, has been a source of strength and motivation. I cannot express enough gratitude to my partner, Alice, for her endless love, patience (sometimes more than needed), and support throughout this challenging journey. Her belief in me has been a constant companion, providing both a comforting presence and a joyful distraction at all times. For all the moments, both big and small, that we shared throughout this process, I am eternally grateful.

Dimitrios Koutsoukos
Zürich, 2024

Contents

CHAPTER 1

INTRODUCTION

1.1 Motivation and Thesis Statement

Data management has changed rapidly over the last few years due to many factors. First of all, technology has become an integral part of our everyday lives in many forms, leading to an increased production of data. Sources [49] estimate that in 2024, 149 zetabytes of data will be produced. The majority of these data need to be stored and processed, with an increasing need for real-time processing [57], especially for Machine Learning (ML) and Deep Learning (DL) workloads. These workloads are another reason that aggressively changes data processing. Since the explosion of deep learning in 2012, it is estimated that ML/DL increases the demand in computing capacity every year by more than 10x [102]. This trend has been exarcebated by the recent introduction of Large-Language Models (LLMs), which among other models are notoriously expensive to be trained from scratch [31]. These new workloads and algorithms have expanded the types of data that are needed to be stored (e.g. vector embeddings) and have also made databases part of a pipeline of systems that are used for analysis. Today, it is very common for data scientists to run SQL queries to extract data from a data management system, to further preprocess them and perform feature engineering, to finally train ML models. DBMSs, therefor, have transitioned from a major player for enterprise data to a piece of a large puzzle that also includes ML/DL during the last decade.

Independently of the above, the end of Moore's law has also shifted attention to accelerators, especially FPGAs and GPUs [243, 151, 131, 123, 128, 246, 226, 144, 148, 147], as CPUs do

not have the processing capacity to analyze the data produced everyday on the web. For example, accelerators such as GPUs have excellent capabilities to perform matrix multiplications, the basic building block of almost all ML/DL algorithms. To take advantage of these trends, industry and academia has built a growing number of frameworks to port ML/DL algorithms into accelerators [219, 8, 6, 7, 15]. Although these frameworks were initially built just for ML/DL algorithms, they have been expanded to perform other tasks in the data stack, e.g. data preprocessing [56]. At the same time, we also observe the opposite trend, i.e. more and more research efforts [262, 109, 132, 73] are pushing ML inside databases. We therefore have two opposing trends that influence data processing. Either expanding ML frameworks to perform data management tasks or expanding databases to execute ML workloads with the additional support of accelerators.

These two opposite directions have acted as Procrustes' bed for today's data processing leading to many versions of specialized and monolithic systems. It is not uncommon for a system to dominate the data plane for a couple of years only to be subsequently replaced by another one, only because of its lack of support to newer advances in algorithms and/or hardware. This leads to a lot of repeated development effort that does not conform with the high degree of specialization we observe.

To this end, in the first part of the thesis we explore two different software abstractions acting as core data processing elements. We first introduce the notion of sub-operators, a form of relational operators that are more modular and fine-grained than traditional relational operators. Using these fine-grained operators, we show how a relational query engine with minimal changes can execute full queries on RDMA clusters, serverless functions, and smart storage with performance comparable or even better than mature systems. Subsequently, we use the tensor abstraction as a core component to workloads other than ML/DL where it is traditionally used. More specifically, we run relational and graph algorithms, showing competitive performance both for CPUs and GPUs compared to custom kernels and specialized libraries. With these two approaches, we show that there is not one correct uniform abstraction and/or framework for data processing. All approaches have their advantages and disadvantages and their suitability. However, when picking one software abstraction instead of combining differently purposed systems together, researchers and industry pave the way for more adaptable and efficient data systems, capable of keeping pace with rapid technological advancements both in the software and the hardware landscape.

Besides the aforementioned variables, an additional component shaping data management is I/O overhead. As data keeps growing and the capacity of main memory cannot increase propor-

tionally, new types of memory have emerged, such as disaggregated DRAM [197, 138, 198], disaggregated persistent memory [202, 247], far memory [186, 77] and smart remote memory [244, 173]. A further I/O challenge is the increasing prevalence of semi-structured data (e.g. CSV, JSON, Parquet) especially for cloud data warehouses [65, 61], a trend that has been intensified by modern data science [182, 139, 58, 59].

Therefore, in the second part of the thesis, we investigate modern techniques in hardware and software that try to alleviate the I/O bottleneck. Starting with hardware, we perform an extensive experimental analysis on Intel Optane Persistent Memory, the first public implementation of Non-Volatile Memory (NVM). Our experiments give guidance on best-usage of NVM and show its role in the future memory hierarchy, but also show the shortcomings of the hardware. Then, motivated by the data movement problem, a difficult problem for end users in the cloud, we present a modern form of database cracking, called serverless cracking. Serverless cracking leverages Function-as-a-Service (FaaS) to cut down on cost/latency for Query-as-a-Service (QaaS) when they are querying external semi-structured data. Our approach can cut down on cost by an order of magnitude and we outline future directions on how we envision serverless cracking will be integrated as a cloud module in QaaS.

Thesis statement. Data management has been influenced by many factors during the last years, including data explosion, machine learning workloads, the memory-CPU gap and the hardware evolution. In this dissertation, we show how we can adapt different components of database systems to leverage these trends, bringing databases up to speed with all the recent advancements. We show how modular operators and tensors can both act as a core processing unit for general data processing that adapts to the newest trends. We then investigate hardware and software solutions to the I/O bottleneck, demonstrating that although many steps have been done to alleviate this problem, there is still a long road ahead to be covered.

1.2 Contributions & Structure

The contributions of this thesis to the state-of-the art of heterogeneous data processing are summarized as follows:

In **Chapter 2**, we develop a modular query engine, called *Modularis* that can execute distributed analytics over heterogeneous platforms. More specifically:

- We make a case for modular operators to adapt to the fast-paced landscape of algorithms and hardware.

- We come up with design principles around modular operators, which we call *suboperators* and we design a query engine around them.
- We show how we can use these operators to execute a distributed join.
- With the majority of the operators that we used for the join, we execute sequences of joins, a distributed `GROUP BY` and end-to-end TPC-H queries.
- We compare our engine against mature systems and specialized operators and we show competitive performance for the vast majority of the usecases.

In **Chapter 3**, we investigate if the tensor abstraction is suitable for non DL/ML tasks and more specifically graph and relational algorithms. Concretely:

- We show how we can use Tensor Computing Runtimes (TCRs) to map a popular graph algorithm (PageRank) into a tensor program.
- We also map to the tensor abstraction cardinality estimation over `DISTINCT` queries and relational joins.
- Our implementations outperform custom kernels and specialized libraries by an order of magnitude in some cases.
- We also sketch future directions on how we envision the tensor abstraction can be used as a core component for unified data processing.

In **Chapter 4**, we explore if Intel Optane Persistent Memory (PMEM/NVM) could work as a drop-in replacement for relational databases and key-value stores for both OLAP and OLTP workloads. Precisely:

- We perform microbenchmarks to measure different quantities (throughput, latency) of the hardware without any system integration.
- We use a popular OLAP benchmark (TPC-H) for a variety of relational databases, to understand the behavior of PMEM under this type of workloads.
- We utilize an OLTP benchmark (TPC-C) to get insights on how NVM operates under heavy-write workloads.
- We exploit a key-value store (RocksDB) together with a widely accepted benchmark (YCSB) to comprehend PMEM characteristics' for this type of task.
- Based on all our results, we provide insights and recommendations on how researchers and practicioners can integrate PMEM into the future memory hierarchy and which of its attributes can be relevant for future memory technologies (CXL).

In **Chapter 5**, we propose a new form of database cracking, called serverless cracking, which is adapted to QaaS. More specifically:

- We show how simple transformations (transcoding, chunking, sorting) affect the runtime and cost of QaaS.
- We study the cost/latency tradeoff of these transformations on QaaS.
- We use FaaS to perform these transformations and analyze how time-consuming/expensive they are.
- We make concrete recommendations on how users can transform their input data to take the most out of sparse workloads run on semi-structured data on QaaS.
- We provide future directions on how we envision serverless cracking will be integrated as a module in cloud environments.

Chapter 6 summarizes and shows future directions of the thesis, and how data processing is going to evolve to integrate different platforms and hardware.

Related Publications This dissertation is based on work that has also been presented in the following publications:

- Modularis: modular relational analytics over heterogeneous distributed platforms by Koutsoukos, D., Müller, I., Marroquín, R., Klimovic, A. and Alonso, G., 2021. Proceedings of the VLDB Endowment, 14(13), pp.3308-3321.
- Tensors: An abstraction for general data processing by Koutsoukos, D., Nakandala, S., Karanasos, K., Saur, K., Alonso, G. and Interlandi, M., 2021. Proceedings of the VLDB Endowment, 14(10), pp.1797-1804.
- NVM: Is it Not Very Meaningful for Databases? by Koutsoukos, D., Bhartia, R., Friedman M., Klimovic, A. and Alonso, G., 2023. NVM: Is it Not Very Meaningful for Databases?. Proceedings of the VLDB Endowment, 16(10), pp.2444 - 2457.
- Serverless cracking for QaaS by Koutsoukos, D., Marroquín, R., Müller, I., Klimovic, A. and Alonso, G. (under review)

Apart the above-mentioned publications, participation in other projects has resulted in the following publications:

- The collection virtual machine: an abstraction for multi-frontend multi-backend data analysis Müller, I., Marroquín, R., Koutsoukos, D., Wawrzoniak, M., Akhadov, S. and Alonso, G., 2020, June. In Proceedings of the 16th International Workshop on Data Management on New Hardware (DaMoN) (pp. 1-10).
- Farview: Disaggregated memory with operator off-loading for database engines Korolija, D., Koutsoukos, D., Keeton, K., Taranov, K., Milojicic, D. and Alonso, G., 2022. Conference on Innovative Data Systems Research (CIDR)

- Wu, B., Koutsoukos, D., Alonso, G. 2023. Efficiently Processing Large Relational Joins on GPUs. arXiv preprint arXiv:2312.00720.

Chapter 2

Modular operators for heterogenous platforms

2.1 Motivation

The growing popularity of machine learning applications and the increasing amount of data that analytics applications must process have had a substantial influence on the way systems are designed and optimized. There is a constant stream of specialized accelerators (TPUs, GPUs, FPGAs, smart NICs, smart storage, near memory processing) and platforms (large appliances, InfiniBand clusters, serverless, cloud instances, data centers) that forces a continuous redesign of data processing engines—often leading to new engines—simply to exploit the capabilities of new hardware [100].

Often, to gain performance, developers design monolithic operators that are highly tailored to the underlying hardware [90, 93, 94, 224, 209]. However, as the algorithms and platforms evolve quickly, it becomes very difficult to reuse these operators in newer versions of the system. Examples abound: For instance, a join optimized for multi-core machines [90] requires fundamental changes to run on Remote Direct Memory Access (RDMA) [93, 94] due to the different communication schemes between NUMA nodes and the network. As another example, although FPGAs are not competitive with multi-core machines for full joins, they can significantly accelerate the partitioning phase [166]. Supporting distributed query processing on serverless computing has

received a lot of attention and requires a specialized exchange operator to allow communication through storage [209, 224]. With the current approach of highly engineered, monolithic operators, it is difficult to exploit the potential of new architectures and platforms without major redesigns.

We argue that to cope with the fast changes in the hardware and platform landscape, query processing needs to become more modular and composable at a finer granularity than conventional relational operators. Having many versions of highly optimized, monolithic operators is not a design approach compatible with the high degree of specialization we observe. In almost all cases where hardware or platform advances offer new opportunities, the potential advantages affect only part of an operator (e.g., only one of partitioning, build, or phase of a join) and often require to change other significant parts (e.g., intermediate data placement in an exchange operator, partitioning strategies, etc.). It is very rare that the whole operator can become faster or that all operators benefit. This effect is notable in accelerators (FPGA, GPU) [243, 151, 131, 123, 128, 246, 226, 144, 148, 147], specialized processor components (AVX, SGX), [90, 74, 172, 210, 115, 237] evolving networks in distributed systems (Infiniband, RDMA, smart NICs, etc.) [93, 270, 194, 199, 274, 101, 232], and even platforms (Infiniband clusters [93, 194, 199], cloud/serverless computing [209, 133, 162, 177, 79, 174, 229, 236, 241, 107, 176, 224]). Yet, the current design approaches often require a complete redesign because they are based on careful tailoring to the underlying platform and hardware.

In this chapter, we focus on the modular design of query processing. We show how to design a modular distributed query processing engine with performance comparable to its monolithic counterparts. The topic of modularity in database operators has been visited many times in the past [126, 157] and recently [125, 165, 180]. However, to our knowledge, we are the first to implement modularity at the hardware platform level, while at the same time formulating concrete design principles. We argue that modularity at this level is a necessity rather than a nice-to-have feature. To this end, we have built *Modularis*—an execution engine aiming to maximize performance without specializing neither the engine nor the bulk of the operators to the target platform. Modularis is based on a collection of composable sub-operators that are both as small and simple as possible, as well as reusable, while retaining the ability to execute entire SQL queries. Modularis' sub-operators share the same goal of composability of operators present in traditional database engines: sub-operators can be freely and easily combined, have a well-defined interface, and adhere to a common execution model. Our sub-operators are similar to the microcode used in processor design to implement more complex instructions. We use them to build up complex plans like TPC-H queries. This allows Modularis to run seamlessly on

three different platforms: an InfiniBand RDMA cluster, a serverless cloud service, and a smart storage engine, by simply replacing in the query plan only those operators actually affected by the change in the platform (e.g., the exchange operator), leaving everything else unaffected.

We have evaluated Modularis' performance and behavior extensively. First, to explore the end-to-end performance of Modularis, we compare it to mature systems using the TPC-H benchmark. When compared to Presto, a system that is general enough to utilize various storage layers and distributed set-ups, Modularis is an order of magnitude faster. When compared to SingleStore (previously called MemSQL), a system that specializes in in-memory analytics using SQL, Modularis is faster for the majority of the queries. The speed-up compared to both of these systems comes from the ability to optimize at the sub-operator level and the usage of RDMA for fast data transfer. Next, we show how Modularis adapts to heterogeneous environments by discussing the minimal changes necessary to go from running on an RDMA cluster to a serverless platform using AWS Lambda or a smart storage engine (S3Select) —a very significant architectural change in the underlying platform. For TPC-H queries, Modularis-on-serverless is competitive against commercial Query-as-a-Service systems (Athena, BigQuery). This last experiment shows that modularity does not need to result in end-to-end performance losses while it enables the execution of workloads on two fundamentally different platforms with minimal development effort, which mainly involves the design of new exchange and executor operators. That way, we get the best of both worlds: maximum performance across platforms, without redesigning the whole system from scratch.

Second, we quantify the potential performance overhead of the modular design by comparing a join composed from several Modularis' sub-operators with a hand-tuned join. For the latter, we use the best monolithic implementations available for RDMA clusters [93, 94]. Modularis is always within 30 % of the performance of the specialized implementation (and often closer). Nevertheless, our system uses 3.8 × fewer lines of code than the monolithic operator and all its sub-operators are not specific to the join but can be reused in other query plans.

Third, we demonstrate the advantages of sub-operators over monolithic approaches when it comes to extending existing operators. We show how to use the same sub-operators that we used for the join for optimizations for sequences of joins and a distributed `GROUP BY` (with one additional sub-operator). In contrast, extending existing manually handcrafted joins [93, 94, 199] to support, e.g., inner, outer, semi, and anti joins plus grouping for partitioned, sorted, and general inputs would be very difficult (and has not been attempted, to our knowledge). In Modularis, once we had the sub-operators for the initial distributed radix hash join, we needed only a small effort to develop the rest of the operators. These results put into perspective the potential per-

formance loss when comparing Modularis, which can run code over different platforms, with handcrafted algorithms tailored to run on a single target system.

2.2 Background

2.2.1 RDMA

Broadly, RDMA allows using the network protocol (InfiniBand [1], RoCE [2]) and the Network Interface Card (NIC) to move data directly from the memory of the sender machine to the memory of the receiving machine. This is accomplished by using DMA features that allow the NIC to read and write from and to the host memory directly over the PCI bus (almost) without involving neither the CPU nor the OS.

For this chapter, we focus on one-sided operations that do not require any involvement of the receiving CPU. One-sided operations have Remote Memory Access (RMA) as their basis. RMA supports access to remote memory regions through one-sided read and write operations. There is an initial set-up procedure where the target memory region (buffer) is reserved, pinned, and registered with the NIC. The registration of the *memory region* to be used involves pinning a portion of memory to avoid other processes or the OS to interact with it. Memory or buffer registration has been identified as a bottleneck when using RDMA for data processing [134], an aspect that our experiments also confirm. After this initialization process, the sending side just puts or gets the data into or from the memory of the remote host through a one-sided write or read operation.

2.2.2 Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a widely used standard for writing parallel, High-Performance Computing (HPC) applications. MPI provides platform-independent communication primitives. Different MPI implementations such as foMPI [140], OpenMPI [4], and MVAPICH [3] are customized for particular platforms and they select the most applicable communication approach among processes.

The user can define the parallelism of an MPI job by specifying the total number of *processes* to be used when dispatching it. Each of these processes is assigned a *rank* as a unique identifier for communication purposes. The dispatch system, *mpirun*, initializes all processes that will run

the same code on the machines assigned to the job. To perform any one-sided RDMA operation each process has to allocate a contiguous section of main memory, called *memory window*. This is accomplished using a collective call named `MPI_Win_create`. This function is implemented as a collective call such that all processes execute this instruction, even if one of them does not need to register memory itself.

After the creation of the window and before the program executes any other operation on it, the user must synchronize the processes with the `MPI_Win_fence` function. The synchronization ensures that all incoming and outgoing RMA operations will complete at that process before the fence call returns. The time mediating between two fence function calls is called an *RMA epoch*. Fence synchronization is also a collective call.

To read and write from a remote window the user should use the `MPI_Get` and `MPI_Put` functions. To ensure that all the pending RMA operations that the calling process starts on the target window are complete without releasing the lock, MPI has the `MPI_Win_flush` function only in passive target synchronization. After this call, the program can reuse or read all the buffers used by previous `MPI_Get` and `MPI_Put` function calls.

2.2.3 Experimental setup

We run all of the RDMA experiments of this chapter using all available cores from a cluster of 8 machines (specifications in Table 2.1). Regarding the MPI implementation, we use OpenMPI 3.1.4 as opposed to foMPI [140] used by [94] because foMPI is specific to Cray machines. Finally, unless otherwise mentioned, we run each experiment five times and report the average among such runs.

2.3 Modular operators

2.3.1 Modularis architecture

In this section, we describe the architecture of Modularis and how it executes a query. We show the system architecture in Figure 2.1. The user writes queries in a UDF-based library interface written in Python (similar to PySpark). When the user submits the query, they specify the target execution platform using a flag (e.g. ,`--rdma`, `--lambda`). The query is then parsed and translated into an IR, representing a DAG of operators. The DAG is serialized and passed to

Component	Specs
CPUs	2 × Intel Xeon E5–2609 2.40 GHz
Cores/Threads	2 × 4/4
RAM	128 GB
L1 Cache	2 × 4 × 64 KB
L2 Cache	2 × 4 × 256 KB
L3 Cache	2 × 10 MB
InfiniBand	Mellanox QDR HCA

Table 2.1: RDMA cluster specification

Modularis' backend, written in C++, which applies a series of both plan-specific and platform-specific transformations.

The plan-specific transformations involve projection and selection push-downs, operations typical in DBMSs to reduce data movement and I/O. Next, Modularis cuts the DAG into tree-shaped sub-plans, each of which represents a pipeline with a materialization point at its end. Then, we transform the query into its distributed equivalent. This step involves wrapping the initial plan into a distributed executor and adding exchange operators in the plan inputs. Depending on the target platform, we specialize the generic operators with hardware-specific ones. In the case of serverless, we use the exchange operator of Lambada [209] and an executor that spawns the workers in a tree-plan fashion. In the case of smart storage, we use a specialized operator to

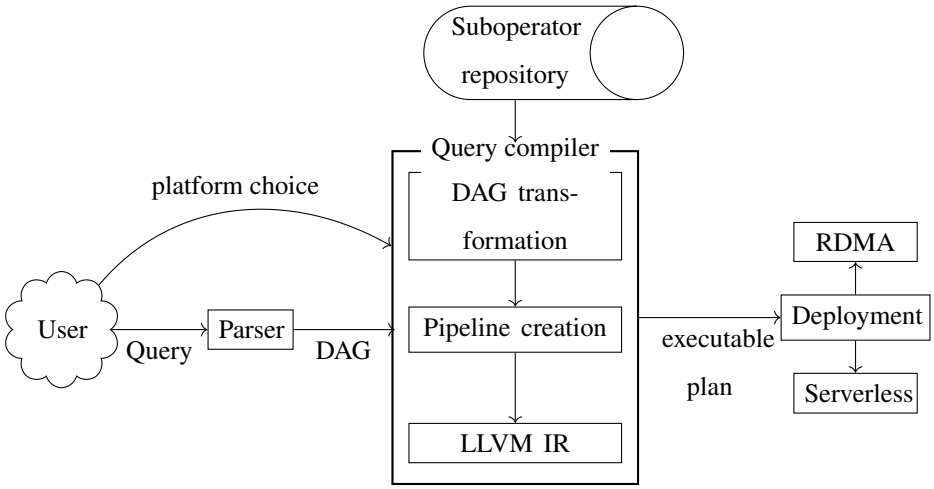


Figure 2.1: System architecture

get data from S3Select. In the case of RDMA, we use an MPI executor and an RDMA-based exchange operator based on the one of Barthels et. al [94] (with modifications to avoid unnecessary data movement). We give concrete examples of the final plan in Sections 2.4.4 and 2.4.5. Finally, the plan is lowered into LLVM IR and Just-in-Time compiled to native machine code. To translate UDFs into LLVM IR, Modularis uses Numba [52] and inlines the generated LLVM code into the remainder of the plan to eliminate any function calls or interpretation in inner loops. The query is then executed on the target platform, based on the specialized operator that it has been wrapped around. When the results are produced, they are returned back to the user.

2.3.2 Design principles

Modularis builds on the observation that the commonly used operators, e.g., for high-speed networks [93] or multi-core CPUs [90], are built around the same *conceptual* building blocks: when the authors describe their algorithms, they use some visual or textual representation of "reading data", "partition by key", "for each partition", etc. To readers, these terms imply the same operation is being done during different phases of query execution. However, there is no code reuse—the implementations differ in intricacies related to how and where data is stored, how it is passed from one phase to the next, how they depend on the state of some enclosing scope, etc.

The goal of Modularis is to identify pieces of code that reoccur in operators in slight variations, factor out their common logic, and package them behind a well-defined interface such that they can be reused and recomposed. In other words, our goal is to derive *actual* building blocks from the conceptual ones. To derive sub-operators systematically, we follow these design principles:

1. *Each sub-operator consists of or is a part of at most one inner loop.* If a high-level algorithm consists of several phases, each of these phases is expressed by at least one sub-operator. Phases often reoccur across and within monolithic operators, sometimes in slight variations. For example, join operators typically use partitioning to improve cache locality. In Section 2.4.3, we show that by factoring out the partitioning logic into a dedicated sub-operator, we can reuse it to improve cache locality in grouping operators as well.
2. *Use dedicated sub-operators for each physical (in-memory) data materialization format.* This decouples the processing of data from where and how it is stored. Consequently, other sub-operators become independent of the physical formats of their inputs and outputs and are more generic. One high-level example is to have different scan sub-operators for reading base tables or intermediate materializations in RDMA buffers. That way, a single partitioning sub-operator im-

plementation can consume inputs of two different scan operators (or any other operator) instead of having two specialized partitioning operators (see Section 2.4.1.2).
3. *Express high-level control flow as (nested) operators.* This allows connecting plan fragments of sub-operators that express the heavy-lifting data processing through the same operator interface. In monolithic operators, such orchestration logic is usually implemented as imperative code specific to that operator and, thus, it makes it necessary to reimplement the data path as imperative code as well. One high-level example is to express the in-memory join of two partition pairs occurring in a classical partitioned hash join as a nested query plan that is executed for each pair of matching partitions. That allows the use of partition-unaware sub-operators in the inner plan. We introduce the `NestedMap` sub-operator for this purpose below (Section 2.4.1.2).

Generality of design principles. The above somewhat different implementations do not only apply to the context of CPUs, but also to other hardware architectures (e.g. FPGAs, GPUs). Therefore, the goal of the design principles is still applicable, as we can use them to derive the basic building blocks that the underlying architectures use. By following our design principles, we do not expect to design operators that work solely in different architectures without any modifications. We rather want to avoid reimplementing the same conceptual building blocks with small modifications. Our goal is to end up with a very similar set of operators for different architectures. For example, an operator that reads data in columnar format is relevant to both CPUs and FPGAs. By having a similar set of building blocks for different architectures, we can offload parts of a query to different hardware configurations seamlessly, just by swapping out the hardware-specific part of the plan. We give a concrete example of how this can be done in the context of smart storage in Section 2.4.5.

2.3.3 The sub-operator interface

We base the interface of sub-operators on one of the most known models in the database community, the Volcano model [145]. The model is based on iterators that pass records along the data path of a tree of `Next()` function calls. Like in a traditional execution engine, the iterator interface allows us to combine operators in almost arbitrary ways, limited only by the schema or types that operators may require.

The main distinctive feature of the sub-operator interface compared to traditional Volcano-style operators is the type system of the records (or "tuples") passed between them. While records in relations (in the First Normal Form) consist of atomic fields, we need a more expressive type system for a generic physical execution layer. For example, to split the materialize and scan

operators of a given physical data format into two distinct sub-operators, these operators need to pass "records" containing the materialization from one to the other. Similarly, if we want to express operators that work on individual records as well as those that work on batches (or morsels [191]) in the same interface, we need to be able to represent the concept of a "batch". We thus extend the concept of tuples with that of "collections", which is the generalization of any physical data format of tuples of a particular type. This allows expressing physical execution properties into the query plan rather than hard-coding them for the entire execution engine.

More formally, sub-operators are iterators over *tuples* and the tuples are of a statically known type from the following recursive type structure:

$$\begin{aligned} \mathit{tuple} &:= \langle \mathit{item}, \ldots, \mathit{item} \rangle \\ \mathit{item} &:= \{\, \mathit{atom} \,|\, \mathit{collection} \text{ of } \mathit{tuples} \,\}, \end{aligned}$$

where a *tuple* is a mapping from a domain of (static) field identifiers to item types, an *item* is a (statically known) atomic or collection-based type, an *atom* is a particular domain of undividable values, and a *collection* is the generalization of any physical data format one might want to use in the execution layer. We denote tuple types by $\langle \mathit{fieldName0} : \mathit{ItemType0}, \ldots, \mathit{fieldNameK} : \mathit{ItemTypeK} \rangle$ and collection types by $\mathit{CollectionType}\langle \mathit{TupleType} \rangle$. Most operators are *generic* in the sense that they require their upstream operator(s) to produce tuples of a type with a particular structure but accept any type of that structure. Their output type usually depends on the type(s) of their upstream(s). For example, the scan operator for a C-array of C-structs (which we call *RowVector*) requires from their upstreams to produce tuples of type $\mathit{RowVector}\langle \mathit{TupleType} \rangle$ and returns tuples of *TupleType*, where *TupleType* is allowed to be any tuple type. Similarly, operators consuming or producing batches can do that by consuming or producing tuples with *RowVector* fields.

We also extend the Volcano-style execution model to DAGs, whereas operators in the original Volcano-style execution model could only have one consumer (i.e., plans had to be trees). Before execution, we cut a DAG of operators into pipelines, where pipelines start with either the original plan inputs or the result of any operator with several consumers. In each pipeline, that result will be read only once, so the sub-plan of the pipeline is a tree and can thus be executed with the iterator model. Pipelines materialize their results, such that multiple downstream pipelines can read them. For simplicity, we present the plans as DAGs in the remainder of the chapter and omit the pipelines and materialization points.

2.3.4 Set of sub-operators

In this section, we introduce the set of sub-operators that we use in Modularis. We choose the sub-operators such that they are expressive enough to support all the major relational operations, e.g. selections, projections, aggregations, and joins. At the same time, we follow our design principles. While the design principle of making operators simple aims indeed at increasing code re-usage and reducing implementation effort, this does not necessarily mean that there is no redundancy. In fact, we do add new operators if they provide a sufficiently large and broad performance benefit. Such an example is the different join implementations (inner, semi, anti). They could all be based on the same hash-build and a separate probe operator. However, having a special operator for each of them provides better performance. Similarly, having a second version for each of them with flipped build and probe sides increases performance further. Finally, while the set of sub-operators is enough for relational analytics, we expect that to expand Modularis to more types of analytics (e.g. linear algebra, ML), we need to expand our sub-operator set.

We present our list of sub-operators in Table 2.2. The sub-operators fall into six categories: orchestration operators, data processing operators, MPI-specific operators (to support RDMA clusters), Lambda-specific operators (to support serverless), Smart storage-specific operators, and generic materialize and scan operators. Orchestration operators enable the execution of nested computations. Data processing operators express the computations carried out on the data inside the inner loops. MPI- and Lambda-specific operators are the ones that are aware of the distributed nature of query execution. Smart storage-specific operators use pushdown computations to smart storage. Finally, materialize and scan operators read and write *tuples* from and to nested *collections*. We give an overview of our operators in Table 2.2. We believe that the semantics of most operators are clear given their names. However, the two orchestration operators merit an explanation, as they differ substantially from other systems.

The `ParameterLookup` operator encapsulates plan inputs in the operator interface, such that other operators can consume them. This operator is the only operator aware of plan inputs. The `NestedMap` operator executes a nested plan independently on each input tuple, which typically contains a nested collection. Each invocation of the nested plan produces an output tuple that may contain nested collections as well. This allows us to process nested collections using the same building blocks regardless of the nesting level. This operator consumes tuples of any type, and the `ParameterLookup` operator(s) in the nested plan return a tuple of that type.

Category	Operators
Orchestration operators	Parameter Lookup, NestedMap
Data processing operators	(Parametrized) Map, Projection, Cartesian Product, Filter, Reduce (By Key), GroupBy, Zip, Local Histogram, Build and Probe, Partition, Semi-join, Sort, Top-K
MPI-specific operators	MPI Executor, MPI Histogram, MPI Exchange
Lambda-specific operators	Lambda Executor, Lambda Exchange
Smart storage-specific operators	S3Select Scan
Materialize and scan operators	Local Partitioning (AVX-based), Partition, Row Scan, Column Scan, Parquet Scan, Materialize Row Vector, Arrow table to collection

Table 2.2: Initial set of sub-operators

2.4 From operators to complex query plans

2.4.1 High-performance distributed join

We illustrate how Modularis' sub-operators can express optimized monolithic operators with a case study of a state-of-the-art distributed join algorithm proposed by Barthels et al. [94].

2.4.1.1 State-of-the-art distributed join

We start with a very brief summary of the algorithm as it was originally proposed (Figure 2.2). The algorithm consists of three phases: (1) histogram computation, (2) multi-pass partitioning including network transfer, and (3) hash table build and probe. The two phases where communication happens amongst processes, namely the histogram calculation and the network partitioning phase, are depicted using black boxes around them. The original algorithm is optimized for a workload involving two relations where both relations consist of 16-byte tuples (8 bytes for the key and another 8 for the payload). For more details, we refer the readers to the original paper [94].

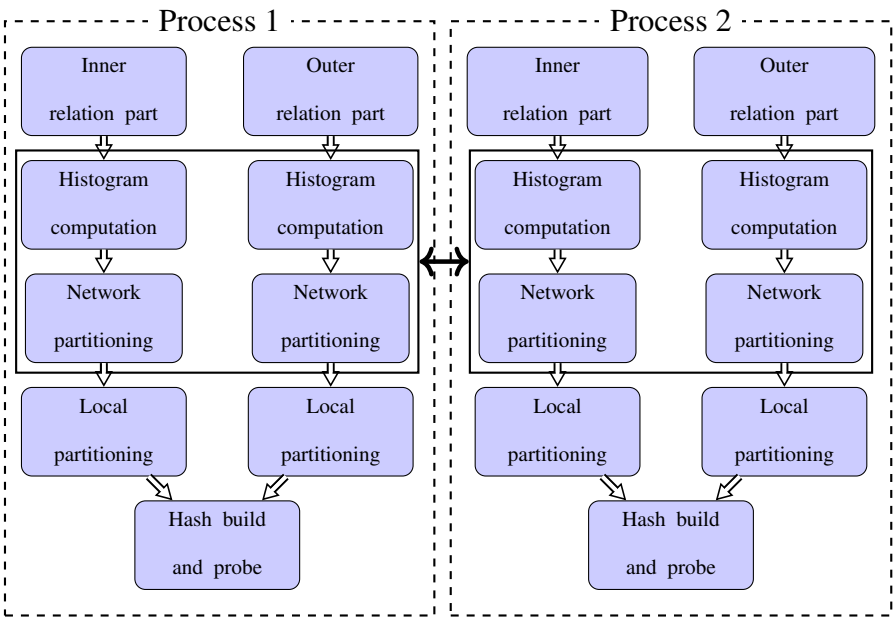


Figure 2.2: RDMA-aware hash join algorithm for two processes proposed by [94]

2.4.1.2 Query plan in Modularis

We now show how the same join algorithm can be expressed using sub-operators in our RDMA backend. The resulting query plan is shown in Figure 2.3. To keep the graphical representation concise, we abbreviate the operator names as shown in Table 2.3. Furthermore, we omit materialization points and, instead, express the plan as a DAG as discussed previously. Finally, most of the operators in the Figure are part of a nested plan inside a `MpiExecutor` operator, which is executed concurrently by all MPI processes (which we call *ranks* in the rest of the section) the cluster in a data-parallel way, illustrated with a stacked frame.

The join starts by computing the histogram of each of the two inputs using the `LocalHistogram` operator. The inputs can be produced by any operator producing tuples with key fields, e.g., a scan operator reading from a base table stored in main memory. On each of the two sides, a

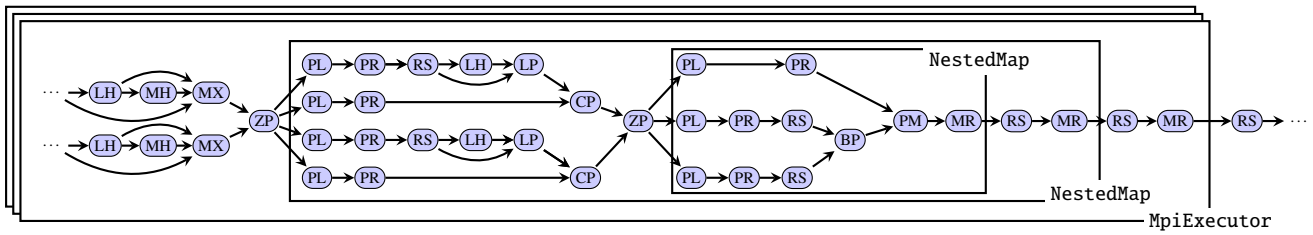


Figure 2.3: Plan that runs the distributed hash join with modular operators across many nodes

`MpiHistogram` computes the global histogram from the local ones until the `MpiExchange` consumes the local histograms, the global histogram, and the original input. With this information, each rank allocates a contiguous memory area in the main memory of the host (called *RMA window*) that will hold all tuples it will receive in this phase and computes the offsets of its exclusive regions inside the windows of the target ranks. Then, each rank reads the input again, computes the target partition for each input tuple the same way as it did for computing the histogram, and writes the tuple into a buffer corresponding to that partition. This partitioning routine is based on well-known techniques using streaming stores and software write-combining [227, 237, 210] to achieve the full memory bandwidth. When the buffer of a partition is full, it is sent to the target rank using an asynchronous RDMA write operation, it replaces the buffer with an empty one, and continues partitioning the input immediately. This overlaps computation with communication and increases performance.

In the network partitioning phase, as in the original algorithm, each rank compresses the 16-byte workload of the algorithm into 8 bytes to reduce the data transmitted by a factor of two. This

Abbreviation	Operator name	SLOC
PL	Parameter lookup	28
NM	Nested map	49
PR	Projection	27
BP	Hash build and probe	103
LH	Local histogram	77
ZP	Zip	44
CP	Cartesian product	54
PM	Parametrized map	51
RK	Reduce by key	75
RS	Row Scan	59
LP	Local partitioning	143
MR	Materialize row vector	56
ME	MPI Executor	140
MX	MPI Exchange	269
MH	MPI Histogram	52

Table 2.3: Source line of code per operator

optimization comes as an additional pass to our query compiler, and although it is very specific, it is useful for dictionary-encoded data. The compression uses the fact that some bits of the key are common for each partition. Specifically, if we use the identity hash function and radix partitioning with a fan-out of 2^F, the first F bits of each partition are identical. Furthermore, we assume that keys and values come from a dense domain and can be represented with P bits each. Thus, key and value can be stored in a single 64-bit word if $2 \cdot P - F \leq 64$. After the partitioning and compression, the operator returns the partitions as $\langle$*networkPartitionID*, *partitionData*$\rangle$ pairs such that all tuples of each partition end up on only one rank. Because we extend the original algorithm with materialization of the input tuples, we recover the missing bits by forwarding the *networkPartitionID* further downstream. As we can observe, the `MPIExchange` operator batches the tuples in order to avoid the overhead of sending a tuple-at-a-time over the network.

The subsequent plan joins the tuples inside two corresponding partitions of the two sides. An imperative implementation would express this as a loop over matching partition pairs. In Modularis, we use the `NestedMap` operator for the same purpose: We take the corresponding $\langle$*networkPartitionID*, *partitionData*$\rangle$ pairs and pass them through a `Zip` operator, which produces $\langle$*networkPartitionID*, *partitionData*, *networkPartitionID*, *partitionData*$\rangle$ tuples (note that they are produced in dense, ordered sequence). This way, all data belonging to one partition pair is represented in a single tuple, and we can express the remaining logic as a nested plan transforming each such tuple. The nested plan starts by dissecting the input tuple. The tuple has four fields: the partition ID and data of the two sides, respectively. A sequence of `ParameterLookup` (which returns the entire tuple) and `Projection` operators (which retains one of the fields) extracts one of the fields, each. The partition data is partitioned further on both sides by a sequence of `RowScan` (which extracts individual tuples from the nested collection inside the *partitionData* fields), `LocalHistogram`, and `LocalPartitioning` operators. Note that each of these sequences returns several $\langle$*localPartitionID*, *partitionData*$\rangle$ pairs. To be able to recover the dropped bits further downstream, we augment each of these pairs with the *networkPartitionID* by using the `CartesianProduct` operator. Its left side only consists of a single tuple (containing the network partition ID), so it does not increase the number of tuples.

The hash and probe phase happens inside another nested plan, which is executed for each pair of sub-partitions. As before, we use a `Zip` operator to combine all information of each pair of partitions into a single tuple, on which we call a nested plan using `NestedMap`. We use sequences of `ParameterLookup` and `Projection` operators to extract the partitions of the two sides. Each partition is read by a `RowScan` operator and individual tuples produced by these two are finally fed into the `BuildProbe` operator, which produces the matching pairs. To recover

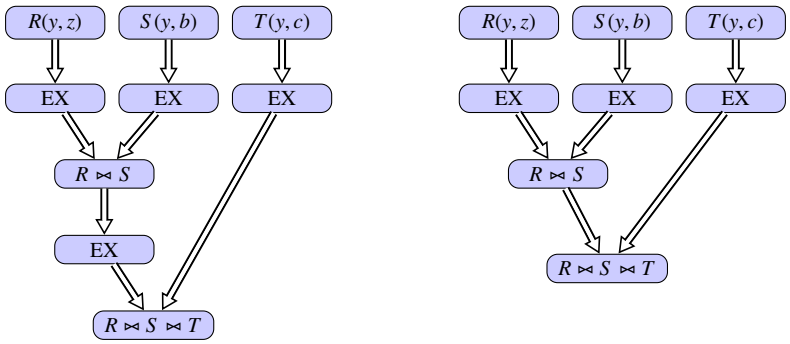


Figure 2.4: Naive (left) and optimized (right) versions for a sequence of two joins on the same attribute

the dropped bits from the network phase, we use a `ParametrizedMap` operator: It contains a function that, given a parameter from upstream (the network partition ID), shifts that parameter by a certain amount and adds the result to the key field of each input tuple from the other upstream.

The remainder of the plan depends on what happens with the join output. The Figure shows a plan that materializes that result. Since each `NestedMap` needs to return a single tuple, the result of each nested plan needs to be materialized using a `MaterializeRowVector` operator. This operator produces a single tuple containing its input tuples as a nested *RowVector*. Each `NestedMap` thus returns several such tuples (one for each input tuples) and the inner tuples can be recovered as a flat stream using `RowScan` operators.

In the above description, we reuse many of our building blocks to construct the high-performance distributed join. Also, we showcase many of our design principles in action, such as the dedicated read/write operators to read data either from RMA windows or local partitions, how the high-level control functions work, and finally how we reuse operators across different phases of the algorithm.

2.4.2 Sequences of joins

A key advantage of Modularis is that, once we have the original join algorithm, it is straightforward to extend the plan to run sequences of joins. None of the work to date on high-performance joins on multi-core CPUs or over RDMA has ever addressed this design due to the complexity of modifying the highly tuned operators. For a cascade of N joins, the output of the $n-1$-th join is joined with the n-th relation, where $n = 1, \ldots, N$. Therefore, in the original plan of Figure 2.3, after the `RowScan` operator, we return the new data to the `LocalHistogram` and `MpiExchange`

operators. On the other side, another upstream operator returns tuples that go through the network partitioning phase. This pattern is repeated on one side of the corresponding join for each output and on the other side for the corresponding new relation, until all of the N joins are performed.

However, if all the joins are on the same attribute (i.e., the attribute y in Figure 2.4) and the relations fit in main memory, we can apply the following optimization: We network-partition all relations at the beginning instead of reshuffling the output of every join through the network. This is possible since we execute the output of the first join again on the attribute y, and therefore we can pre-partition all the relations from the beginning of the query instead of waiting for the result of each join . This way, for a cascade of N joins, we shuffle through the network $N + 1$ instead of $2N$ relations. We show our optimization for a sequence of two joins in Figure 2.4, where instead of shuffling four relations through the network, namely R, S, T, and the output of $R \bowtie S$, we shuffle only R, S, and T. For conciseness, we depict with the operator `EX` the chain of a LocalHistogram, MPI Histogram, and MPI Exchange operators.

This optimization is easily applied because of the operator modularity. In the case of monolithic operators, a system engineer would have to take special care of this case by adapting a large part of the system and possibly by reimplementing parts of the algorithm. In contrast, Modularis restructures the sub-operators inside the query plans and takes advantage of the common attribute in a sequence of joins. After it performs all the network partitioning phases at the beginning of the inner plan of Figure 2.3, it carries out all the local partitioning phases in the first nested map. Finally, it forms a sequence of `BuildProbe` operators where the output of the $n-1$-th `BuildProbe` is the input of the n-th `BuildProbe` and the final build probe output is the input of the `ParametrizedMap` operator.

2.4.3 Distributed `GROUP BY`

To illustrate how Modularis simplifies operator development and provides extensibility, we implement a distributed `GROUP BY` operator by re-using components from the previous use cases.

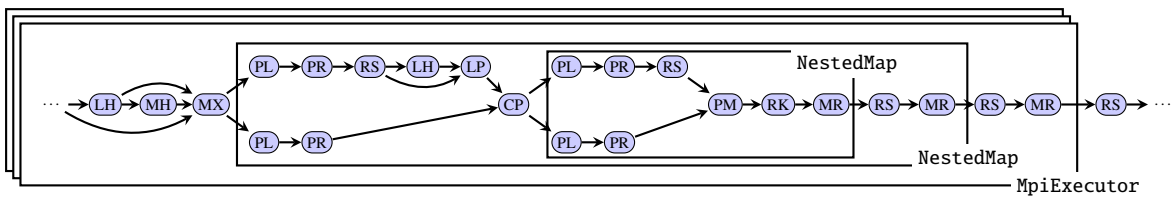


Figure 2.5: Plan that runs the distributed `GROUP BY` with modular operators across many nodes

We show the corresponding plan in Figure 2.5. The algorithm workload is a 16-byte tuple (8 bytes for the key and 8 bytes for the value).

The plan starts with any upstream operator that returns tuples to the `LocalHistogram` and `MpiExchange` operators. Since we have multiple consumers from one operator, these tuples have to be materialized and put into a separate pipeline (the materialization is not shown in the Figure as discussed earlier). After the local and global histogram calculations, the tuples are partitioned and distributed through the network. The operator performs a similar compression scheme as in Section 2.4.1.2. As before, this compression allows us to reduce the network traffic in half, which is crucial for performance. Every output tuple (which consists of $\langle networkPartitionID, partitionData \rangle$ pairs) coming from the `MpiExchange` operator is the input of a `NestedMap`, which executes its nested plan for every input partition.

The execution of the nested plan starts with the `ParameterLookup` operators. The tuples returned by these operators are passed to `Projection` operators, which ensure that each downstream operator gets the correct input. Specifically, the network partitioning data are passed to a `RowScan` operator that returns a tuple at a time to the `LocalHistogram` and `LocalPartitioning` operators. After the histogram calculation, the `LocalPartitioning` consumes both the input data and the calculated histogram to calculate the necessary prefixes inside a partition. It then performs the data partitioning. The corresponding partitioned data is concatenated with the network bits that the `MpiExchange` removed, using a `CartesianProduct`. Lastly, the $\langle networkPartitionID, localPartitionID, partitionData \rangle$ triples are the input to `NestedMap` operator, which executes the final aggregation for each input partition.

To perform the final aggregation, we first have to restore the original keys as we did in the join algorithm after the hash build and probe phase. The difference now is that we have to restore the full keys using the `ParametrizedMap` operator before we forward each tuple to a `ReduceByKey` operator, which aggregates the data per local partition. Afterward, we materialize the output tuples of the `ReduceByKey` operator with a `MaterializeRowVector` operator. Like in the distributed hash join case, we finish the plan with the `RowScan` operators that remove nesting levels of the `MaterializeRowVector` operators that should end every nested plan. Finally, the individual results from the workers return to the driver.

Based on the previous description, it is evident that the distributed `GROUP BY` plan is very similar to the distributed hash join plan. The main differences are 1) the total number of input relations and 2) that for distributed `GROUP BY` operator we do not perform a hash build and probe phase in the end but an aggregation using a `ReduceByKey` operator. The large overlap between the two plans shows how Modularis uses a similar set of sub-operators to implement different relational

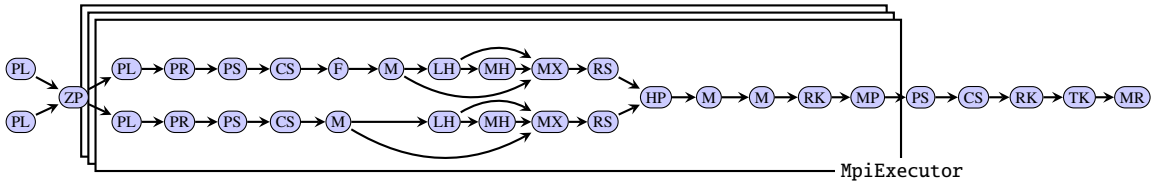


Figure 2.6: Modularis plan for TPC-H Q12 on RDMA

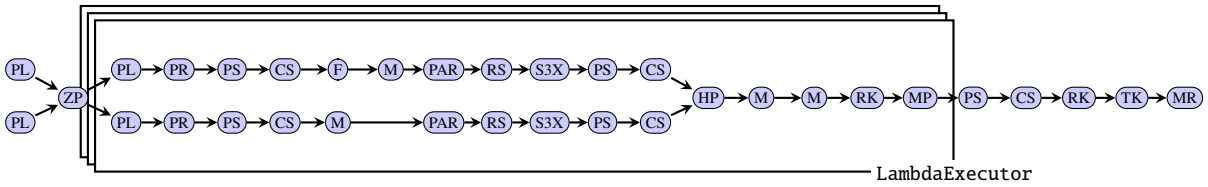


Figure 2.7: Modularis plan for TPC-H Q12 on Serverless

database operators in contrast to using monolithic operators, where the different operators would probably have to be reimplemented almost from scratch, although the logic behind shares many similarities (e.g., the partitioning phases).

2.4.4 TPC-H queries

So far we have used Modularis to implement key components of a relational query processor. We can use the same sub-operators to implement TPC-H queries. In fact, by altering only our MPI-specific sub-operators to Lambda-specific we can execute the same set of TPC-H queries on a different hardware platform. The latter is a strong argument towards modularity, as systems that operate on different hardware platforms have fundamentally different execution layers making it almost impossible to apply the techniques used in one for the other without major reimplementation.

We take as an example TPC-H Q12 and show (simplified) query plans of Modularis in Figures 2.6 and 2.7 for RDMA and serverless respectively. Although the plans are simplified, they still preserve the semantics of the system. Since the network bandwidth in serverless is rather slow (around 80 Mbit/s, see [209]), we use the partitioning only as a pre-processing step required by the exchange operator and do not partition the exchanged data further. This means that each worker processes only one data partition after the exchange, so we do not have any `NestedMap` operators in that platform. Both plans take Parquet file paths to the base tables as input, either in S3 or NFS depending on the platform. The paths of the left and right input are zipped and each resulting pair is used as the input for a nested plan instance by the executor of the respective platform. The execution of a nested plan starts again with `ParameterLookup` operators, from

which we project the paths for the left and right relations, pass them to `ParquetScan` operators, and finally extract individual tuples from the column chunks produced by that operator using the `ColumnScan` operator. The next operators express the corresponding operations of Query 12.

At this point, we differentiate the plans with the operators that constitute the exchange routine for each platform. Note that we do not perform any compression of the tuples, such as in the case of the distributed join and the `GROUP BY`. In the case of RDMA, the plan looks similar to the ones that we have presented before. In the case of serverless, we use the exchange algorithm of Lambada [209]: First, we partition the data into a sequence of partitions using a `Partition` operator. Subsequently, the `GroupBy` operator takes the `<pid, data>` partitions and groups them by `pid`. Then the `RowScan` operator reads each partition and forwards it to the `S3Exchange` operator, which writes the data into a file on S3 whose file name is based on the ID of the sender containing one row group per receiver. The `S3Exchange` then returns triples of worker-specific S3 paths and the first and last row group to read from that file, which is then read by the subsequent `ParquetScan` operator. This pattern implements the "write combining" optimization of Lambada, which significantly reduces the number of write requests to S3. Finally, the column chunks produced by the `ParquetScan` operator are consumed by a `ColumnScan`, which extracts individual tuples.

After the exchange phase finishes, the plans converge again: We perform the join of the two relations using the `HashProbe` operator. The matched tuples are transformed with a series of `Map` and `ReduceByKey` operators to get the final result, and we use a `MaterializeParquet` operator to return the individual results of the workers to the driver process. We read the individual results using a sequence of `ParquetScan` and `ColumnScan` operators. Finally, we merge the results of the workers using a `ReduceByKey` operator, we select the first tuple using a `TopK` operator and we return the results to the user with a `MaterializeRowVector` operator.

We therefore show how altering only a handful of operators that are hardware-specific, allows us to run the same TPC-H query on two very different hardware platforms. That way, implementation effort is reduced vastly. Using the same ideas, we could extend the TPC-H implementation to use an exchange operator based on TCP. The addition of more backends only requires changing the executor and the operators that comprise the network exchange phase.

2.4.5 Integration with smart Storage

To show how Modularis can use a smart storage component offered by a major cloud provider, we integrate S3Select [53] into our system. S3Select is a smart storage engine offered by Ama-

zon that follows the trend of pushing computation into storage [258, 161] to overcome the I/O bottleneck. S3Select pushes computations directly into S3 and thus it pulls only the data that the user needs from these objects.

S3Select takes as parameters an SQL query, the S3 input path, and an output serialization format (either JSON or CSV). In our case, the user writes an SQL expression in the frontend, which pushes selections and projections to S3Select. We have created an additional sub-operator called `S3SelectScan` that our query compiler decomposes into three simpler, more re-usable separate operators. The first sub-operator performs an API call to S3Select and requests the data, which S3Select returns in CSV format. The sub-operator then uses Apache Arrow to convert the CSV to an Arrow Table and forwards the table downstream. The next operator converts the Arrow Table to a *collection of tuples* as this is defined in Section 2.3.3. Finally, the third operator is a `ColumnScan` that gets this collection and returns the individual tuples. Then, we continue with the execution of the rest of the plan in our serverless backend.

Thus, by following the same design principles, we only develop the sub-operators needed to integrate S3Select into Modularis. We did not have to redesign the system from scratch to adapt to this system component. We only had to make small adjustments, whereas other systems would have to make a major redesign to incorporate such a change. Finally, this integration shows the generality of our design principles, because the implementation done is not limited to S3Select. We could use the same architecture to request data from a different smart storage engine that is based on an accelerator (like Amazon AQUA [50]). As databases push more computation to storage or to accelerators, we expect that systems like Modularis will be able to use these components with only minimal changes.

2.5 Modularis evaluation

In this section, we first compare Modularis to commercial systems on TPC-H queries on two hardware platforms: RDMA and serverless. We then analyze the performance of the system against a distributed hash join. Finally, we show the runtime of a distributed `GROUP BY` operator and variations of plans for sequences of joins.

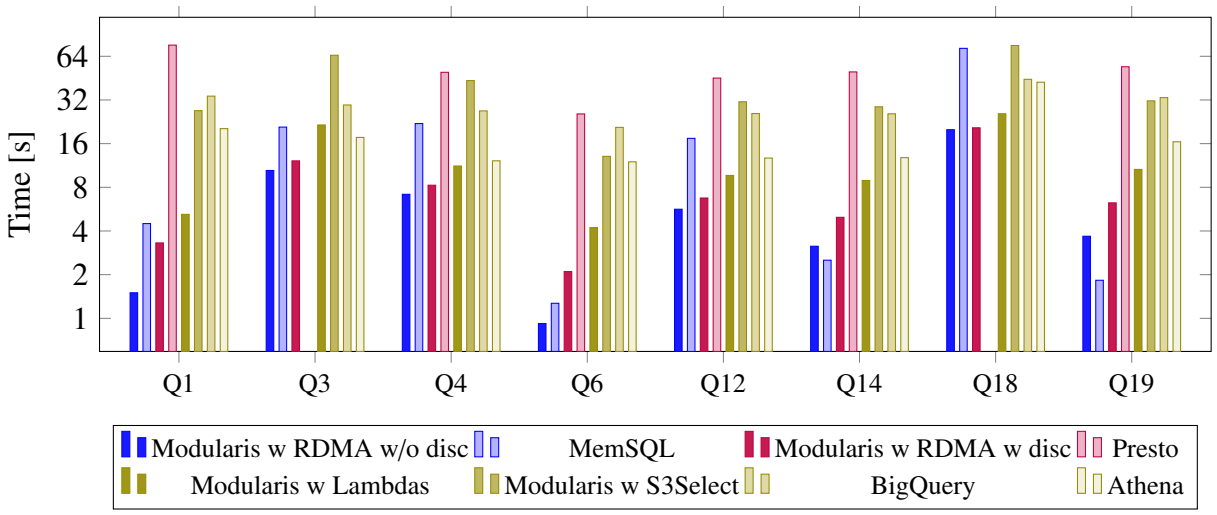


Figure 2.8: TPC-H queries runtime using SF-500

2.5.1 TPC-H queries

We start our evaluation by comparing Modularis to commercial systems, both for RDMA and serverless using TPC-H queries. We present the results for scale factor 500 in Figure 2.8. We pick TPC-H Queries 1, 3, 4, 6, 12, 14, 18, and 19, which is more than a third of the benchmark and a representative subset of all the challenges that the benchmark poses. More specifically, Q1 has a large aggregation, Q3 has a large join, Q4 involves an inclusion test, Q6, Q14, and Q19 have selective filtering that largely reduces the processed tuples of the input tables, Q12 involves almost all of the major operators a relational engine should implement (join, selection, projection, aggregation, sorting), and Q18 has a high-cardinality aggregation. As mentioned, the challenges involved in these queries together make up for the most important challenges of the benchmark itself and an execution layer that has a good performance on these queries has a high probability to perform well on the whole benchmark [169, 104]. The inclusion of more queries is not a limitation of the system but involves the implementation of a more sophisticated optimizer, which is part of future work and out of the scope of this chapter; for now, we concentrate on the execution layer alone.

2.5.1.1 RDMA

For the RDMA backend, we compare Modularis against two different systems, Trino (previously Presto) and SingleStore (previously MemSQL). Both of these systems do not use RDMA for the network exchange but we configure them to use the InfiniBand network for data transfer

at higher rates. We verify this by monitoring the network traffic. For all the RDMA experiments, we run Modularis using 64 workers. For SingleStore, we use version 7.0.12 and deploy a master aggregator node and 7 leaf nodes. We do a warm run for each of the queries and report averages of 5 runs. We deploy the TPC-H database using both a row-store [55] and a column-store [54] format. The row-store format is completely in-memory and has very good random seek performance. The disk-backed column-store format has optimizations such as indices, compression, fast aggregations, and table scans, most of which are not supported by Modularis. We use the column-store format, as it is the best for all the queries. Although the column-store format is disk-backed, SingleStore serves all queries from an in-memory cache; for a fair comparison, we thus exclude the time Modularis needs to read the data from disk as well. We observe that Modularis is between 30% up to more than 3x faster for all the queries except Q14 and Q19.

For Q19, Modularis is slower because the histogram-based RDMA network exchange is slower than a broadcast operator for small joins. Modularis is 40% slower in Q14, because the `Map` operator does not perform the selective `LIKE` as fast as SingleStore. For the queries where Modularis is faster, all except Queries 1 and 6 have large joins, where the RDMA network exchange has better performance. Another operator responsible for a large part of the overall runtime is `ReduceByKey`, which uses a highly optimized version of a parallel hash map in Modularis and thus executes very fast large aggregations. The difference is obvious in Queries 1 and 18 where Modularis is 3 and 3.5 times faster, respectively.

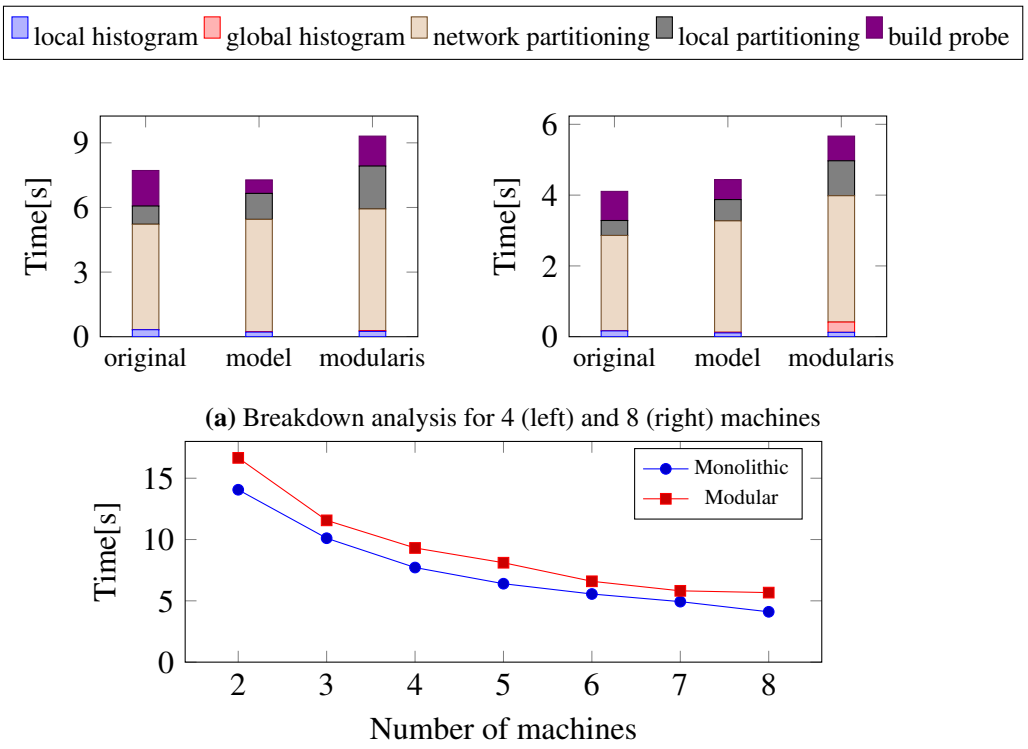
For Trino, we deploy Trino SQL version 327 along with HDFS (Hadoop version 2.6.0) in the eight-machine cluster using one node exclusively as coordinator and NameNode. We configure HDFS to use replication factor 3 and Trino to use as much memory as possible. We run each query four times, use the first run as a warm-up and then report the average of the other runs. To have a fair comparison, we include also the time that Modularis needs to read the input data from disk. For Queries 3 and 18, Trino cannot execute the queries and fails because of insufficient resources. Our system is 6-9x faster than Trino, depending on the query, partly because of the optimized RDMA network exchange and partly because of the highly-performance sub-operators.

2.5.1.2 Serverless

For the serverless backend, we use the two Query-as-a-Service systems Athena and BigQuery as baselines. These systems run queries over cloud storage without the need to start or maintain any infrastructure. For BigQuery, we use external tables [66] that point to 512 Parquet files per

relation stored on a single-region bucket in Google Cloud Storage in the standard storage tier. Creating such a table is just a metadata operation and takes < 1s, i.e., no data is loaded. Instead, the query runs against the original files on cloud storage. We use the same files for Athena, for which we created an external table [60] as well. The remaining configuration is done by the cloud provider. For Modularis we use the same Parquet files. For the integration with S3Select, we use 512 workers and for the experiments with the `ParquetScanOperator` we use 256 workers. The workers for both these configurations have 2 GiB of main memory each. For all three systems we run each query 6 times and report averages. As we observe, Modularis with S3Select is comparable to BigQuery for the majority of the queries but slower than Athena. To investigate the reason, we isolate the calls to S3Select and time them independently of Modularis. We find out that they make up for the majority of the runtime (e.g. for Q1, 24 out of 27 seconds are spent getting data from S3Select). The problem is enlarged when we read many relations (e.g. Q3, Q18) and is almost negligible for high selective queries that read only one relation (e.g. Q6). In fact, for Q6 Modularis with S3Select is very close to Athena and faster than BigQuery. The larger running times of calls to S3Select are because the service returns chunks of uncompressed CSV data, whereas our `ParquetScan` reads data in compressed format and also pushes down projections. Our observations agree with the ones made in [266]. This problem is ameliorated if we read only one Parquet file per worker, which is why we use 512 workers in this experiment.

We can improve the performance significantly by using the specialized `ParquetScan` operator. In that case, Modularis outperforms both baselines in all queries except for Query 3, where Athena is marginally faster thanks to a better query plan. For Queries 1 and 6, the running time difference is due to the `ParquetScan` operator, which is optimized to push down projections while reading data in compressed format. For the other queries, the specialized `LambdaExchange` operator can run workloads involving data exchange between workers faster than the commercial baselines. This is evident because the workers are mostly bounded by network bandwidth and latency. The current performance of S3Select illustrates the advantage of modularity: today, we can use the much faster alternative of our optimized `ParquetScan`, but using `S3SelectScan` can be enabled easily if AWS improves the performance of their smart storage engine in the future.



(a) Breakdown analysis for 4 (left) and 8 (right) machines

(b) Comparison across machines

Figure 2.9: Modularis distributed join execution time per phase and compared to Monolithic design

2.5.2 State-of-the-art distributed join

2.5.2.1 Implementation effort comparison

Before we delve into a performance comparison of the execution of the distributed hash join algorithm between Modularis and the original codebase, we measure the implementation effort between the two approaches by using the number of lines of code of each of them. Although this metric is not always reliable, most of the time it gives a good indication of the implementation effort. The operators that are used in the plan according to Table 2.3 sum up to 1152 lines of code while the original implementation adds to 1754 lines of code, leading to a 35% reduction. One can argue that this can be attributed to coding style but the main take-away from this comparison is not only the size reduction but the extensibility of our sub-operators. While to support other join types (e.g. semi-joins, anti-joins) we only need to modify the `HashProbe` operator that consists of 103 lines, the original codebase has to be replicated for every join variant. Furthermore, the only operators that are platform-specific used are: `MpiExecutor`, `MpiHistogram`, and `MpiExchange`, which sum up to 461 lines of code. In contrast, the original monolithic code

would have to be rewritten from scratch if we wanted to change the target platform, involving 3.8 × more code.

2.5.2.2 Performance comparison

We compare the distributed hash join code by Barthels et al. [93], which consists of a monolithic operator, against our equivalent Modularis plan. For a fair comparison, we extend the original code base with a similar materialization operation to our `MaterializeRowVector` operator. The workload consists of two relations with 2048 million tuples each. Unless otherwise mentioned, we use a 1-to-1 correspondence between the keys in the inner and outer relation. This setup is consistent with the workload used in the original paper [93] in the scale-out experiment (shown in Figure 7(a) in [93]).

We present our results in Figure 2.9. To understand how our system performs, we also microbenchmark our sub-operators. These microbenchmarks show the model performance that Modularis' components can achieve. We compare the total runtime of these microbenchmarks (referred to as *model*) against the entire Modularis query plan and the original code base. We present our results for two machine configurations in Figure 2.9a. We start our analysis by comparing the three execution times phase-by-phase.

Starting with the *local histogram phase*, we observe that, compared to the original code, both the model and the whole query plan have a small speedup. We attribute this speedup to the fact that, as we mention in Section 2.4.1.2, the local histogram calculation is isolated in a small pipeline because its input has to be consumed by multiple readers. This allows for compiler optimizations (e.g. automatic usage of SIMD instructions, function inlining) that remove our sub-operators abstractions. These optimizations are not possible in larger pipelines, because in larger pipelines the compiler cannot inline all the `next()` functions effectively.

The *global histogram phase* has almost the same execution time in our model and the original code. However, in the full query plan and especially when the join is executed on more machines, the total time is significantly larger. This is associated with the `MPI_Allreduce` function that calculates the histogram, which is a collective operation that requires data from all the processes. In case a process is stalled in a previous phase of the algorithm, then every other process must wait until it has the required data from it. In the original algorithm, this phenomenon is not present because the histograms are calculated sequentially for both relations. This also holds for the model. On the other hand, during the execution of the join in Modularis, the global histogram calculation happens in two distinct phases, one for each upstream path and a network

partitioning phase is between the two global histogram phases. Because the network partitioning phase has a slight variation in its execution time, it causes tail latencies for some processes during the calculation of the global histogram of the second relation.

The *network partitioning phase* is slower in the model and the query plan than the original code base for two reasons. First, this operator is part of a large pipeline in our generated code and therefore, as mentioned before, the compiler cannot perform all the possible optimizations and remove all of our abstractions. To validate this assumption and find the cause of the slowdown, we run the following benchmark: We generate 1 billion integers and record the time that `RowScan` needs to read them and compute their sum, compared to a simple C++ program that does the same. `RowScan` needs about 1 second, whereas the C++ program needs around 0.8 seconds. The second reason for the slowdown is due to tail latencies because, as before, the window allocation/synchronization function calls are collective operations. In the original code base, they are called almost at the same time for both relations but, in Modularis, they happen at different times, one for each upstream path.

The *local partitioning phase* is faster in the original code base than in the model and the query plan. Part of the slowdown can be again explained due to the complex pipelines that this phase belongs to, which causes a comparative slowdown in the `RowScan` operator. This effect is more eminent in the query plan because there the pipeline is even bigger than the one in our microbenchmarks. Another cause of the slowdown is that in the query plan, we cannot exactly isolate the local partitioning phase, but we instead measure the whole sub-plan present in the first `NestedMap` of Figure 2.3 and subsequently subtract the one present in the second `NestedMap`. This nested plan includes an extra materialization of the output partitions, and the processing of metadata necessary for later phases of the algorithm. Although the latter is not significantly compute-heavy, it attributes to a small part of the slowdown present.

The *build probe phase* is faster in the model, compared to the original code base and the query plan. The slowdown in the first case is explained by the fact that the `MaterializeRowVector` uses the `realloc` function to request more memory, compared to the allocator interface of the original code base. In the second case, build probe is again part of a large pipeline. Therefore we lack some compiler optimizations. On 8 machines, each process materializes fewer tuples and requests less memory, and these effects are ameliorated.

Finally, in Figure 2.9b, we depict the total runtime of the monolithic operator compared to the Modularis plan for the distributed join algorithm. Our modular integration is from 12 to 28% slower, depending on the number of machines used, due to the reasons explained before. However, the operators present in this plan exhibit the advantage that they can be reused in a

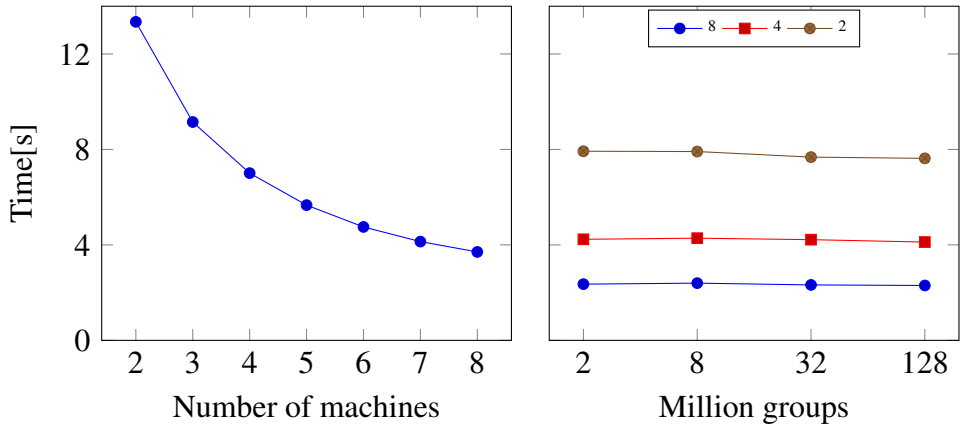


Figure 2.10: Distributed `GROUP BY` runtime: varying cluster size with fixed key cardinality (left); varying key cardinality for different cluster sizes (right)

variety of queries.

2.5.3 Distributed `GROUP BY`

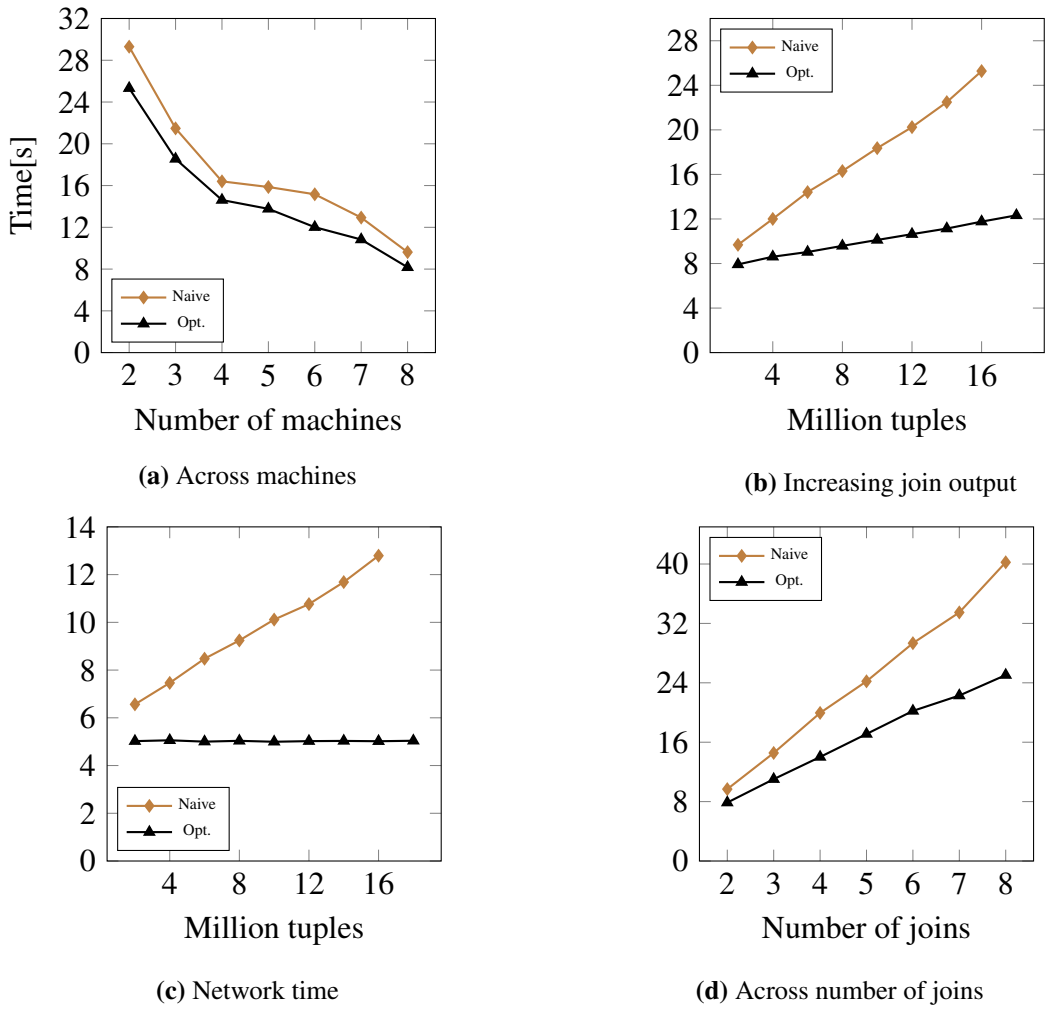
In this section, we run the distributed `GROUP BY` plan presented in Section 2.4.3. We show our results in Figure 2.10. On the left side of the Figure, we run the plan across different machine configurations for a workload of 2048 unique million keys. As expected, the total runtime decreases as the algorithm load is distributed across more nodes. On the right side of the Figure, we increase the number of distinct keys in the input (and hence the number of groups in the result) and execute the plan for three different machine configurations. Because the total execution time is dominated by the network time and the materialization of the tuples, the time is almost steady for each machine configuration. Overall, we observe that Modularis performs grouping of billion of elements in a few seconds without having to design a specialized monolithic operator for this cause but mainly by reusing existing system components.

2.5.4 Sequences of joins

Finally, we present the results of executing sequences of joins. As a baseline, we use the naive version of the plan that shuffles four relations through the network. We compare the naive plan against an optimized one that shuffles only three. Both of these versions are implemented in our system. For this experiment, we use multiple relations with 2048 million tuples each, similar to

the relations used in [93]. Additionally, we use a 1-to-1 correspondence between the keys in the inner and outer relation unless otherwise mentioned.

Figure 2.11a shows the execution time when performing a sequence of two joins across several machines with the two variants of the algorithm, *naive* and *optimized*. We observe that there is a constant speedup in the optimized version compared to the baseline, which is partly due to the network shuffling of one less relation and partly due to the materialization of only the final result instead of materializing both the intermediate and the final join output. The execution time has a sublinear speedup as the number of machines increases because the tail latencies mentioned in the previous section in the network phases are even more eminent. Because the network phases constitute a larger part of the execution time of the optimized version, these tail latencies are the



(a) Across machines

(b) Increasing join output

(c) Network time

(d) Across number of joins

Figure 2.11: Sequences of joins in Modularis

main reason that the speedup between the baseline and the optimized version is decreased as the number of machines increases.

However, the advantage of the optimized version is more conspicuous when the first join has an increasing join output. We show the total runtime of such an experiment across 8 machines in Figure 2.11b and the time spent on partitioning data through the network in Figure 2.11c. While the naive version increases linearly with a high rate as the algorithm materializes and shuffles through the network an extra relation that has an increasing size, the optimized version has a sublinear increase in its total execution time. To analyze further the cause of this time difference, we show in Figure 2.11c that for the optimized version the time spent on shuffling data through the network is constant, as all three relations are pre-partitioned at the beginning of the plan execution while the network time is increasing linearly as more data are shuffled through the network. In these two plots, we cannot execute the baseline algorithm for more than 18 million tuples due to memory constraints.

Lastly, we compare the two versions while increasing the number of joins performed. We show the results in Figure 2.11d. The difference in the total runtime between the two versions is proportional to the number of joins because for N joins, the optimized plan performs $N - 1$ materializations and $N + 1$ network shuffling phases less than the naive plan. This shows, as in the `GROUP BY` case, that we can apply optimizations that have a significant performance impact. This comes without rewriting the whole system from scratch, as in the case of monolithic operators, but mainly by restructuring operators across plans.

2.6 Related Work

Database operators design. Operator modularity is a crucial design decision as it significantly affects performance. Determining the right granularity of operators has been a reoccurring topic of research: from the bracket model [124] for parallelization in the early days of databases to objected-oriented modular designs [126], record-oriented components adapted at runtime [157], morsel-driven parallelism [191], and "deep" query optimization [125] more recently. In Modularis, we use the Volcano model [145] as the basis for the interfaces between operators, but we also support collections instead of just flat records. Modularis shares a similar vision to that of Dittrich and Nix [125, 51], Bandle and Giceva [91], and Kohn et al. [180] in having operators defined at a finer granularity. Dittrich and Nix argue that this enables a deeper level of query optimization and easy implementation of research ideas without sacrificing performance. Bandle

and Giceva analyze how sub-operators can be used for general data analytics. Both these works sketch a vision for modular operator design, and they focus on a few variations of aggregation using different indexes (sorting/hashing) or on how to build more complex algorithms (e.g hash joins, k-means) out of sub-operators. In contrast, Modularis is a full system comprising a variety of operators and runs full TPC-H queries using plans tailored to three very different platforms. Kohn et al. [180] develop a framework that uses modularity to speed up query execution when the query contains multiple aggregates. Modularis follows a similar spirit but builds a more generic execution layer that achieves a similar goal for query execution in general. In contrast to these approaches, we tackle the problem at the platform level and formulate design principles for these operators. We also do not focus on the optimization aspect from the view of query rewriting. Our goal is to reduce the implementation effort and keep up with the fast pace with which the hardware evolves, without having to rewrite systems completely. Finally, efforts seeking to optimize sets of operators are orthogonal to Modularis as they could be applied to the query plan generated by Modularis as additional optimization passes. For instance, Leeka et al. [190] fuse similar operators into a single *super-operator* by using a streaming interface.

Emerging technologies in data processing. Two of the most recent emerging technologies in distributed data processing are RDMA and serverless computing. On the one hand, RDMA has been used to implement highly distributed relational operators [199, 93, 94] and several projects have explored the network bottleneck and the use of RDMA for query processing and database design in a variety of contexts [96, 232, 235, 100, 194]. On the other hand, serverless computing has been extensively studied over the last few years for a variety of applications [133, 162, 177, 176, 79, 174, 229, 236, 241, 107, 224]. In both cases, research focused on solving platform-specific challenges and was often carried out in built-from-scratch prototypes that were limited in functionality. In Modularis, by leveraging the granularity of the sub-operators, we develop specialized platform-specific operators that leverage the latest technologies of both platforms but keep most of the other operators and the rest of the system hardware-agnostic. This shows that we can achieve competitive performance without having to redesign the entire system.

Query compilation for data processing. Modularis is also related in part to systems that perform Just-in-Time (JiT) query compilation as pioneered in HyPer [213]. Similar techniques were later used and extended in several other systems: Tupleware [117], which targets machine learning workloads; Weld [218], which incorporates a large class of data analytics algorithms by translating frontend languages into an intermediate representation (IR) that is later Just-in-Time compiled; LegoBase [178], which builds a database using a high-level language; and Flare [129**?**], which combines data processing tasks with machine learning. Modularis uses JiT compilation

techniques similar to those in such systems to eliminate the potential performance overhead of a modular design. The systems above either have an IR tailored towards single-machine execution and rely on systems such as Spark [269] for their distributed setup (e.g. Weld) or, if they have a distributed setting (e.g. Tupleware), they use a very generic model that is very hard to be adapted in case of platform changes (e.g. from VMs to serverless functions). Modularis on the other hand, targets directly distributed analytics and focuses on tailoring execution to the underlying platforms by only using minimal changes to the plans.

2.7 Conclusion

In this chapter, we propose Modularis—an execution engine based on sub-operators. After sketching the design principles that the sub-operators should follow, we propose an initial set of sub-operators and demonstrate how they can be combined to build traditional database operators such as a distributed hash join or a `GROUP BY` query as well as more complex query plans like sequences of joins and TPC-H queries. We show that by changing a small subset of our sub-operators we can execute the same TPC-H query plans on diverse hardware platforms (RDMA, serverless clusters, smart storage). Through extensive experiments, we show that modularity reduces implementation effort without requiring users to sacrifice performance. Modularis is an order of magnitude faster than Presto, a data warehouse supporting various storage layers and distributed setups, and more performant in the majority of cases than SingleStore, an in-memory analytics SQL engine. Modularis also outperforms Query-as-a-Service systems such as BigQuery and Athena by simply changing the RDMA-specific operators with dedicated serverless-based ones so that queries run on the cloud.

Chapter 3

Tensor Computing Runtimes for graph and database workloads

3.1 Motivation

Applications such as large-scale data analytics and machine learning (ML) are driving an unprecedented demand for computing capacity. ML model training alone is responsible for a 10× yearly increase in demand for computing capacity [102]. Informal data suggest that it costs as much as several millions of dollars to train a complex model from scratch [31]. This huge compute demand has been partially met by distributed computing, leveraging data center and cloud infrastructures to speed up data processing [8, 238].

To further boost processing efficiency, but to also reduce the number of machines required [206], specialized hardware and hardware acceleration are gaining momentum. The former refers to hardware tailored to the task at hand (typically tensor computations), while the latter to the use of standard components such as GPUs or FPGAs to replace CPUs, as their architecture is more suitable to tensor computations. However, specialized hardware increases the complexity for the programmer and often requires careful optimizations to improve performance. Thus, in parallel to these developments, a significant effort and investment is being made on software frameworks such as PyTorch [219], Tensorflow [8], or TVM [112] that simplify the development, management, deployment, and optimization of Deep Learning (DL) models. In such tools, DL models

are created as a composition of tensor operations. This *tensor abstraction* is the basis for automatic optimizations [175, 160, 33, 272, 113, 250] and enables the compilation of the same code over heterogeneous hardware, e.g., CPUs, GPUs, TPUs [164], IPUs [18], FPGAs [208, 29], etc.

The investments in Tensor Computing Runtimes (TCRs) is likely to continue and specialized hardware tailored to tensor computations will evolve accordingly. It thus makes sense to consider whether TCRs can be used to support computations other than just DL—this would allow additional classes of computation to benefit from the continuous advances in TCRs. The challenge behind the vision of making tensors a general purpose abstraction for data processing lies in the tensor abstraction itself: the close relation between tensors and DL algorithms is what has made these frameworks so efficient. But is this abstraction, which is so far dedicated to DL, a good match for other use cases too?

Recent work has mapped traditional ML algorithms (e.g., decision trees) to the tensor model [211]. The resulting system, HUMMINGBIRD, is the starting point for what we present in this chapter: to investigate whether such mappings exist and can be beneficial for conventional data processing tasks as well. In particular, we explore how to map selected graph processing and relational operator algorithms to tensor computations. We use PyTorch and TVM to compile these implementations into code executable on CPUs and GPUs. Despite leveraging a high-level abstraction and using the same source code regardless of the underlying hardware, the resulting machine code often outperforms custom-built C++ and CUDA kernels. These initial results are promising and support the idea that the tensor abstraction and existing TCRs can be a good fit for data processing tasks well beyond ML.

3.2 Background

3.2.1 The Tensor Abstraction

DL models are a family of ML models based on artificial neurons [143]. Frameworks used to implement DL models include TensorFlow [8], PyTorch [219], CNTK [6], MXNet [7], and ONNX [22]. DL models are expressed in terms of *operations over tensors*. DL frameworks such as TensorFlow and PyTorch provide hundreds of tensor operators. A new class of compilers for tensor programs has recently emerged. Systems like Halide [230], XLA [33] and TVM [112] employ the tensor abstraction to generate highly optimized code targeting heterogeneous hardware. While such frameworks and compilers have been mostly used in the computer vision

and DL domains, in this chapter we explore their potential to serve as generic compilers and runtimes for hardware accelerators. In the remainder of the chapter we refer to compilers (e.g., TVM) and DL frameworks as Tensor Computing Runtimes (TCRs).

3.2.2 Tensor Operators

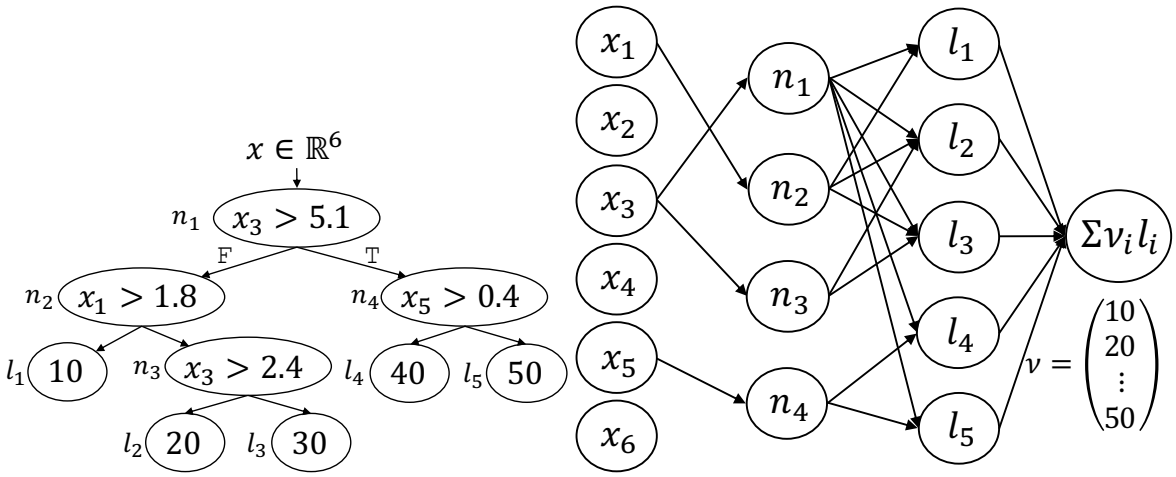
TCRs offer a rich set of operators over tensors. To help less familiar readers, we provide a quick overview of the operators commonly found in TCRs and the categories that they belong to (for the sake of explanation, we will use the PyTorch API here and in the following sections, but similar operators can be found in other TCRs).

Transformation operators. This category involves operations for selecting one or more elements of a tensor (using the square bracket notation), slicing a row/column (`slice`, `index_select`), concatenating two tensors (`cat`), sorting (`sort`), and reorganizing the elements of a tensor (`gather, scatter`).
Reduction operators. This category contains operations for calculating aggregates such as `sum` or `mean`, as well as the `size` of a tensor. It also includes more complex operations like `scatter_add` and histograms (`bincount, histc`).
Arithmetic operators. These operators implement basic arithmetic operations between tensors or between scalars and tensors (`add, mul, div, sub`), matrix-vector (`mv`) or matrix-matrix (`mm`) multiplications, and cross product (`cross`).
Logical operators. Finally, logical operations contain element-wise comparisons between tensors (=, >, <), as well as operations such as `where` implementing conditional statements.

3.2.3 Hummingbird: Traditional ML to Tensors

The first step in motivating the work outlined in this chapter was HUMMINGBIRD [211], which enables inference of traditional ML algorithms (i.e., non-DL ML algorithms, such as linear regression, decision tree, one-hot encoding) on TCRs. Unlike DL, traditional ML algorithms and featurizers do not follow a common pattern as the tensor abstraction. Instead, they are often implemented using imperative programming in Python (e.g., scikit-learn [222]), .NET (ML.NET [75]) or Java (eg., H2O [5]). Expressing them efficiently using tensor computations is not trivial. We briefly explain the key intuition behind HUMMINGBIRD with an example [211].

Decision tree inference involves traversing a tree by comparing input features to the inner nodes to define the path from the root to a leaf node. This are *sparse* computations as only a small part



(a) Decision tree. **(b)** Neural Network conversion.

Figure 3.1: Translation of a decision tree (left) to an equivalent Neural Network (right) using HUMMINGBIRD.

of the tree is activated each time. For example, the decision tree of Figure 3.1a is composed of four inner nodes over three input features. Starting from the root node n_1, feature x_3 is compared with 5.1, and based on the result, either the left (n_2) or the right (n_4) child are selected. The process continues until a leaf node is reached.

HUMMINGBIRD supports three different strategies for translating a decision tree to tensor computations, and it picks the most promising one using a mix of heuristics and structural information of the input tree. Figure 3.1b depicts one of these strategies, whereby the decision tree of Figure 3.1a is translated into a neural network composed of two hidden layers: the first containing all the internal nodes, and the second containing all the leaf nodes. At inference time, the input features are multiplied with an adjacency matrix that encodes which feature is used by each internal node. The output is then compared against the thresholds, thereby returning the active internal nodes. Finally, another matrix multiplication and comparison are used to check which (unique) leaf node is active and to return the final leaf value.

A characteristic of the translation is that *all* internal and leaf nodes are evaluated at inference time. In other words, the computation becomes *dense*. Despite the *redundancy* introduced, the approach is implemented using two matrix multiplication operations that can be efficiently executed by TCRs both on CPU (e.g., using BLAS libraries such as MKL [19]) and GPU. This strategy is able to deliver state-of-the-art performance when the input trees are not too deep. For example, in [211] we show up to 3× better performance than the XGBoost [111] custom implementation both on CPU and GPU, as well as better performance than RAPIDS FIL [9] just

by using PyTorch, without having to implement any of the custom operations that these systems implement on CPU and GPU.

3.2.4 Hardware Setup

For the experiments in this chapter we use an Azure NC6 v2 machine with 112 GB of RAM, an Intel Xeon CPU E5-2690 v4 @ 2.6GHz (6 virtual cores), and an NVIDIA P100 GPU. The machine runs Ubuntu 18.04 with PyTorch 1.7.0, TVM 0.8-dev, and CUDA 10.2. We run TVM with `opt_level` 3. TVM models are compiled automatically from the PyTorch models. We report averages of 5 runs.

3.3 Graphs to tensors

Existing efforts such as GraphBLAS [17] have already studied how to express graph algorithms using linear algebra. These approaches rely on the fact that graphs can be represented using adjacency or incidence matrices. Therefore, one could use such linear-algebra implementations to directly execute graph algorithms over TCRs. However, such implementations turn out to be suboptimal when executed over TCRs, because they rely on sparse representations of the graph, while TCRs are not efficient for sparse computations. Hence, novel implementations are required.

As an example, below we show how to efficiently execute PageRank [217], one of the most common graph algorithms, over TCRs. Several other graph algorithms (e.g., affinity propagation, all-pairs shortest paths, connected components) perform operations similar to PageRank, i.e., iterations and aggregations over sparse graphs. Therefore, we believe that TCRs can also be used for such algorithms, yielding similar performance benefits. Additionally, one can also express graph algorithms using more traditional approaches, such as BFS and DFS, over TCRs. Although the necessary data structures (e.g. queues, stacks) may not exist directly in TCRs, they can be implemented using TCRs' available operations.

3.3.1 PageRank

PageRank is used to rank web pages or, in general, influential nodes on a graph. It is based on the idea that the important nodes are more likely to receive links from other nodes. The algorithm

receives as inputs a matrix M with dimensions $N \times N$, where N is the number of nodes that need to be ranked. Each entry $M_{(i,j)}$ contains a number, specifying a probability transition from j to i, such that for all $j, \sum_i M_{(i,j)} = 1$. A damping factor d represents the probability that a person will continue clicking on links. The output of the algorithm is a vector v with dimensions $N \times 1$, which contains the rank v_i of each page. The vector v is normalized and its entries sum up to 1.

The algorithm consists of two initialization steps and a loop that runs for a specific number of iterations n_{iter} or until the ranks v in two consecutive iterations change less than a predefined threshold. In the initialization steps the vector v is filled with random values and a matrix $\hat{M}$ is calculated using the following equation for each element: $\hat{M}_{(i,j)} = d \cdot M_{(i,j)} + \frac{1}{N} \cdot (1 - d)$. The core step of the algorithm is to multiply the matrix $\hat{M}$ with the vector v and assign the result to v for the next iteration.

3.3.2 Tensor Implementation

Implementing the algorithm on the tensor abstraction may look straightforward. Since its core is a matrix-vector multiplication, an operation that is heavily used and optimized in TCRs, PageRank should run efficiently on them. However, web or social graphs are extremely sparse and the number of connections is significantly smaller than the quadratic number of connections that a full graph would have. Therefore, by naïvely implementing the algorithm over dense tensors, we cannot execute PageRank even for a graph with $100K$ nodes (orders of magnitude smaller than real world examples) because it will require the allocation of terabytes of main memory. A first solution to this problem is to use the sparse modules that TCRs frameworks offer. Nevertheless, as the majority of calculations in DL models involves dense operations, these modules are highly experimental and not optimized. This is confirmed by our experimental results in Section 3.3.3.

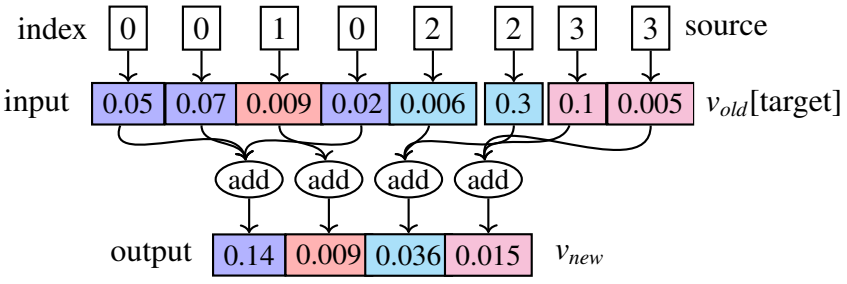


Figure 3.2: Schematic representation of one iteration of PageRank using `scatter_add` (before normalization). The source nodes are used to index the values contained in v_{old}[target]. The resulting sums are stored into v_{new}.

In this algorithm, the matrix-vector multiplication is the bottleneck. To see why, let's assume that $d = 1$ and therefore $\hat{M} = M$. We also assume that M is an adjacency matrix, i.e. it contains only 0s and 1s. Without these assumptions, $\hat{M}$ is an affine transformation of our simplified matrix and we will re-introduce them after we have explained how the algorithm works. Note that M cannot be represented explicitly for very large inputs because it is a sparse matrix. Therefore we represent M as two 1-dimensional tensors which contain the *source* and *target* nodes for every edge.

Since M contains only 0s and 1s, its multiplication with vector v consists of the following steps: (1) for every node i *gather* the values of v for the target nodes to which i is connected (that's where $M(i, *)$=1); (2) *sum* them and store the result in a copy of v that contains the ranks of the next iteration, as follows:

$$v_{new}\text{[i]} \mathrel{+}= v_{old}\text{[M[i, *] != 0]}$$

The steps of gathering indices and adding values together is implemented in TCRs as a single `scatter_add(dim, Index, Input)` operator, where `Input` is the input tensor (v_{old}[target] in our case) and `Index` contains the indexes to gather (source). We have dim=0, since our input tensors (source and v) are 1-d. Briefly, the `scatter_add` operator calculates the following quantity for 1-dimensional tensors:

$$\texttt{output.scatter\_add(0, Index, Input)} \rightarrow \texttt{output[index[i]] += input[i]}$$

We also show `scatter_add` schematically for a single PageRank iteration in Figure 3.2. Since we used two 1-d tensors that contain the source and target nodes for every edge, and the $\hat{M}$ contains an affine transformation of the adjacency matrix, the final form of the function is:

$$v_{new}\texttt{.scatter\_add(0, source, } d\cdot\ v_{old}\texttt{[target]} + \tfrac{1}{N} \cdot (1 - d))$$

To keep the semantics of the algorithm intact and because M is a probability matrix where for all j, $\sum_i M_{(i,j)} = 1$, we have to normalize v_{new} at the end of every iteration by dividing each entry with the number of ingoing connections. Although the scatter-gather approach is well-known in distributed frameworks [187], to the best of our knowledge we are the first to apply it to PageRank with the tensor abstraction.

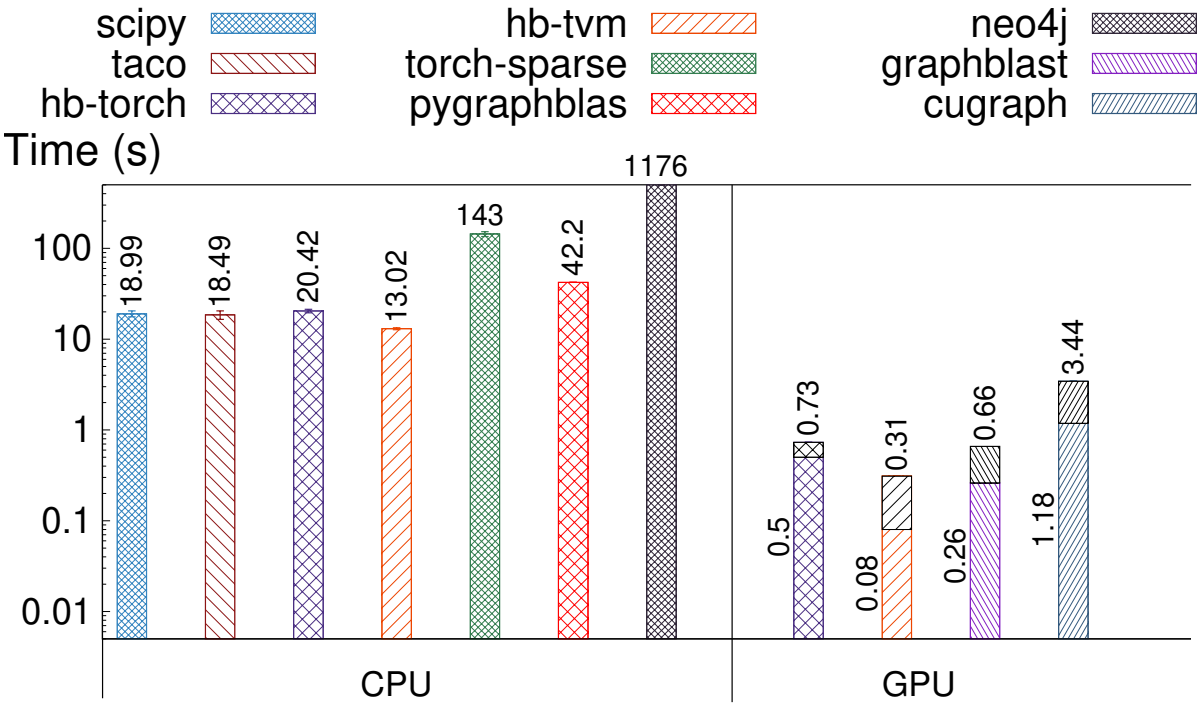


Figure 3.3: PageRank runtime comparison on CPU and GPU. For the GPU we report the computation time as well as the total time including data transfer (stacked gray boxes). For `cugraph`, the data transfer time includes the time to read the data from disk.

3.3.3 Performance Analysis

To evaluate our algorithm we use the LiveJournal Social Network,[1] consisting of 4,847,571 nodes and 68,993,773 edges. This dataset is often used to benchmark libraries performing graph computations [263, 242]. We compare the `scatter_add` approach implemented on PyTorch and TVM (denoted `hb-torch`, and `hb-tvm`, respectively), both on CPU and GPU. For the CPU experiments, we use the following baselines: 2 specialized libraries able to efficiently handle sparse linear algebra operations (`scipy` [26] and `taco` [175]); a PyTorch implementation using sparse matrix-vector multiplications (denoted `torch-sparse`); `pygraphblas` (a Python wrapper around GraphBLAS [17], a specialized library that expresses graph operations as linear algebra blocks on top of BLAS [10]); and `neo4j` [20] (a graph database). Besides Neo4j that is implemented in Java, all other CPU baselines are implemented in C/C++ with Python bindings.

[1] `https://snap.stanford.edu/data/soc-LiveJournal1.html`

For the GPU experiments we further compare against `graphblast` [263] (a GPU implementation of GraphBlas); and `cugraph` [15] (a library developed by NVIDIA and implementing GPU accelerated graph algorithms).

We run 50 iterations of PageRank, which are enough for the algorithm to converge across all frameworks. The running time does not include the construction of the prerequisites of the algorithm (e.g. the construction of the sparse adjacency matrix M from the original indices), because this procedure and the optimal format for the sparse matrix are different across frameworks.

CPU Results. As shown in Figure 3.2, our PyTorch implementation is on par with `scipy` and `taco`. By adding TVM on top of our PyTorch implementation, we gain an additional 45% performance, as TVM unrolls the loops and fuses the different operators into one, and we outperform the aforementioned baselines by almost 30%. The Pytorch and TVM implementations are more than 2 and 3× faster, respectively, than `pygraphblas`. Finally, if we compare our implementations with `neo4j` and `torch-sparse`, we outperform both by orders of magnitude.

GPU Results. In this setting, if we take into account only the computation time, our PyTorch implementation is 2× slower than `graphblast` and 2.3× faster than `cugraph`. However, TVM improves the performance and it is 5× faster than the PyTorch implementation, making the overall runtime 3.2× faster than `graphblast` and 15× faster than cugraph. If we take into account both the transfer and the computation time, PyTorch is 10% slower than `graphblast` and 3× faster than `cugraph`. With TVM, HUMMINGBIRD is 2× faster than `graphblast` and 10× faster than `cugraph`. The `scatter_add` operation is non-deterministic in GPUs for floating point numbers due to the use of atomic add instructions which do not enforce a deterministic ordering. However this problem does not arise for integers (i.e., in our PageRank implementation).

3.4 Relational to Tensors

Among the operators composing relational database engines, selections, projections, and simple aggregates (e.g., COUNT(*) but also SUM, AVG, MAX, MIN) are already naïvely expressible in TCRs operators. As an example of a non-trivial relational operation over tensors, below we describe the implementation of *cardinality calculation*.

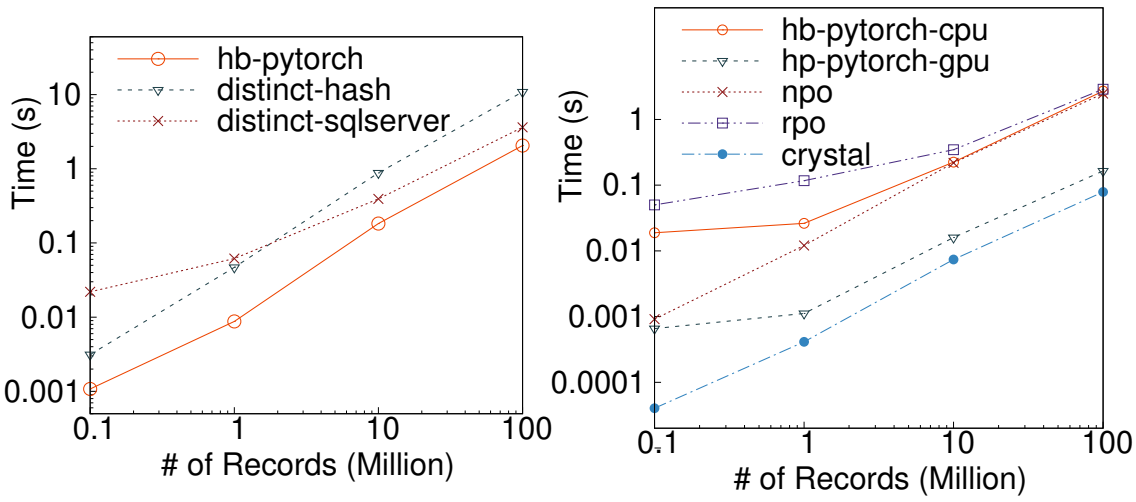
3.4.1 Cardinality Calculation

Knowing the cardinality of operators is crucial in query optimization [12] and to answer DISTINCT queries. TCRs can also calculate histograms on the input data, which can be used by the query optimizer to decide the right type of scan or join operators [28] or to distribute the data evenly across nodes on a distributed join [95]. We thus show how to calculate the output cardinality of relational operators like DISTINCT or JOIN using tensor computations.

3.4.2 Tensor Implementation

Our cardinality estimation implementation is based on the `bincount(T)` operator, which counts the frequency of each value in the input one-dimensional tensor `T`, with the assumption that `T` contains only non-negative integers. The result of this operation is a frequency histogram over the input values. Specifically, the result is another 1-d tensor `O` of length equal to the maximum element present in `T`, where `O[i]` contains the number of times the number i appears in `T`. Using `bincount`, we can calculate the output cardinality of a DISTINCT operator by counting the number of elements that have a value > 1 in `O`. In case the input has negative numbers, we can apply tuple renaming, e.g., map the negative numbers to positive ones by using hashing or by adding a positive constant.

To estimate the output cardinality of a join if the keys are unique, we simply concatenate the inner and outer relation keys and call directly the function in the concatenated 1-d tensor. Every element > 1 is a match, therefore we only need to count the number of elements satisfying this condition. If the keys are not unique, we slightly modify the algorithm. We first call `bincount` function on the keys of the individual relations `S`, `R`. For the result of these calls, namely $\mathtt{B}_S$, $\mathtt{B}_R$, we calculate $\mathtt{l}_{\min} = \mathtt{min}(\mathtt{size}(\mathtt{B}_S), \mathtt{size}(\mathtt{B}_R))$, since the matching keys are only in this range. We truncate $\mathtt{B}_S$, $\mathtt{B}_R$ to length $\mathtt{l}_{\min}$. Finally, to cover all the possible pairs of keys with more than one match in case the keys are not unique, we multiply $\mathtt{B}_S$, $\mathtt{B}_R$ and `sum` the entries in the resulting one-dimensional tensor. Although our algorithm is very simple, it is sub-optimal when the keys do not come from a dense domain. We can overcome this obstacle similar to the `distinct` case, e.g. by tuple renaming.



(a) DISTINCT (CPU only).

(b) JOIN (CPU and GPU)

Figure 3.4: Cardinality estimation comparison.

3.4.3 Performance Analysis

To evaluate the performance of our cardinality calculation, we compare against an handwritten C++ program that calculates the cardinality of `distinct` using an optimized hashmap (denoted with `distinct-hash` in Figure 3.4a). We choose this baseline as hashing is one of the main techniques used to calculate the distinct number of elements in DBMSs [16]. We additionally preallocate the memory for the hashmap and we do not report this as part of the runtime. Finally, we also compare against an instance of the latest edition of SQLServer to qualify our performance against a real-world DBMS. For SQLServer, we create a table that we fill with unique keys with the desired cardinality and we run the query `select count(distinct(id)) from table`.

For the `join` output cardinality, we compare against CPU and GPU baselines that compute the full join. In order to have a fair comparison, we modify these latter baselines such that they return only the number of matching tuples (e.g., the cardinalities) instead of performing materialization. For the CPU baseline, we use the hash-join implementations provided by Balkesen et. al [89] that, to our knowledge, are the state-of-the-art on CPU. We run all of the algorithms contained in the chapter (hardware oblivious join, hardware conscious join, parallel-radix hash join histogram-based, parallel-radix hash join histogram-based optimized), but for the sake of conciseness, we report only the numbers for the hardware oblivious join (denoted by `npo`) that is the fastest among all implementations, and the hardware conscious join (denoted by `rpo`) which

is the fastest of all the parallel-radix implementations. Finally, for the GPU experiments, we compare against crystal [239], a recent library specifically tailored for relational operators on GPU.

DISTINCT Results. For the `distinct` cardinality estimation the input consists of a vector of integers where each element occurs only once. We run the C++ baseline and the PyTorch implementation on a single thread, while we let the SQLServer query run over all the available virtual cores (6). We report the results in Figure 3.4a. As we observe, our implementation (denoted with `hb-pytorch`) is 2–4× faster than the C++ implementation, because of the vectorized processing that PyTorch offers, compared to the scan loop of the C++ program. Compared to SQLServer, our implemenation is almost 2-20× faster. The difference is larger for small inputs, as a DBMS has a larger overhead to distribute and parallelize the plan.

JOIN Results. For the join cardinality estimation, we use as input {`key, value`} tuples where each field is a 32-bit integer. We generate the inner and the outer relation such as all the keys are distinct and there is a 1-on-1 match for the inner and the outer side.

We use 6 threads both for the PyTorch implementation and the baselines. For small inputs, the hardware-oblivious implementation outperforms our implementation (denoted `hb-pytorch-cpu`), but the difference gets smaller as we increase the number of elements. On the other hand, the PyTorch implementation is from 50% to 5× faster than the hardware-conscious implementation, but the runtime is almost the same when the input is more or equal than 10 million elements for each relation. However, we observe that our implementation does not fully utilize the CPU, and therefore there is plenty of room for improvement.

For the GPU experiments, our implementation is 2-16× slower than `crystal`, but the difference is decreasing as the input grows. The difference comes from the fact that `crystal` leverages its optimized CUDA implementation, while PyTorch show some constant overhead (due to the Python interpreter) for small sizes.

Remarks. Our join cardinality estimation outperforms or is comparable to full join implementations performing counting (i.e., they do not execute materialization). The observed speedup shows the promise of TCRs in simple relational calculations. We are actively working towards supporting full joins with materialization and other relational operators. Finally, compiling the code with TVM did not give us additional performance but, as for scatter_add, support for the bincount function was only recently introduced.

3.5 Related work

Much as the MapReduce [122] abstraction helped democratizing distributed computing, the tensor abstraction could play a similar role for hardware accelerators. Similarly to how Apache Spark [268] proved that is possible to run ML [207], graph [142], and relational [82] workloads end-to-end on the MapReduce abstraction, in this chapter we test the expressivity of the tensor abstraction, as well as the efficiency of TCRs for different classes of computations.

Hardware-portable languages [23] and compilers [52, 189] allow targeting both CPU and accelerators. However, they require programmers to write custom code in low-level languages, whereas our goal is to leverage already available optimized CPU and accelerator kernels exposed through the tensor abstraction of TCRs. Dandelion [234], Hocelot [152], and Voodoo [225] share the same goals, although they provide custom kernels or they use OpenCL to target the different hardware with a single implementation. More recently, offload annotations [267] suggests annotating functions implemented in high-level languages (i.e., Python) with a corresponding function from an accelerator library. This approach works well for specific domains where efficient kernels are already available (e.g., Numpy), while it is not clear how the approach will work in the generic case, i.e., how to map a tree-traversal algorithm.

In the database domain, in the past few years several efforts [255, 154, 183] have suggested to map tensor algebra into relational algebra for leveraging database optimizer over linear algebra expressions. We deem this approach complimentary, and we plan to explore in the future the opportunities brought by the unified representation. Tensor execution of relational operators has natural commonalities with vectorized execution over columnar data [105]. Indeed tensor operators are mapped to vectorized instruction sets on CPU (when available). Interestingly, we find that TCRs (and the tensor abstraction) is flexible enough to support both vectorized and compiled execution [214]; the latter implemented through the TVM compilation stack for example. In both cases, no low-level details on vectorized execution or LLVM compilation are required, which we think makes easier the exploration of novel (hybrid) strategies.

It is well known that graph algorithms can be mapped over linear algebra operations [17]. To our knowledge, we are however the first showing that graph algorithms can be run efficiently over TCRs. As for relational operators, TCRs are flexible enough to allow switching between a linear algebra-based implementation to an imperative one, as we describe in Section 3.4.

Finally, several proposals exist on how to run relational operators on hardware accelerators (e.g., GPU [239], FPGA [130] and even TPU [153]). The same applies for graph algorithms and tra-

ditional ML [263, 111]. In this work we claim that similar, or even better, performance can be reached by targeting the tensor abstraction and using TCRs, rather than using low-level languages and primitives (e.g., CUDA). In recent years, several companies have proposed to accelerate specific workloads or systems, e.g., Spark-RAPIDS [27], BlazingSQL [11], OmnisciDB [21]. Compared to these, we think that the tensor abstraction will allow us to be hardware and library agnostic, while enabling similar performance. There is a growing amount of work [118, 137, 149] that proves that indeed tensor-based systems have competitive performance.

3.6 Conclusion

In this chapter we investigate whether the tensor abstraction can be used for programming general computations that can be run end-to-end on hardware accelerators. Our experience [211] and experiments show that good performance is achievable, thanks to the massive investments poured into TCRs, as well as the related optimizations [175, 160, 33, 272, 113, 250]. Our implementations outpeform custom kernels and specialized libraries both for graph and relational workloads. We now present future directions and lines of investigation that could stem from our work.

TCR coverage landscape. TCRs have an excellent coverage of dense arithmetic operations as well as operations manipulating statically sized tensors. Additionally, sorting and simple aggregation is very efficient, and we therefore expect sort-based grouping and sort-merge joins to have good performance. However, there are also "pain points" that hinder TCRs from executing efficiently and easily operations common in other classes of algorithms. For example, as we saw in the experimental evaluation of PageRank, sparse operations are very inefficient compared to other libraries (e.g. scipy). However, the current trend towards Graph Neural Networks (GNNs) and sparsity in DNNs in general [30] has increased the interest of the community towards this important topic. As GNNs are getting more popular, we expect better TCR coverage of sparse data operations in the near future. Additionally, there is limited coverage of group aggregates (except for very basic cases such as `scatter_add`, but the community is trying to fix this limitation [24]) and pipelining operators to avoid intermediate materialization of results (although TVM is improving this thanks to its ability to do operation fusion). There is almost no support in regards to other data structures (e.g., hash maps), non-numeric data types (e.g., strings), and inputs of irregular shape and size. These features are specifically required for supporting,

for example, compression or text data types. Finally, we find that the tensor abstraction shines when one needs to do bulk data transformations, while it lacks flexibility and performance for fine-grained operations. For example, predication [233] is hard to express using the tensor API.

Challenges. While our initial results are promising, there are several limitations that need to be addressed. The two bigger challenges are (1) compilation time, (2) data movement between CPU and memory. The database community is actively working on improving compilation performance for SQL queries [179], and similarly the systems community is working on Just-In-Time compilation approaches for DL models [159]. Driven by these ideas, we could devise a scheme where interpreted code is executed first while generating machine code on-the-fly. Regarding data movements, our goal is to compile computations end-to-end such that data movement is minimized. Other techniques such as pipelining data access with compute, paging, and batching can be used both at conversion and runtime to improve data movement performance.

Hardware trends. In this chapter, we focused on hardware acceleration over GPUs, given their widespread adoption and availability. An emerging trend in hardware acceleration is dataflow-based architectures both in academia [252] and in industry by companies such as SambaNova [25], Graphcore [18], or Cerebras [13]. Interestingly, those approaches also expose TCRs as a programming interface [208, 32, 14]. We are planning to test our approach on this hardware for relational and graph workloads.

Compiled vs vectorized execution. Main memory query processors [268] and databases [214, 105] are increasingly getting compute-bound rather than memory-bound. This is because compilation [214] and vectorization [105] approaches allow achieving even better hardware utilization. TCRs are built from the ground up for compute-bound workloads (i.e., DL). Additionally, the high-level tensor abstraction on top of TCRs considerably simplifies the process of implementing new algorithms, since there is no need to understand LLVM internals, or to write GPU kernels. We therefore believe that using TCRs as a component of main memory query processors will enable a new wave of systems and innovative algorithms. For instance, any vectorized algorithm (or in general any algorithm over columnar data) can be implemented on top of TCRs. Since TCRs also allow compilation and operator fusion, an exciting direction is the exploration of vectorized vs compiled implementations [170], and when to pick one or the other based on, e.g., the hardware.

Cross-optimizations. Apache Spark [268] has shown that Data Frames can be used to express relational, graph, and ML computations, although cross-optimizations between them is limited. SystemML [171] and others [255] have considered using relational algebra (and database-

style optimizers) for logically (co-) optimizing linear and relational algebra. For example in Raven [167] we explored how to cross-optimize relational and traditional ML, beyond linear algebra (e.g., decision trees). With the approach presented in this chapter, a new level of cross-optimizations will be possible since all computations are expressed using the same (tensor) abstraction.

Chapter 4

NVMe as a storage solution

4.1 Motivation

To alleviate the memory pressure [186] and to accelerate I/O, Persistent or Non Volatile Memory (PMEM) has been proposed as a solution. PMEM is both cheaper and has a higher capacity than DRAM, it is byte-addressable, and it persists data. This comes at the price of higher latency and lower bandwidth.

Database researchers have extensively studied how to integrate PMEM into a DBMS [83, 84, 85, 86, 231]. Nevertheless, most of these studies are based on simulating PMEM since they were done before it was commercially available. The picture changed with the release of Intel®Optane©DC [68]. Intel®Optane©DC can be configured in Memory, AppDirect or Mixed mode. In Memory mode it operates as a volatile memory extension, in AppDirect mode it serves as byte-addressable persistent memory, and finally in Mixed mode part of it runs in AppDirect mode and the rest in Memory mode.

Optane sparked numerous studies benchmarking and explaining the behaviour of the hardware [158, 220, 223, 257]. These studies evaluate the hardware's characteristics and compare them to DRAM and SSDs. For example, they show the large latency difference between sequential and random accesses and the asymmetric behaviour of PMEM between read and write bandwidth.

However, DBMSs are complex systems with many knobs that behave very differently as workloads and input size vary. Existing studies [119, 106, 260] give only a partial picture as they provide only high-level conclusions and are mostly based on micro- or small-scale benchmarks. Furthermore, they do not explore how database knobs and different operations (e.g. join types) or query plans affect PMEM's behaviour. To this end, in this chapter we provide the first extensive analysis of database engines using Intel®Optane©DC Persistent Memory. We run OLAP, OLTP, and key-value store workloads on various engines (PostgreSQL, MySQL, SQLServer, DuckDB, VoltDB, and RocksDB) under various PMEM configurations altering a variety of database/workload knobs.

Our results provide a complex picture showing that using PMEM does not always lead to better performance, despite its nominal advantages. Although PMEM is faster than SSDs, the I/O path is not fully optimized since there is no OS prefetching and the CPU is involved in I/O when using PMEM as persistent memory, wasting valuable and often scarce processing capacity. Furthermore, in systems with resource contention due to mixed or write-heavy workloads, PMEM experiences a large performance drop. This makes SSDs competitive in scenarios such as CPU-heavy queries with high selectivity. Similarly, although PMEM offers extra volatile memory capacity in Memory mode, this does not translate to performance benefits. In conclusion, our findings indicate that in its current form, PMEM does not provide large advantages to database engines, particularly when considering its additional cost.

4.2 Background

4.2.1 Persistent Memory

Persistent memory, also known as non-volatile or storage class memory, combines the byte-addressability and low latency of DRAM with the persistence and high capacity of disks. The technology is available in the Intel®Optane©DC Persistent Memory Module [68] (referred to as PMEM in the rest of the chapter). It is faster and more expensive than NAND Flash but slower and cheaper than DRAM. PMEM comes in a DIMM form factor in 128, 256, or 512 GB sizes and it is attached to a memory channel.

Each memory channel connects to the CPU via an integrated Memory Controller (iMC) (Figure 4.1), which maintains read/write queues for PMEM DIMMs and ensures persistence on power failure. With a maximum memory capacity of 6TB for a 2-socket server, PMEM's inter-

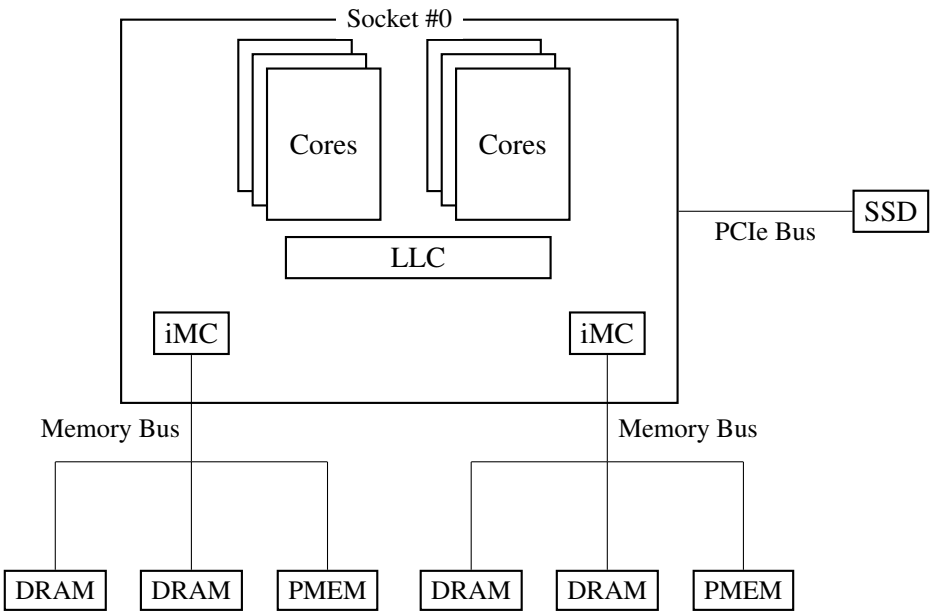


Figure 4.1: Socket topology

nal access granularity is 256 bytes, which can cause read/write amplification [158]. Sequential access patterns have highest bandwidth, just 2-3x lower than DRAM [158].

PMEM operates in three modes: Memory Mode, AppDirect Mode, and Mixed Mode. In Memory Mode, PMEM acts as volatile memory while DRAM serves as an L4 direct-mapped cache. The iMC manages data traffic between DRAM and PMEM, with applications experiencing DRAM latency for cache hits and both PMEM and DRAM latency for cache misses. Applications do not need any source code changes to use this mode.

In AppDirect Mode, PMEM serves as persistent storage, offering striped read/write operations in interleaved or standalone regions. Configurable namespaces modes include `fsdax`, `devdax`, `sector`, and `raw` mode [71], with Intel recommending `fsdax` for DAX-aware file systems. [71] `Fsdax` eliminates the OS page cache in the I/O path, allowing `mmap(2)` to directly map to persistent memory media. Finally, Mixed Mode combines Memory and AppDirect modes, using DRAM as an L4 cache. Users can configure the PMEM percentage for volatile memory or disk usage, but Intel suggests a minimum 1:4 DRAM to PMEM ratio for cost efficiency.

In the rest of the section, we briefly compare memory management in traditional systems and AppDirect Mode, covering the OS page cache, prefetching, and Direct Memory Access (DMA) operations, as these are crucial for the results of the chapter.

The OS page cache caches recently accessed data from disk. This improves the performance of

I/O operations. User-level programs access the page cache through system calls like `read(2)` and `write(2)`. The page cache uses techniques such as prefetching to further improve I/O. Prefetching anticipates the data that will be accessed and proactively caches them to ensure that the data is readily available in memory when needed.

Direct memory access (DMA) is a feature that allows data to be transferred between peripheral devices (such as network or storage devices) and memory without going through the CPU. DMA operations transfer data directly between the peripheral device and memory, bypassing the CPU and the page cache altogether.

In a traditional system, the OS page cache and prefetching are used to improve the performance of I/O operations. However, in the case of AppDirect mode, the OS page cache is not available and applications use directly `mmap(2)` to establish mappings to persistent memory. Also since Intel Optane Persistent Memory is not a peripheral device, DMA is not available and the CPU is necessarily involved in every data transfer between PMEM and the CPU. Finally, although PMEM has its own prefetcher, it cannot use the OS page cache for extra memory capacity and it has to rely directly in the memory allocated by the application.

4.3 Experimental setup

4.3.1 System specification

All the experiments in this chapter were conducted on a dual-socket server (Table 4.1) and the socket topology shown in Figure 4.1. Each CPU socket has two memory controllers and three channels per controller. The DRAM and Intel Optane DIMMs are installed in a 1-1-1 topology. We disable the turbo mode and we set the CPU power governor to performance mode with 2GHz as the maximum frequency, so that we can have reproducible experiments with minimum variation. We refer to *Default Mode* as the configuration that does not use PMEM at all and, unless otherwise mentioned, to *AppDirect mode* as the configuration that mounts PMEM in AppDirect mode with `fsdax` enabled, since this is the one recommended by Intel. When we use PMEM in the AppDirect mode, we do not use the SSD at all. In *Memory mode*, we configure all the available PMEM capacity (512GB) as volatile memory. We collect system statistics with the Intel Vtune Platform Profiler [34]. To get more accurate statistics, we modify the source code of the profiler to increase the sampling hardware counter frequency. We use PostgreSQL 13.2, MySQL 8.0.23, SQLServer 2019-15.0.4178, DuckDB v0.6.1 919cad22e8, VoltDB 11.4.0.beta1

Table 4.1: Server specifications

Component	Specs
Sockets	2
CPU	Intel(R) Xeon(R) Gold 6248R
Microarchitecture	Cascade Lake
Cores	48 physical (96 logical)
L1 cache	64KB
L2 cache	1MB
L3 cache	36MB
RAM	128GB (16GB DDR4 @ 2666 MHz × 8)
PMEM	512GB (128GB × 4)
SSD	KIOXIA KPM6XRUG1T92 (2TB)

Community Edition, YCSB 0.17.0 and RocksDB 5.11.3. Finally, we are aware that our SSD is not the fastest, but having a cheaper SSD gives a better perspective on the average case and shows even more clearly how small can the performance gap be besides the hardware difference. We expect that with a high-end SSD disk, the performance advantage of PMEM would become even smaller.

For all our benchmarks, we pin database processes to socket 0 to avoid remote NUMA access to better interpret the results, as done by other studies [106, 260]. That also ensures the working set is larger than the DRAM capacity for most queries. Before executing each query or workload, we clear the OS page and the database cache.

For TPC-H, we use SF-100 and a buffer cache of 16GB across databases. We also add foreign indexes, when the DBMS does not do that automatically. We use PMEM in interleaved mode, since the data together with the indexes do not fit in a single PMEM DIMM.

We use 1000 warehouses for all TPC-C experiments and set the warmup and running time to 3 minutes to sufficiently load the buffer pool and stabilize database activity. To reduce the proportion of reads to writes, we employ a large buffer pool (48GB). Additionally, we found that using only one of the two available PMEM DIMMs in the socket was sufficient to reach maximum tpmC for most experiments. Thus, unless otherwise noted, we use non-interleaved mode (e.g., one PMEM DIMM) for TPC-C in AppDirect mode, ensuring a better comparison with Default mode, which uses only one SSD. For MySQL, PostgreSQL, and SQLServer, we use HammerDB [47] as a TPC-C implementation, while for VoltDB, we use the official implementation provided in the VoltDB repository [48].

We adjust DBMS checkpointing strategies and parameters for frequent checkpoints, offering a clearer picture of hardware impact on performance. Specifically, for PostgreSQL (using PGTune [37]), we set `checkpoint_timeout` to 30 seconds and `max_wal_size` to 5GB. For SQLServer, we do automatic checkpoints every minute. For MySQL, we disable binary logs, keeping default log buffer and log file sizes, causing checkpointing every second. For VoltDB, we set the flush interval to 1 second and use one snapshot copy that's continuously overwritten, exploring PMEM's behaviour to concurrent writes. Frequent checkpoints, while atypical, expose PMEM's weaknesses under stress, providing valuable insights.

Furthermore, SQLServer requires `fsdax` to store data and `sector` to store logs [43]. We thus have to use both memory sockets, as it is not possible to create a mixed namespace in one socket that has the capacity for 1000 warehouses. Finally, although we run the Memory mode for TPC-C for many configurations across DBMSs, in most cases its performance is almost identical to the Default mode as requests are served by the L4 DRAM cache and not by PMEM. We therefore do not report any Memory mode results for TPC-C. Finally, we do not compare against Intel Optane SSDs, because Optane Persistent Memory is significantly faster that Optane SSDs [106].

For RocksDB, we use the Yahoo Cloud Serving Benchmark [116], specifically Workload A (update-heavy), Workload B (read-heavy), and Workload C (read-only). We generate 110GB of data per workload and set the operation count to 10 million to enable a more comprehensive analysis of the hardware behavior. We leave all other parameters of the benchmark to their default values.

Finally, we do not run experiments for Mixed mode. This mode requires a minimum volatile memory ratio of 1:4, which Intel suggests for PMEM to be cost and performance-effective over a DRAM only solution. Furthermore, different sizes and ratios are needed to understand this mode but each one requires its own hardware configuration and individual insights might not generalize.

4.3.2 Basic microbenchmarks

As a baseline, we measure the latency and bandwidth of our system for different memory types (Tables 4.2 and 4.3), following a similar methodology to Wang et al. [264]. We use one socket for all microbenchmarks to avoid NUMA-node effects, employing AVX-512 instructions and PMEM in interleaved mode. We don't perform measurements for PMEM in Memory Mode due to Intel's lack of public information on its operation [70, 72].

Table 4.2: Latency measurements for different memory types

Memory type	Read latency [us]	Write latency [us]
DRAM	0.147	0.250
PMEM	0.333	0.262
SSD (random)	4.97	108.9
SSD (sequential)	4.85	94.9

For PMEM and DRAM, we measure load and store latencies by issuing a 64-byte instruction for sequential and random memory accesses and not using any caching. For loads, we issue a 64-byte load instruction with a cold cache. For stores, we load the memory address into the cache, and afterwards measure the 64-byte store, followed by a flush (clwb) and a fence (sfence) instruction. For the SSD we use ioping [42]. The SSD has 10× larger read latency than PMEM and more than 30× larger read latency than DRAM. The gap is much larger for write latencies, where the SSD has up to 435× larger latency than both DRAM and PMEM.

We measure bandwidth on sequential and random accesses and we vary the number of threads from 1 to 8. We pick the access granularity that maximizes bandwidth for every hardware type (64B for DRAM, 256B for PMEM, 4KB for SSD). For DRAM and PMEM, we execute a flush and a fence instruction after each store. For loads, we only execute a fence instruction. For SSDs we use fio [41]. We notice that for 1 thread, DRAM has 2.9× more bandwidth for sequential reads and 12% more bandwidth for sequential writes than PMEM. The gap increases as we increase the number of threads, where DRAM has more than 5× larger bandwidth for reads and more than 3× more bandwidth for writes. For random accesses and 1 thread, DRAM has 2.4× more read bandwidth and the same write bandwidth. Increasing the thread count to 8, shows that DRAM has 4.7× larger read bandwith and 3× larger write bandwidth. If we compare PMEM with the SSD, there is a very large gap for reads, but a smaller gap for writes. For 8 threads and sequential accesses, PMEM has 25× higher read bandwidth and 5× larger write bandwidth. However, for random accesses and 8 threads, PMEM has 9× higher read bandwidth and 2.8× larger write bandwidth. The difference is analogous for smaller number of threads with random accesses. For sequential accesses and 1 thread, PMEM has 20× more read bandwidth and 5× more write bandwidth than SSD.

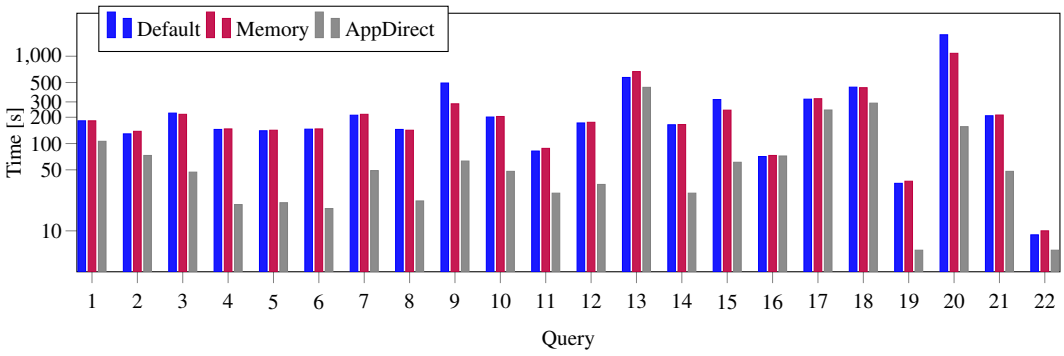
Table 4.3: Bandwidth for different memory types

Memory type	Read bw [MB/s]	Write bw [MB/s]
DRAM (random, 8 threads)	20480	2529
DRAM (sequential, 8 threads)	67550	7615
DRAM (random, 1 thread)	3185	324
DRAM (sequential, 1 thread)	11460	969
PMEM-storage (random, 8 threads)	4341	830
PMEM-storage (sequential, 8 threads)	12628	2425
PMEM-storage (random, 1 thread)	1306	323
PMEM-storage (sequential, 1 thread)	3977	861
SSD (random, 8 threads)	470	293
SSD (sequential, 8 threads)	504	477
SSD (random, 1 thread)	160	157
SSD (sequential, 1 thread)	191	166

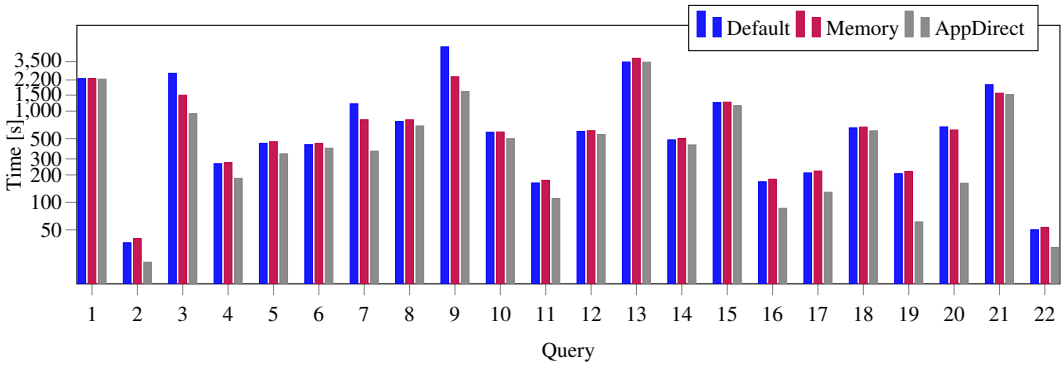
4.4 OLAP workloads (TPC-H)

4.4.1 General observations

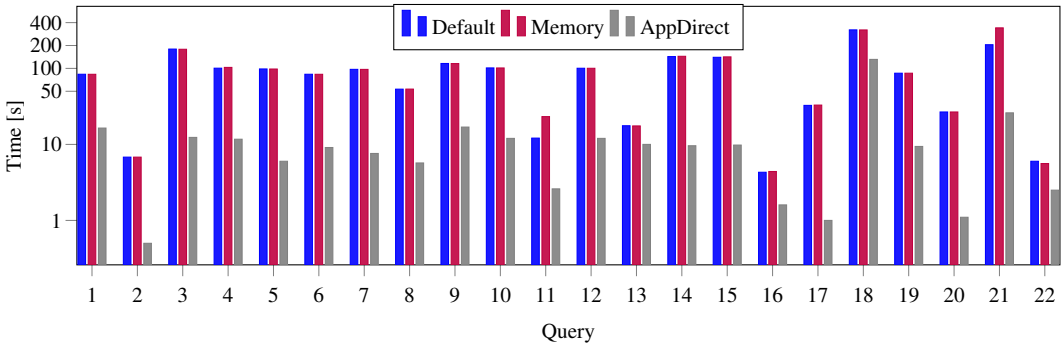
We present the running times of the TPC-H benchmark for all four systems in Figures 4.2. DuckDB cannot execute queries 17 and 21 because of insufficient memory.In general, the AppDirect mode is consistently faster than the Default mode. This happens because PMEM has lower latency and higher througput than SSDs and, thus, all I/O is faster. MySQL is the slowest database, since it does not support multiple threads per query, except for particular type of queries, e.g., `SELECT COUNT(*)` [36]. When doing I/O with only one thread, the bandwidth in both PMEM in AppDirect mode and the SSD are far from their maximum. PostgreSQL is in the middle, as it uses 8 threads for processing and I/O. SQLServer and DuckDB use in general all the available logical cores in socket 0 (48 in total) and that is why the running time is much lower than the other two databases. The PMEM read bandwidth is up to 8 GB/s for some queries of SQLServer. Besides using all the available logical cores in socket 0, DuckDB uses a native compressed column format to which it pushes selections and projections and therefore it reduces I/O costs vastly. For MySQL, PostgreSQL, and SQLServer, the time difference is more obvious for queries involving larger tables (e.g. the `lineitem` table, which is around 100GB of storage together with indexes). For DuckDB, although the AppDirect mode is faster for the majority



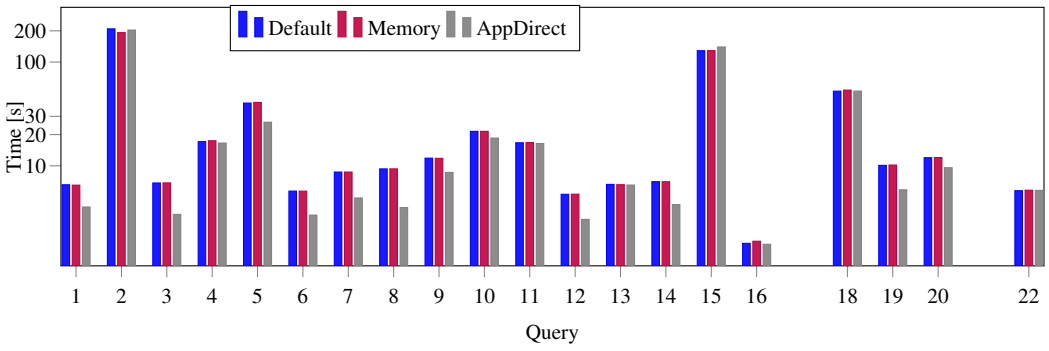
a) PostgreSQL



b) MySQL



c) SQLServer



d) DuckDB

Figure 4.2: Running time (log-scale) for TPC-H SF-100 on various DBMSs for different system configurations

of the queries, the time difference is smaller, since I/O is not in the critical path. Additionally, as mentioned in Section 4.2.1, DMA is not available in AppDirect mode. As a result, CPU resources are used for I/O. In scenarios with CPU-intensive queries or a low number of threads, the CPU spends computation time on I/O, leading to suboptimal PMEM bandwidth and making the Default mode's performance competitive. We can observe this for DuckDB when the optimizer chooses CPU-heavy operations (e.g. index joins, hash group-bys) where the AppDirect mode is very close or has worse performance than the Default mode (e.g. queries 2, 15, 18).

The Memory mode has similar or slightly worse running times than the Default mode for the vast majority of queries. To understand where the overhead comes from, in Figure 4.3 we show the DRAM read and write bandwidth consumed by each query for the Default and Memory modes, respectively, only for PostgreSQL. We observe similar results for MySQL and SQLServer. DuckDB has a different behaviour, since it minimizes I/O because of its columnar compressed format. For all the queries, we read and write more data from DRAM in Memory Mode compared to the Default mode across databases. For reads, that happens because a DRAM (L4 cache) read miss in Memory mode results in a read from PMEM, which consequently leads to a DRAM write. Another reason for the increased DRAM reads is that during a DRAM write, the system has to read the dirty lines and flush them to PMEM. Due to prefetching by the iMC, the DRAM writes are more than the L4 cache misses. Also, the increased DRAM bandwidth for both reads and writes is because there are collisions in the L4 direct-mapped cache, since multiple PMEM memory lines map to the same DRAM memory line. If we allocate memory space such that two PMEM memory lines map to different DRAM memory lines (e.g. by having the exact same capacity in PMEM and DRAM), that will decrease the cache conflicts and makes the two modes comparable.

Insights. For general OLAP workloads, Memory mode does not offer a significant performance advantage, as it slightly hinders performance for the majority of the queries. AppDirect mode can significantly speed up workload execution, if the queries are I/O intensive and there are enough system resources dedicated to the database. If system resources are limited, the design restricts the number of cores/threads involved or the queries are CPU-intense, SSDs offer a competitive and cheaper alternative.

4.4.2 Scans

Sequential scans in AppDirect mode have a much lower latency in PMEM [119, 158]. We observe this advantage for the AppDirect mode compared to the Default mode for all the systems.

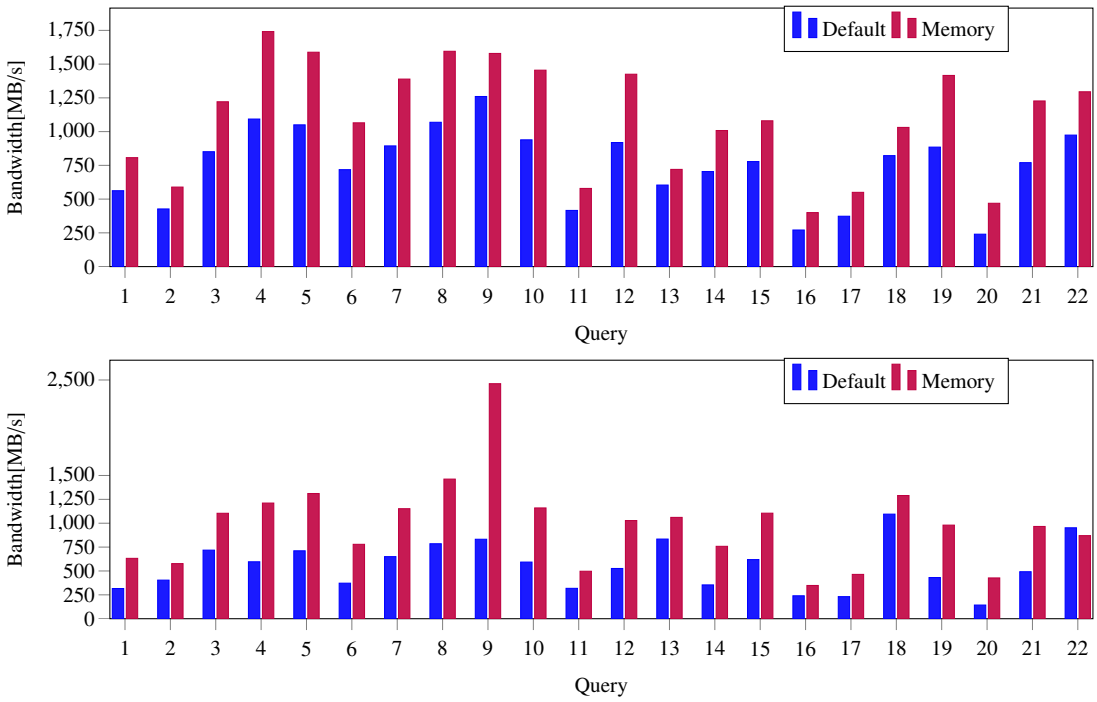


Figure 4.3: DRAM average write (top) and read (bottom) bandwidth on PostgreSQL for different system configurations

However, this advantage becomes more pronounced for queries that involve larger sequential scans (e.g. queries 3, 4, 5, 6, 7, 8, 11, 12, 19, and 20), especially the more threads are used. Although the lower latency does provide a significant benefit, it is not as substantial as we initially anticipated based on our microbenchmarks. We expected to see a 9-25× difference across all queries, but the actual advantage is somewhat lower. This is due to the fact that, as explained in Section 4.2.1, the AppDirect mode with `fsdax` does not utilize DMA or the OS page cache.

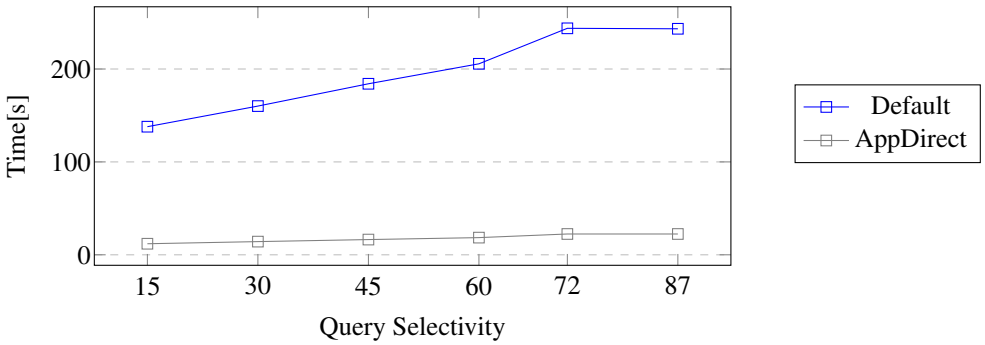


Figure 4.4: Increasing selectivity microbehmark for PostgreSQL

This is especially true for queries that have large sequential scans and are also compute-heavy (such as query 1), where valuable CPU resources are used by PMEM for memory transfers in the absence of DMA, and the Default mode gains an advantage. In contrast, simpler queries that mostly involve scans (such as query 6) benefit greatly from PMEM. To validate that, we run a variant of TPC-H SF-100 query 6 using PostgreSQL, where we increase the selectivity of the predicates (i.e. we select more data) and we present the results in Figure 4.4. As we observe, by increasing the selectivity, both PMEM and the SSD have a sublinear time increase but their relative difference stays constantly between 9-10x and it is due to the sequential scan of the lineitem table. Furthermore, SQLServer writes data directly to physical hardware, bypassing the OS page cache [44], and it is also more efficient at pipelining I/O and processing. As a result, there is a larger time difference between queries, since SQLServer can take full advantage of the higher bandwidth. Finally, MySQL prefers nested loop joins over hash or merge joins. Nested loop joins have many random accesses when there are no indexes available compared to hash or sort/merge joins. Especially in the case of MySQL, the secondary indexes are non-clustered and in case of matches, the DBMS has to go into the base table to materialize the result. However, as these accesses have large block sizes (i.e. 16KB), PMEM has similar bandwidth to sequential accesses. In contrast, prefetching and the OS page cache cannot help in Default mode for random accesses. ***Insights.*** The AppDirect mode is largely beneficial for non-CPU intensive queries with sequential scans. When the majority of the data has to be heavily processed, then the lack of the page cache and prefetching in AppDirect mode reduces most of PMEM's advantages. Users should tune the block size to a large size (≥ 8KB), when the DBMS performs random scans, because then PMEM has similar bandwidth to sequential accesses.

4.4.3 Index lookups

Index lookups cause random accesses. Thus, we would expect a performance drop in AppDirect mode. However, because all tested databases use large block sizes (≥ 8KB) the lookup accesses are about 10× faster for the AppDirect mode (queries 2, 4, 5) or have the same latency as a sequential scan for PMEM (queries 8, 9, 10, 19, 20, 21). We also observe that the type of index can affect the lookup time. More specifically, MySQL stores primary clustered indexes together with the data and secondary indexes separately. That puts PMEM at a disadvantage, because an access to a secondary index needs to locate the data in the tables and in the Default mode part of these accesses are served by the OS page cache. This behaviour is not present in the other engines, because PostgreSQL does not have the notion of a clustered index and SQLServer by-

passes the page cache. DuckDB uses indexes only to perform index joins once the data are already in DRAM, and therefore the storage medium is irrelevant at this point in the query plan. Compared to the other systems it has much less I/O due its compressed columnar format, making the effects of the page cache inconspicuous.

Insights. When creating indexes or running query plans that operate on indexes (e.g. joins), the block size should be tuned to be large (≥ 8KB) to maximize the read bandwidth in AppDirect mode for PMEM. Additionally, whenever possible, clustered indexes should be used. The additional lookup in a non-clustered index has a performance penalty for PMEM due to the lack of an OS page cache.

4.4.4 Working set size in Memory mode

As databases try to minimize I/O as much as possible, we would expect that adding a larger volatile memory capacity with PMEM in Memory Mode would improve the running time of queries with a large working set. However, across all three engines, we observe that the majority of the queries perform the same or slightly (less than 5%) worse in Memory Mode because of the increased traffic between the L4 DRAM cache and PMEM as mentioned in Section 4.4.1. We can split the queries in 2 categories based on the working set size. Queries 1, 4, 5, 6, 12, 14 and 17 have a working set of less than 64GB and therefore most of the accesses are served from the L4 DRAM cache. The rest of the queries have a working set larger than the L4 and/or the OS page cache, but the majority of them does not benefit from the larger volatile memory capacity and any improvements of the Memory mode over the Default mode are due to the memory structures that the systems use to handle memory and how the DBMS utilizes the OS page cache. For PostgreSQL, these are queries 9, 15, and 20. For MySQL, these are queries 3, 7, 9, and 21. For SQLServer, no query is significantly faster in Memory mode as the DBMS does not utilize the OS page cache. Finally, DuckDB has very small working sets for all the queries. Thus, in Memory mode the data are directly transferred to the L4 DRAM cache and PMEM is not used.

Insights. A very large working size can run slightly (up to 10%) faster in some cases in Memory mode, due to the larger OS page cache available in main memory. On the other hand, the running times of queries with small working sets is not affected at all by the larger volatile memory capacity.

4.4.5 Query plans

PostgreSQL, MySQL and DuckDB as open-source databases offer valuable statistics about their query plans. We isolate a subset for all, providing valuable observations on how the DBMSs use the underlying hardware. In this section we only compare the Default with the AppDirect mode.

4.4.5.1 PostgreSQL

Query 1 has a parallel sequential scan and a parallel aggregation of `lineitem`. The selection predicate is satisfied by 98% of the rows. In AppDirect mode, 85 out of the 106 seconds are spent in the aggregations, and only 17 seconds are spent in scanning the data. PMEM has a latency of 10us for 4KB application reads [35]. Therefore, for the block size of Postgres (8KB), the latency should be less than 20us. We can confirm that because we have 10 parallel workers with an overall bandwidth of 4-5 GB/s which explains the 17 seconds needed to scan the 86GB table. In contrast, in Default mode, SSDs have a 10x latency, but only 5x of the time is spent in scanning because of the OS page cache and prefetching.

In query 3 the Default mode spends the majority of the time scanning the `lineitem` table. The time is larger than query 1 because the prefetcher does not work so effectively. That happens because the inter-arrival time of read calls is lower as fewer operations are performed in higher-level operators. For the AppDirect mode, the scan time increases to 19 seconds compared to query 1. This is due to the fact that parallel workers are also involved in writing data to PMEM during the join and sorting of the query. It is well known that writing in PMEM requires more CPU and memory resources and provides lower bandwidth [119, 158]

Query 13 has an index-only sequential scan on `customer`, which takes the same time in both modes due to prefetching in the Default mode. There is also an index scan in `orders`, which takes approximately the same time in the AppDirect and Default mode. That happens because the index is not correlated, and therefore the database can scan more than one block at the same time. This leads to a large working set that is larger than the buffer cache, which subsequently leads to many reads from PMEM in AppDirect mode. The amount of data read in AppDirect mode is 12× more than that in the Default mode. However, AppDirect mode still performs better, because PMEM bandwidth is 6× larger and it does not involve the synchronous read from DRAM as the Default mode.

In query 15, we observe the asymmetric behaviour of PMEM as we have to read `lineitem` twice. The maximum read bandwidth is 4 GB/s and the maximum write bandwidth is 1 GB/s.

To maintain the price/performance ratio, we need to increase the working memory or use a separate storage drive for temporary writes.

Lastly, in query 17 there is a hash join between `lineitem` and `part` with `lineitem` on the probe side without any filtering. For the sequential scan of lineitem, prefetching is effective and the main performance difference is caused by index lookups on lineitem. In such CPU intensive queries, when CPU operations overshadow I/O requests, prefetching is very effective in sequential accesses. On the other hand, for random accesses asynchronous I/O is more beneficial. Therefore, if Postgres adopts asynchronous I/O, the performance difference between the two modes would be negligible.

4.4.5.2 MySQL

In query 1, most of the time is spent on aggregation and filtering. The scan time of `lineitem` in the Default mode is 436s and in AppDirect mode it is 392s. The small performance difference has two reasons. First, prefetching and the OS page cache is very effective for sequential scans of large tables. Second, the average block size is around 100KB for both modes. The increase in response time for larger block sizes is sublinear in SSDs due to the inherent parallelism. On the other hand, in AppDirect mode the CPU is involved in reading, and the response time is exactly linear.

In query 3, there is a table scan on `customer` and index lookups on `orders` that use a secondary index. The cost of a lookup is 140us for the AppDirect mode and 250us for the Default mode. The difference is not as large as we would expect based on the hardware, but the Default mode has the advantage of prefetching and the OS page cache. Additionally, PMEM has a larger latency when secondary indexes are used because some accesses are served by the OS page cache in the Default mode.

In queries 4 and 5 there are index lookups on `lineitem` using a primary clustered index, and the cost of a lookup is 15us for the AppDirect and 30us Default mode. The lookup times are close, because the queries have a smaller working set that mainly fits in the buffer cache. In general, for smaller working sets secondary and primary index lookups have comparable lookup times since they fit in the buffer cache and the OS page cache is not useful. We also observe this behaviour in query 7.

In query 13, both modes take the same amount of time, because the working set is larger than the buffer cache but smaller than the page cache. Thus, a memory copy from DRAM to DRAM has similar latency to a memory copy from PMEM to DRAM. Queries 19 and 20 involve a join with

`lineitem` using a secondary index lookup. Every lookup returns 30 and 7 tuples, respectively. PMEM is faster, because of large block random accesses, even when a small number of tuples is returned within a lookup.

4.4.5.3 DuckDB

Query 1 has a sequential scan on lineitem that takes 2.6s for PMEM and 5s for the Default mode which is a 2x performance increase, much less than our microbenchmarks suggest. DuckDB streams data in chunks using a vectorized push-based model, meaning that CPU resources have to be split between I/O and processing. That gives an advantage to the Default mode, which is visible in the subsequent operations that take 1.4s for the Default mode and 1.8s for the AppDirect mode.

Query 2 is very CPU intensive and does not perform a lot of I/O, because it does not involve `lineitem`. Most of the time is spent on index joins between the tables, and therefore since I/O is a negligible portion of the query execution time, the total runtime for the Default and the AppDirect mode is almost the same.

Query 15 has two sequential scans on `lineitem`, that for PMEM take 3.04s while for the Default mode they take 5.75s combined. However, since there is a hash group by and an index join afterwards that are very CPU intensive, the absence of DMA gives an advantage to the Default mode. These two operations take 20s for the Default mode and 23s for the AppDirect mode, essentially covering PMEM's bandwidth advantage completely.

Query 19 has a lot of projections and selections on the lineitem and part tables. As DuckDB pushes most of these directly to the columnar storage, the I/O operations contain effectively most of the running time for this query. The Default mode takes 7.5s to scan the lineitem table, whereas the AppDirect mode does 3.5s.

Insights. A few of the characteristics presented in the last sections are now revealed in more detail through the query plans. We see that the absence of DMA and the page cache closes the gap from 25 (the value observed in our microbenchmarks) to 2 times between PMEM and the SSD in some queries. We also notice the advantage of prefetching in CPU intensive queries (e.g. queries 17, 18) that involve large tables on the probe side of a join.

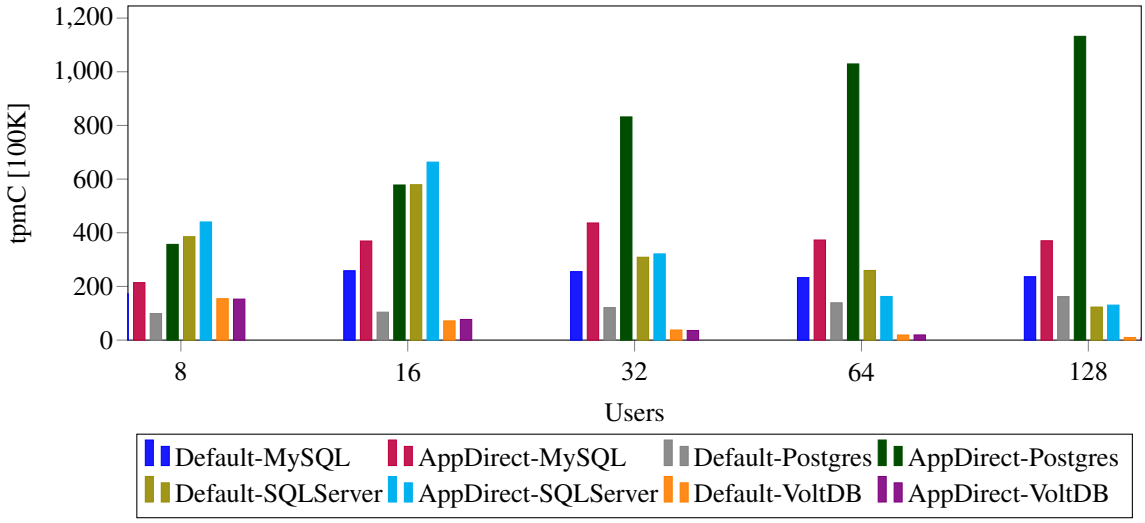


Figure 4.5: tpmC for TPC-C with 1000 warehouses on all three systems for different system configurations

4.5 OLTP workloads (TPC-C)

4.5.1 General observations

We present the TPC-C results for MySQL, PostgreSQL, SQLServer, and VoltDB in the Default and AppDirect modes in Figure 4.5. In general, and for different number of concurrent users, the AppDirect mode performs better than the Default mode but depending on the engine the differences vary.

For PostgreSQL, the AppDirect mode outperforms the Default mode by 3-10× for 8-128 threads respectively. That happens because the checkpoint, background, and WAL writer processes use only one thread and this write combining minimizes resource contention for PMEM. We also conduct experiments with the `no-dax` mode, where the page cache is enabled, but the tpmC drops drastically. This happens because the latency of PMEM is low compared to that of SSDs/HDDs and since PostgreSQL relies on the OS for prefetching, keeping/updating the page cache is an additional overhead.

SQLServer, as mentioned, uses all the available cores in socket 0. As we notice, for small number of concurrent users, the higher bandwidth PMEM offers gives a small performance advantage of around 10% compared to the Default mode, due to the lower latency PMEM has compared to SSDs. However, as the number of concurrent users increases, the Default mode

is very competitive and outperforms the AppDirect mode for 64 concurrent users by 37%. For such high number of users, there is a lot of interference between I/O and processing in the system and the tpmC drops drastically because of thrashing. Additionally, there is a large number of concurrent reads and writes, which causes a significant performance drop for PMEM [119]. Finally, we observe that CPU utilization is very low for the AppDirect mode, indicating that I/O is in the critical path, despite the much lower latency that PMEM offers.

In MySQL, the AppDirect mode consistently outperforms the Default mode by up to 20%. This happens because of the smaller latency that PMEM offers and also due to the fact that MySQL does not use a large number of threads for writing. The latter helps in avoiding write contention in the iMC, which is one of the main reasons that affect significantly PMEM's performance.

VoltDB's performance is worse than other databases in all but a few cases, with the AppDirect mode showing only marginal improvement over the Default mode. The lower transaction count is due to several factors. Firstly, VoltDB is designed for distributed, high-throughput environments, limiting its performance in single-node setups. Furthermore, VoltDB is the only tested system written in Java, while the others use C/C++. Additionally, upon examining VoltDB's TPC-C implementation, we find that it relies heavily on writing to the storage medium whole snapshots instead of just the changes in the database/log like the other databases. High concurrency high volume writing does not scale well, particularly for PMEM, which struggles with numerous concurrent users. This is also the main reason that contributes to the minimal performance difference between AppDirect and Default modes.

Insights. As TPC-C is a write-heavy workload, we confirm what other studies have found about the effect of writes on the performance of PMEM [158]. Having low number of write threads and/or combining writing operations in one thread provides a much larger transaction rate than the Default mode and it also avoids thrashing, as we observe in the case of MySQL and PostgreSQL. Conversely, heavy contention and mixed read/write workloads significantly impact PMEM performance, bringing it close to or even worse than the Default mode, as seen with SQLServer and VoltDB.

4.5.2 Log and data placement

We assess PMEM's effectiveness as a data or WAL store for MySQL, PostgreSQL, and SQLServer (Table 4.4). We could not separate the log and data checkpointing process for VoltDB. We tested configurations with data and WAL on SSD (`SSD-both`), data on SSD and WAL on PMEM

Table 4.4: tpmC (in 100K) for different database configurations and concurrent users

DB/Users / Configuration	PostgreSQL					SQLServer					MySQL				
	8	**16**	**32**	**64**	**128**	**8**	**16**	**32**	**64**	**128**	**8**	**16**	**32**	**64**	**128**
SSD-both	170.8	239.8	332.7	387	474.4	385.2	578.9	308.7	259.4	123	180.6	258.7	262.6	274.1	282.2
SSD-data-PMEM-wal	210.3	313.3	373.5	502.6	495.9	414.3	613.5	321.2	162.5	130.3	189.3	271.5	270	279.7	288.6
PMEM-data-SSD-wal	324.4	520.1	541.6	598.7	645.7	406	662.9	328	172.6	133.2	-	-	-	-	-
PMEM-both	350.6	577.4	854.2	1001.7	1088.4	440.2	663.2	321.2	162.5	130.3	-	-	-	-	-

(`SSD-data-PMEM-wal`), data on PMEM and WAL on SSD (`PMEM-data-SSD-wal`), and data and WAL on PMEM (`PMEM-both`).

For PostgreSQL, for a small number of clients, when we place the WAL in PMEM instead of the SSD, we see only a slight increase in tpmC. That means that buffered I/O does not significantly affect performance in log writing. However, the gap increases when we increase the clients due to the bandwidth limitations of the SSD. On the other hand, when we place the data in PMEM instead of the SSD, we see a large increase in tpmC, even for a small number of clients. Thus, PMEM should be used as a data rather than just as a log store, because databases can utilize PMEM's read bandwidth more effectively than its write bandwidth.

In SQLServer, placing the log in PMEM offers only a small performance advantage, compared to placing the data in PMEM, indicating that I/O is more on the critical path than the redo logs. As the number of clients increases, log placement does not affect performance significantly. Lastly, placing the data on PMEM and the log on SSD has a higher throughput than placing both on PMEM, but this is due to the remote accesses to socket 1 for the log (that is necessary due to the configuration enforced by SQLServer).

For MySQL that log placement in PMEM does not offer a significant performance advantage since the tpmC increases marginally (up to 5%). Thus, redo logs are not the bottleneck and buffered I/O hides the latency for the Default mode.

Insights. Placing only the logs in PMEM does not offer a significant performance advantage irrespective of the DBMS. We therefore conclude that PMEM should be used as a data or a data and log store rather than just a WAL store, because DBMSs can utilize PMEM's read bandwidth more effectively than its write bandwidth.

4.5.3 Varying database configurations

As open source DBMSs, PostgreSQL and MySQL give us the freedom to experiment with a number of parameters that affect transactions. We analyze various options in this section.

4.5.3.1 WAL compression

WAL compression is widely used for PostgreSQL to close the CPU-storage gap. However, in our case even though it is slightly useful for the Default mode, it causes a drop in tpmC for the AppDirect mode. The number of transactions is almost 50% less for 16 clients. Due to the

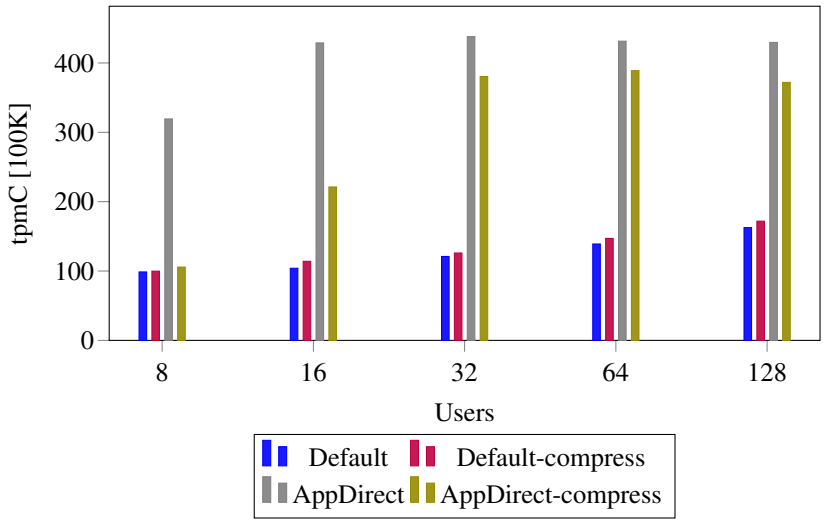


Figure 4.6: tpmC for TPC-C with 1000 warehouses on PostgreSQL with and without log compression for different configurations

lower latency of PMEM, the compression step involving an additional read and write from/to memory becomes unnecessary. Lastly, because PostgreSQL depends on the OS page cache for read-ahead and buffering of writes, we enable the page cache in AppDirect Mode with the no-DAX option. However, as we have mentioned, in AppDirect mode the CPU is involved in reading/writing and this causes an increase in CPU usage and a drop in tpmC as we can see in Figure 4.6. Therefore, although PostgreSQL is very effective at hiding the storage latency of HDDs and SSDs, this is not the case with PMEM and the techniques it uses for HDDs and SSDs hurt performance in PMEM.

4.5.3.2 Double write buffer

In MySQL, the double-write buffer is a storage area where dirty pages are flushed to, before writing to their respective positions in the data files [39]. This buffer does not incur additional costs, because InnoDB writes data in a large sequential chunk with a single `fsync` call at the end. We disable the double-write buffer and present the results in Figure 4.7 for the two modes. We can observe that the tpmC for the Default mode stays stable, since double writes are a part of the background flushing. Therefore, the storage can follow along and transactions are not affected. Conversely, the tpmC increases significantly by up to 20% for the AppDirect mode when double-writes are disabled, confirming again that concurrent writes are a pain point for PMEM.

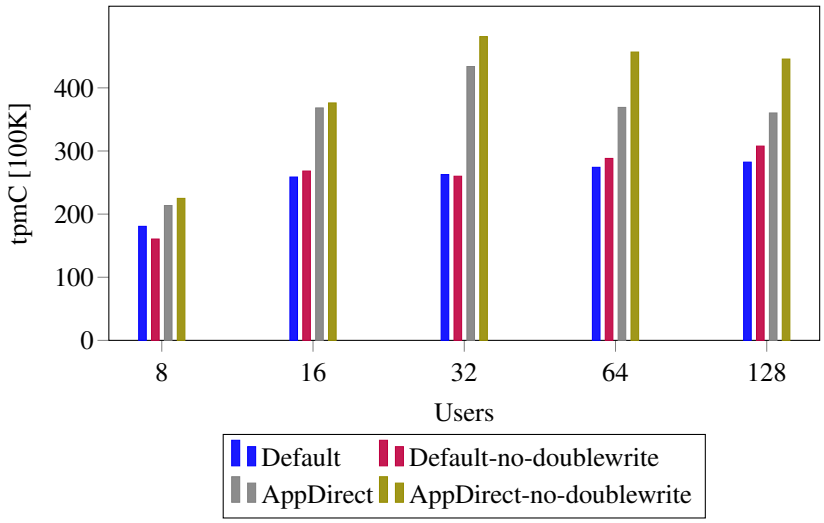


Figure 4.7: tpmC for TPC-C on MySQL for different number of clients with and without the double-write buffer

4.5.3.3 Number of write threads

By default, MySQL uses async I/O to perform reads and writes. This removes control over how many concurrent requests are pending, which might be detrimental to PMEM's performance. By using sync I/O, we can control the number of background write threads. Read threads are mainly used for prefetching and with a large buffer pool they do not play a significant role in the TPC-C workload. We vary the number of write threads from 1 to 16 for different numbers of concurrent users and we present the results in Figure 4.8. The throughput reaches a maximum for 8 threads across all numbers of concurrent users and decreases after that, confirming again that PMEM is not effective at performing many concurrent accesses.

4.5.3.4 Flushing methods

In MySQL, we have to opportunity to experiment with different flushing methods of InnoDB, e.g. enabling the `O_DIRECT` flag, which uses direct I/O (bypassing the OS page cache) for data files and buffered I/O for logs. We present the results in Figure 4.9. In general, the AppDirect mode is faster than the Default mode, both with and without direct I/O, due to the smaller latency that PMEM offers. However, direct I/O does not offer a significant performance advantage in the AppDirect mode, because it does not provide any additional benefit on top of DAX. Furthermore, using the AppDirect mode with the `no-dax` option and the `O_DIRECT` flag, provides the same performance as the default AppDirect mode (with `dax` enabled). This is expected, since both

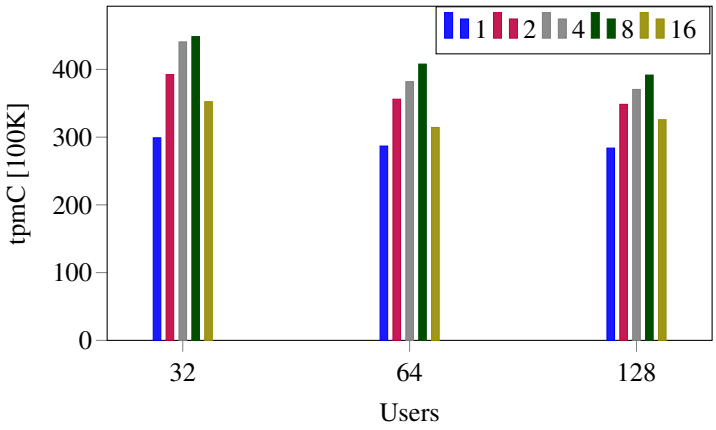


Figure 4.8: tpmC for TPC-C on MySQL for different number of write threads

configurations skip the OS page cache. For the Default mode, when we have a small number of clients this happens because MySQL manages its own buffer pool and does not depend upon the OS page cache for prefetching and caching. As the number of clients increases, the Default mode with direct I/O performs much better, since we have restricted the execution to one socket. Thus, the database is not doing unnecessary memory copy from/to the OS page cache, which in turn reduces contention and leaves more resources available.

We also investigate how disabling `fsync` affects performance. This function is used to ensure that writes are flushed to disk and not to the device cache. Because these calls may degrade performance, MySQL provides a flushing method called `O_DIRECT_NO_FSYNC` that takes them

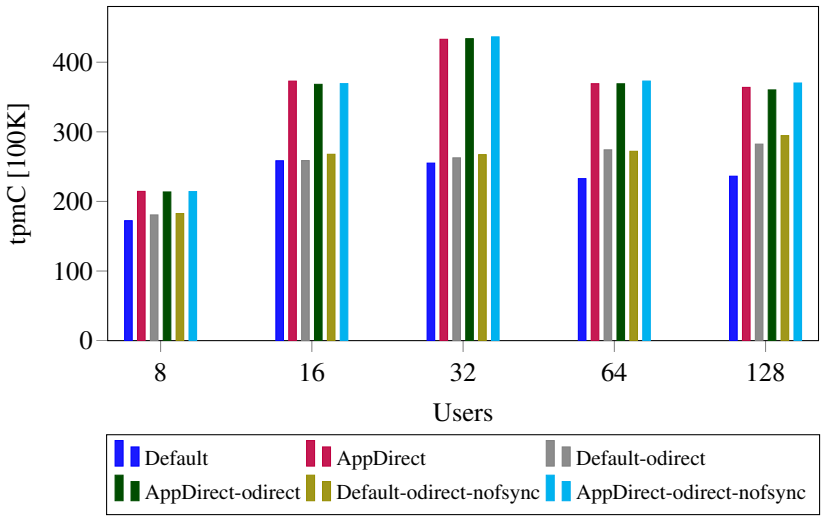


Figure 4.9: tpmC for TPC-C on MySQL with different flushing methods

away. As we see in Figure 4.9, when using this flag, the performance increase is negligible, especially in AppDirect mode. This happens because of two reasons. First, `fsync` calls are not totally eliminated and they are still used for synchronizing file system metadata changes, e.g. during appends [40]. Second, writing dirty pages to the tablespace is done in batches that require a single `fsync` call that happens in the background. Therefore, this does not affect transactions directly [38]. Contrary to the Default mode, when we disable `fsync` we see a large improvement in the AppDirect mode. This happens because overwrites in AppDirect mode with DAX use non-temporal stores. Thus, no cache lines are flushed and only a `sfence` instruction is issued to make sure writes complete.

4.5.3.5 ext4 block size

In MySQL, we evaluate how the ext4 block size affects PMEM's performance. (Table 4.5). We set the ext4 block size to 1KB and 4KB in AppDirect mode with `nodax` and `O_DIRECT` enabled, because direct access is only supported when the block size is equal to the system page size. As we observe in the figure, for a small number of clients 1KB block size performs worse than 4KB, because the amount of I/O is increased and I/O is more in the critical path for a small number of clients. Contrary to that, tpmC is larger for 1KB block size for larger number of clients, because access size must be lower for higher thread counts [119].

4.5.3.6 `no-dax` mode

Regarding different configurations in the AppDirect mode, we experiment with the `no-dax` mode in both PostgreSQL and MySQL. In MySQL, the `no-dax` options is on par with `fsdax`. However, on PostgreSQL, the `no-dax` options performs significantly worse. This is because PostgreSQL depends heavily on the OS page cache for read-ahead and buffering of writes and as we have mentioned, in AppDirect mode, the CPU is involved in reading/writing from DRAM

Table 4.5: tpmC [in 100K] for TPC-C on MySQL for different block sizes on AppDirect mode

Configuration \ Users	8	16	32	64	128
AppDirect-nodax-bs-1kb	210.2	358.9	450.4	404.2	415.8
AppDirect-nodax-bs-4kb	213.6	368.2	433.7	389.7	372.5

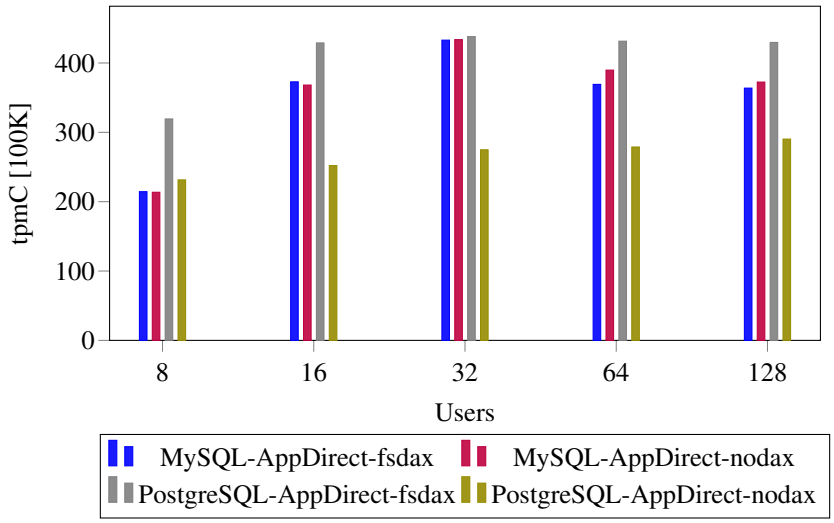


Figure 4.10: tpmC for TPC-C on MySQL and PostgreSQL with and without fsdax

to PMEM as there is no DMA available. This causes an increase in CPU usage and a drop in tpmC.

4.5.3.7 Interleaved vs. non-interlaved

We compare the AppDirect mode in PostgreSQL using only one PMEM DIMM with the AppDirect Mode using both PMEM DIMMs (denoted with AppDirect-interleaved) in Table 4.6. The interleaving access block size is 4KB and any read/write for PostgreSQL happens in multiples of 8KB. Therefore, when using PMEM in interleaved mode, we would expect a large increase in tpmC, since the access latency is halved and the max bandwidth is doubled. However, as we see in the figure, the increase in tpmC is minimal. For a small number of clients, that happens

Table 4.6: tpmC for TPC-C on PostgreSQL using different number of DIMMs

Configuration \ Users	8	16	32	64	128
AppDirect	319.3	428.9	438	431.4	429.6
AppDirect-interleaved	346.8	524.2	524.6	533.9	477.4
AppDirect-interleaved-increased-wal	356.6	578	831.7	1029.1	1132

because the workload is already CPU-bound even when using a single PMEM DIMM. For a larger number of clients, the bottleneck is the checkpointing process. Thus, we increase the `max_wal_size` to reduce the number of checkpoints and the tpmC for 128 clients is more than 2x higher and is bounded by the max bandwidth of the interleaved setup.

Insights. As PMEM is sensitive to write workloads and its performance can deteriorate heavily, we should carefully tune the underlying parameters to avoid additional writes (e.g. the double-write buffer), use small access sizes for higher thread counts and limit the number of write threads to at most 8 threads. Additionally, since the bandwidth of PMEM in AppDirect mode is much larger than the one that SSDs offer, we do not have to resolve to I/O optimizations, especially if these involve CPU operations.

4.6 Key-value stores (YCSB)

We evaluate RocksDB, an open-source key-value store, using YCSB. We choose YCSB due to its use in PMEM-native key-value store studies [97]. Figure 4.11 presents throughput and read latency results, while Figure 4.12 shows write latency for two workloads as the thread count increases. Workload A is update-heavy (50% reads - 50% writes), Workload B is read-heavy (95% reads - 5% writes), and Workload C is read-only (100% read).

In all the graphs, the AppDirect mode consistently achieves higher throughput and lower read latency across all workloads, with the gap widening as the number of threads increases. Meanwhile, Memory mode performs similarly to the Default mode. To understand why, we must examine the workload details. Each workload accesses all fields of a 1000-byte row. With RocksDB processing vast amounts of wide rows, the AppDirect mode experiences read and write amplification, particularly as thread count rises. This results in up to 6.5 GB/s read bandwidth for PMEM (compared to 0.9 GB/s for SSD) and up to 1.3 GB/s write bandwidth for PMEM (compared to 0.4 GB/s for SSD).

We can also make some interesting observations. For Workload A, as we increase the number of threads, reading and writing concurrently, causes the throughput to plateau and slightly drop for the Default and Memory modes, and to heavily drop for the AppDirect mode. In fact, the throughput for 48 threads for the AppDirect mode drops to the level of 16 threads for the same configuration, showing that mixed read-write workloads should keep a moderate number of threads to achieve maximum performance. The plateau effect for the AppDirect mode is also visible if we have a very small percentage of writes such as in Workload B, but it is not visible

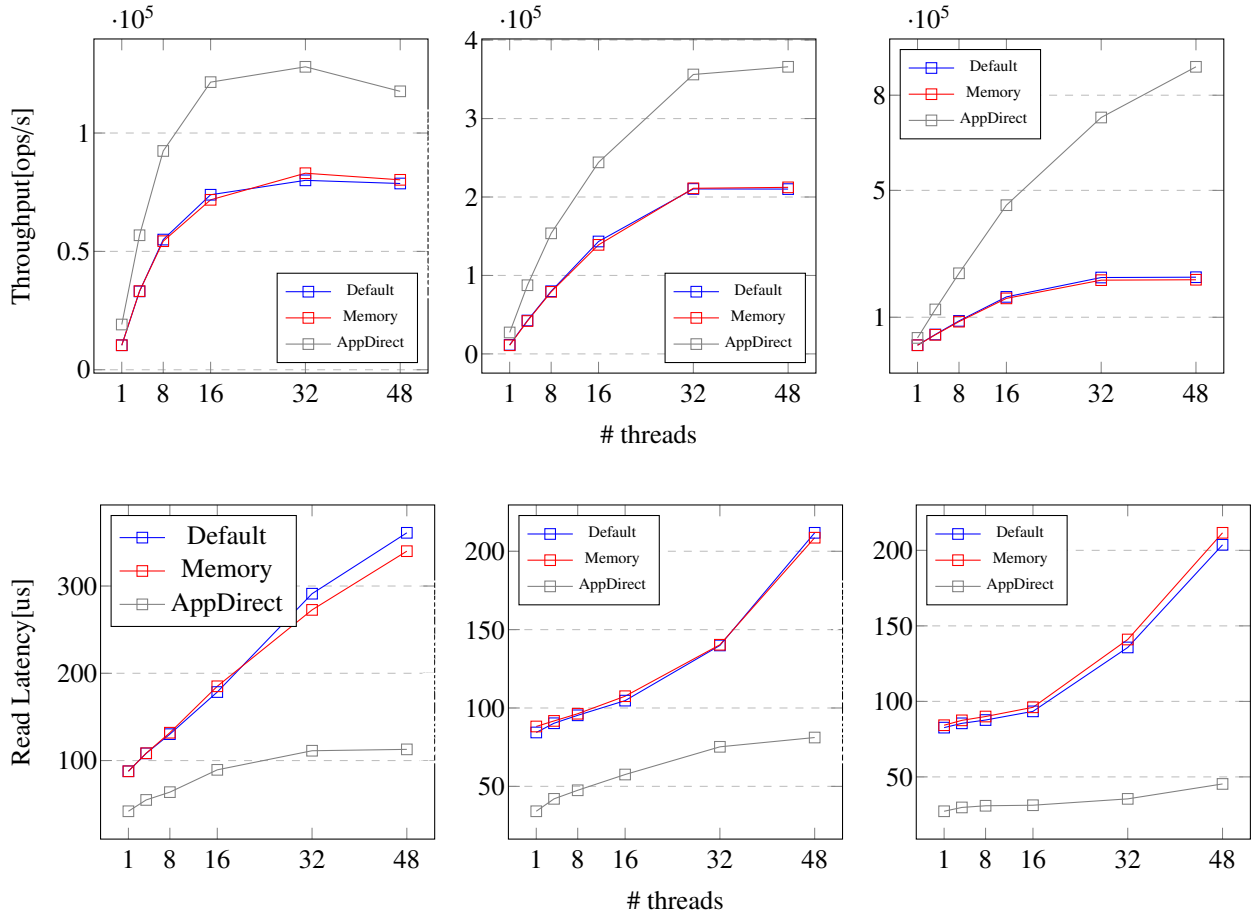


Figure 4.11: RocksDB throughput (left) and read latency (right) for Workload A, B, and C of the YCSB respectively

in Workload C where the throughput continues to increase as we increase the number of threads. For the Default and Memory modes the throughput plateaus consistently after 32 threads because we reach the bandwidth limits of the SSD.

While the read latency of the Default and Memory mode keep increasing for all the workloads, for PMEM in AppDirect mode we observe a plateau in Workload A for the read latency. For the rest of the workloads in AppDirect mode, the read latency keeps slighly increasing. Additionally, the read latency for the Memory mode is slighly lower for higher number of threads compared to the Default mode and the corresponding throughput is higher. Since the workload is 110GB in total the additional volatile memory capacity comes handy as the number of threads increases, because the DRAM capacity is not enough and the requests go to a faster storage medium (PMEM) instead of the SSD. Finally, the write latency of PMEM in AppDirect mode

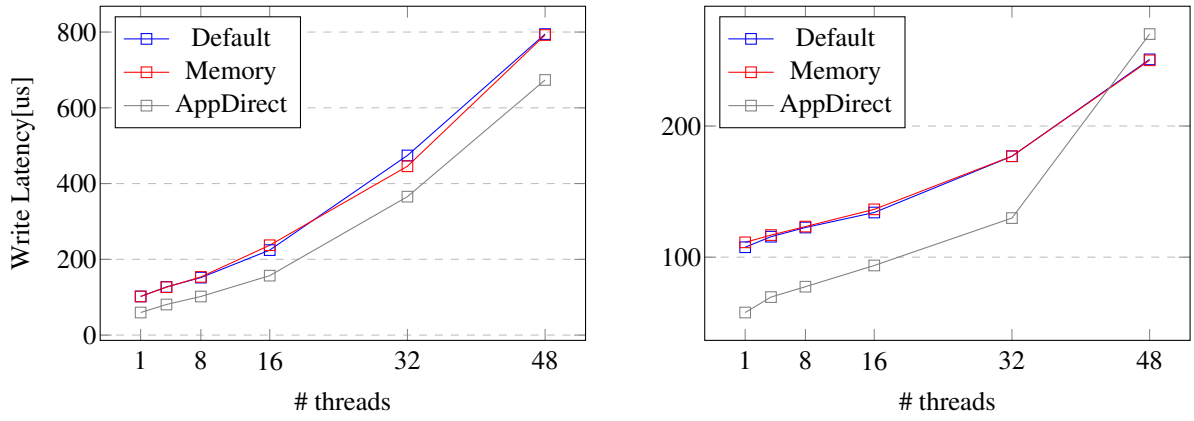


Figure 4.12: RocksDB write latency for Workload A and B of the YCSB respectively

is consistently lower than the rest of the modes, with the exception of 48 threads for Workload B, where the latency is marginally higher. This confirms that even a very low number of writes with a high number of threads can be highly problematic for PMEM.

Insights. The AppDirect mode consistently achieves higher throughput and lower latency across all workloads. However, increasing the number of threads leads to a throughput plateau or drop for mixed workloads. Additionally, the results indicate that even a low number of writes combined with a high number of threads can be problematic for PMEM, highlighting the importance of proper tuning in achieving peak performance. Finally, the Memory mode performs almost identically to the Default mode. Even if the larger volatile memory capacity is handy, the advantage is very marginal.

4.7 Related work

Even before the first PMEM commercial implementation, researchers adapted DBMS components to PMEM characteristics. Chatzistergiou et al. [108] developed a library for transactional updates in data structures designed for PMEM and Wang et al. [254] came up with a distributed logging protocol for PMEM. Similarly, Arulraj and Pavlo adapted different parts of a database engine to PMEM, such as the buffer manager [273], and the storage and recovery [87, 85] protocol. With their findings, they developed a DBMS, Peloton [221, 69], built to leverage PMEM. Similarly, SAP HANA adapted their engine to integrate PMEM [78], Alibaba developed their own durable PMEM database [141] and Liu et al. [200] built a log-free OLTP engine for PMEM.

Van Renen et al. [231] evaluated how DBMSs perform when they use either PMEM exclusively or a hybrid approach with a page-based DRAM cache in front of PMEM. There has also been considerable work on algorithms such as index joins [228], or data structures such as B+-Trees [110, 251, 265, 155, 114], hash tables [201, 275, 121, 135, 212, 136], and other indexes [193, 150, 271] on PMEM.

Most of the above efforts simulate PMEM using software tools [127]. With the introduction of Intel Optane, it is possible to experiment on the actual hardware. This has led to several performance studies. Outside of the database context, Izraelevitz et al. [158] provided the first comprehensive performance measurements of Intel Optane including read/write latency with sequential and random access patterns as well as the effect of factors such as the number of threads on different PMEM modes (Memory-mode and AppDirect-mode). This study was followed by others [223, 264] providing more in-depth analysis and elaborating on the insights of Izraelevitz et al. Xiang et al. [261] study the read/write buffering performance of PMEM and they provide new insights on how read/write buffers are managed. In HPC, a number of studies [220, 257] have shown promising results in hybrid DRAM-NVM configurations.

Database researchers have primarily focused on OLAP workloads. Benson et al. [119, 106] employ microbenchmarks and OLAP benchmarks (Star Schema, TPC-H) to identify best practices for using Intel Optane and assess its viability as an NVMe SSD replacement, exclusively utilizing the AppDirect mode (with and without `fsdax`). Shanbhag et al. [240] evaluate the AppDirect mode in an OLAP context using the SSB benchmark. Wu et al. [260] conduct a brief study running microbenchmarks, TPC-H, and TPC-C on SQLServer 2019 using PMEM in Memory and AppDirect mode (with and without `fsdax`), presenting high-level conclusions on DBMSs leveraging PMEM, primarily related to SQLServer, without experimenting with database knobs or examining how query plans affect PMEM performance. Renen et al. [249] measure Intel Optane Persistent Memory's bandwidth and latency, while tuning log writing and block flushing. Lastly, Benson et al. [98] develop a benchmark framework to evaluate various customizable database-related PMEM access characteristics [98].

Table 4.7 summarizes the landscape of available benchmarking papers, denoting our contributions with the symbol ‡ and prior work with references. We use NVMe for Intel Optane SSD disks and AppDir and Mem rows to represent the corresponding Intel Optane Persistent Memory modes. As shown in the table, we evaluate five DBMSs in both AppDirect and Memory modes using complete benchmarks (TPC-H and TPC-C) rather than microbenchmarks. In TPC-H, we demonstrate how query plans, operations, and system statistics are affected by using PMEM. We also extensively evaluate Memory mode, providing insights on its usefulness with working

Table 4.7: Landscape of Intel Optane benchmark papers in the DB community (our contributions are denoted with ‡)

	PostgreSQL	MySQL	SQLServer	DuckDB	VoltDB	Benchmark
SSD	‡	‡	‡, [260], [259]	‡	-	TPCH
SSD	‡	‡	‡, [260], [259]	-	‡	TPCC
NVMe	[106]	-	[259]	-	-	TPCH
NVMe	-	-	[259]	-	-	TPCC
AppDirect	‡, [106]	‡	‡, [260]	‡	-	TPCH
AppDirect	‡	‡	‡, [260]	-	‡	TPCC
Memory	‡	‡	‡, [260]	‡	-	TPCH
Memory	‡	‡	‡, [260]	-	‡	TPCC

sets larger than DRAM. We use both row and column stores to show how memory accesses and compression affect Intel Optane. In TPC-C, we explore the impact of database knobs. Furthermore, we are the only ones benchmarking a key-value store (RocksDB) under varying read-write workloads.

4.8 Conclusion

We evaluated PMEM on various engines and popular benchmarks (TPC-H, TPC-C, and YCSB). Our study sheds light on how to efficiently integrate PMEM into the memory hierarchy and how to tune relational and non-relational systems to get the best performance out of each configuration, given the higher cost both in terms of storage and CPU. In Memory mode, increasing volatile memory capacity using PMEM does not offer any performance advantage. In AppDirect mode, PMEM offers a lower latency than SSDs, which can increase performance significantly when I/O is in the critical path. However, when there is a lot of resource contention or mixed workloads and because the I/O path is not fully optimized in the case of PMEM, SSDs still remain a competitive solution. In the rest of the section we summarize the main insights from our study and we present future directions of our research.

4.8.1 Insights

(i) Memory mode does not offer a significant performance advantage: As we observe for all engines and benchmarks, in the vast majority of cases, Memory mode with its larger volatile

memory capacity does not offer any performance advantage. There are even cases, where performance is slightly worse. This happens due to the conflict cache misses in the DRAM L4 cache, which also causes additional memory traffic between PMEM and DRAM. The only case where Memory mode is useful is for queries with sequential accesses and low number of reads/writes to memory. That happens because the DBMS can take advantage of the larger OS page cache offered by the larger memory capacity and together with prefetching, these queries can have a small speedup compared to configurations that do not use PMEM at all. Nevertheless, it is worth noting that other studies in the HPC context [220] show that a working set considerably larger than typical DRAM capacities, makes the Memory mode useful.

(ii) Performance gains when using PMEM vs. SSDs can be due to application limitations rather than differences in hardware: Although PMEM has superior performance compared to SSDs for sequential read accesses with many threads, this is not the same for random accesses. If applications adopt asynchronous I/O for random accesses (e.g., index lookups) for workloads in which CPU requests overshadow I/O, the Default and the AppDirect mode will have a very small performance difference.

(iii) The lower latency of PMEM in AppDirect mode does not translate to an advantage equal to the hardware characteristics: Our microbenchmarks reveal PMEM has 30x lower latency than SSDs and, in some cases, over 10x higher read and write bandwidth. However, this difference does not result in the expected runtime reduction, as the I/O path is not fully optimized. In AppDirect mode with `fsdax` (recommended by Intel), there is no OS page cache, and DMA is unavailable since PMEM is managed by the iMC, not as a peripheral device. Consequently, CPU resources are spent on I/O in AppDirect mode, while in Default mode, DMA and the OS page cache can cover part of the lower latency, especially in CPU-intensive, I/O-heavy queries or in cases involving secondary indexes with table data stored in the OS page cache.

(iv) In systems where resources are limited, PMEM in AppDirect mode is not as useful: In general PMEM in AppDirect mode involves the CPU as no DMA is available. When there is a lot of resource contention, the hardware advantage of PMEM is almost negligible. We notice this behaviour in various DBMSs (in MySQL for TPC-H, when we restricted PostgreSQL to one core, and in SQLServer and VoltDB for TPC-C).

(v) PMEM requires fine-tuning for write or mixed workloads: As noticed by previous studies [119, 158] heavy-write workloads and read/write interference decrease PMEM's performance dramatically. We notice the same behaviour when running TPC-C or when we use Workload A and B in the YCSB. Even a small number of writes with a high thread count in mixed workloads can hinder the performance of PMEM significantly. Thus, the number of

(write) threads has to be carefully tuned to avoid interference. Especially in a DBMS context, several configurations that increase additional writes should be turned off to increase performance (e.g., the double-write buffer).
(vi) Optimizations made for SSDs/HDDs need to be re-designed when using PMEM: Many traditional optimizations avoid I/O as much as possible due to the latency gap between DRAM and storage (e.g. log compression). However, this gap is not as large with PMEM and these optimizations may not offer any performance advantage. In general, as the CPU is involved in I/O in AppDirect mode, it is preferable to avoid devoting CPU resources to optimize I/O.
(vii) Log placement in PMEM does not increase performance significantly: Across DBMSs, placing only the log in PMEM does not increase the transaction rate for TPC-C significantly. Placing data or both the log and data in PMEM increases throughout for TPC-C due to the lower latency and higher bandwidth of PMEM compared to SSDs/HDDs.

4.8.2 Future directions

As of July 2022, Intel decided to discontinue the Optane series [46]. Although DRAM may reach capacities up to 512GB with DDR5, those will come with a significant cost. Therefore, there is still a need for an intermediate storage tier between DRAM and SSDs. That gap can be filled with CXL [45], which is a cache-coherent interface for connecting CPUs, memory and accelerators. CXL-attached memory can increase the memory capacity of servers, offering a fast storage memory tier to place "hot" data that does not fit in DRAM. Several characteristics of PMEM are relevant to CXL-attached memory, such as byte-addressability and near-DRAM performance with higher capacity and opportunities for higher energy efficiency. Especially the last point will be highly valuable in the context of data centers, where CXL will be deployed first. Therefore our study provides insights on what system function (e.g., pre-fetching, low CPU utilization, query optimization hints, etc.) are valuable in the database context, to avoid performance regressions.

Chapter 5

Adaptive storage layout for QaaS

5.1 Motivation

Data management and analysis has changed rapidly over the last few years, influenced by both the increasing amount of data produced every day and the effect that Machine and Deep Learning (ML/DL) had over data processing. This evolution is further exacerbated by the prevalent use of semi-structured data (e.g. CSV, JSON, Parquet), which are very popular both for the scientific domain and for ML datasets [182, 139, 58, 59]. Especially CSV offers simplicity and aligns well with the flexible nature of ML/DL algorithms, but on the other hand lacks complex schemas and data structures. Organizations, influenced by the evolving landscape, find themselves storing an increasing portion of their data in CSV, which they need to analyze, get insights from, and use to their benefit. To do so, they can use cloud data warehouses or analytics platforms (e.g. Databricks, Snowflake) or QaaS systems (e.g. Amazon Athena, Google Big Query). Data warehouses are a very good fit for data that need to be continuously processed and maintained. However, there are many use cases that produce data that will only be used and processed a handful of times or even just once. Take for example a particle physicist producing insights on a single experiment for High-Energy Physics [146]. For these sparse, infrequent workloads, maintaining an always-on infrastructure is not efficient, and QaaS are an ideal solution. By declaring semi-structured data as external tables, users can query them and extract the necessary information while only paying for the amount of data they scanned and without the need to worry about scaling the infrastructure or optimizing the input data layouts.

	CSV		Parquet	
	Runtime[s]	Cost[$]	Runtime[s]	Cost[$]
BigQuery	227	9.05	71.5	2.11
Athena	273.3	9.09	132.3	0.76

Table 5.1: Runtime and cost of running TPC-H SF-100 on Parquet and CSV for Amazon Athena and Google BigQuery

Between the two extremes of data warehousing and QaaS, there are many opportunities for improvement. For example, it is widely known that transforming the input data to a columnar format (e.g. Apache Parquet) can have a huge impact on the execution time and cost of QaaS. Consider the cost and runtime for TPC-H SF-100 on two popular QaaS (Amazon Athena, Big Query) in Table 5.1. We observe that the runtime can be reduced more than 3x and the cost by an order of magnitude just by making this simple transformation. Nevertheless, for sporadic execution, converting the whole dataset to Parquet might be ineffective in terms of cost and/or latency.

Hence, in this chapter we present our vision for *serverless cracking*, a modern form of database cracking for QaaS. Serverless cracking relies on simple transformations (e.g. transcoding, sorting, chunking) that have a large impact on the cost and/or execution time of QaaS. Serverless cracking is based on FaaS, as serverless functions are a good fit for occasional, bursty workloads [209]. In this way QaaS could invoke FaaS only when additional budget is given by the customer such that it does not interfere with query processing. To understand the various trade-offs, we use Amazon Athena and AWS Lambda as a use case for our transformations. Based on our observations, we provide actionable advice on where and which transformation pays off and how users can reduce their cost and latency for their sporadic workloads. Our insights are easily transferable between different cloud providers and also pave the way to practical storage optimization in the cloud, an area that is becoming increasingly important through data lakehouses [81, 245, 248]. Finally, we sketch future directions on how serverless cracking can be integrated in the cloud as part of QaaS and what are the challenges there. For example, cloud providers can offer a cost/runtime advisor that can help clients to perform incremental transformations based on their own needs.

5.2 Background

In this section we give an overview of the technologies that we use in the chapter. We start with Apache Parquet, a popular columnar storage format and then present QaaS and serverless functions.

5.2.1 Apache Parquet

Apachet Parquet [62] is an open-source columnar storage file format, designed to improve storage and access efficiency of tabular data compared to row-based files like CSV. Internally, Parquet files are divided into smaller units called row groups. Each row group contains a subset of the data and within a row group, data are organized into column chunks. Each column chunk is divided into pages, which include data for a specific column in a specific row group. Finally, each file has footer metadata that contain information about row groups and columns chunks (e.g. data types, total row group size, encodings, min-max indexes). The columnar storage, together with the footer metadata allow for numerous compression techniques such as dictionary or run-length encoding. Additionally, using metadata, query engines can apply various optimizations (e.g. predicate pushdown, bloom filters, column statistics) to skip reading irrelevant row groups based on query predicates, therefore improving query performance. All these features, together with its wide tooling and ecosystem support and the fact that it is language-agnostic have made Parquet the backbone of lakehouse formats, such as Deltalake [65] and Apache Iceberg [61].

5.2.2 Query-as-a-Service (QaaS)

Query-as-a-Service (QaaS) systems are a new wave of cloud computing systems that allow users to execute SQL or SQL-like queries directly on data stored in the cloud. They offer significant advantages compared to traditional databases, as they are able to query datasets on demand through external tables without the need to ingest the input for creating tables. Additionally, there is no need to manage the underlying infrastructure. Popular examples of such systems are Amazon Athena [60] and Google BigQuery [66]. QaaS support a variety of data formats including CSV, JSON, Avro, and Parquet. One of their biggest benefits is that they only charge the client based on the amount of data scanned of individual queries. However, when working on external data, they are lacking other traditional optimizations (e.g. join reordering).

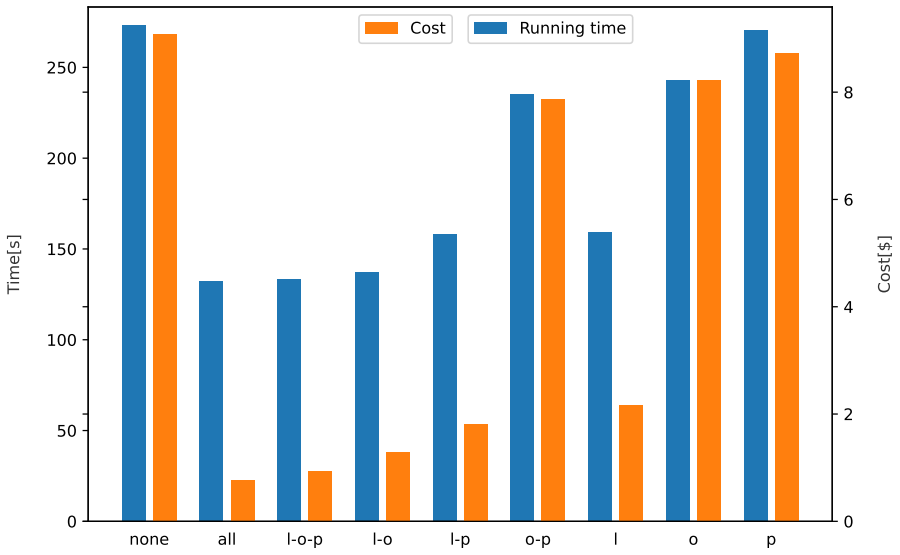


Figure 5.1: Runtime and cost of running TPC-H on Amazon Athena with different tables on Parquet

5.2.3 Serverless functions

Serverless functions, also known as Function-as-a-Service (FaaS) are a cloud execution model where cloud providers dynamically manage the allocation and provisioning of servers. They are called serverless as managing the instrastructure is abstracted away from developers, who only need to focus on writing the functions that comprise the application code. The functions are executed in response to events (e.g. HTTP requests, file uploads to cloud storage, messages from a queue). Popular examples of serverless functions are AWS Lambda [63], Azure Functions [64], and Google Cloud Functions [67]. FaaS offer a number of advantages for sparse, bursty workloads. First of all, they are on a pay-per-use base, meaning that users only pay for the compute time they consume. They are also highly scalable, reaching up to a few thousand requests per second, while keeping a simple and quick deployment. Additionally, they are especially useful for data-independent tasks that can start up and shut down quickly and run on an infrequent basis [209].

5.3 Data transformations

To quantify the potential of serverless cracking, we study three different transformations (transcoding, chunking, and sorting) and how they affect the overall runtime and cost of QaaS. We an-

alyze the runtime and cost of performing these conversions on serverless functions. We use AWS Athena as an example to run the QaaS workloads and AWS Lambda to run the data transformations. To reduce the variability of serverless function latency, in particular due to cold starts [196, 253, 163], we pre-warm the functions and take the median of 10 runs for each experiment.

5.3.1 Transcoding to columnar format

Conversion to a columnar format (or transcoding), such as Apache Parquet, is one of the most effective transformations users can perform to reduce cost and latency for QaaS, because Parquet has a number of advantages (Section 5.2.1) compared to CSV or JSON.

Although transcoding is generally beneficial, for sparsely-running workloads, it is not obvious which tables a user should convert to Parquet to maximize its benefit. To understand the various trade-offs, we run TPC-H SF-100 with 300 files per table on Amazon Athena. We transform combinations of tables in Parquet and we show the execution time and cost in Figure 5.1. For the transformation, we pick either all the tables or a combination of the three bigger tables of the workload (`lineitem`, `orders`, and `partsupp`), which constitute 97% of the total dataset size. Transforming all data to Parquet, reduces the overall runtime in half, and the total cost around 10x. Since the first three tables make up almost the entire dataset, transforming them brings the cost and the runtime very close to having all data on Parquet.

We now experiment with transcoding combinations of two tables from `lineitem`, `orders`, and `partsupp`. Transforming `lineitem` and `orders` is very similar to transforming all three tables, whereas converting `lineitem` and `partsupp` has both higher runtime and cost, respectively 20s and 0.5$. This increase is much larger proportionally than the 5GB difference between `orders` (17GB) and `partsupp` (12GB). We make similar observations when we convert only `orders` and `partsupp` or individual tables in the right side of the figure. Finally, we observe the impact of only changing `lineitem`. The table is 70% of the workload but having it in Parquet reduces the runtime almost 40% and the cost more than 4x. Overall, the size of a table is only one of the factors that influence the runtime and cost of the workload. There are other aspects that affect these quantities, such at the frequency that each table appears in the query set, and the projection and the selection predicates of each query.

To understand how expensive and time consuming the transcoding of the tables is, we show the runtime and cost of converting the same table combinations as in the previous experiment

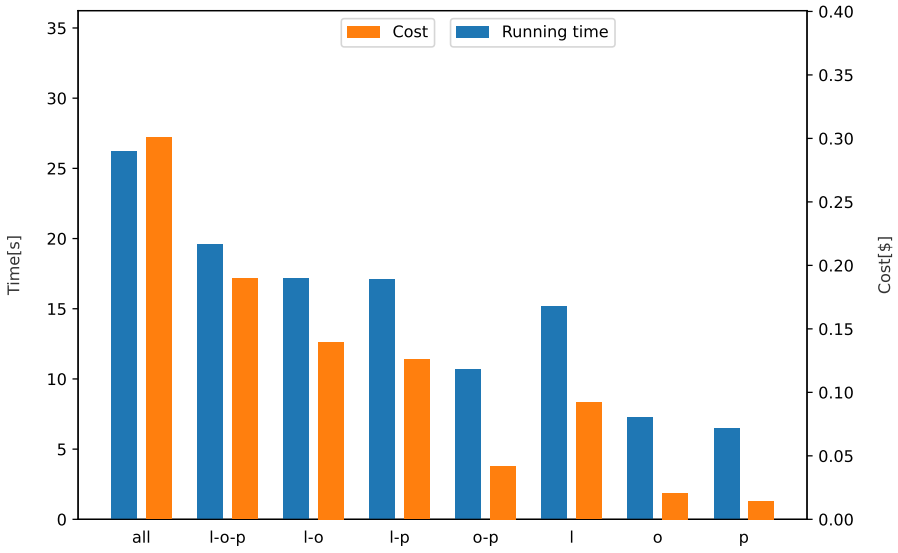


Figure 5.2: Runtime and cost of transcoding different tables on Amazon Lambda

to Parquet in Figure 5.2. Every table is split across 300 files of equal size, and we invoke one function per file for the conversion using a large degree of parallelism for the function invocation.

The experiment mirrors up to a degree what we noticed previously. First, we observe a sub-linear behavior, depending on the table size, both for latency and cost. That comes from the fixed "setup" cost (e.g. invocation of functions, time until a function starts etc.), which is independent from the input size. Additionally, converting `lineitem` with other table combinations takes roughly the same time as converting `lineitem` alone, although the cost increases as we convert more tables. That indicates that the transformation of a large table dominates the running time and it is the last step that completes when the overall number of function invocations is not very large (up to a 1000). On the other hand, when we convert all tables to Parquet, the overall runtime and cost increase significantly, in a disproportionate way compared to the additional data transformed. In particular, the rest of the tables are only 5GB, but the overall increase both in runtime and cost is more than 30%. That shows the influence of the setup cost in the overall runtime as well as the impracticality of always transforming everything into a more suitable format as it may not be cost effective, especially for large data amounts.

5.3.2 Splitting by size (chunking)

Changing the number of files a table consists of is another effective transformation that influences the latency but not the cost of a workload. However, it is not obvious what partition

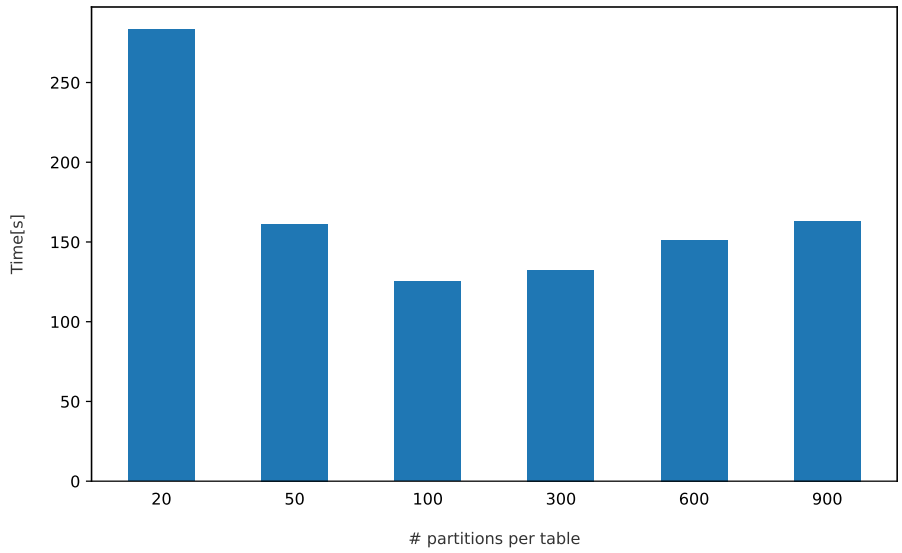


Figure 5.3: Runtime of running TPC-H on Amazon Athena with different number of parquet partitions per table

file size to choose because cloud providers do not provide explicit instructions. For example, Athena's documentation says that "datasets that consist of many small files result in poor overall query performance"[1] but does not give indications on what the file size should be.

The performance drop that Athena mentions happens because, first, QaaS need to keep a list of all the partition locations and therefore with an increasing number of partitions the listing time increases and, second, because QaaS usually schedule the number of tasks based on the number of partitions. If there are too many small files, each task has minimal work and most of the time is spent communicating across different tasks in exchange phases. However, a few large files can also be detrimental because not enough tasks are scheduled and the overall CPU power is not sufficient.

To understand the file size effect, we alter the number of partitions a table comprises of from 20 to 900 using CSV as the input format. In a plot not shown, we see that the number of partitions on CSV has no influence on the execution time. Because the actual implementation of Athena is not open source, we can only speculate about why parallelism does not increase with higher number of partitions on CSV. When observing the query plans, we notice that the number of tasks stays the same independently of the number of partitions and is also quite high. Therefore,

[1] https://docs.aws.amazon.com/athena/latest/ug/performance-tuning.html

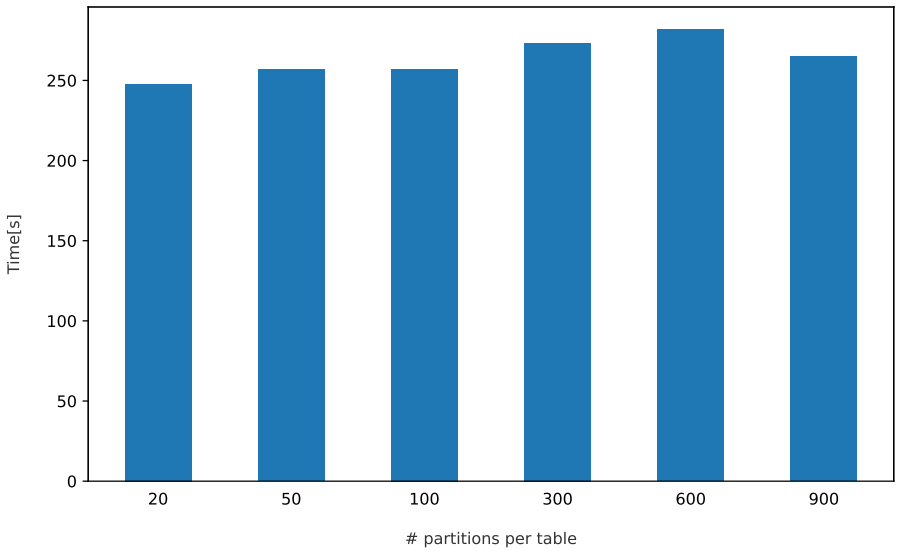


Figure 5.4: Runtime of running TPC-H on Amazon Athena with different number of csv partitions per table

a plausible explanation is that Athena sets a high degree of parallelism directly when reading CSV and splits the data across the workers with a mechanism that is independent of the number of files.

In contrast, the number of Parquet files has a big influence on the query runtime. We observe this in Figure 5.3, where we run TPC-H SF 100 with an increasing number of Parquet files per table. The plot has a U-shape: If we have very small or very large number of files, the performance is not optimal. Especially, for small numbers of files (e.g., 20 files per table), the overall runtime is even larger than running the same workload on CSV. The optimal performance occurs with 100 partitions per table. After that, increasing the number of partitions increases gradually the execution time. We look again into the query plans to understand why this happens. We observe that Athena sublinearly increases the number of tasks as the number of partitions increases. For small a number of partitions, based on the CPU time, there is not enough parallelism to process data as fast as possible, whereas for large number of partitions, the CPUs are underutilized.

We repeat a similar experiment to Section 5.3.1. Specifically, we run the benchmark using 20 partitions per table while we use 100 chunks for the tables denoted in the x-axis in Figure 5.5. We pick either all tables or combinations of the three largest tables. We notice that chunking `orders`, `partsupp`, or both at the same time gives a marginal or even no performance advantage, especially when taking into account their size in the overall benchmark (around 25%). On

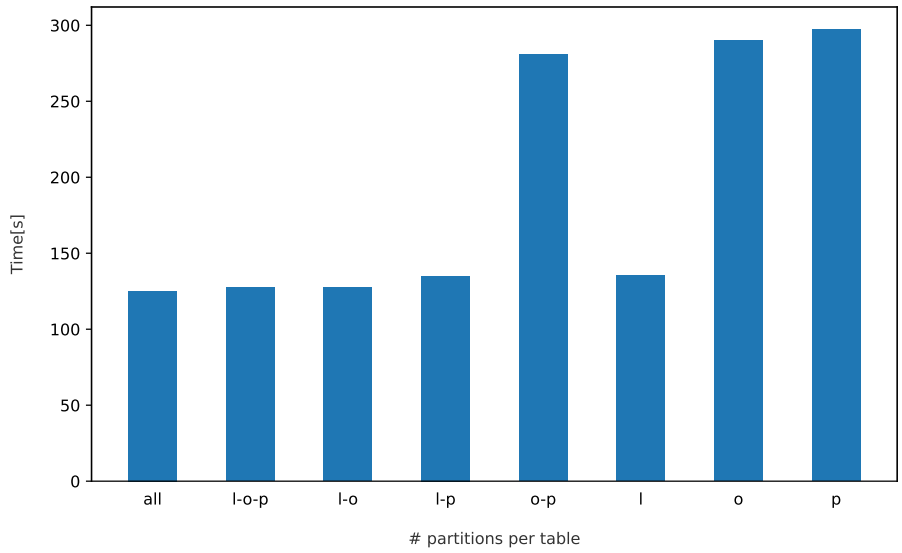


Figure 5.5: Runtime of running TPC-H on Amazon Athena when having different number of parquet partitions in a subset of tables

the other hand, transforming only `lineitem` is 8% slower than splitting all the tables into 100 partitions and dividing more tables other than that gives marginal increases, if any. The rationale for that is that the number of tasks that can be scheduled for a particular table increases for tables like `lineitem` but not for small tables like `partsupp`.

To gain insights into how time and cost-effective chunking is, we use serverless workers to divide an individual Parquet file into smaller ones for various TPC-H tables. We use one function per input file, load the file gradually from S3, and transform it in batches to be as memory efficient as possible. We present the results in Figure 5.6. We see that the overall cost is much lower than if we convert files from CSV to Parquet because we use a smaller number of functions compared to transcoding and we transfer less data from S3 to Lambda due to the compression that Parquet has.

A second key observation is that any transformation involving `lineitem` has roughly the same execution time, except for partitioning all the tables. The difference between splitting `lineitem` alone or together with other tables comes from the natural variation of cloud environments. Ideally, all these experiments should take exactly the same time because `lineitem` is the last table to finish transforming. Using this, we can partition a couple of additional smaller tables together with a very large table without increasing the overall execution time and with minimal additional cost. Finally, transforming `orders` and `partsupp` separately or together, has exactly

the same runtime and a negligible difference in cost. This occurs because, for transformation of small tables and for a small number of files, the setup cost has minor duration compared to the actual work the function does.

5.3.3 Sorting a selection column (sorting)

As a final transformation, we explore sorting Parquet files based on a column: each FaaS invocation reads the Parquet file it gets assigned, sorts the input by the sorting column, and writes the result into a new Parquet file. This improves compression and makes min/max indexing more effective. Sorting does not offer any performance advantage on semi-structured file formats such as CSV.

We first experiment with isolating `l_shipdate`, the column in `lineitem` that has the most selection predicates. This will give an indication of the largest reduction that we can get since (1) `lineitem` is the largest table and (2) one third of the benchmark discards large amounts of data based on `l_shipdate`. We run these queries on tables split over 20 files with and without the sorted column and present the results in Table 5.2. The queries have a ranging degree of selectivity from 1% to 99%. However, the decrease in runtime and cost is also affected by other factors such as the query plan and the other tables the query scans from. The biggest decrease both for latency and cost is for Queries 6 and 20. Both of these have a 43% cost and latency

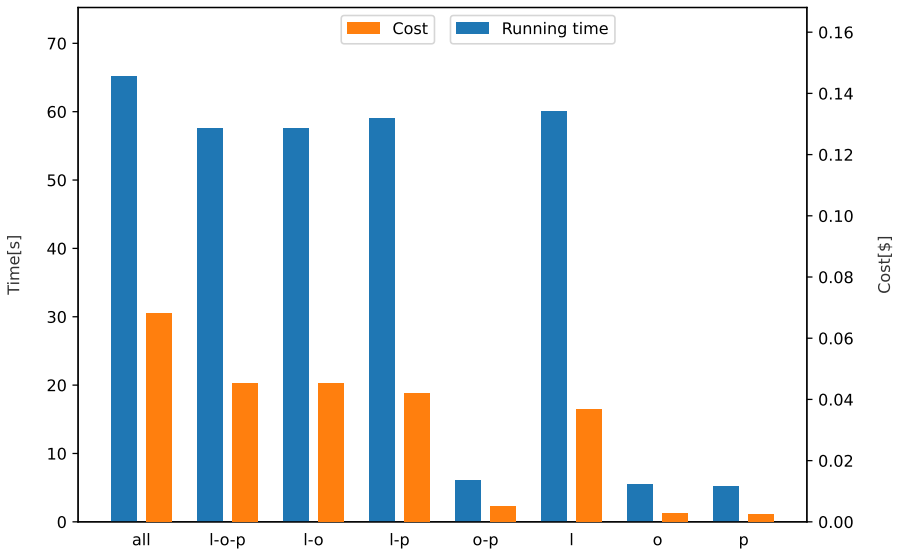


Figure 5.6: Runtime (left axis) and cost (right axis) of splitting tables into smaller files on Amazon Lambda

Table 5.2: TPC-H SF-100 queries with sorted and unsorted selection predicate

Query	Selectivity	No sorting		`l_shipdate` sorting	
		Runtime[s]	Cost[$]	Runtime[s]	Cost[$]
Query 1	1%	23.21	0.026	21.75	0.021
Query 3	46%	12.33	0.033	11.58	0.040
Query 6	85%	9.53	0.023	5.31	0.013
Query 12	51%	14.44	0.021	11.55	0.020
Query 14	99%	11.18	0.037	7.07	0.020
Query 15	96%	12.52	0.070	8.21	0.046
Query 20	84 %	13.37	0.039	7.95	0.022
TPC-H total (common data)	-	283.19	0.718	272.6	0.737
TPC-H total (separate data)	-	283.19	0.718	260.36	0.650

improvement due to less data scanned. In contrast, for Query 3, we notice an interesting trend: although the latency is reduced, the cost increases, meaning that Athena scans more data.

We investigate why this happens. By sorting on a column, we are disrupting the order of other columns. That may lead to unwanted results, e.g. for tables that are sorted on the primary key, which is the case for TPC-H by default. As a consequence, we cannot predict the effect of altering the primary key order on the runtime and cost. To verify this assumption, we run the whole benchmark with the data sorted on `l_shipdate` (Table 5.2). The total runtime is reduced only by 4% and the cost is increased by 3%. To avoid this and to make sure that sorting will have a positive effect, we keep two separate copies of the data: (1) the original data for queries that do not include a selection predicate on the columns that is sorted and (2) the transformed data for the rest of the queries. With this approach, we maximize the expected return. We verify this by running TPC-H with two data copies (Table 5.2). The execution time is reduced by 8% and the total cost is decreased by 10%.

Sorting also interacts with chunking. When we have small files sorted by a previously unsorted column, the value range of each fixed-size row group after sorting is larger. That effectively means larger distance between the min and max values per row group and, hence, less effective min/max filters.

We verify this hypothesis by increasing the number of files and running TPC-H SF 100 with sorted and unsorted data on `l_shipdate` (Figures 5.7 and 5.8). If running with sorted data, we have two separate data copies. We observe similar trends for both plots: As we increase

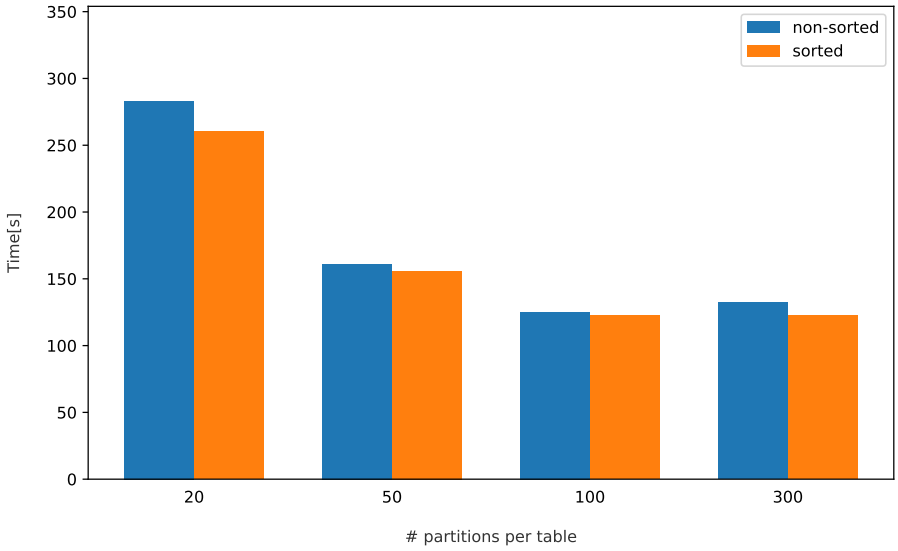


Figure 5.7: Runtime of unsorted vs sorted on `l_shipdate` for different number of partitions on Amazon Athena

the number of partitions, sorting provides a smaller latency/cost reduction. Especially for 100 partitions, the time difference is less than 2%. We conclude that that sorting and chunking are negating each other and that we have to pick only one of them if the input file sizes are large.

Finally, we discuss the cost and runtime of sorting the `l_shipdate` column of `lineitem` using AWS Lambda. While we used serverless functions with 2GB of main memory for the previous transformations, for sorting, this is not sufficient. As every file is around 1.1GB and sorting is memory-intensive, we need to use the lambda functions with the highest memory (10GB). Thus, sorting might not be feasible for every workload, table, and partition size. The total execution time is 63s and the cost is 0.25$. From a first look, sorting does not provide any advantage, given the runtime and cost of transforming the files. While this is the case for TPC-H, the actual advantage might be different for other workloads or queries and depending on how many times we run a workload.

5.4 Serverless cracking

In the previous section, we present insights on the effect that different transformations have on the latency and cost of QaaS. We also give details on how expensive these transformations are, both in terms of cost and time. In this section, we give practical recommendations on how users

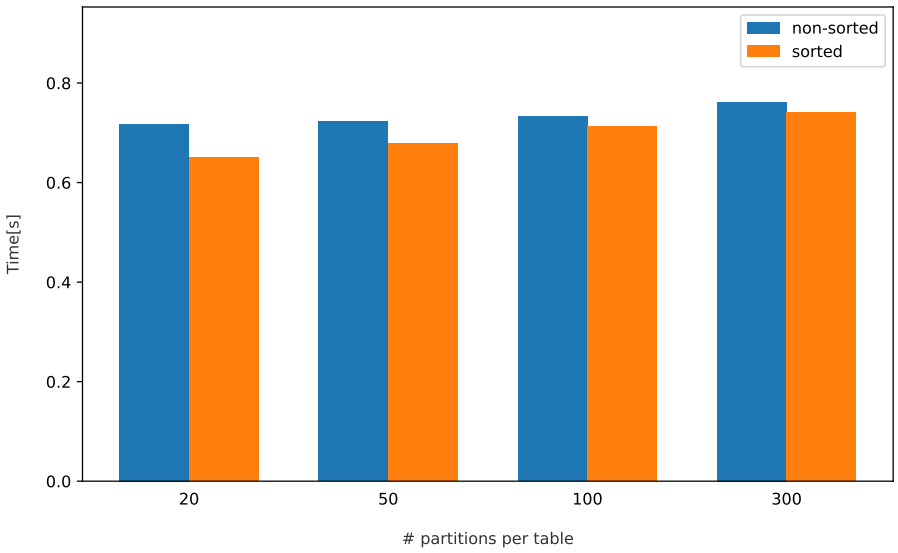


Figure 5.8: Cost of unsorted vs sorted on `l_shipdate` for different number of partitions on Amazon Athena

can (gradually) transform their data to minimize latency and cost for QaaS and sketch an initial architecture that integrates this module into a cloud offering.

5.4.1 Insights

(i) Converting a table to Parquet is a good starting point but not always a solution: As we have seen, converting to Parquet can make a big difference on runtime and improve cost by an order of magnitude. Although converting all the input data to Parquet might not give always such margins, it is a very good starting point for large tables that are used frequently in the query workload, especially if the queries have many projections and predicates with high selectivity (discarding a lot of data). In contrast, for very small or even medium-sized, infrequently used tables (e.g., `partsupp` in TPC-H), converting to Parquet might spend more resources than the expected gain as the transformation on serverless has a fixed setup cost, e.g., spinning the workers and transferring data from/to S3, which increases significantly as we are converting more tables.

(ii) Parquet files should always be less than 200MB: Although chunking does not have any effect on CSV, it can have a very large effect when running QaaS workloads on Parquet. QaaS' parallelism increases with more files. In our experiments, we see that the optimal Parquet size file is around 200MB and splitting below that does not offer any advantages. Given that chunk-

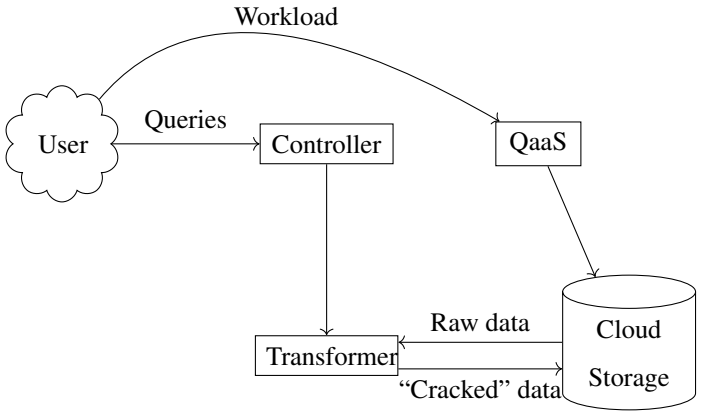


Figure 5.9: System architecture

ing has a very low cost on AWS Lambda, it is beneficial to devote a very small budget to lower the overall latency.

(iii) Sorting by a column only pays off if the workload has many queries and highly selective predicates: Although sorting Parquet files can, in theory, have a large impact both on latency and cost, in practice we found out that it makes little sense given its relatively high cost. It only pays off for very large files and queries with a very high selectivity. Moreover, given that the additional memory on the serverless workers increases the cost further and that we can get almost the same benefit simply by splitting into smaller files for a considerably lower cost, we consider sorting mainly suitable for workloads with large and highly selective queries.

5.4.2 System architecture

We briefly describe how we imagine serverless cracking will be integrated as a cloud module (Figure 5.9). The user submits a workload, consisting of data and queries to QaaS. Our system has visibility on these queries. After analysis, the controller launches serverless functions through the transformer, which has access to the data through the cloud storage, to perform the required transformations. The "cracked" data are then available to the QaaS for running the initial workload. Subsequent executions of the workload are again follow the same loop.

5.5 Related work

Our vision is inspired by database cracking [156] but has many important differences. First, our approach examines data copying and not only reorganization through different formats and other transformations and it is not necessarily done on a per-query basis but can be also done ahead of time. Additionally, it is tailored for QaaS and not for in-memory databases. Serverless cracking is also similar in spirit with many in-situ and caching efforts [216, 215, 88, 76]. However, all these approaches are based on on-premise, disk-based systems, which consequently do not consider data copies but substituting data, or use different data layouts in caches. Our approach is cloud-based and does not use caching but different data copies. Additionally, data formats like the one proposed by Snowflake [120] or in self-organizing data containers [203] are orthogonal to our work since we do not explore new data structures but instead reorganize data on a per-need basis to take advantage of various QaaS characteristics for querying external data. Systems that use FaaS to do query processing [209, 99, 224] are independent from our approach as we use FaaS not to perform query processing but to lower the runtime and cost for QaaS. Finally, in parallel to our strategy, Boncz et al [103] propose adaptive indexing for query optimization in the cloud given a particular budget and Bang et al [92] suggest to avoid using ML techniques to optimize cloud systems but instead to use systems as oracles and perform convex optimization. Serverless cracking does not necessarily build indexes but performs data reorganization tailored for QaaS infrastructures and could also potentially build on linear/convex optimization to investigate which transformations are feasible and worth given a particular cost/latency budget.

5.6 Conclusion

In this chapter, we introduce the notion of serverless cracking, a form of database cracking for QaaS. We show how the end user can leverage simple transformations to reduce the cost and latency of infrequent workloads that run on QaaS, leveraging FaaS infrastructure. This opens a few exciting avenues for future work.

"Infinite"-resource cloud. Compared to traditional database deployments, the cloud has "unlimited" resources in terms of compute and storage. Although in our study, which uses FaaS, we are minimizing cost and chose the workers with the least amount of memory with the task given, there are many potential optimizations that can speed up transformation latency with additional

cost [168]. For tasks that are not data-parallel and require communication, we could potentially explore function communication [256] or using VMs as a coordinator [177]. This would allow exchanging data across multiple files that are being sorted and thus increase the effectiveness of our current per-file sorting to a point where it is actually beneficial. In terms of storage, databases are usually limited by the amount of system memory, which is not the case for the cloud. Theoretically, we could keep a large amount of copies tailored to minimize runtime and cost of a specific queries inside a workload. However, this would become infeasible in terms of cost and version management. Although storage solutions like S3 are relatively cheap, keeping multiple copies of the same dataset and querying them can increase the cost significantly. Deltalake and Iceberg have versioning that could potentially help with the organization but this versioning tracks data evolution and it is not about different views of the same dataset.

Workload-instance advisor. Based on our study, we can develop an advisor to minimize workload latency given a particular cost budget. The advisor will be part of the controller in the system architecture of Figure 5.9. Besides the queries, it can take as input the queries of the workload and metadata about the dataset (e.g., data types) and use optimization algorithms or ML techniques to propose transformations. This type of advisor comes with various advantages but also a number of challenges. The cloud has a lot of variability due to tail latencies. It is also very difficult to understand how systems like Athena or BigQuery work without having access to the internals. However, this type of advisor is also data-agnostic as it does not need access to the actual data to calculate the transformations. It is also very easily extensible to more transformations that have impact on QaaS and it is transferable between cloud deployments if we can re-calibrate our method on different providers. Although in our work, we mainly focused on CSV and Parquet, the advisor could also work with other unstructured data formats (e.g. JSON, Avro).

Incremental transformations. In database cracking, the data is continuously being transformed as queries are coming. Should transformations be applied a per query or per batch of queries? This is still an open question. We made the implicit assumption that every workload is run only once but the transformations applied could be very different if a workload has many queries or is executed multiple times. The same question applies on how long we have to keep the different data versions. Is the five-minute rule [80] going to apply here as well? Given that we have transcoded a table to Parquet, do we have to keep the original CSV version? We imagine that serverless cracking would work on a query-batch basis to alleviate for the FaaS setup cost. Periodically, we will need to clean up our storage by deleting unnecessary dataset versions.

CHAPTER 6

CONCLUSION

This chapter concludes the thesis by summarizing the main findings and their implications. It also outlines future directions.

6.1 Summary and implications

In this thesis, we took a step forward in understanding how analytical database engines can adapt to the quickly changing landscape of hardware and software. In the first part, we explored the correct software abstraction for this step, utilizing both traditional relational operators at a finer granularity and also tensor computing runtimes. In the second part, we focused more on storage and we explored both NVMe and also adaptive data reorganization as a solution to the I/O bottleneck.

Chapter 2 introduces the notion of *sub-operators*, which are traditional operators at a finer granularity and builds Modularis, an analytical query engine around them. The resulting system shows competitive performance against mature systems both installed locally and on the cloud, while vastly reducing the development effort.

While using a different software abstraction for relational operators is indeed feasible, a completely different path is to utilize tensor computing runtimes like PyTorch for graph and relational workloads. Therefore, in Chapter 3, we show how the tensor abstraction is not limited to just DL and ML workloads but can be extended to more general data processing. Our implementations outperform custom solutions by orders of magnitude in some cases, proving that tensors is another software granularity, suitable for data analysis.

While Chapter 2 and Chapter 3 focused on the software abstraction side, in the rest of the thesis, we investigate solutions to the I/O pressure, one of the biggest bottlenecks that database systems traditionally face.

Chapter 4 investigates if Intel Optane Persistent Memory can operate as a drop-in storage replacement for databases. Our results are negative, showing that the commercial hardware implementation sometimes is even slower than cheaper and older storage hardware. We showcase this with numerous experiments both on OLAP and OLTP workloads for a variety of systems. Furthermore, we summarize our main findings and advice on how to best use PMEM to get the most out of it.

Finally, Chapter 5 tackles the lack of traditional data structures on QaaS, by introducing serverless cracking. Serverless cracking uses FaaS to perform simple data transformations to reduce the latency and cost of infrequent workloads of QaaS. We give actionable recommendations on how users can take advantage and transform their input data based on their budget and runtime restrictions.

Overall, this thesis shows how database engines can adapt to the newest trends both on the hardware and software front, making them a first-class citizen for data processing.

6.2 Future Directions

Modularis extensions. In Chapter 2, we show how the sub-operator approach produces promising result for relational analytics. A natural extension of this is to cover more ground in the analytics space, e.g. by expanding Modularis in the ML space. This would require the introduction of additional data processing operators, and the incorporation of more elaborate query compiler rules. To facilate the analytics unification, we could use compiler infrastructure like MLIR [188]. MLIR could make the compiler that Modularis uses in the backend more mature and advanced and also extend Modularis to arithmetic computations. Finally, we can explore using Farview [181] as an alternative smart storage engine instead of S3Select. By using the FPGA as a smart buffer cache, we could offload simple selections, projections and aggregations, thereby transferring a decreased amount of data to the main engine for further query processing.

CXL as a memory pool for data processing. In Chapter 4, we observed the disadvantages of Optane and we also gave light to the future of the memory hierarchy through CXL after the Optane discontinuation. As CXL is a relatively new technology and the standard is not completely finalized or brought to production through a commercial implementation, there are many open

questions. For example, how would similar technologies to Optane (e.g. flash memory) be incorporated in the memory hierarchy? Also, several of the characteristics that we found missing in Optane (e.g. hardware prefetching) could be relevant to CXL, especially as the memory pool would be larger there. Although, there has been initial work in this area [195], we believe that CXL would make scale-up databases increasingly relevant [192].

Data-layout advisor for cloud databases. In Chapter 5, we gave initial results on workload-instance data layout optimization for QaaS, using Faas infrastructure. This paves the way for an advisor that will be plugged as a module on OLAP engines in the cloud with the intend to transform data based on monetary and/or latency constraints. As data management in the cloud transitions more and more to semi-structured data and datalakes [81], and workloads require more on-the-fly computations and analysis, instance-optimized advisors become more relevant. The advisor could operate using either mathematical optimization, or it could be based on the many ML techniques applied to database components [204, 184, 185, 205] over the last few years.

List of Figures

List of Tables

BIBLIOGRAPHY

[1] Infiniband™ Architecture Volume 1 and Volume 2. https://www.infinibandta.org/ibta-specifications-download/.

[2] Infiniband™architecture specification release 1.2.1 annex a16: Roce. https://cw.infinibandta.org/document/dl/7148.

[3] MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. http://mvapich.cse.ohio-state.edu/.

[4] Open mpi: Open source high performance computing. https://www.open-mpi.org/.

[5] 2015. H2O Algorithms Roadmap. https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/product/flow/images/H2O-Algorithms-Road-Map.pdf.

[6] 2018. CNTK. https://docs.microsoft.com/en-us/cognitive-toolkit/.

[7] 2018. MXNet. https://mxnet.apache.org/.

[8] 2018. TensorFlow. https://www.tensorflow.org.

[9] 2019. RAPIDS Forest Inference Library. https://medium.com/rapids-ai/rapids-forest-inference-library-prediction-at-100-million-rows-per-second-19558890bc35.

[10] 2020. Blas. http://www.netlib.org/blas/.

[11] 2020. BlazingSQL. https://blazingsql.com/.

[12] 2020. Cardinality estimation in sqlserver. https://docs.microsoft.com/en-us/sql/relational-databases/performance/cardinality-estimation-sql-server?view=sql-server-ver15.

[13] 2020. Cerebras. https://cerebras.net/.

[14] 2020. Cerebras Software. https://cerebras.net/product/#software.

[15] 2020. cugraph. https://github.com/rapidsai/cugraph.

[16] 2020. Distinct using hashing. https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query?view=sql-server-ver15.

[17] 2020. Graphblas. https://people.engr.tamu.edu/davis/GraphBLAS.html.

[18] 2020. GraphCore. https://www.graphcore.ai/.

[19] 2020. Intel MKL. https://software.intel.com/content/www/us/en/develop/documentation/mkl-developer-reference-c/top/blas-and-sparse-blas-routines.html.

[20] 2020. Neo4j. https://neo4j.com/.

[21] 2020. OmnisciDB. https://www.omnisci.com/.

[22] 2020. ONNX. https://github.com/onnx/onnx/blob/master/docs/Operators.md.

[23] 2020. Opencl. https://www.khronos.org/opencl/.

[24] 2020. PyTorch scatter with different reduce modes. https://github.com/pytorch/pytorch/issues/22378.

[25] 2020. Sambanova: Massive Models for Everyone. https://sambanova.ai/.

[26] 2020. Scipy. https://github.com/scipy/scipy/.

[27] 2020. Spark-RAPIDS. https://nvidia.github.io/spark-rapids/.

[28] 2020. Statistics in sqlserver. https://docs.microsoft.com/en-us/sql/relational-databases/statistics/statistics?view=sql-server-ver15.

[29] 2020. Xilinx vitis platform. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[30] 2021. The future of sparsity in deep neural networks. https://www.sigarch.org/the-future-of-sparsity-in-deep-neural-networks/.

[31] 2024. Cost of training GPT-3. https://bit.ly/361o7I0.

[32] 2024. PyTorch Release for IPU. https://medium.com/pytorch/graphcore-announces-production-release-of-pytorch-for-ipu-f1a846de1a2f.

[33] 2024. Tensorflow XLA. https://www.tensorflow.org/xla.

[34] Accessed 2021-10-25. Intel Vtune Platform Profiler. https://www.intel.com/content/dam/develop/external/us/en/documents/vtuneplatformprofilerwhitepaper.pdf.

[35] Accessed 2021-11-01. Faster access to more data. https://www.intel.ca/content/www/ca/en/architecture-and-technology/optane-technology/faster-access-to-more-data-article-brief.html.

[36] Accessed 2021-11-02. InnoDB: Parallel read of index. https://dev.mysql.com/worklog/task/?id=11720#tabs-11720-2.

[37] Accessed 2021-11-04. PGTune. https://pgtune.leopard.in.ua/.

[38] Accessed 2021-11-09. Fsync performance on storage devices. https://www.percona.com/blog/2018/02/08/fsync-performance-storage-devices/.

[39] Accessed 2021-11-09. Mysql 8.0 reference manual: doublewrite buffer. https://dev.mysql.com/doc/refman/8.0/en/innodb-doublewrite-buffer.html.

[40] Accessed 2021-11-09. Mysql 8.0 reference manual: innoDB startup options and system variables. https://dev.mysql.com/doc/refman/8.0/en/innodb-parameters.html#sysvar_innodb_flush_method.

[41] Accessed 2021-11-16. fio - Flexible I/O tester rev. 3.27. https://fio.readthedocs.io/en/latest/fio_doc.html.

[42] Accessed 2021-11-16. Ioping. https://github.com/koct9i/ioping.

[43] Accessed 2021-11-23. Configure persistent memory (PMEM) for SQL Server on Linux. https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-configure-pmem?view=sql-server-ver15.

[44] Accessed 2021-11-25. Performance best practices and configuration guidelines for SQL Server on Linux. https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-performance-best-practices?view=sql-server-ver15.

[45] Accessed 2022-11-22. Compute Express Link. https://www.computeexpresslink.org/.

[46] Accessed 2022-11-22. Discontinuation of Intel Optane. https://www.intel.com/content/www/us/en/support/products/99743/memory-and-storage/intel-optane-memory/intel-optane-memory-series.html.

[47] Accessed 2023-03-22. HammerDB. https://www.hammerdb.com/.

[48] Accessed 2023-03-22. VoltDB-TPCC. https://github.com/VoltDB/voltdb/tree/master/tests/test_apps/tpcc.

[49] Accessed 2024. 53 important statistics about how much data is created every day in 2024. https://financesonline.com/how-much-data-is-created-every-day/.

[50] Accessed 2024. Amazon aqua. https://aws.amazon.com/redshift/features/aqua/.

[51] Accessed 2024. Mutable project. https://bigdata.uni-saarland.de/projects/mutable/.

[52] Accessed 2024. Numba python package. http://numba.pydata.org/.

[53] Accessed 2024. S3select. https://aws.amazon.com/blogs/aws/s3-glacier-select/.

[54] Accessed 2024. Singlestore column-store format. https://docs.memsql.com/v7.1/concepts/columnstore/.

[55] Accessed 2024. Singlestore row-store format. https://docs.memsql.com/v7.1/concepts/rowstore/.

[56] Accessed 2024. tf.data. https://www.tensorflow.org/guide/data.

[57] Accessed 2024. Top 5 real-time data processing trends of 2023. https://www.voltactivedata.com/blog/2023/12/top-5-real-time-data-processing-trends-of-2023/.

[58] Accessed 2024-01-04. CSV explained. https://ai-jobs.net/insights/csv-explained/.

[59] Accessed 2024-01-04. CSV Files: Use cases, Benefits, and Limitations. https://www.oneschema.co/blog/csv-files.

[60] Accessed 2024-01-10. Amazon Athena. https://aws.amazon.com/athena/.

[61] Accessed 2024-01-10. Apache Iceberg. https://iceberg.apache.org/.

[62] Accessed 2024-01-10. Apache Parquet. https://parquet.apache.org/.

[63] Accessed 2024-01-10. AWS Lambda. https://aws.amazon.com/lambda/.

[64] Accessed 2024-01-10. Azure Functions. https://learn.microsoft.com/en-us/azure/azure-functions.

[65] Accessed 2024-01-10. Deltalake. https://delta.io/.

[66] Accessed 2024-01-10. Google BigQuery. https://cloud.google.com/bigquery/.

[67] Accessed 2024-01-10. Google Cloud Functions. https://cloud.google.com/functions/.

[68] Accessed 2024-02-29. Intel®optane©dc persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[69] Accessed 2024-02-29. Peloton. https://pelotondb.io/.

[70] Accessed 24-03-2023. How does the dram caching work in memory mode using intel® optane™ persistent memory? https://www.intel.com/content/www/us/en/support/articles/000055901/memory-and-storage/intel-optane-persistent-memory.html.

[71] Accessed 24-03-2023. Intel® optane™ persistent memory startup guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel_Optane_Persistent_Memory_Start_Up_Guide.pdf.

[72] Accessed 24-03-2023. Intel® optane™ persistent memory startup guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel_Optane_Persistent_Memory_Start_Up_Guide.pdf.

[73] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. In-database learning with sparse tensors. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 325–340.

[74] Sandeep R. Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venka tanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, and Eric Sedlar. 2017. A many-core architecture for in-memory data processing. In *MICRO*.

[75] Zeeshan Ahmed, Saeed Amizadeh, Mikhail Bilenko, Rogan Carr, Wei-Sheng Chin, Yael Dekel, Xavier Dupre, Vadim Eksarevskiy, Senja Filipi, Tom Finley, et al. 2019. Machine learning at Microsoft with ML.NET. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '19, page 2448–2458, New York, NY, USA. Association for Computing Machinery.

[76] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. Nodb in action: adaptive query processing on raw data. *Proceedings of the VLDB Endowment*, 5(12):1942–1945.

[77] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16.

[78] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Suren-

dra Vishnoi, et al. 2017. Sap hana adoption of non-volatile memory. *Proceedings of the VLDB Endowment*, pages 1754–1765.

[79] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. 2018. Sprocket: A serverless video processing framework. In *SoCC*, pages 263–274.

[80] Raja Appuswamy, Renata Borovica, Goetz Graefe, and Anastasia Ailamaki. 2017. The five minute rule thirty years later and its impact on the storage hierarchy. In *Proceedings of the 7th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*, CONF.

[81] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, volume 8.

[82] 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. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA. Association for Computing Machinery.

[83] Joy Arulraj and Andrew Pavlo. 2017. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1753–1758.

[84] Joy Arulraj and Andrew Pavlo. 2019. Non-volatile memory database management systems. *Synthesis Lectures on Data Management*, pages 1–191.

[85] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722.

[86] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. 2019. Multi-tier buffer management and storage system design for non-volatile memory. *arXiv preprint arXiv:1901.10938*.

[87] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind logging. *Proceedings of the VLDB Endowment*, pages 337–348.

[88] Tahir Azim, Manos Karpathiotakis, and Anastasia Ailamaki. 2017. Recache: Reactive caching for fast analytics over heterogeneous data. *Proceedings of the VLDB Endowment*, 11(ARTICLE):324–337.

[89] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96.

[90] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*.

[91] Maximilian Bandle and Jana Giceva. 2021. Database technology for the masses: Sub-operators as first-class entities. *PVLDB*.

[92] Tiemo Bang, Conor Power, Siavash Ameli, Natacha Crooks, and Joseph M. Hellerstein. 2023. Optimizing the cloud? don't train models. build oracles!

[93] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-scale in-memory join processing using rdma. In *SIGMOD*.

[94] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5).

[95] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed join algorithms on thousands of cores. *Proceedings of the VLDB Endowment*, 10(5):517–528.

[96] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2019. Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores. *PVLDB*, 12(13).

[97] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An efficient hybrid pmem-dram key-value store. *Proceedings of the VLDB Endowment*, pages 1544–1556.

[98] Lawrence Benson, Leon Papke, and Tilmann Rabl. 2022. Perma-bench: benchmarking persistent memory access. *Proceedings of the VLDB Endowment*, pages 2463–2476.

[99] Haoqiong Bian, Tiannan Sha, and Anastasia Ailamaki. 2023. Using cloud functions as accelerator for elastic data analytics. *Proc. ACM Manag. Data*, 1(2).

[100] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The end of slow networks: it's time for a redesign. *PVLDB*, 9(7).

[101] Spyros Blanas, Paraschos Koutris, and Anastasios Sidiropoulos. 2020. Topology-aware Parallel Data Processing: Models, Algorithms and Systems at Scale. In *CIDR*.

[102] OpenAI Blog. 2018. AI and Compute. `openai.com/blog/ai-and-compute/`.

[103] Peter Boncz, Yannis Chronis, Jan Finis, Stefan Halfpap, Viktor Leis, Thomas Neumann, Anisoara Nica, Caetano Sauer, Knut Stolze, and Marcin Zukowski. 2023. Spa: Economical and workload-driven indexing for data analytics in the cloud. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 3740–3746. IEEE.

[104] Peter Boncz, Thomas Neumann, and Orri Erling. 2013. Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*.

[105] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237. www.cidrdb.org.

[106] Maximilian Böther, Otto Kißig, Lawrence Benson, and Tilmann Rabl. 2021. Drop it in like it's hot: An analysis of persistent memory as a drop-in replacement for nvme ssds. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*, pages 1–8.

[107] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A serverless framework for end-to-end ml workflows. In *SoCC*, pages 13–24.

[108] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. 2015. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment*, pages 497–508.

[109] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. 2016. Towards linear algebra over normalized data. *arXiv preprint arXiv:1612.07448*.

[110] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, pages 786–797.

[111] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA. ACM.

[112] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 579–594, Berkeley, CA, USA. USENIX Association.

[113] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3393–3404, Red Hook, NY, USA. Curran Associates Inc.

[114] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. Utree: A persistent b+-tree with low tail latency. *Proceedings of the VLDB Endowment*, page 2634–2648.

[115] John Cieslewicz and K.A. Ross. 2007. Adaptive Aggregation on Chip Multiprocessors. In *VLDB*.

[116] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154.

[117] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. 2015. An architecture for compiling udf-centric workflows. *PVLDB*, 8(12).

[118] Wei Cui, Qianxi Zhang, Spyros Blanas, Jesús Camacho-Rodríguez, Brandon Haynes, Yinan Li, Ravi Ramamurthy, Peng Cheng, Rathijit Sen, and Matteo Interlandi. 2023. Query processing on gaming consoles. In *Proceedings of the 19th International Workshop on Data Management on New Hardware*, pages 86–88.

[119] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing persistent memory bandwidth utilization for olap workloads. In *Proceedings of the 2021 International Conference on Management of Data*, pages 339–351.

[120] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226.

[121] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (ATC 2018)*, page 373–385.

[122] Jeffrey Dean and Sanjay Ghemawat. 2008. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

[123] Christopher Dennl, Daniel Ziener, and Jurgen Teich. 2012. On-the-fly composition of fpga-based sql query accelerators using a partially reconfigurable module library. In *FCCM*.

[124] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. 1990. The gamma database machine project. *TKDE*, 2(1).

[125] Jens Dittrich and Joris Nix. 2020. The Case for Deep Query Optimisation. In *CIDR*.

[126] Klaus R Dittrich and Andreas Geppert. 2000. *Component database systems*. Elsevier.

[127] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–15.

[128] Kayhan Dursun, Carsten Binnig, Ugur Cetintemel, Garret Swart, and Weiwei Gong. 2019. A morsel-driven query execution engine for heterogeneous multi-cores. *PVLDB*, 12(12).

[129] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *OSDI*.

[130] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. 2020. In-memory database acceleration on fpgas: a survey. *VLDB J.*, 29(1):33–59.

[131] Yuanwei Fang, Chen Zou, and Andrew A Chien. 2019. Accelerating raw data analysis with the accorda software and hardware architecture. *PVLDB*, 12(11).

[132] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a unified architecture for in-rdbms analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 325–336.

[133] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *USENIX ATC*, pages 475–488.

[134] Philip Werner Frey and Gustavo Alonso. 2009. Minimizing the hidden cost of RDMA. In *ICDCS*.

[135] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. Nvtraverse: In nvram data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*, page 377–392.

[136] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. *Mirror: Making Lock-Free Data Structures Persistent*, page 1218–1232.

[137] Apurva Gandhi, Yuki Asada, Victor Fu, Advitya Gemawat, Lihao Zhang, Rathijit Sen, Carlo Curino, Jesús Camacho-Rodríguez, and Matteo Interlandi. 2022. The tensor data platform: Towards an ai-centric database system. *arXiv preprint arXiv:2211.02753*.

[138] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 249–264.

[139] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative distributed csv data parsing for big data analytics. In *Proceedings of the 2019 International Conference on Management of Data*, pages 883–899.

[140] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2018. Enabling highly scalable remote memory access programming with mpi-3 one sided. *CACM*, 61(10).

[141] Caixin Gong, Chengjin Tian, Zhengheng Wang, Sheng Wang, Xiyu Wang, Qiulei Fu, Wu Qin, Long Qian, Rui Chen, Jiang Qi, et al. 2022. Tair-pmem: a fully durable non-volatile memory database. *Proceedings of the VLDB Endowment*, pages 3346–3358.

[142] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 599–613, USA. USENIX Association.

[143] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[144] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management Naga. In *SIGMOD*.

[145] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processmg System. In *SIGMOD*.

[146] Dan Graur, Ingo Müller, Mason Proffitt, Ghislain Fourny, Gordon T Watts, and Gustavo Alonso. 2023. Evaluating query languages and systems for high-energy physics data. In *Journal of Physics: Conference Series*, volume 2438, page 012034. IOP Publishing.

[147] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational Query Coprocessing on Graphics Processors. *TDS*, 34(4).

[148] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2008. Relational Joins on Graphics Processors. In *SIGMOD*.

[149] Dong He, Supun Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *arXiv preprint arXiv:2203.01877*.

[150] Yuliang He, Duo Lu, Kaisong Huang, and Tianzheng Wang. 2022. Evaluating persistent memory range indexes: Part two. *arXiv preprint arXiv:2201.13047*.

[151] Zhenhao He, David Sidler, Zsolt István, and Gustavo Alonso. 2018. A flexible k-means operator for hybrid databases. In *FPL*.

[152] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6(9):709–720.

[153] Pedro Holanda and Hannes Mühleisen. 2019. Relational queries with a tensor processing unit. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, New York, NY, USA. Association for Computing Machinery.

[154] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. Laradb. *Proceedings of the 4th Algorithms and Systems on MapReduce and Beyond - BeyondMR'17*.

[155] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in Byte-Addressable persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200.

[156] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. 2007. Database cracking. In *CIDR*, volume 7, pages 68–78.

[157] Florian Irmert, Michael Daum, and Klaus Meyer-Wegener. 2008. A new approach to modular database systems. In *Proceedings of the 2008 EDBT workshop on Software engineering for tailor-made data management*, pages 40–44.

[158] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*.

[159] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dongjin Shin, and Byung-Gon Chun. 2019. JANUS: fast and flexible deep learning via symbolic graph execution of imperative programs. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 453–468. USENIX Association.

[160] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA. Association for Computing Machinery.

[161] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. 2016. Yoursql: a high-performance database system leveraging in-storage computing. *PVLDB*, pages 924–935.

[162] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *SoCC*, pages 445–451.

[163] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*.

[164] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760.

[165] Michael Jungmair and Jana Giceva. 2023. Declarative sub-operators for universal data processing. *Proceedings of the VLDB Endowment*, 16(11):3461–3474.

[166] Kaan Kara, Jana Giceva, and Gustavo Alonso. 2017. Fpga-based data partitioning. In *SIGMOD*.

[167] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending relational query processing with ML inference. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org.

[168] Simon Kassing, Ingo Müller, and Gustavo Alonso. 2022. Resource allocation in serverless query processing. *arXiv preprint arXiv:2208.09519*.

[169] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB*, 11(13).

[170] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222.

[171] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222.

[172] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *PVLDB*, 2(2).

[173] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 297–312.

[174] Youngbin Kim and Jimmy Lin. 2018. Serverless data analytics with flint. In *IEEE CLOUD*, pages 451–455.

[175] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29.

[176] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding ephemeral storage for serverless analytics. In *ATC*, pages 789–794.

[177] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444.

[178] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. *PVLDB*, 7(10).

[179] A. Kohn, V. Leis, and T. Neumann. 2018. Adaptive execution of compiled queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 197–208.

[180] André Kohn, Viktor Leis, and Thomas Neumann. 2021. Building advanced sql analytics from low-level plan operators. In *SIGMOD*, pages 1001–1013.

[181] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. 2021. Farview: Disaggregated memory with operator off-loading for database engines. *arXiv preprint arXiv:2106.07102*.

[182] Jan Kossmann, Daniel Lindner, Felix Naumann, and Thorsten Papenbrock. 2022. Workload-driven, lazy discovery of data dependencies for query optimization. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[183] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. A relational approach to the compilation of sparse matrix programs. Technical report, USA.

[184] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2021. Sagedb: A learned database system.

[185] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*, pages 489–504.

[186] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. 2019. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 317–330.

[187] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. 2019. Gpop: A cache and memory-efficient framework for graph processing over partitions. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 393–394.

[188] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE.

[189] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *arXiv e-prints*, page arXiv:2002.11054.

[190] Jyoti Leeka and Kaushik Rajan. 2019. Incorporating super-operators in big-data query optimizers. *PVLDB*, 13(3).

[191] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *SIGMOD*.

[192] Alberto Lerner and Gustavo Alonso. 2024. Cxl and the return of scale-up database engines. *arXiv preprint arXiv:2401.01150*.

[193] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment*, pages 574–587.

[194] Feng Li, Sudipto Das, Manoj Syamala, and Vivek Narasayya. 2016. Accelerating relational databases by leveraging remote memory and rdma. In *SIGMOD*.

[195] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms.

[196] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2022. Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84.

[197] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news*, 37(3):267–278.

[198] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE.

[199] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2017. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. In *EuroSys*.

[200] Gang Liu, Leying Chen, and Shimin Chen. 2021. Zen: a high-throughput log-free oltp engine for non-volatile main memory. *Proceedings of the VLDB Endowment*, pages 835–848.

[201] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302*.

[202] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric nvm architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 757–773.

[203] Samuel Madden, Jialin Ding, Tim Kraska, Sivaprasad Sudhir, David Cohen, Timothy Mattson, and Nesime Tatbul. 2022. Self-organizing data containers. *Memory*, 1:2.

[204] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1275–1288.

[205] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*.

[206] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland. USENIX Association.

[207] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7.

[208] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. A hardware-software blueprint for flexible deep learning specialization.

[209] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *SIGMOD*.

[210] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-Efficient Aggregation: Hashing Is Sorting. In *SIGMOD*.

[211] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A tensor compiler for unified machine learning prediction serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 899–917. USENIX Association.

[212] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing (DISC 2017)*, pages 37:1–37:16.

[213] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9).

[214] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550.

[215] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. 2017. Slalom: Coasting through raw data via adaptive partitioning and indexing. *Proceedings of the VLDB Endowment*, 10(10):1106–1117.

[216] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. 2020. Adaptive partitioning and indexing for in situ query processing. *The VLDB Journal*, 29:569–591.

[217] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab.

[218] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating end-to-end optimization for data analytics applications in weld. *PVLDB*, 11(9).

[219] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. In *NIPS-W*.

[220] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. 2019. Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules. In *Proceedings of the International Symposium on Memory Systems*, pages 288–303.

[221] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-driving database management systems. In *CIDR*, volume 4, page 1.

[222] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830.

[223] Ivy B Peng, Maya B Gokhale, and Eric W Green. 2019. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, pages 304–315.

[224] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A scalable query engine on cloud functions. In *SIGMOD*, pages 131–141.

[225] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - a vector algebra for portable database performance on modern hardware. *Proc. VLDB Endow.*, 9(14):1707–1718.

[226] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *SIGMOD*.

[227] Orestis Polychroniou and Kenneth A. Ross. 2014. A Comprehensive Study of Main-Memory Partitioning and its Application to Large-Scale Comparison- and Radix-Sort. In *SIGMOD*.

[228] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the latency gap between nvm and dram for latency-bound operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–8.

[229] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *NSDI*, pages 193–206.

[230] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530.

[231] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1541–1555.

[232] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-speed query processing over high-speed networks. *PVLDB*, 9(4).

[233] Kenneth A. Ross. 2004. Selection conditions in main memory. *ACM Trans. Database Syst.*, 29(1):132–161.

[234] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: a compiler and runtime for heterogeneous systems. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 49–68. ACM.

[235] Abdallah Salama, Carsten Binnig, Tim Kraska, Ansgar Scherp, and Tobias Ziegler. 2017. Rethinking distributed query execution on high-speed networks. *IEEE Data Eng. Bull.*, 40(1).

[236] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. 2018. Serverless data analytics in the ibm cloud. In *Proceedings of the 19th International Middleware Conference Industry*, pages 1–8.

[237] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. 2015. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning. *PVLDB*, 8(9).

[238] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in tensorflow.

[239] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of gpus and cpus for database analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1617–1632, New York, NY, USA. Association for Computing Machinery.

[240] Anil Shanbhag, Nesime Tatbul, David Cohen, and Samuel Madden. 2020. Large-scale in-memory analytics on intel® optane™ dc persistent memory. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, pages 1–8.

[241] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. Numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679*.

[242] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1135–1149, New York, NY, USA. Association for Computing Machinery.

[243] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *SIGMOD*.

[244] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. Strom: smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16.

[245] Sivaprasad Sudhir, Wenbo Tao, Nikolay Laptev, Cyrille Habis, Michael Cafarella, and Samuel Madden. 2023. Pando: Enhanced data skipping with logical data partitioning. *Proc. VLDB Endow.*, 16(9):2316–2329.

[246] Jens Teubner and Rene Mueller. 2011. How Soccer Players Would do Stream Joins. In *SIGMOD*.

[247] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated {Key-Value} stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48.

[248] Suketu Vakharia, Peng Li, Weiran Liu, and Sundaram Narayanan. 2023. Shared foundations: Modernizing meta's data lakehouse. In *The Conference on Innovative Data Systems Research, CIDR*.

[249] Alexander Van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent memory i/o primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–7.

[250] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically. *ACM Trans. Archit. Code Optim.*, 16(4).

[251] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST 2011)*, page 5.

[252] Matthew Vilim, Alexander Rucker, Yaqi Zhang, Sophia Liu, and Kunle Olukotun. 2020. Gorgon: accelerating machine learning from relational data. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 309–321. IEEE.

[253] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146.

[254] Tianzheng Wang and Ryan Johnson. 2014. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment*.

[255] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. Spores: Sum-product optimization via relational equality saturation for large scale linear algebra. *Proc. VLDB Endow.*, 13(12):1919–1932.

[256] Mike Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. 2021. Boxer: Data analytics on network-enabled serverless platforms. In *11th Annual Conference on Innovative Data Systems Research (CIDR 2021)*.

[257] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. An early evaluation of intel's optane dc persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–19.

[258] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An intelligent storage engine with support for advanced sql offloading. *PVLDB*, pages 963–974.

[259] Kan Wu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Rathijit Sen, and Kwanghyun Park. 2019. Exploiting intel optane ssd for microsoft sql server. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–3.

[260] Yinjun Wu, Kwanghyun Park, Rathijit Sen, Brian Kroth, and Jaeyoung Do. 2020. Lessons learned from the early performance evaluation of intel optane dc persistent memory in dbms. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, pages 1–3.

[261] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the performance of intel optane persistent memory: a close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 488–505.

[262] Lijie Xu, Shuang Qiu, Binhang Yuan, Jiawei Jiang, Cedric Renggli, Shaoduo Gan, Kaan Kara, Guoliang Li, Ji Liu, Wentao Wu, et al. 2022. In-database machine learning with corgipile: Stochastic gradient descent without full data shuffle. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1286–1300.

[263] Carl Yang, Aydin Buluc, and John D. Owens. 2019. Graphblast: A high-performance linear algebra-based graph framework on the gpu.

[264] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 169–182.

[265] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 2015)*, page 167–181.

[266] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. Pushdowndb: Accelerating a dbms using s3 computation. In *2020 IEEE ICDE*.

[267] Gina Yuan, Shoumik Palkar, Deepak Narayanan, and Matei Zaharia. 2020. Offload annotations: Bringing heterogeneous computing to existing libraries and workloads. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 293–306. USENIX Association.

[268] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*.

[269] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95.

[270] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking database high availability with rdma networks. *PVLDB*, 12(11).

[271] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, Guoan Wu, and Andy Rudoff. 2022. Plin: A persistent learned index for non-volatile memory with high performance and instant recovery. *Proceedings of the VLDB Endowment*, pages 243–255.

[272] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association.

[273] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2195–2207.

[274] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing distributed tree-based index structures for fast rdma-capable networks. In *SIGMOD*.

[275] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. In *Proceedings of the ACM on Programming Languages (PACMPL 2019)*.