Towards Network-Accelerated Databases

Lasse Beck Thostrup

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Towards Network-Accelerated Databases

TECHNISCHE UNIVERSITÄT DARMSTADT

Doctoral thesis by
Lasse Beck Thostrup

submitted in fulfillment of the requirements for the degree of *Doctor rerum naturalium (Dr. rer. nat.)*

Reviewers
Prof. Dr. rer. nat. Carsten Binnig
Prof. Tianzheng Wang, Ph.D.

Department of Computer Science
Technical University of Darmstadt

Darmstadt 2023

Lasse Beck Thostrup: *Towards Network-Accelerated Databases*

Day of the viva voce: 10.11.2023
Year published in TUprints: 2024

Darmstadt, Technical University of Darmstadt

Erklärung laut Promotionsordnung

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, September 25, 2023

Lasse Beck Thostrup

Abstract

Throughout the last years, data processing systems have seen substantial changes, notably moving towards disaggregation of resources. This shift separates compute and storage resources into distinct servers for better resource utilization, as they can now be scaled independently based on demand. This development is crucial for cloud-native Database Management Systems (DBMS), which mainly build on such disaggregated structures.

This thesis examines two significant hardware trends in disaggregated architectures for DBMSs: modern networks and heterogeneous computing. Modern networks such as Remote Direct Memory Access (RDMA) are critical for efficient, high-throughput, low-latency data transfer, but present challenges for achieving optimal performance for DBMSs. The reason for this is that RDMA comes with a low-level interface with a plentitude of performance-critical aspects to consider. To address this challenge, this thesis introduces a high-level programming interface, the Data Flow Interface, specifically targeting the needs of data-intensive processing systems.

In addition, this thesis highlights the emerging trend toward programmable network devices that offer data processing capabilities in the network. This trend is especially interesting for distributed DBMSs as they have to transfer large amounts of data over the network due to the disaggregated architecture, but also typical distributed data processing operations such as joins have to shuffle data between compute nodes. In the thesis, in-network processing devices are evaluated with typical DBMS operations to investigate the benefits and potential shortcomings.

Another trend in the data center is the increasing heterogeneity of computing units such as GPUs and FPGAs due to their fast processing capabilities. Incorporating these heterogeneous devices into disaggregated architectures with fast networks has many merits. The reason is that specialized compute units can be exposed as network-attached disaggregated accelerator pools and thus provide flexible and scalable high-performance data processing. This integration of heterogeneous compute units and fast RDMA-capable networks is however non-trivial since networks like Remote Direct Memory Access (RDMA) are typically not directly supported for devices besides CPUs and are as such non-trivial to integrate efficiently. The challenge of how to achieve efficient communication between different types of compute devices is addressed by proposing a network-driven communication scheme that leverages a programmable switch to carry out the network communication on behalf of the compute devices.

Zusammenfassung

In den letzten Jahren haben sich die Datenverarbeitungssysteme erheblich verändert, insbesondere in Richtung Disaggregation von Ressourcen. Dabei werden Rechen- und Speicherressourcen in verschiedene Server aufgeteilt, um eine bessere Ressourcennutzung zu erreichen, da sie nun unabhängig voneinander je nach Bedarf skaliert werden können. Diese Entwicklung ist entscheidend für Cloud-native Datenbankmanagementsysteme (DBMS), die hauptsächlich auf solchen disaggregierten Strukturen aufbauen.

In dieser Arbeit werden zwei wichtige Hardware-Trends bei disaggregierten Architekturen für DBMS untersucht: moderne Netzwerke und heterogenes Computing. Moderne Netzwerke wie Remote Direct Memory Access (RDMA) sind für einen effizienten Datentransfer mit hohem Durchsatz und geringer Latenz entscheidend, stellen jedoch eine Herausforderung dar, wenn es darum geht, eine optimale Leistung für DBMS zu erzielen. Der Grund dafür ist, dass RDMA über eine Low-Level-Schnittstelle verfügt, bei der eine Vielzahl von leistungsrelevanten Aspekten zu berücksichtigen ist. Um dieser Herausforderung zu begegnen, wird in dieser Arbeit eine High-Level-Programmierschnittstelle, das Data Flow Interface, eingeführt, die speziell auf die Bedürfnisse datenintensiver Verarbeitungssysteme ausgerichtet ist.

Darüber hinaus wird in dieser Arbeit der sich abzeichnende Trend zu programmierbaren Netzwerkgeräten hervorgehoben, die Datenverarbeitungsfunktionen im Netzwerk bieten. Dieser Trend ist vor allem für verteilte DBMS interessant, da diese aufgrund der disaggregierten Architektur große Datenmengen über das Netzwerk übertragen müssen, aber auch typische verteilte Datenverarbeitungsoperationen wie Joins müssen Daten zwischen Rechenknoten verschieben. In dieser Arbeit werden netzinterne Verarbeitungsgeräte mit typischen DBMS-Operationen evaluiert, um die Vorteile und potenziellen Nachteile zu untersuchen.

Ein weiterer Trend im Rechenzentrum ist die zunehmende Heterogenität von Recheneinheiten wie GPUs und FPGAs aufgrund ihrer schnellen Datenverarbeitungsmöglichkeiten. Die Einbindung dieser heterogenen Geräte in disaggregierte Architekturen mit schnellen Netzwerken hat viele Vorteile. Der Grund dafür ist, dass spezialisierte Recheneinheiten als netzgebundene disaggregierte Beschleunigerpools eingesetzt werden können und so eine flexible und skalierbare Hochleistungsdatenverarbeitung ermöglichen. Diese Integration von heterogenen Recheneinheiten und schnellen RDMA-fähigen Netzwerken ist jedoch nicht trivial, da Netzwerke wie RDMA typischerweise nicht direkt auf anderen Recheneinheiten als CPUs unterstützt werden und daher nicht einfach effizient zu integrieren sind. Die Herausforderung, wie eine effiziente Kommunikation zwischen verschiedenen Arten von Recheneinheiten erreicht werden kann, wird durch den Vorschlag eines netzwerkgesteuerten Kommunikationsschemas angegangen, das einen programmierbaren Switch nutzt, um die Netzwerkkommunikation an Stelle der Recheneinheiten durchzuführen.

Publications

[1] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. “DFI: The Data Flow Interface for High-Speed Networks.” In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. Best Paper Award. ACM, 2021, pp. 1825–1837. DOI: 10.1145/3448016.3452816. URL: https://doi.org/10.1145/3448016.3452816.

[2] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. “DFI: The Data Flow Interface for High-Speed Networks.” In: *SIGMOD Rec.* 51.1 (2022). Research Highlight Award, pp. 15–22. DOI: 10.1145/3542700.3542705. URL: https://doi.org/10.1145/3542700.3542705.

[3] Lasse Thostrup, Daniel Failing, Tobias Ziegler, and Carsten Binnig. “A DBMS-centric Evaluation of BlueField DPUs on Fast Networks.” In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2022, Sydney, Australia, September 5, 2022*. Ed. by Rajesh Bordawekar and Tirthankar Lahiri. 2022, pp. 1–10. URL: http://www.adms-conf.org/2022-camera-ready/ADMS22_thostrup.pdf.

[4] Jaco A. Hofmann, Lasse Thostrup, Tobias Ziegler, Carsten Binnig, and Andreas Koch. “High-Performance In-Network Data Processing.” In: *10th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2019, Los Angeles, California, USA, August 26, 2019*. Ed. by Rajesh Bordawekar and Tirthankar Lahiri. 2019, pp. 64–73. URL: http://www.adms-conf.org/2019-camera-ready/hofmann_adms19.pdf.

[5] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. “Distributed GPU Joins on Fast RDMA-capable Networks.” In: *Proc. ACM Manag. Data* 1.1 (2023), 29:1–29:26. DOI: 10.1145/3588709. URL: https://doi.org/10.1145/3588709.

[6] Matthias Jasny, Lasse Thostrup, and Carsten Binnig. “Zero-sided RDMA: Network-driven Data Shuffling.” In: *Proceedings of the 19th International Workshop on Data Management on New Hardware, DaMoN 2023, Seattle, WA, USA, June 18-23, 2023*. Ed. by Norman May and Nesime Tatbul. ACM, 2023, pp. 82–85. DOI: 10.1145/3592980.3595302. URL: https://doi.org/10.1145/3592980.3595302.

[7] Matthias Jasny, Lasse Thostrup, Sajjad Tamimi, Andreas Koch, Zsolt István, and Carsten Binnig. “Zero-sided RDMA: Network-driven Data Shuffling for Disaggregated Heterogeneous Cloud DBMSs.” In: *Proc. ACM Manag. Data* 2.1 (Mar. 2024). DOI: 10.1145/3639291. URL: https://doi.org/10.1145/3639291.

Due to the nature of the synopsis and for better readability, selected paragraphs from these publications were transferred verbatim throughout the synopsis without explicit labeling as suggested in the department regulations “Kumulative Dissertation und Eigenzitate in Dissertationen” (21.09.2021) §1.

Acknowledgments

I want to start by expressing my sincere thanks to Prof. Carsten Binnig for his outstanding support and guidance throughout my Ph.D. journey. His mentorship went beyond just academic matters. Prof. Carsten Binnig consistently showed genuine care for my personal well-being, ensuring that I was not only progressing in my research but also thriving personally.

I am also very thankful to Prof. Tianzheng Wang for taking the time to review this dissertation.

I would like to express my gratitude to the Systems Group at TU Darmstadt, and especially my colleagues Benjamin, Nils, Matthias, Tobi, and Muhammad. Their feedback and help have been crucial, as well as the supportive and enjoyable office environment they have provided.

A special thanks to Mona, who always keep a positive attitude and has been incredibly helpful in handling all kinds of organizational matters.

Additionally, I want to thank my parents and siblings for their constant support throughout this journey. Their encouragement has been indispensable.

Lastly, a heartfelt thank you to my wife, Florencia, and our daughter Luisa for their unconditional love, support, and understanding. I am incredibly grateful for their presence in my life.

Contents

Acronyms

DB	Database
DBMS	Database Management System
ML	Machine Learning
NUMA	Non-Uniform Memory Access
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
RDMA	Remote Direct Memory Access
NIC	Network Interface Controller
ASIC	Application-specific Integrated Circuit
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
DPU	Data Processing Unit

Part I

Synopsis

1 Introduction

1.1 Context & Motivation

In the past decade, cloud computing and the landscape of data processing systems have undergone a significant transformation. Notably, there has been a shift towards disaggregation, separating compute and storage into distinct resources to offer improved resource utilization, as each resource can be scaled independently based on demand [11, 28, 147, 150]. This trend has been transformative for cloud-native Database Management Systems (DBMSs), which typically build on top of such disaggregated architectures [4, 23, 130]. When pairing this architectural trend with the innovation and shift in data center hardware, we see that especially two hardware trends play an important role in disaggregated architectures, namely modern networks and heterogeneous compute units.

Modern Networks - Fast & Programmable. In contrast to the traditional shared-nothing DBMS architecture (shown in Figure 1.1a) where data storage and compute are realized on the same physical servers, in disaggregated architectures (shown in Figure 1.1b), data is stored separately from the compute nodes to enable efficient scaling of resources. As such, one of the most important factors to performance is the rate at which data can be transferred over the network and has therefore caused considerable attention on developing fast network solutions. Remote Direct Memory Access (RDMA) is one of these, which has already been widely adopted by the major cloud vendors as it offers an efficient means of moving data from one server to another [5]. RDMA achieves efficient networking by offloading parts of the networking stack to specialized Network Interface Controllers (NICs) and completely bypassing the operating system. While RDMA networks provide high throughput and low latency, it comes with a plentitude of aspects to consider in order to achieve good performance for distributed data processing systems. As such, blindly upgrading to faster networks does often not directly translate into performance gains for DBMSs.

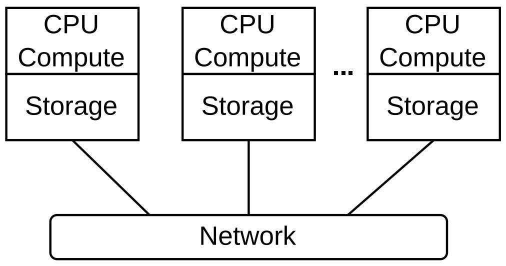
Data center networks are not only becoming faster with technologies such as RDMA, but they are also becoming programmable in the sense that networking devices such as NICs and switches can be programmed for specific purposes beyond their fixed routing and forwarding functionality. This opens up many new possibilities for realizing new communication protocols and for disaggregated data processing systems to not only process data at the end-hosts but also in the network while data is being transferred. A concrete example is the possibility of pushing down whole database operators into the network.

Heterogeneous Compute. Another important trend for data processing systems is that compute devices are becoming increasingly heterogeneous due to the performance stagnation observed for CPUs. Heterogeneous compute devices such as Graphics Processing Units (GPUs) or Field Programmable Gate Arrays (FPGAs) offer ample opportunities for accelerating many processing jobs, and are as such seeing a lot of research interest. This is especially the case for single-node hardware acceleration which has already shown great potential for DBMSs [75, 120, 122]. Incorporating heterogeneous compute units into disaggregated architectures with fast networks has many merits. The reason is that specialized compute units can be exposed as network-attached disaggregated accelerator pools, as illustrated in Figure 1.1c, and thus provide flexible and scalable high-performance data processing. This integration of heterogeneous compute units and fast RDMA-capable networks is however a rather unexplored avenue. The reason for this is that networks like RDMA are typically not directly supported for devices besides CPUs and are as such non-trivial to integrate efficiently.

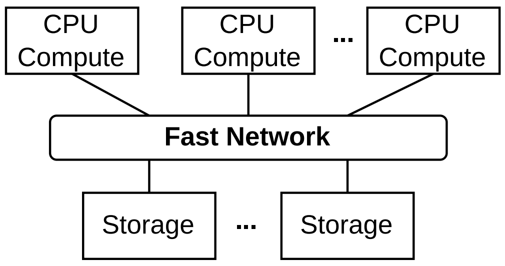
1.2 Problem Statement & Challenges

While innovations in data center hardware bring a lot of promise for improvement, they also bring new challenges and tradeoffs to consider. The challenges that arise from the integration of modern networks and hardware are now iterated in order to set the context for the thesis.

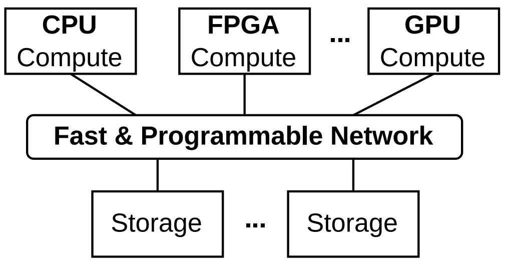
Challenge 1: Adoption of High-Speed Networks for DBMSs. RDMA has become a crucial component in data center networking due to its unprecedented performance and ability to remove network-related CPU overhead. With RDMA, an application can leverage different communication primitives that can be categorized as one-sided (*read / write*) or two-sided (*send / receive*) operations, which refer to the involvement of the sender and receiver in the communication. For one-sided operations, only the sender is



(a) Shared-nothing architecture.



(b) Disaggregated architecture.



(c) Disaggregated architecture with heterogeneous compute.

Figure 1.1: Overview of DBMS architectures for the cloud. In the shared-nothing architecture (a), the network is avoided at all costs. In the disaggregated architecture (b), resources can be scaled in or out separately by interconnecting each resource with fast networks. In (c), heterogeneous compute units are integrated in the disaggregated architecture to reap the benefits of accelerated compute with the scaling properties of disaggregated resources.

actively involved, but as a consequence, the sender also has to decide where the data should be placed on the remote node. With two-sided operations, the receiver is also actively involved in the communication since it needs to issue *receive* requests before *send* requests can be issued on the sender side. It can thus also decide where to place data, simplifying remote memory management.

While one-sided operations have been shown to provide the best performance as they help to remove remote CPU-overhead [28, 156], it introduces additional complexity as all senders have to coordinate remote memory accesses. In addition to deciding which RDMA operations to utilize, RDMA also has many low-level parameters such as transport protocols, queue sizes, and additional communication flags.

As such, navigating these performance tradeoffs and design decisions is non-trivial and hinders the adoption of RDMA for DBMSs.

Challenge 2: Implications of In-network Processing. Data center networks are evolving to be more active in the sense of providing the ability to process data as it is transferred through the network. This change has fostered a lot of research as the possibilities for distributed systems are many [60, 63, 79, 153]. Especially DBMSs can benefit from in-network processing as they have to transfer large amounts of data over the network both due to the disaggregated architecture, but also typical distributed data processing operations such as joins have to shuffle data between compute nodes.

In-network processing is enabled by different networking devices such as switches and NICs which are equipped with different processing units such as CPUs, FPGAs and Application-specific Integrated Circuits (ASICs).

Due to the variety in devices, processing capabilities, and performance, it is difficult to determine when DBMSs can benefit from in-network processing. As such, for future DBMSs to build on and use in-network processing, it is important to first understand and evaluate performance implications and potential benefits when offloading database operators and data structures into the network.

Challenge 3: Distributed Heterogeneous Compute. Also for the compute units, modern hardware is increasingly making its entry into the data center. While historically the CPU has been the primary platform used for data processing, the challenges of continuously scaling the processing power (typically referred to as the end of Moores Law and Dennards Scaling), are causing increased interest in more specialized compute devices. Examples of such devices are GPUs and FPGAs, which come with different processing models as compared to traditional CPUs but in turn, can provide speedups for a wide range of DBMS tasks in single-node setups [75, 120, 122, 127, 139].

However, many modern processing tasks require multiple interconnected compute units to fulfill the processing needs due to high compute demands and dataset sizes. In addition, the predominant disaggregated DBMS architecture builds on a shared storage model where data is transferred from storage nodes to compute nodes. What this means for compute accelerators is that it is essential for the overall device performance that data can be transferred efficiently in and out of device memory. This is a major challenge, as efficient networking stacks such as RDMA are rarely available for processing units other than CPUs. For this reason, the networking control flow is handled by the CPU which can in turn heavily affect the achieved processing performance as it increases the synchronization and coordination overhead between the accelerator and the CPU. Therefore, in order to integrate compute accelerators into disaggregated setups, it is essential that the accelerators can communicate independently.

Another challenge with the integration of compute accelerators is the need to redesign and adapt existing database operators to take full advantage of the processing capabilities such accelerators can offer. In a distributed setup, this means that networking has to be incorporated directly into the operator design to efficiently make use of both resources concurrently.

1.3 Contributions

This thesis aims to investigate the benefits, challenges, and suggest solutions to bring data processing systems into the era of fast programmable networks and heterogeneous compute devices. The thesis contains the following contributions that address the aforementioned challenges:

Addressing the Complexity of RDMA. This contribution addresses Challenge 1 by simplifying the adoption and integration of RDMA in DBMSs. By providing an abstraction, it aids system developers in navigating complex design decisions by simplifying distributed memory management and performance trade-offs associated with RDMA operations. The abstraction is centered around the requirements of distributed DBMSs by providing a set of tailored networking primitives typically used in DBMSs. As such, it facilitates the integration of RDMA, helping to remove the barriers hindering its wider adoption in database systems.

Understanding and Evaluating In-network Processing. As data center networks evolve, understanding and assessing the implications of in-network processing becomes vital. This contribution addresses Challenge 2 by conducting an evaluation of in-network processing in the context of DBMSs. The evaluation benchmarks various devices, processing capabilities, and performance metrics. It establishes a foundation for future DBMSs, outlining when and how they can benefit from in-network processing, thus offering guidance for offloading database operators and data structures into the network.

Realizing a Distributed GPU Join Operator on Fast RDMA-capable Networks. This contribution addresses Challenge 3 by presenting a novel distributed GPU join operator that leverages RDMA-capable networks. A pipelined approach is introduced to efficiently incorporate network communication into the join execution and by that leverage both the limited GPU memory and the network optimally. By designing and adapting existing database operators to harness the power of GPUs and high-speed RDMA networks, significant improvements in processing performance are demonstrated. This work addresses the challenges of efficient data transfer, synchronization, and coordination between accelerators.

Designing Network-driven Communication for Disaggregated Accelerators. With modern hardware making its entry into data centers, enabling efficient networking stacks like RDMA for heterogeneous processing units is a significant challenge. To further address Challenge 3, this contribution introduces a network-driven communication scheme designed explicitly for disaggregated accelerators. A key contribution is that offloading and driving the network communication from the network, allows accelerators to do efficient RDMA communication without having to implement the communication scheme in the accelerators or CPUs.

Together, these contributions not only address the specific challenges brought by innovations in data center hardware but also pave the way for new possibilities and advancements in database systems. The ensuing chapters of this thesis will delve into the methodologies, experiments, and results that underline these contributions.

1.4 Outline

In the following chapters, each of the main contributions are presented. In Chapter 2 an interface is proposed which addresses the challenges of leveraging RDMA for DBMSs. Chapter 3 takes a step further into the network by evaluating the effectiveness of in-network processing for DBMS use cases. Subsequently in Chapter 4, a distributed GPU join operator is presented that optimally integrates the network into join execution on the GPUs. In Chapter 5, a network-driven communication scheme for integrating various accelerators with fast and efficient networking is presented. The synopsis concludes with a summary of the results and an outlook of future research directions in Chapter 6.

2 Addressing the Complexity of RDMA

This chapter summarizes the work on addressing the complexities of RDMA through an interface for DBMSs. The contributions are based on the following peer-reviewed publications.[1]

- **"DFI: The Data Flow Interface for High-Speed Networks"** published in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021.* **Best Paper Award**. [125], (cf. Chapter 7).

  Contributions of the authors: Lasse Beck Thostrup is the leading author and was thus responsible for the implementation, benchmark design, and evaluation of the Data Flow Interface, and the manuscript. The co-author Jan Skrzypczak contributed to the implementation and evaluation of the distributed consensus (state machine replication) use-case. The remaining authors Matthias Jasny, Tobias Ziegler, and Carsten Binnig contributed invaluable feedback. All authors agree with the use of the publication for this dissertation.

- **"DFI: The Data Flow Interface for High-Speed Networks"** published in *SIGMOD Rec.* 51.1 (2022). **Research Highlight Award**. [126][2].

  Contributions of the authors: Lasse Beck Thostrup is the leading author and was thus responsible for adapting the manuscript of the original publication ("DFI: The Data Flow Interface for High-Speed Networks" SIGMOD 2021) to the SIGMOD Record publication format. The authors Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig contributed invaluable feedback. All authors agree with the use of the publication for this dissertation.

RDMA has been gaining in popularity, especially in the cloud due to its unprecedented performance both in terms of low latency and high bandwidth. For disaggregated

[1]Several passages in this chapter were transferred verbatim from these publications.
[2]Publication is not included in Part II as it is a condensed version of the original SIGMOD 2021 publication (cf. Chapter 7).

DBMSs, RDMA is particularly interesting as the network plays a crucial role in the overall performance since data transfers are on the critical path. However, the performance of RDMA comes at the cost of a complex and hard-to-use interface, with many low-level knobs and parameters to tune in order to achieve the optimal performance.

In the following, the challenges of RDMA are addressed by proposing an abstraction called The Data Flow Interface (DFI) which aims to make it easier for data processing systems to exploit high-speed networks without the need to deal with the complexity of RDMA.

2.1 RDMA Background

Before diving into the Data Flow Interface, a background on RDMA is first provided.

The InfiniBand RDMA verb interface is a low-level interface providing low latency and high bandwidth communication. The interface exposes one-sided verbs (*write*, *read* & *atomics*) and two-sided verbs (*send* & *receive*) which refer to the involvement of end-points (i.e., one-sided verbs only involves the CPU of the sender). The high performance of RDMA is in parts achieved by zero-copy functionality where data is transferred directly from the application over the network to the receiver without any intermediate copies which therefore avoids spending CPU resources on copying. In addition, RDMA has an asynchronous nature making it possible to pipeline computation and communication such that the CPU is not busy idling during network communication. RDMA requires specialized network cards (RNICs) that implement a part of the networking stack directly in hardware.

The RDMA verb interface is a userspace library, meaning all networking functionality is executed directly in the application processes. To issue RDMA verbs (one- or two-sided), the application has to register a memory region in which the RNIC can directly access memory, leaving communication-related memory management to be a responsibility of the application. RDMA can be set up with different kinds of transport types, such as Reliable Connection or Unreliable Datagram. The most commonly used transport is the Reliable Connection which requires stateful connections between any end-points. Since reliability is a must for most applications and only the Reliable Connection supports one-sided verbs, in this thesis, only Reliable Connection is used[3].

[3]With one exception, that is for supporting native RDMA multicast, only Unreliable Datagram is supported.

In summary, due to the RDMA verb interface's very low abstraction level, it provides a huge design space. This requires, that applications need carefully explore this design space and optimally make use of the available low-level options [35, 69, 156, 157]. This is however burdensome and requires a lot of engineering to realize.

2.2 The Data Flow Interface

To address the complexity of the RDMA interface, the Data Flow Interface (DFI) is now introduced. In this section, the central design goals of DFI are first highlighted before discussing the flow-based programming model, as well as the high-level idea of the execution model behind flows.

2.2.1 Key Design Principles

The aim of DFI is to provide a high-level abstraction that provides efficient support for a broad set of data processing systems. The key design principles of DFI to ideally support the needs of these systems:

- *(1) Pipelining:* As many data-centric applications are often dominated by data transfers (i.e., data shuffling), it has been shown to be crucial that computation and communication can be overlapped in order to fully utilize both the network and CPU at the same time [10].

- *(2) Thread-centricity:* Multi-threading is essential not only in achieving high degrees of parallelism in modern data-centric architectures but also in saturating the network. Hence, different from process-centric libraries (e.g., MPI), DFI should be designed from the ground up to enable a thread-centric execution and communication model.

- *(3) Low-overhead synchronization:* Another important aspect that goes along with thread-centricity is that DFI aims to provide low-overhead synchronization between sender and receiver threads as well as between sender threads that target the same receiver. By providing low-overhead synchronization, DFI should enable scalability to a high number of sender and receiver threads.

- *(4) Declarative optimization:* A last important goal is that DFI exposes parameters as a handle for applications to declare what optimizations are desired. Examples of such optimizations are whether applications are bandwidth or latency-sensitive, but also other guarantees such as the global ordering of messages when data is sent across flows (which is important, for example, for data replication protocols).

These design principles form the foundation of DFI and are realized through a flow abstraction that offers applications a high-level interface to declare data movements between processing nodes.

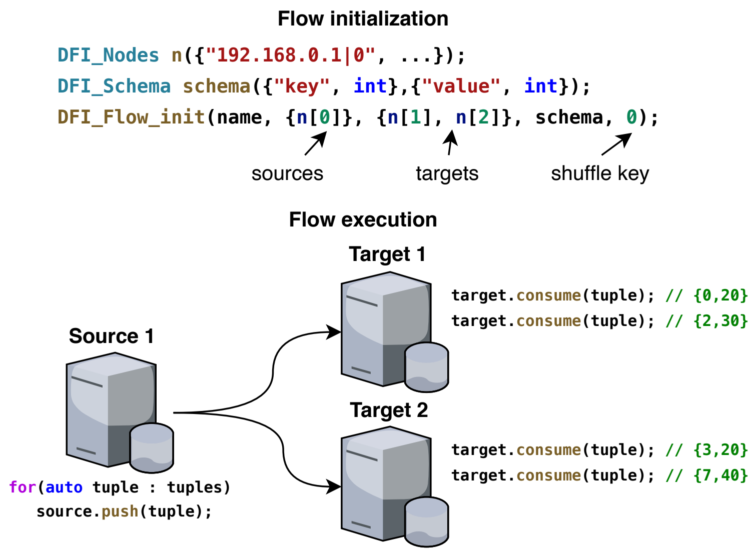
2.2.2 Flow-based Programming Abstraction

Central to the DFI abstraction are *flows*. These flows define data movement between endpoints in distributed applications, presenting *sources* and *targets* as data entry and exit points on a per-thread basis. This intuitive abstraction allows applications to design various communication topologies, ranging from point-to-point to many-to-many communications between worker threads across nodes. It's versatile enough to compose many different data processing scenarios, from bandwidth-intensive distributed join algorithms to latency-sensitive consensus protocols, as later shown in Section 2.3.

The following is an example of a concrete many-to-many flow type in DFI, which is one out of multiple other flow types. The most prevalent instance of many-to-many communication in data processing systems is key-based data shuffling across multiple sources and targets. Figure 2.1 depicts such a shuffle flow in DFI.

Setting up and using a flow in DFI works as follows:

- *Initialization*: To deploy a flow, one must first initialize it, specifying details like a unique flow name identifier, the locations of source and target threads (which are identified using node addresses and thread IDs under *DFI_Nodes*), and the tuple schema. This also involves determining the key for tuple shuffling. While applications can designate specific partition functions, by default, DFI uses a key-based hash function for tuple partitioning.
- *Metadata Registry*: After initialization, the flow's metadata is registered centrally, typically on a master node in a distributed system. This centralization ensures other nodes can access and participate in the flow, with information such as distributed memory locations, tuple schema, and flow type. Before execution, sources and targets fetch the flow metadata from the registry.
- *Flow Source Execution*: During execution, a thread pushes tuples into the source. The push primitive is asynchronous and returns immediately after the tuple to be transferred is copied into an internal send buffer. This allows for overlap of pipeline computation and communication and thus uses both CPU and network resources in parallel.



```
DFI_Nodes n({"192.168.0.1|0", ...});
DFI_Schema schema({"key", int},{"value", int});
DFI_Flow_init(name, {n[0]}, {n[1], n[2]}, schema, 0);
```

```
target.consume(tuple); // {0,20}
target.consume(tuple); // {2,30}
```

```
target.consume(tuple); // {3,20}
target.consume(tuple); // {7,40}
```

```
for(auto tuple : tuples)
   source.push(tuple);
```

Figure 2.1: DFI's Programming and Execution Model. Example of flow initialization for setting up a shuffle-based flow. The flow execution exemplifies the tuple-based push and consumes primitives on DFI.

- *Data Tranfers*: Internally, DFI transfers tuples from sources to targets given the distribution function. This is done with one-sided RDMA in configurable transfer sizes depending on the declared optimization goal of the application.
- *Flow Target Execution*: At the target, tuples are internally detected by end-of-message polling in a circular memory buffer, and directly handed over to the application without intermediate copies.

Beyond the shuffle flows involving N senders and M receivers, DFI also supports other flow types and topologies to accommodate various data processing applications, which is covered next.

2.2.3 DFI Flows Overview

DFI introduces flows tailored to the diverse requirements of data processing systems. As outlined in Table 2.1, DFI features three distinct flow types. Each type supports specific communication topologies and is equipped with its own set of declarative flow options.

The strength of the flow abstraction is in its inherent flexibility. Different flow types can be easily interchanged to achieve different communication patterns. For example, to transform a symmetric re-partition join algorithm into a fragment-and-replicate join, it is possible to simply switch from using a shuffle flow that directs tuples based on the

Flow type	*Communication topology*	*Flow options*
Shuffle flow	1:1, N:1, 1:N, N:M	Bandwidth/latency
Replicate flow	1:N, N:M	Bandwidth/latency + ordering guarantees
Combiner flow	N:1	Bandwidth/latency + various aggregations

Table 2.1: DFI flow types for a wide range of data-centric applications. Communication topologies and flow options further allow applications to adjust the behavior of flows based on application requirements.

join key to a replicate flow that replicates the inner table. Achieving such modifications in conventional systems utilizing the RDMA verb interface would typically demand a substantial rewrite of communication-related components.

Below, the distinct flow types and their relevance to data processing systems are iterated:

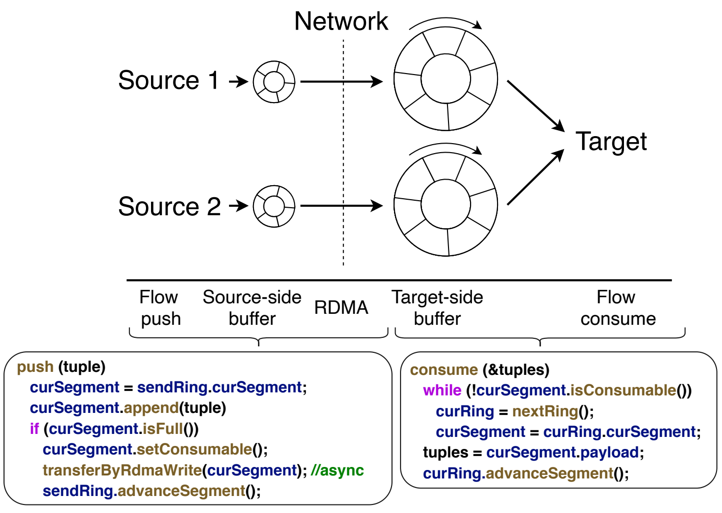
Shuffle Flow. The shuffle flow accommodates many communication patterns and routing preferences. The communication pattern is indirectly defined by declaring the participating sources and targets in the flow initialization, and can therefore follow 1:1, N:1, 1:N, and N:M communication patterns between sending and receiving threads.

The routing of tuples in a shuffle flow can be determined in various ways, such as by supplying a shuffle key or defining a routing function mapping tuples to target destinations.

Replicate Flow. The replicate flow caters to data processing tasks involving data duplication, for example, replicated state machines and fragment-and-replicate join operations.

However, a straightforward replication approach using multiple RDMA operations can be limited by the source node's outgoing link speed. To address this, DFI utilizes RDMA multicast. This ensures that messages are efficiently replicated within the network, circumventing potential bottlenecks at the source's outgoing link.

Combiner Flow. DFI's third flow is the combiner flow, designed for many-to-one communication patterns prevalent in aggregation scenarios like SQL aggregation or distributed machine learning parameter servers [83]. While a basic approach might conduct reduction at the target node, network-assisted acceleration is possible. For instance, the SHARP protocol in InfiniBand [41] facilitates in-network aggregations for fast InfiniBand networks, addressing potential bottlenecks at the receiver's inbound network.



```
push (tuple)
  curSegment = sendRing.curSegment;
  curSegment.append(tuple)
  if (curSegment.isFull())
    curSegment.setConsumable();
    transferByRdmaWrite(curSegment); //async
    sendRing.advanceSegment();
```

```
consume (&tuples)
  while (!curSegment.isConsumable())
    curRing = nextRing();
    curSegment = curRing.curSegment;
  tuples = curSegment.payload;
  curRing.advanceSegment();
```

Figure 2.2: DFI flow implementation using ring buffers. In DFI flows, each source allocates a private target-side ring buffer to minimize coordination overhead.

2.2.4 Realizing DFI Flows

The following covers a brief overview of how flows are internally realized and designed.

On a high level, to achieve the previously presented design goals for shuffle flows, DFI uses a private send/receive buffer for each pair of source and target threads as illustrated in Figure 2.2. The design of source- and target-side buffers follows a ring-based design where each ring is composed of a configurable number of segments and is allocated as one consecutive region in memory. The segment itself can be sized to contain a single tuple up to a batch of tuples. Therefore, the segment size is a tuning parameter that allows DFI to either optimize for bandwidth or latency independent of the tuple sizes used by the application.

One key question is how such a segmented ring design enables pipelining of tuples with low-overhead synchronization. In order to achieve pipelined data transfers between buffers (i.e., a decoupling of senders and receivers), one-sided RDMA writes are used to copy data asynchronously from sources to targets. To ensure that data is not overwritten on the target side, synchronization is handled through a credit-based approach with the target buffer such that only occasionally the state of the remote target buffer has to be read.

2.3 Evaluation & Methodology

In this section, DFI is evaluated by first benchmarking the shuffle flow, and subsequently, the usability and performance are evaluated through a distributed join use case.

Evaluation Environment. All experiments were conducted on an 8-node cluster with Intel Xeon Gold CPUs equipped with 100 Gbps ConnectX-5 InfiniBand NICs. DFI is implemented with C++17 and compiled with gcc-7.3.0.[4]

2.3.1 Shuffle Flows Evaluation

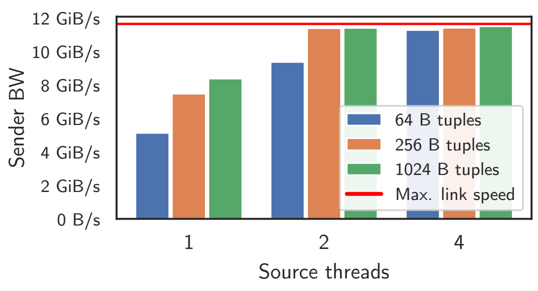
In the following, shuffle flows are evaluated with bandwidth and latency optimization and lastly, a scale-out experiment is presented.

Bandwidth-Optimized. The first experiment evaluates performance for the shuffle flow from 1 server to 8 servers with varying tuple sizes. Further, the number of sources (threads) pushing tuples into the flow is varied. The batch size for the bandwidth-optimized version in the experiments is 8 KiB. A batch size of 8 KiB is chosen as this offers a good tradeoff between network bandwidth and time until the batch is filled.

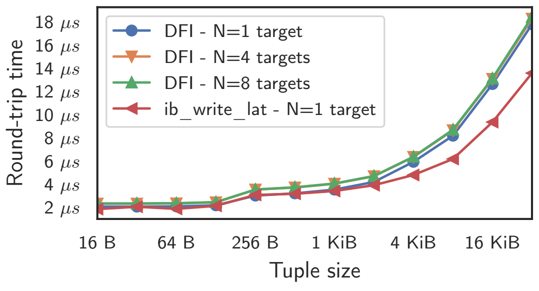
Figure 2.3a reports results for the bandwidth-optimized flow. As can be seen, in most settings full network bandwidth is achieved. Only the single-threaded scenario shows some overhead since batches must first be filled on the source side with individual tuples before they can be transferred to the target. This overhead can, however, be amortized by using more threads per server. Due to the efficient multi-threading support of DFI, we see that from two source threads on, the bandwidth is limited by the speed of the outgoing link (100 Gbps / 11,64 GiB/s - red line) for tuple sizes larger than 128 B. Moreover, when using 4 threads the maximal bandwidth is achieved independent of tuple sizes.

Latency-Optimized. Next, the shuffle flow with latency optimizations is evaluated. Using two shuffle flows, the round-trip time between nodes is measured. In Figure 2.3b, the latency is reported for different numbers of targets. The latency is compared with *ib_write_lat*[5], which is a standard tool for performance testing that uses low-level verbs to implement the round-trip. Only a small latency overhead can be detected for bigger tuple sizes due to the internal buffering in DFI. Multi-target latency in DFI is slightly increased due to shuffle flow's internal routing, whereas *ib_write_lat* supports only one target.

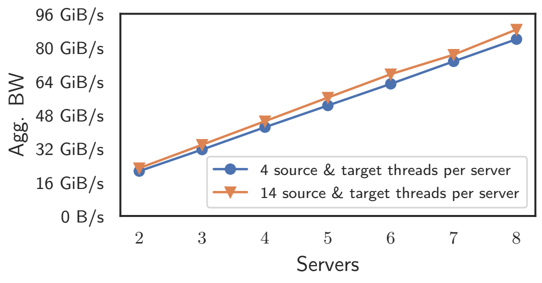
[4]For a full description of the test environment see Section 7.6
[5]`https://github.com/linux-rdma/perftest`.



(a) Sender bandwidth (1:8)



(b) Median latency (1:N)



(c) Aggr. sender bandwidth (N:N)

Figure 2.3: Shuffle flow performance. DFI achieves max. bandwidth and low latency for various scenarios.

Scale-out. A scale-out experiment for the shuffle flow was conducted, expanding the number of servers. Utilizing 14 sources and targets per node resulted in 12,544 unique connections for 8 servers. As shown in Figure 2.3c, DFI scales linearly with the number of nodes (as indicated by the x-axis), effectively increasing the aggregated bandwidth with the link speed of each added node.

2.3.2 Join Use Case

Distributed joins are crucial operators in OLAP due to large amounts of data having to be transferred across the network, and therefore a good candidate to evaluate bandwidth-optimized flows of DFI.

Distributed Radix Join. A distributed radix hash join was implemented using DFI and compared against a state-of-the-art implementation [9] using the Message Passing Interface (MPI) as an alternative high-level interface with RDMA capabilities. The DFI join was realized with two shuffle flows to re-distribute each input table on the join key.

The DFI join achieves the best performance largely due to two DFI-centric design decisions, as seen in Figure 2.4: (1) The DFI radix join does not need to first compute

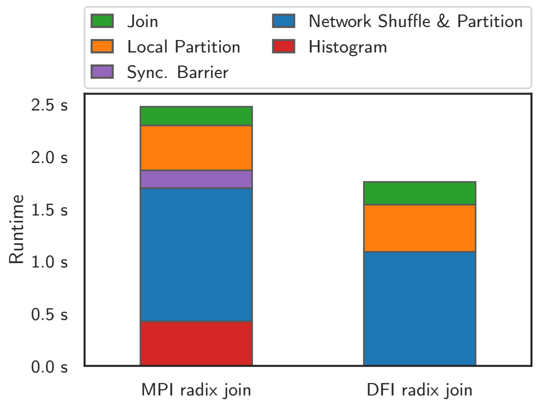


Figure 2.4: Distributed radix join - 8 nodes, 64 threads (DFI)/64 processes (MPI) in total. 2.56 B ⋈ 2.56 B tuples.

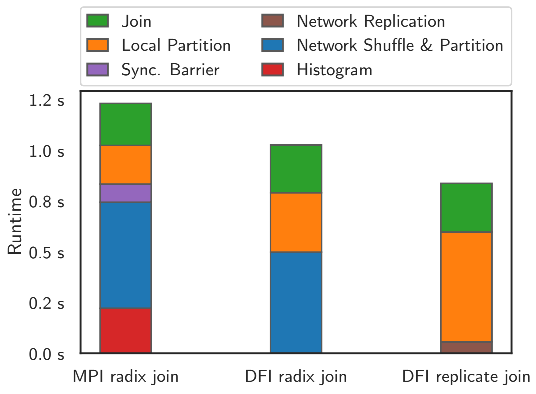


Figure 2.5: Distributed joins - 8 nodes, 64 threads (DFI)/64 processes (MPI) in total. 2.56 M ⋈ 2.56 B tuples.

a global histogram of the partition buckets. The MPI radix join in [9] makes use of one-sided *MPI_Put* primitives. In order to achieve coordination-free writes, it thus has to compute exclusive writing offsets for each partition using one additional pass. Different from this, DFI encapsulates the memory management through the buffer design which makes the additional pass superfluous. (2) The MPI join mandates a synchronization after the network partition phase, ensuring all data is fully received before progressing to a local partitioning step. This synchronization barrier can incur high overhead even for uniformly distributed data. With DFI, this constraint is unnecessary, as incoming data can be processed immediately upon arrival.

Join Adaptability. Flows in DFI offer a high-level abstraction that encapsulates the data transfer of applications. As a result, it is trivial to adapt algorithms to use a different communication pattern. To demonstrate this, we adapted the radix hash join implementation to a fragment-and-replicate join variant which uses one replicate flow that replicates the inner table on all nodes. Figure 2.5 shows the runtimes of the three different join implementations with a smaller inner table (1000× smaller than the outer table). The replication of the small inner table is comparably cheap compared to shuffling the big outer table over the network. Overall, this setup helps to further reduce the overall runtime by another 20%,

2.4 Related Work

Before concluding with a summary of this chapter, the related work is first iterated.

Liu et al. [84] conclude that MPI is not a fitting RDMA interface for data processing systems due to its processes-centric and synchronous behavior. To limit the number of roundtrips needed for typical remote operations, the authors suggest an extension of RDMA besides the *read* and *write* that are specifically targeting low-level DBMS operations. Contrary to DFI, this work does not aim to raise the level of abstraction for data processing systems, but instead to enrich the existing RDMA interface with new primitives.

Along similar lines, Burke et al. [17], suggest an extension of the RDMA interface to limit the number of roundtrips needed for, e.g., key-value stores, replicated storage, and distributed transactions. The extension permits to chain remote operations together such the latency is reduced due to fewer network round-trips.

Fent et al. [35] propose the communication library L5. While both DFI and L5 aim to raise the abstraction for RDMA, L5 is built around simplifying the use of Shared Memory and RDMA for external database communication (ODBC-like), whereas DFI targets distributed data processing. As such, DFI provides various communication patterns for, e.g., data shuffling and replication which L5 does not.

2.5 Summary

DFI presents data-centric systems with an easy-to-use interface for fast RDMA-capable networks, without having to deal with the low-level and complex aspects of RDMA. DFI provides a high-level interface that offers efficient means of shuffling, replication, or combining data.

The design of DFI is centered around four design principles that offer efficient support for a broad set of data-processing systems. With the implementation of DFI, it has been shown that DFI adds only minor overhead compared to low-level abstractions such as RDMA verbs. Moreover, the join use case demonstrated that DFI can efficiently support data-centric applications at high performance while maintaining a high level of abstraction.

3 Evaluating In-network Processing for DBMSs

This chapter summarizes the work on evaluating in-network processing for DBMSs. The contributions are based on the following peer-reviewed publications.[1]

- **"A DBMS-centric Evaluation of BlueField DPUs on Fast Networks"** published in*International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2022, Sydney, Australia, September 5, 2022* [124], (cf. Chapter 8).

  Contributions of the authors: Lasse Beck Thostrup is the leading author and was thus responsible for the implementation, benchmark design, evaluation, and manuscript. The co-author Daniel Failing contributed to the initial benchmark implementation and evaluation that was not included in the publication. The remaining authors Tobias Ziegler, and Carsten Binnig contributed invaluable feedback. All authors agree with the use of the publication for this dissertation.

- **"High-Performance In-Network Data Processing"** published in*10th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2019, Los Angeles, California, USA, August 26, 2019* [50], (cf. Chapter 9).

  Contributions of the authors: Lasse Beck Thostrup is the leading author and was thus responsible for the implementation, benchmark design, evaluation, and manuscript. The co-author Jaco A. Hofmann contributed to the switch FPGA implementation and evaluation. The co-author Tobias Ziegler contributed to the initial system design as well as the join cost models. The remaining authors Carsten Binnig, and Andreas Koch contributed invaluable feedback. All authors agree with the use of the publication for this dissertation.

While RDMA facilitates high network bandwidth and low latency in database systems, it is not the only option of modern network technologies that distributed data processing systems can leverage. A recent trend in modern networks is that network components

[1]Several passages in this chapter were transferred verbatim from these publications.

such as switches and NICs become programmable by providing additional computation on the device. Such devices thus enable processing or manipulation of data as it is traversing the network. It, therefore, opens up many possibilities for tailoring the network stack to data processing, ranging from opportunities such as in-network caching to the execution of distributed SQL operations inside network components [12, 36, 60, 116, 153]. The programmable networking devices that are available today span far in terms of compute architectures, such as ASICs, FPGAs, and general CPU cores, and therefore come with trade-offs in programmability and performance.

As such, it is non-trivial to decide in which areas and on which compute architectures DBMSs can benefit from in-network processing. In the following, this challenge is addressed by first evaluating how well programmable NICs equipped with CPU cores can be integrated into typical DBMS workloads with RDMA, and subsequently in Section 3.2, how a database join can be offloaded completely into the network onto a programmable switch.

3.1 Evaluation of Programmable NICs for DBMSs

A programmable NIC, also referred to as a Data Processing Unit (DPU), that has seen a lot of attention is the BlueField Network Interface Card [102] since it provides RDMA capabilities, flexible programmability due to its general-purpose ARM CPU cores along with other embedded specialized ASICs for tasks such as data encryption and decryption. While evaluations have shown that security-related tasks or tasks to provide tenant-isolation in data centers [20] can be provided in an efficient manner, there is no previous study that shows the offloading capabilities of DBMS tasks to such RDMA-enabled DPUs.

In this work, a first systematic study is provided that evaluates the basic performance characteristics of the BlueField network cards in the context of typical DBMS operations. For the evaluation, the offload potential of using BlueField as an RDMA-enabled DPU for two important use cases is analyzed: (1) a remote B-tree and (2) an end-host sequencer (i.e., remote counter). The remote B-tree is chosen because it is a frequently used data structure for DBMSs and it is also used in disaggregated architectures to avoid transferring all data across the network [135, 157]. On the other hand, the end-host sequencer is a commonly used building block for many distributed system tasks such as global ordering [82], coordinating write access to shared memory [11], or implementing optimistic concurrency control [146].

3.1.1 Background on Programmable NICs

Before diving into the evaluation, the relevant background on the Data Processing Units (DPUs) is provided. Many of the major network hardware vendors are including DPUs in their product offerings, with examples like the Intel IPU, Broadcom Stingray, AMD Pensando, or the Nvidia BlueField. DPUs come with different compute architectures ranging from P4 programmable ASICs to general-purpose CPU cores. This evaluation focuses on the BlueField cards from Nvidia. The BlueField DPU cards are equipped with general-purpose ARM cores which are capable of executing any program logic in contrast to the more rigid ASIC architectures. The DPU runs its own OS (e.g., Ubuntu) and as such resembles another independent server with an added set of networking features. Internally, the DPU consists of the RDMA-capable networking component (ConnectX) which provides hardware-offload of the network stack for more efficient (i.e., less CPU intensive) networking. Moreover, the ARM cores which act as computational resources on the DPU are equipped with DDR4 memory and are connected to the ConnectX over an internal PCIe switch. The BlueField (from the second generation on) is additionally equipped with hardware accelerators for compression/decompression, encryption, and regex pattern matching.

3.1.2 Experiment Setup

In the experimental setup, a typical disaggregated storage and compute setup is used. The storage node is equipped with a BlueField 2 card and runs with an Intel Xeon Gold 6326 CPU, 512 GB DDR4 memory, and PCIe 4.0. The compute node uses a non-programmable RDMA ConnectX-5 NIC and is equipped with an Intel Xeon Gold 5220 CPU with 512 GB DDR4 memory[2]. The servers are connected with RDMA over Converged Ethernet (RoCE) v2. Note that the BlueField card used in this setup only provides a maximum of 25 Gbps per network link. However, since the BlueField NIC is equipped with two links, a total of 50 Gbps connection between the compute and storage server can be used by splitting the traffic over the two links.

The BlueField card is configured in a so-called *Embedded Mode* where two virtual interfaces are created which route to either the ARM subsystem or the host. The routing

[2]For a complete overview of the hardware used, see Table 8.2

is offloaded into the eSwitch and does therefore not introduce measurable overhead on the ARM subsystem.

3.1.3 Use case 1: Remote B-Tree with RPC & One-sided RDMA

In the first experiment, the offloading potential of the BlueField-2 DPU is evaluated by utilizing the DPU together with the host CPUs of the storage server. On both the DPU and the host, all 8 CPU cores are used, which is fixed regardless of the partition sizes on the host or DPU. To use both compute resources, a B-tree is range-partitioned between the host memory and the DPU memory.

Different partition setups are used, ranging from 0-100% of the B-tree being stored on the DPU. Moreover, the index requests are uniform in the whole key range such that the relative partition sizes of the tree also match the workload generated to each device (i.e., the host CPU or the BlueField-2 DPU).

B-Tree with RPC. For the first experiment, the remote B-tree is accessed via RPC calls. The RPC framework employs two-sided RDMA SEND/RECEIVE verbs, incorporating existing optimizations like door-bell batching on both the client and server-side and inlining to minimize PCIe overhead [67]. The B-tree adopts an OLC (optimistic lock-coupling) synchronization protocol to ensure scalable reads [78]. Both keys and values are represented by 8-byte integers, and the experiment contains both a balanced mix of 50/50 read-write or exclusively read-only workloads.

In Figure 3.1, we see the results when we gradually increase the range partition of the B-tree on the DPU and decrease it on the host, indicated by the x-axis. With 0% DPU offload, the host contains the full B-tree and as such all requests are handled by the host and the DPU is not processing any requests. Instead, with 25% offload, 3/4 of the B-tree is on the host and 1/4 is on the DPU. Overall, in Figure 3.1 we initially see a steady increase in throughput as more requests are routed to the DPU up until around 25% whereas for the tree with 1M keys, the overall throughput increases by 47%. For larger tree sizes, the observed throughput increase is slightly less, with around 30% for the B-tree with 256 M keys.

However, offloading more than about 25% of the B-tree to the DPU is detrimental to the throughput since the DPU is then overloaded and the performance degrades to DPU-only throughput. This degradation of throughput is not surprising as both the CPU on the DPU is weaker and the main memory is slower than that of the storage

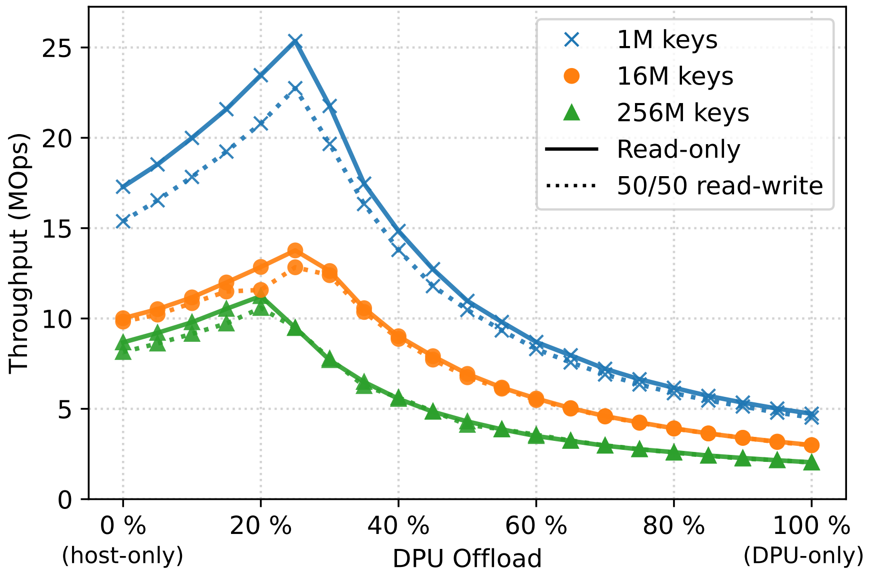


Figure 3.1: Remote B-tree with increasing offload on BlueField-2 for various B-tree sizes. RPC requests with read-only or 50/50 read-write.

host.[3] Moreover, the read-only and 50/50 read-write workload only differs slightly for the higher throughput and smaller B-tree cases for the same reasons as discussed before.

These results indicate that while the BlueField-2 is not powerful enough to achieve high throughput in comparison to the host, it yields a significant speedup by using the DPU resources in addition to the host CPU. However, this imposes challenges for real-world use cases, as the optimal partitioning of the B-tree is dependent on the workload (i.e., potential access skew) and the performance of the host server in relation to the DPU. As such, more sophisticated adaptive solutions could come into play, which re-balances and re-partitions the B-tree between the host and DPU based on utilization metrics, to automatically adapt to the most optimal partitioning between the host and DPU.

B-Tree with One-sided RDMA. More and more designs are utilizing one-sided RDMA to access remote data structures in disaggregated memory setups [2, 135, 157, 159]. The reason for this is that one-sided operations help to remove the load on the (potentially weak) remote memory servers. The BlueField-2 struggled to achieve good RPC performance due to the relatively slow memory and CPU cores, one-sided access is, therefore, a promising use case as it does not incur any CPU overhead on the DPU.

[3]Full server specifications are reported in Table 8.1 and Table 8.2.

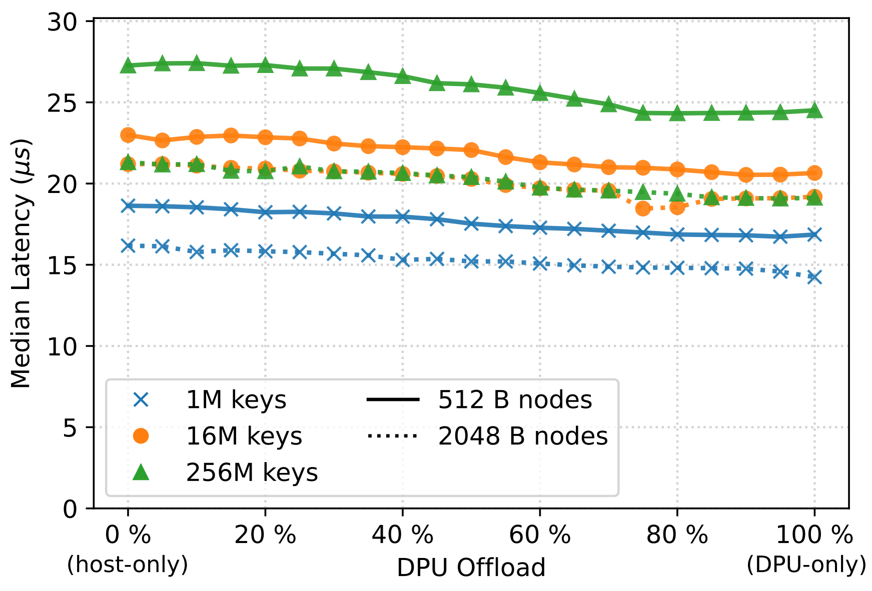


Figure 3.2: Remote B-tree latency with increasing offload on BlueField-2. Read-only with one-sided RDMA.

In this experiment, a remote B-tree accessed only over one-sided RDMA read operations is evaluated. The way a remote B-tree lookup works for one-sided operations is that clients first issue a read on the root node and locally perform a binary search to determine the next child node. This is repeated until the leaf level. As such, an RDMA read request is issued for each level of the B-tree and since the reads are interdependent (i.e., the location of one read depends on the previous) they cannot be overlapped.

Since the local DPU memory is in close proximity to the network, it can be expected that one-sided network requests can be answered faster as compared to the host.

In Figure 3.2 B-tree lookups are executed with different partition sizes offloaded to the DPU. A clear trend is shown here that the DPU provides faster lookups than the host. For the different tree sizes and node sizes, the improvement is around 11-13%. The difference in latency observed for the evaluated tree sizes and node sizes is due to the different latency of the RDMA read and the depth of the tree. As an example, a node size of 512 B has a fanout of $512B/16B = 32$ (with 16 B used as key and child pointer), so for a tree with 256 M keys, the depth will be $\lceil log_{32}(256M) \rceil = 6$, whereas a node size of 2048 B only has a depth of 5 and therefore one less RDMA read. The experiment shows that the lower latency of a 512 B read with respect to a 2048 B read does not amortize the cost of an extra read in the B-tree.

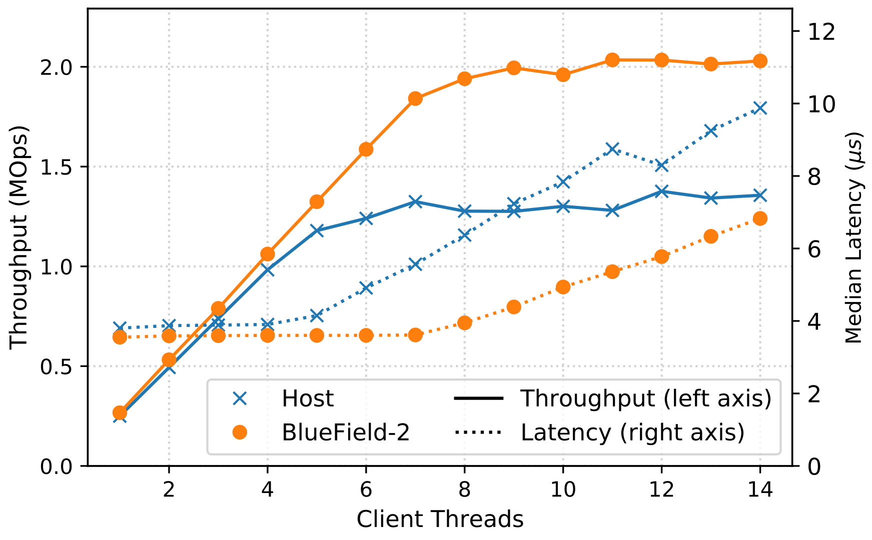


Figure 3.3: Throughput and latency of one-sided RDMA fetch-and-add on either the storage host or the BlueField-2 DPU.

Discussion. In conclusion, since the relatively weak CPU cores of the BlueField-2 are not engaged with one-sided access, the DPU has better offloading potential. The strongest benefit comes with lower access latency due to the co-location of the CPU cores and the network on the same physical board.

Another interesting benefit given by offloading the one-sided accesses to the DPU is that read or write pressure on the local main memory of the host is alleviated, which might benefit concurrent memory-intensive applications. This is even more noticeable with faster networks such as the BlueField-2 model with 200 Gbps.

3.1.4 Use case 2: Remote Sequencer

A common building block in distributed systems is global counters. They are among others used for global timestamps, asserting message ordering or coordinating access to shared memory [67]. In the next experiment, the performance of one-sided RDMA atomic operations on the storage host and the BlueField-2 DPU is evaluated.

One-sided RDMA Atomics. Atomic operations are already provided in the collection of RDMA primitives such that multiple clients can perform fetch-and-add or compare-and-swap operations over the network without any additional coordination. In general, this can facilitate one-sided access to remote data structures without any locking. For this use case, a remote counter (i.e., sequencer) is accessed with one-sided RDMA fetch-and-add operations.

In Figure 3.3 the throughput and latency are reported with an increasing number of client threads on the compute node accessing the same counter. In this use case a substantial benefit of the DPU in terms of achieved throughput can be seen. Placing the atomic counter on the DPU achieves an almost 50% throughput speedup. This is also reflected in the number of clients needed to saturate the remote server where the throughput of the DPU is saturated by around 8 clients whereas the storage host only can scale up to around 6 clients.

The latency is almost identical with a slight benefit on the DPU. Beyond the saturation point, the latencies increase linearly with more clients added as contention is created and requests are increasingly queued.

3.1.5 Conclusion

In conclusion, it was evaluated how well the BlueField-2 DPU performs with respect to typical DBMS tasks such as remote B-trees or a global sequencer. The main findings are that an acceleration potential exists for one-sided accesses both in terms of latency and throughput whereas two-sided accesses easily overload the DPU. However, using the DPU in combination with the host CPU of the storage server can yield promising performance benefits provided that the workload is carefully distributed with regard to the relative performance difference.

3.2 Distributed Join on a Programmable Switch

Besides programmable NICs, modern networks also include the possibility to program network switches. This new type of hardware is especially interesting for distributed DBMSs as many operations require shuffling huge amounts of data over the network. With programmable switches, it opens up many interesting possibilities of offloading and redesigning database operations, especially, since the switch is centrally placed in the network and therefore sees the traffic for many connected nodes.

However, for current available programmable switches such as Intel Tofino switches [54], a major weakness is that the memory capacity is severely limited. This limits the applicability of data-intensive operations which require stateful data structures exceeding just tens of megabytes. One example here is the common database hash join which uses a hash table built on one of the two tables. As the size of database tables easily go beyond the available switch memory, in this line of work, a custom switch approach is taken to be able to offload a database join to the network.

An FPGA-based switch design is derived which provides a larger amount of DDR3 main memory capable of supporting more memory-intensive operations. In the following, it is outlined how distributed join processing can integrate and make use of a programmable switch, and subsequently, the switch design is outlined before presenting the evaluation.

3.2.1 Join Processing with INP

To illustrate the main idea of distributed query processing that utilizes a programmable switch, classical distributed join execution is first reviewed, and subsequently, a new execution scheme for INP is discussed. A typical setup of a shared-nothing database consists of one master and several compute nodes, as well as one switch connecting the nodes. As an example query, consider the execution plan in Figure 3.4 that could result from the SQL statement `SELECT * FROM A JOIN B JOIN C`.

In the classical distributed query processing, `A` and `B` are first shuffled according to the join key. Then, each node builds a hash table over `B` (assuming `B` is the smaller table) and uses tuples from `A` to probe in that hash table. For the subsequent join, the intermediate result of $A \bowtie B$ as well as relation `C` need to be shuffled again, such that the joins can be executed by building and probing into the hash table of `C`. Thus, each join (if data is not co-partitioned) typically requires one expensive shuffle operation for each input table.

The equivalent query plan for INP execution is shown on Figure 3.5. As can be seen in the two figures, the classical (Figure 3.4) and the INP-based plan (Figure 3.5) consist of two types of pipelines (probe-pipelines and build-pipelines), the INP-based plan splits the plan into multiple pipelines that can be placed on worker nodes or the switch respectively. As a consequence, different from the traditional plan, many of the `Shuffle` operators can be completely avoided since the probe steps are executed all in the switch. In the following, the implications of these differences are discussed in more detail.

Discussion. As mentioned before, the main conceptual difference of the INP-scheme is the elimination of shuffling, and in particular the re-shuffling of intermediate join results. This is beneficial since shuffling comes with several challenges.

First, shuffling operations are so-called pipeline breakers, since the streaming of tuples through an operator pipeline is stopped (i.e., the shuffle operation only starts once the previous intermediate result has been materialized completely). This, however, limits the

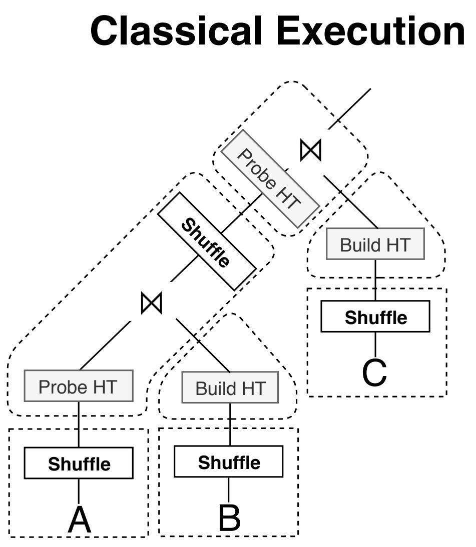


Figure 3.4: Example of Query Plan for Classical Execution.

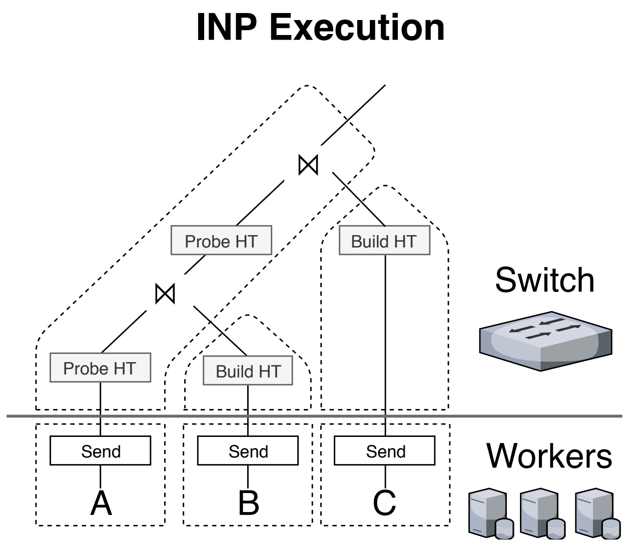


Figure 3.5: Example of Query Plan for INP Execution with Operator Placement.

degree of parallelism of the execution since the following phases of a query need to wait for the completion of previous ones. For instance, the second join of the example query can not be computed until the result of the first join has been materialized.

Second, shuffling usually means that significant amounts of data need to be transferred via the network since also the intermediate results need to be partitioned and sent to all workers. The cost of shuffling intermediate results is even higher in a data warehouse setup. This is because, in a star schema with one very big fact table and multiple smaller dimension tables that need to be joined, the cost of shuffling the fact table and the resulting intermediate results dominates the overall query execution cost. Considering the example query plan shown in Figure 3.4, the fact table could be represented by relation `A` and the dimension tables by `B` and `C`.

Third, by processing the join in the network, it is less sensitive to skew, since with skew present in classical join execution, one node receives more data than the others. As such, when relatively more data is sent to one node than the rest, the network link of the skewed node gets congested and slows down overall execution (also known as incast problem). This is avoided with INP, as each node has its own link connected to the switch (for ToR switches) and as such independent of the key distribution can send with full bandwidth to the switch.

Finally, the INP join scheme accommodates the disaggregated DBMS architecture well. The reason for this is that tables can also be streamed directly from storage nodes through the switch without the need for first performing the shuffle operation on worker/compute nodes.

3.2.2 Query Compilation

The query compilation resembles a physical execution plan for a given query. One important decision in distributed systems during optimization is *where* to execute pipelines optimally. Consequently, the adapted query compilation takes the FPGA switch as a processing unit into account. When employing an FPGA for query processing, it is not feasible to synthesize a complete configuration (a so-called bitstream, i.e., the executable logic on the FPGA) on a query-to-query basis, as bitstream generation can take multiple hours. However, once the bitstream is generated and installed on the switch FPGA, re-configuring the switch to use a different pre-installed bitstream only takes a few milliseconds. Hence, the system allows to install a set of bitstreams for pre-generated pipelines to execute multiple different queries efficiently.

As shown in Figure 3.5 for the example query, each worker is only responsible for sending its part of the relation to the switch, which executes the main query pipelines, i.e., building and probing pipelines for executing joins. To support generic queries inside the switch, the pre-generated pipelines provide different signatures. For instance, the intermediate hash table for table `B` needs to store keys and values of 8 Bytes, whereas table `C` needs 4-byte keys and 10-byte values.

The master node in a distributed DBMS thus tries to choose the best-fitting signature, if there is no exact match it takes the next larger one. This clearly induces memory overhead, e.g., if the relation has a 64 Byte value, then the master chooses the 128 Byte pipelines. However, this should not be a common case, since optimal signatures can be generated as soon as the workload is known.

3.2.3 Evaluation

In the following, the initial results of the new switch design in a distributed database is presented.

Setup and Workload. The experiments were executed on a five-node cluster - one master node and four compute nodes. Each server has an Intel Xeon Gold 5120 CPU processor and 384GB RAM, running Ubuntu 18.04. The four compute nodes are connected to a Zyxel XS3700 switch (without INP) and an FPGA-based switch (with INP capabilities).

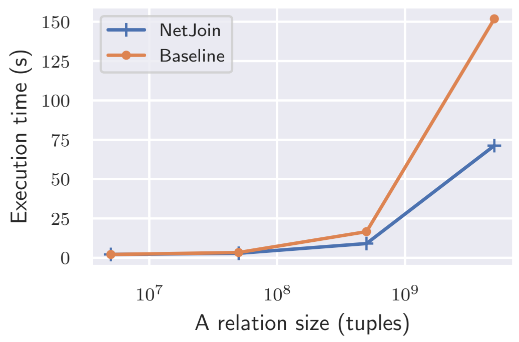


Figure 3.6: Four nodes joining ranging `A` relation sizes ($5e06$ to $5e09$ tuples) with fixed `B`, `C` & `D` relations ($5e07$ tuples). Link speed on each node at $5Gbps$.

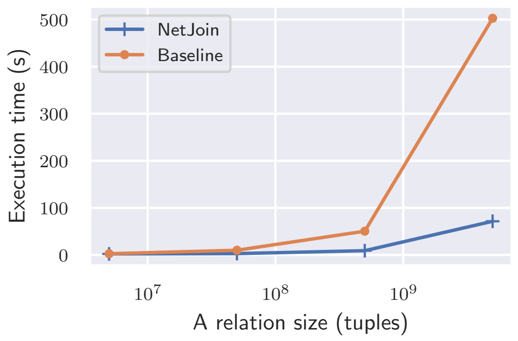


Figure 3.7: Shuffle skew on four nodes joining ranging `A` relation sizes ($5e06$ to $5e09$ tuples) with fixed `B`, `C` & `D` relations size ($5e07$ tuples). Link speed on each node at $5Gbps$.

Based on this setup, multiple experiments were conducted to demonstrate the performance of the proposed architecture (referred to as *NetJoin*) over a baseline without INP. The experiment represents a shuffle-heavy scenario. A table `A` is joined together with three other tables `B`, `C` & `D`. The join shows similarity to a data warehouse setup with `A` being the fact table with foreign keys to the dimension tables `B`, `C` & `D`.

In the setup, all tables are pre-partitioned such that no join partners can be found locally without transferring one of the tuples over the network.

Uniform keys. The first experiment shown in Figure 3.6 scales the size of the `A` relation in comparison to relations `B`, `C` & `D`. The left graph (a) shows the runtime of the distributed hash join over the varying sizes of the `A` relation. The `B`, `C` & `D` relations sizes are $5e07$ tuples, and with the `A` relation ranging from $5e06$ to $5e09$ tuples. Since the join keys are uniform, each of the four nodes receives the same amount of tuples when shuffling the relations.

The results show that as the `A` relation size is small, the *NetJoin* does not perform better than the baseline since reshuffling the intermediate results is inexpensive due to `A`'s small size. As the `A` relation size grows, the *NetJoin* outperforms the baseline. Even though the nodes in the *NetJoin* have to completely send their local partitions of all relations to the switch, the reduced cost of reshuffling compensates for this. For the largest table workload, *NetJoin* outperforms the baseline with over $2\times$.

Skewed keys. As previously mentioned, *NetJoin* is more resilient against skew as in-cast congestion is avoided. To evaluate this, this experiment uses a workload with skewed join keys. The skew is such that when shuffling the relations on four nodes, 80% of all tuples go to Node 1, 13% to Node 2, 5% to Node 3, and the remaining 2% to Node 4.

As shown in Figure 3.7 such a skewed shuffling scenario heavily affects the performance of a distributed join, not only because the compute intensity and memory consumption are not equally distributed, but also because of incast congestion in the network switch. Since Node 2, 3 & 4 all need to send 80% of their local relation to Node 1, the in-going link is acting as a bottleneck and other nodes throttle down their sending rate.

However, with our *NetJoin*, skew on the join key does not play a role since no network shuffling is taking place. Figure 9.9a shows an identical runtime of the *NetJoin*, but with the baseline performance severely suffering in comparison to Figure 9.8a. The speedup shown on Figure 9.9b reports a speedup of 7× for the largest `A` relation size.

3.2.4 Conclusion

This work is motivated by the observation that existing programmable switches cannot process memory-intensive operations due to their limited memory and thus are not suited for distributed query processing.

The main idea is to avoid expensive shuffling operations by offloading more complex query pipelines to the switch. It was shown that the proposed execution scheme can speed up query processing for left-deep join plans by up to 7×.

3.3 Summary

With the presented work on evaluating SmartNICs and offloading join execution to a programmable switch, it was shown that INP has great potential. INP allows for revisiting existing distributed DBMS data structures and algorithms in a new light since it provides a novel way to process data that was previously exclusively possible at the end hosts.

Discussion of In-network Processing for DBMSs. INP for DBMSs is still in its infant years with several open questions and challenges that limit the adoption. These entail:

- *Fault tolerance*: DBMSs have undergone a lot of research to ensure that high availability and durability can still be achieved in the case of failures. With the advent of programmable devices in the network, ensuring fault tolerance is further complicated as these devices typically carry state and are responsible for potentially critical processing. As such, proper failover mechanisms must be devised that allow failures to happen without violating, e.g., any of the ACID properties.

- *System complexity*: Adding heterogeneity in the form of programmable networking devices often comes with the cost of higher system complexity. This is because such devices are typically programmed differently and even have completely different processing schemes to consider than traditional compute units like CPUs. To address this, good abstractions are needed that manage to hide the complexity from the DBMS but still provide the benefits of INP.
- *Cloud offerings*: Leveraging INP in the cloud is often not possible as networking devices are typically not exposed to the cloud users. There are however exceptions such as Microsoft Catapult [36] which exposes FPGA-based SmartNICs. With more and more cloud-native DBMSs, limited access to INP platforms means that such DBMSs are not able to integrate INP into their systems.
- *Multi-tenancy*: In the cloud, multi-tenancy is desired for the cloud provider in order to maximize resource utilization. With, e.g., programmable ToR switches connecting many nodes in a rack serving multiple users, multi-tenancy becomes crucial. However, challenges such as performance isolation, virtualization, and resource partitioning make it a non-trivial problem [109].

These challenges are instrumental for full-scale adoption of INP for DBMSs and form the foundation for future research directions.

4 Realizing Distributed Query Processing on GPUs

This chapter summarizes the work on designing a distributed GPU join operator for fast RDMA-capable networks. The contributions are based on the following peer-reviewed publication.[1]

- **"Distributed GPU Joins on Fast RDMA-capable Networks"** published in *Proc. ACM Manag. Data* 1.1 (2023) [123], (cf. Chapter 10).

  Contributions of the authors: Lasse Beck Thostrup is the leading author and was thus responsible for the implementation, benchmark design, evaluation, and manuscript. The co-author Gloria Doci contributed to the blocking GPU baseline implementation. The remaining authors Nils Boeschen, Manisha Luthra, and Carsten Binnig contributed invaluable feedback. All authors agree with the use of the publication for this dissertation.

As explored in the previous chapter, the network is becoming increasingly heterogeneous with the advent of networking devices offering processing capabilities. The same trend can be observed for compute units, where different devices than CPUs have seen increasing focus. The reason behind this trend is that CPUs are experiencing performance stagnation due to the challenge of continuously scaling the processing power by fitting in more transistors (typically described as the end of Moore's Law and Dennard Scaling). This performance stagnation is however different for other compute architectures such as GPUs and FPGAs. This work therefore looks into the potential of GPUs as a means of speeding up distributed query processing given their high amount of processing power and internal memory bandwidth. As will later be shown, distributed query processing with GPUs is especially interesting in combination with fast RDMA-capable networks as the data transfer to and from devices can be realized very efficiently.

Before diving into the deep end of distributed query processing on GPUs, single-node GPU acceleration is first discussed. In the context of DBMS workloads, most previous

[1]Several passages in this chapter were transferred verbatim from this publication.

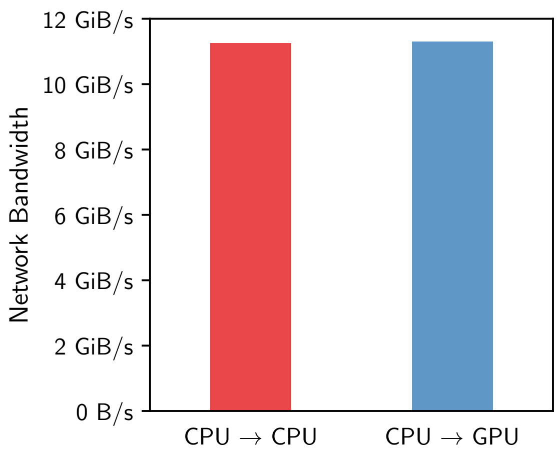
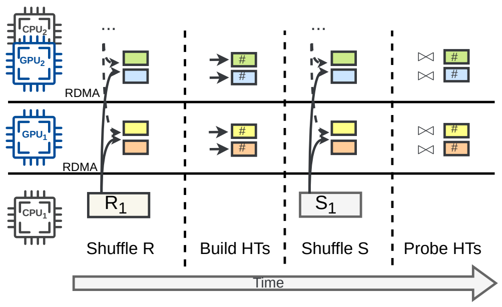


Figure 4.1: Network communication to remote CPUs and GPUs share the same characteristics (reported by *ib_write_bw*).

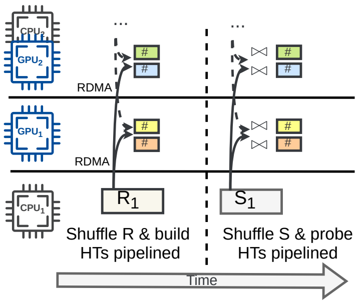
work on the integration of GPUs as accelerators has focused on the acceleration of OLAP workloads for single-node DBMSs [14, 15, 40, 122, 141, 144]. This is because OLAP queries map inherently well to the vectorized execution model of GPUs and hence are clearly an interesting workload for being accelerated by a GPU. This way, the DBMS is able to execute query operators such as joins or aggregations in a massively parallel manner on a single-node DBMS. However, how to scale GPU joins and accelerate join queries in the context of distributed DBMSs is rather unexplored.

This line of work aims to show the potential of GPUs as accelerators in distributed DBMSs on clusters with fast RDMA-capable networks. Here, an important aspect is that using high-speed network cards with GPUDirect RDMA [97, 101], data shuffling over the network has the same cost independent of whether the target of the data transfer is remote CPU memory or remote GPU memory as shown in Figure 4.1. This is the case since with GPUDirect, the network card is able to read and write RDMA data directly into the memory of the GPU without having to relay it through the main memory. As such, in a distributed DBMS where data anyways need to be shuffled for executing the join operators, the higher speed of GPU joins compared to CPU joins can be leveraged without paying any higher cost for transferring data to the GPU.

However, when leveraging GPUs in a distributed setting, the traditional execution scheme is suboptimal for the following reasons. In a typical execution scheme for join operators in distributed DBMSs the data first needs to be shuffled across the network before the operator itself can be executed. For example, in a partitioned join, the data of



(a) Naive Blocking Execution Model


CPU2
GPU2
RDMA
GPU1
RDMA
CPU1
R1
S1
Shuffle R & build HTs pipelined
Shuffle S & probe HTs pipelined
Time

(b) Pipelined Execution Model

Figure 4.2: Two execution schemes: (a) a naive blocking execution where the building and probing phases are distinct from network shuffling of R and S versus (b) a pipelined execution where the GPU execution is overlapped with the network shuffling.

the tables to be joined is first shuffled on the join keys over the network before the join is then executed in parallel on the resulting partitions. A key observation is that using the traditional sequential scheme, the GPUs remain *idle* when the CPU cores execute the shuffle operation. For the distributed partitioned join, this opens up an opportunity to better utilize GPUs during the shuffle operation in a *pipelined* manner by overlapping data transfer and actual join computation on the GPUs.

4.1 Overview

Before looking at the execution scheme of the join, it is important to first clarify the DBMS setup that this work targets. As for typical in-memory distributed DBMS, the database tables are partitioned across the CPU memory of the different nodes. As such, the input tables for the join are located in CPU memory and have to first be transferred to the GPUs for join execution.

This scheme provides benefits over a scheme that stores data to be joined in just GPU memory for the following reasons: First, storing data in CPU memory allows to support joins over input tables (and intermediate results) that are larger than GPU memory. Second, distributed joins typically need to first shuffle the input tables based on the join key. Since GPUDirect RDMA allows data to be shuffled as fast from a CPU as well as from a GPU, the location of input tables does play a large effect on the performance as long as the tables are not pre-partitioned on their join keys.

The Case for Pipelining. The join execution follows a partitioned hash-join where each GPU executes a build and a probe phase over the partitions resulting from shuffling. The main novelty is that the shuffling and join execution is pipelined, which has many benefits, such as support for much larger input tables and even arbitrarily large probe-side tables, together with overlapping of materialization of the join result back to local CPU memory.

This effect is illustrated in Figure 4.2b and contrasted to the blocking execution, as shown in Figure 4.2a. In the following, blocking vs. our pipelining approach is discussed.

Naive Blocking GPU Join. Naively mapping the blocking execution scheme of the state-of-the-art distributed hash join approach [9] to a distributed GPU-accelerated join (as illustrated in Figure 4.2a) does not only come with severe limitations, but it also does not leverage the GPUs in the most optimal manner.

The main reason why a blocking execution scheme of the distributed join is not ideal for GPUs is that the GPU cores would stay idle until the network shuffling phase is finished. The same is true for the building and probing phases where the GPU would be active while the CPU cores would stay idle, resulting in an overall higher runtime as illustrated in Figure 4.2a (i.e., no work is executed on the CPUs in the build and probe phases of the GPU). Hence, this line of work takes the approach of pipelining the execution such that the CPU-driven network shuffling is overlapped with the GPU join processing.

Moreover, another significant limitation of a blocking scheme for GPUs is that only tables of a certain limited size can be processed. The reason is that when executing the phases of a distributed join non-overlapped on the GPU, the GPUs need to be able to hold all intermediate data, e.g., the output of shuffling the build and probe tables in GPU memory. As such, when using a blocking model for executing a GPU-based join in distributed DBMSs, only joins where the input table sizes and intermediate data size do not exceed the aggregated memory of all available GPUs can be supported. In the following, it is presented how the pipelined join approach overcomes these challenges.

Pipelined GPU Join. The pipelined GPU join approach overlaps the execution of the shuffling of input tables with the building and probing on the GPUs. The conceptual reason why the pipelined scheme is more efficient is that such a scheme, as shown in Figure 4.2b, helps to efficiently hide the join processing on the GPUs under the data transfer by making use of CPU and GPU cores concurrently. Moreover, such a pipelined scheme clearly reduces the GPU memory consumption since only a chunk of data, instead of a full partition resulting from shuffling, needs to be stored in GPU memory. For the probe phase, this means that arbitrarily large probe inputs can be supported and streamed through the GPUs. For the build phase, the pipelined model also has benefits since more GPU memory is available for the hash tables, effectively supporting larger build input tables.

Lastly, the pipelining approach allows to overlap not only the probing of the hash tables but also to hide additional processing given the efficient vectorized execution of GPUs by chaining multiple operations (e.g., multiple joins). Such *chained processing* can be used for typical OLAP queries with several joins between dimensions and the same fact table where small (replicated) dimension tables are cached on the GPUs.

4.2 Pipelined GPU Join Design

After having established the best fitting execution scheme for distributed GPUs, the next question is how to design such a pipelined GPU join best. The design options can be categorized into two dimensions: (a) how to use the GPUDirect RDMA communication primitives for implementing the data flow for shuffling data over the network from CPU to GPU memory and (b) the control flow of how the GPU kernel execution is triggered to consume incoming data for building hash tables and probing into them.

For (a), to make use of RDMA on GPUs, GPUDirect RDMA provides a means of transferring data directly over the network from and to the GPU memory using the same RDMA primitives for one- or two-sided communication (RDMA background presented in Section 2.1). However, there are some important differences. Using one-sided primitives with GPUDirect works the same as for CPUs since GPUDirect allows CPUs of the sending nodes can write in the remote GPU memory without involving the remote GPUs. When using two-sided operations instead, it is important to note that the RECEIVE requests are driven by the CPU (and not the GPU) in GPUDirect, since the CUDA library does not support calls to RDMA functions. Hence, in a two-sided communication, the remote CPU is always involved in the data flow even though the GPU is the target.

The other aspect (b) is whether the CPU or the GPU is driving the control flow, i.e., detecting when new data has arrived (on the GPU) and triggering the execution on the GPU for building/probing into the hash tables. With these two design dimensions (one- or two-sided RDMA and CPU- vs. GPU-driven execution) two approaches can be contrasted: CPU-driven execution with two-sided RDMA and GPU-driven execution with one-sided RDMA.

(i) In a CPU-driven approach with two-sided RDMA, the CPU actively detects that a new chunk of tuples has arrived (by polling for so-called RDMA completion events).

Afterward, the CPU then instructs the GPU for subsequent processing by launching a GPU kernel. Important to note with this approach is that a GPU kernel is executed on each batch of arriving tuples. This in turn introduces synchronization overhead for kernel launches and additionally takes up CPU resources.

(ii) In the GPU-driven approach with one-sided RDMA, the kernel launch overhead is removed by using persistent kernels where the GPU instead actively detects new incoming data by polling on a particular memory region for newly arrived data. With persistent kernels, the GPU kernel is only called initially for processing each of the build and probe tables. Polling for new data directly in memory is also often applied in traditional CPU-based RDMA communication as memory polling has a smaller overhead in comparison to polling after RDMA completion events [28, 156].

Summary. Based on these observations the GPU-driven approach with one-sided RDMA is superior due to the fact that it avoids CPU overhead and GPU kernel synchronization. In an experimental evaluation of the two designs, the GPU-driven design can better parallelize and utilize the many cores of the GPUs due to the reduced synchronization. In the following evaluation, the GPU-driven approach with one-sided RDMA is used.

4.3 Evaluation & Methodology

In this evaluation, the GPU-acceleration potential is analyzed along with comparing the pipelined approach to a blocking GPU join. Last, a complete Star-Schema-Benchmark (SSB) query with multiple joins is evaluated.

4.3.1 Setup & Workloads

Setup. All experiments were conducted on a 5-node cluster, each node equipped with two Intel(R) Xeon(R) Gold 5120 CPUs (14 cores) and 512 GB main memory split between both sockets. Each node has two Mellanox ConnectX-5 NICs (100 Gbps) and two Nvidia Tesla V100 GPUs with 16 GB memory, supporting GPUDirect RDMA.

Join Variants. In the evaluation, the following join implementations are used:

- CRJ - *CPU Radix Join:* A state-of-the-art implementation of a CPU baseline (distributed radix hash join [125]).
- GPJ-B - *GPU Partitioned Blocking Join:* This is a distributed variant of the state-of-the-art single-node GPU partitioned join [122]. The shuffle phase (i.e., histogram creation & data shuffling from [9]) and the GPU execution phase are

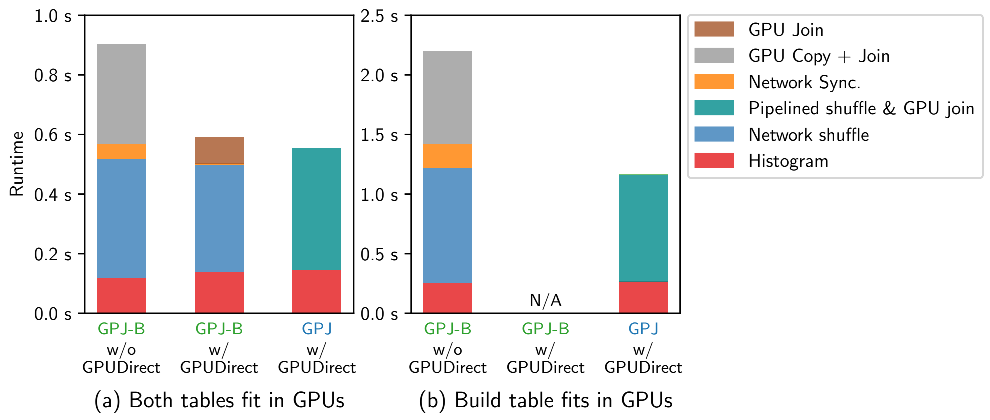


Figure 4.3: Blocking (GPJ-B) vs. pipelined (GPJ) GPU join with build-side of 600×10^6 tuples and probe-side of (a) 1.2×10^9 tuples and (b) 4×10^9 tuples. GPJ provides a speedup of approx. 2× while supporting arbitrarily sized tables.

executed subsequently. Data shuffling is either realized with GPUDirect or without GPUDirect.

- GPJ - *GPU Partitioned Pipelined Join*: The GPU-accelerated distributed join that supports pipelining of the network shuffling and the GPU join phases.

4.3.2 Comparison with a Blocking GPU Baseline:

First, the benefits of the proposed pipelining model are evaluated by comparing it to a distributed blocking GPU join (GPJ-B). The blocking join takes a sequential approach where the two tables are only joined on the GPU once all data has been shuffled.

In this experiment, the input tables are partitioned across 4 nodes such that no two tuples can be joined locally. The tables have randomized tuple order with 16-byte tuples.

GPJ-B is executed both with and without GPUDirect to better show the effect of the direct data path. As shown in Figure 4.3a, using GPUDirect greatly improves the acceleration potential of the GPU. Moreover, we see that the pipelined GPU join, which also uses GPUDirect, can further improve over the blocking GPU join since with pipelining the join phases can be overlapped with data transfers.

Another significant advantage of the pipelined GPU join over the blocking GPU join is that it does not need to accumulate the shuffled tables in GPU memory and as such can support larger joins. In Figure 4.3b, this effect is shown by increasing the probe-side table such that only the build-side table fits on the GPUs. Here, the blocking GPU join is only supported without GPUDirect, whereas the pipelined GPU join can support arbitrary probe-side table sizes and, as such, provides a speedup of approximately 2×.

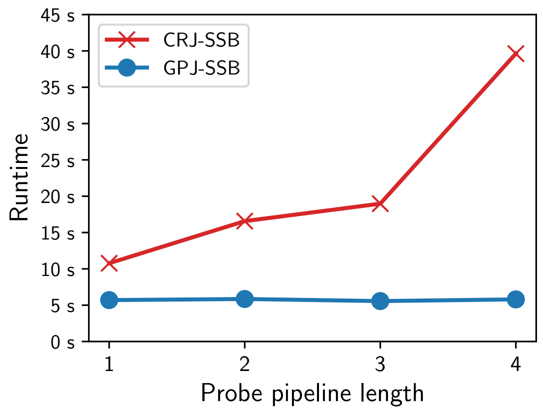
4.3.3 Complete SSB Query

In this experiment, the benefit of the pipelined join is shown for a full query with multiple operations. When running full queries, multiple joins can be *chained* together (e.g., to chain multiple probe steps for a multi-way join in one pipeline on the GPU). The main intuition why chaining on GPUs is beneficial is that GPUs typically have an abundance of processing power in comparison to their i/o speeds. In fact, when executing a single join, as in the previous experiment, there are still untapped computational resources left for chaining multiple operators together.

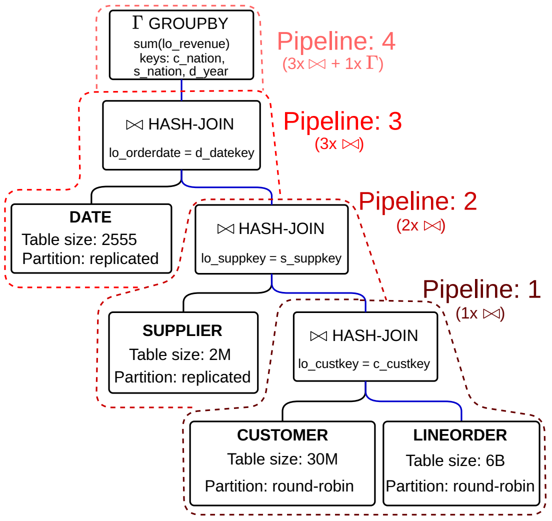
To show these effects of chaining, in this experiment, 3.1 of the Star-Schema-Benchmark (SSB) is used. Query 3.1 involves three hash joins and an aggregation. For comparison, two implementations of the queries are realized: one using only the pipelined join GPJ (referred to as *GPJ-SSB*) with the aggregation also on the GPU, and one that runs completely on CPUs using CRJ (*CRJ-SSB*). For both GPJ-SSB and CRJ-SSB, the same execution strategy is used where first the hash tables are built on the dimensions tables and then chain together the probing of the LINEORDER tuples into these hash tables, followed by a final aggregation.[2]

To test the impact of the size of the probe pipeline and as such the amount of processing needed during the probing stage of the LINEORDER table, the number of joins in the query is varied. The experiment is based on SSB query 3.1 as shown in Figure 4.4b, but with a varied number of joins in the query (pipeline length) and without any filters on the dimension tables. For instance, a probe pipeline length of 2 will join together the LINEORDER with CUSTOMER and SUPPLIER and a length of 4 contains the complete query. As can be seen in Figure 4.4a, the GPU-accelerated query execution (GPJ-SSB) is not affected by the query size since the additional probing is all hidden under the network shuffling. This is in contrast to the CPU-only approach (CRJ-SSB) where more joins result in a higher runtime. For the full query without dimension table filters, a speedup of 6.8× can be observed over CRJ-SSB.

[2]PART and CUSTOMER are partitioned while DATE and SUPPLIER are replicated to enable chaining.



(a) Query Runtime



(b) SSB Query 3.1 Plan w/o Filters

Figure 4.4: Execution (a) with a different number of joins of SSB Query 3.1 (b) with SF 1000, on 4 nodes. Probe-side joining is chained together both in CRJ-SSB and GPJ-SSB. With longer probe pipelines, the runtime of GPJ-SSB is unaffected due to its pipelined design resulting in a reduction in runtime by up to 6.8× against CRJ-SSB.

4.4 Related Work

In the context of high-speed networks, join processing for scale-out distributed DBMSs has been studied by a few works [9, 10, 38, 39]. Barthels et al. [10] implemented a distributed radix hash join over RDMA networks by utilizing efficient one-sided RDMA primitives. While the authors do not explore GPU acceleration, many findings of their work are still applicable, such as the efficiency and use of one-sided RDMA.

The approach by Guo et al. [43] is closest to this line of work, which also explores distributed GPU joins over RDMA. However, they cover only more naive blocking GPU joins and assume a fundamentally different setup where database tables are already stored in GPU memory.

4.5 Summary

Accelerators such as GPUs have the potential to redesign existing database operators to overcome previous bottlenecks. This work showed the possibility of leveraging the high processing power of GPUs for distributed operators in connection with fast RDMA networks. The main finding is that accelerators are especially beneficial when they can be used without having to pay an additional transfer cost compared to the traditional execution plan. This finding is unique to distributed query processing as with fast RDMA-capable networks, data can be moved in and out of GPUs with the same throughput as for main memory.

However, while GPUs are arguably the most common type of accelerator, this work leaves the challenge of query processing over more heterogeneous hardware as an open problem. This challenge is addressed in the next chapter by introducing a network-driven approach to achieving RDMA communication between various different types of accelerators.

5 Network-driven Communication for Accelerators

This chapter summarizes the work on realizing a network-driven communication scheme for heterogeneous accelerators. The contributions are based on the following peer-reviewed publication and paper currently under review.[1]

- **"Zero-sided RDMA: Network-driven Data Shuffling"** published in *Proceedings of the 19th International Workshop on Data Management on New Hardware, DaMoN 2023, Seattle, WA, USA, June 18-23, 2023* [58], (cf. Chapter 11).

  Contributions of the authors: Lasse Beck Thostrup and Matthias Jasny are both the leading authors and contributed equally to the implementation, benchmark design, evaluation and manuscript. The remaining author Carsten Binnig contributed invaluable feedback. All authors agree with the use of the publication for this dissertation.

- **"Zero-sided RDMA: Network-driven Data Shuffling for Disaggregated Heterogeneous Cloud DBMSs"** published in *Proc. ACM Manag. Data* 2.1 (Mar. 2024) [59], (cf. Chapter 12)

  Contributions of the authors: Lasse Beck Thostrup and Matthias Jasny are both the leading authors and contributed equally to the implementation, benchmark design, evaluation and manuscript. Sajjad Tamimi contributed to realizing zero-sided RDMA communication to an FPGA. The remaining authors Andreas Koch, Zsolt István, and Carsten Binnig contributed invaluable feedback. All authors agree with the use of the publication for this dissertation.

[1]Several passages in this chapter were transferred verbatim from these publications.

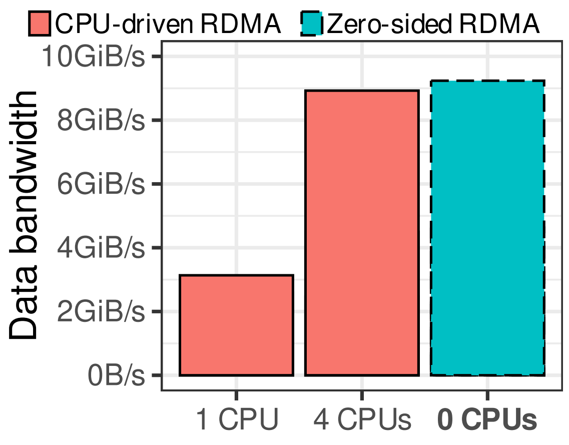


Figure 5.1: GPU to GPU network transfers, driven by either the CPU or the network (zero-sided RDMA).

5.1 The Need for Network-driven Communication

A significant trend seen in the cloud data centers is that, in response to the performance stagnation of CPUs, heterogeneous accelerators are becoming commonplace as an alternative to CPU-based compute resources [95]. Accelerators have been shown to provide significant speed-ups for DBMS workloads as demonstrated in the previous chapter. However, looking ahead, a major question for future cloud DBMSs is how to efficiently use disaggregated heterogeneous compute resources such as GPUs or FPGAs. Similar to disaggregated CPU-based compute resources, these accelerators should be deployed as network-attached accelerator pools, but today this is not the case. Accelerators depend on the presence of a host CPU for executing the control flow of moving data in and out of the accelerator. This strong coupling, however, both introduces CPU overhead, but also limits the acceleration potential of accelerators due to the increased coordination. To overcome this, a zero-sided RDMA communication scheme is proposed to remove the CPU from the critical path and enable efficient accelerator-to-accelerator data movement powered by a programmable switch.

The pitfalls of RDMA for accelerators. As already covered in Section 2.1, RDMA provides different primitives where the READ and WRITE operations only involve the initiator CPU of the communication channel. This already helps to integrate accelerators on the target side as shown in Chapter 4, because the target side communication channel can be realized with only memory operations. However, if we want to fully utilize accelerators in the cloud, the involvement of CPUs, even if just on the initiator side, will lead to bottlenecks. As seen in Figure 5.1, multiple CPU cores need to be dedicated to coordinating data transfers on behalf of the GPU.

Accelerator-driven RDMA as an alternative? One way to remove the CPU would be to implement RDMA-based operations directly on the accelerators. Even though this is technically possible [73], it is challenging for the following reasons: (1) RDMA primitives, such as one-sided RDMA verbs, might not be available for a given accelerator, requiring a full RDMA stack to be implemented per accelerator type. Beyond the engineering cost, the additional problem is that executing communication logic on specialized hardware devices consumes compute resources that could otherwise be utilized for processing. (2) Implementing advanced RDMA-based communication schemes, such as many-to-many data shuffling or multicast, on top of the RDMA stack on each accelerator type requires re-implementing features and has different challenges and limitations on each hardware type.

The case for network-driven RDMA. To remove the CPU from the critical path and to reduce the engineering effort in disaggregated accelerators, in this work, a novel network-driven scheme is proposed where neither a CPU co-located with an accelerator is actively needed nor a custom RDMA implementation on the accelerator. The approach fully offloads the RDMA stack and the RDMA-based communication logic between devices to the network, particularly to a programmable switch. The main idea of offloading the communication logic to the network is that the switch acts as a coordinator of data transfers; i.e., it issues an RDMA READ to a sender and rewrites the read response into an RDMA WRITE for the receiver using the programmable data plane of the switch. Since this does not involve the sender or receiver actively, this communication scheme is termed *zero-sided RDMA*.

In this work, a programmable off-the-shelf Tofino switch is used in contrast to the FPGA-based switch used in the previous Section 3.2. The reason is that, as will later be explained, realizing zero-sided RDMA does not require a lot of state but instead relies on fast line-rate processing which is guaranteed by the ASIC-based pipelined execution model of Tofino switches.

5.2 Zero-sided RDMA Overview

Zero-sided RDMA builds on the ability of the centralized programmable switch to initiate and direct data traffic directly in the data-plane at the aggregated line rate of all connected

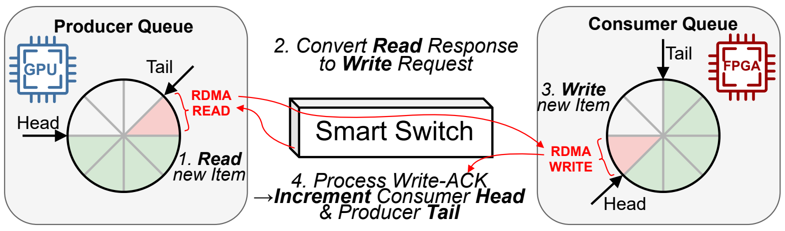


Figure 5.2: In zero-sided RDMA, producers & consumers interact only with their buffers in local memory. The switch reads items from the sender buffer (step 1) and moves them to the receiver buffer (step 3) by converting RDMA READS into WRITES (step 2). The switch coordinates the communication using head/tail pointers (step 4).

accelerators — referred to as programming units (PUs). By placing the communication scheme in the switch data-plane, the switch can initiate one-sided RDMA operations (READ & WRITE) to PUs. At the same time, the PUs are completely oblivious to the network communication scheme, which not only has the benefit that the PUs do not have to issue communication primitives actively but also removes the distributed coordination for communication flows such as data shuffling or even replication.

Communication abstractions. On the side of the PUs (i.e., accelerators), a circular buffer is the core abstraction to participate in zero-sided RDMA data transfers. The design goal of the buffer is to allow PUs to push and pop items with only simple local memory operations while the switch transfers data fully asynchronously without any sender/receiver PU involvement. The coordination between the switch and PU is handled with two pointers to the buffer, a head and tail pointer. The head pointer indicates where the next data item can be written, and the tail pointer indicates where the next data item can be read.

Flow of data transfers. The overall flow of transferring a data item between one producer and one consumer using zero-sided RDMA is illustrated in Figure 5.2. The switch mirrors the state of the producer and consumer buffers (head and tail pointers) to know when a producer has an item to send and whether the consumer has free space. To transfer an item, (1) the switch first issues an RDMA READ on a data item in the producer buffer. (2) The READ response is then converted on the switch into an RDMA WRITE and (3) written into the next free slot at the consumer. When converting a READ response into a WRITE request, the switch does not need to buffer or modify the data payload, only the header of the network packets. (4) After the remote NIC acknowledges the RDMA WRITE, the head pointer of the consumer is incremented to indicate the new item. The tail pointer is incremented on the producer, which frees up the item in the buffer for reuse.

Hardware requirements. Finally, the aim of zero-sided RDMA is to be able to integrate many different heterogeneous devices, as such only a few requirements must hold for participating in zero-sided RDMA: (1) the device must have memory in which to store the buffer data structure, (2) the memory must be accessible by an off-the-shelf RDMA-enabled NIC, e.g., with PeerDirect[2] and (3) the memory consistency model must ensure that a write to an item and the subsequent update to the head pointer is executed in order. In the case of weaker memory consistency models, this behavior is most commonly achievable through memory fence primitives.

5.3 Realizing Zero-sided RDMA

To realize zero-sided RDMA where no active involvement is required for producers or consumers, the communication scheme has to be offloaded to a programmable switch. In the following, the core challenges are summarized and afterwards, three aspects of the design are discussed.

Core challenges of realizing zero-sided RDMA. A programmable Tofino switch follows a pipelined execution model with only a limited set of instructions and memory per stage. In a switch-driven scheme, all connected PUs are not directly connected to each other but are connected to the switch. As such, the switch must implement and adhere to the exact protocol to be compliant with off-the-shelf RDMA NICs. This challenge includes managing stateful RDMA connections with reliability and correctly propagating congestion- and flow-control information from the consumers to the producers within the switch's data-plane to guarantee execution at line-rate.

RDMA switch state. In order to make the switch able to initiate and steer RDMA traffic, it must store and maintain the necessary state. The state can be grouped into the static connection state which does not change during communication and the dynamically changing state. For RDMA, the static connection state includes a remote key and destination queue-pair number to identify the remote RDMA queue-pair, along with its IP and MAC addresses. Since none of these header fields change during communication, it is stored on the switch upon connection setup. However, for the dynamically changing state, such as packet sequence numbers (PSNs) and virtual remote addresses, these need updating on a per-packet basis and therefore need to be stored directly in registers on

[2] https://enterprise-support.nvidia.com/s/article/howto-implement-peerdirect-client-using-mlnx-ofed

the data-plane. Since the switch is in the middle of the communication channel between the PUs, the switch must maintain PSNs and addresses for all connected end-points. In addition to the RDMA-specific state, also the PU buffers' head and tail pointer values are stored in switch registers.

Initiating data transfers. As a first step before reading the first item (step 1. in Figure 5.2), the switch evaluates whether it is possible to initiate the data transfer.

In the switch pipeline, it is checked whether the producer has items in the buffer to send and whether the consumer has space. This is realized by passing through a network packet in the pipeline stages which evaluates these statements based on the head and tail pointers. If neither the producer has data nor the consumer has space available, this packet is recirculated to the beginning of the pipeline. Recirculating the packet is necessary because while-loops are not supported in the pipelined processing model. If these checks pass, the packet is forwarded through the pipeline, where necessary header fields are set to create an RDMA READ request to the producer which adheres to the RDMA protocol.

Data payload transfer. In the second step (2. in Figure 5.2), the read response containing the data payload to transfer to the consumer is streamed through the switch. Here the switch rewrites the response of the RDMA READ into an RDMA WRITE. Rewriting the RDMA type entails updating the relevant header fields to change the operation type and to make it adhere to the correct packet sequence numbers and virtual address expected by the consumer. This step does not require the switch to buffer any packets but will be forwarded at line-rate.

The read response from the producer might be fragmented into several packets. The challenge here is that the network might reorder these packets such that naively assigning new packet sequence numbers (PSNs) to the RDMA WRITE packets in the order they arrive might produce incorrectly written data on the consumer. Instead, an offset is calculated on the fly between the producer-side and consumer-side sequence numbers such that if two packets are reordered, the assigned PSNs on the consumer-side will reflect this, allowing the target consumer-side NIC to assert the right order for the memory write.

5.4 Zero-sided Communication Flows

Switch-driven data transfers can provide more advanced flows than the 1:1 connection illustrated in Figure 5.2. In fact, with zero-sided RDMA, many producers can be connected to many consumers in a flow. In addition, different flow features are supported such as load balancing, replication, or fine-grained quality of service guarantees. In the following, the load balancing and replication features are presented.

Load balancing. Distributing work evenly across processing units in a distributed DBMS is non-trivial, given unforeseen network congestion or processing contention. Multiple schemes have been devised to overcome this [31, 77, 155], which require additional coordination overhead, e.g., through work-stealing or a centralized server-side dispatcher.

With zero-sided RDMA, a communication flow between N producers and M consumers can automatically provide load balancing across all consumers. This flow is realized without any form of producer- or consumer-side coordination since the switch will transparently initiate the data transfers between producers and consumers.

The key to achieving this communication flow is by introducing consumer-specific thread-like processing on the switch, where it is evaluated when a data transfer can be initiated from any of the producers, essentially adapting the data transfer rate from all producers to each consumer independently.

Replication with global ordering. Lastly, zero-sided RDMA also provides a replication-based communication flow in which items from each producer are multicasted out to all consumers.

Multicast has many applications in distributed DBMSs, such as replicated joins [126], or state replication [51, 82] for providing availability. While RDMA already has multicast capabilities, it is only supported through the Unreliable Transport and two-sided verbs. As such, it comes with the cost of higher CPU overhead at the communication endpoints due to the two-sided communication and the cost of ensuring reliability. With zero-sided RDMA, data replication can be realized reliably without any coordination or computational involvement on the processing units by initializing, steering, and multicasting the data directly in the data-plane of the switch.

Traditionally, effects like reordering of packets in the network can cause the received data at each consumer to observe a different order. However, as the zero-sided approach transfers data sequentially with separate acknowledgments from each consumer, it can be ensured that data transfers to all consumers are globally ordered without introducing any overhead at the processing units.

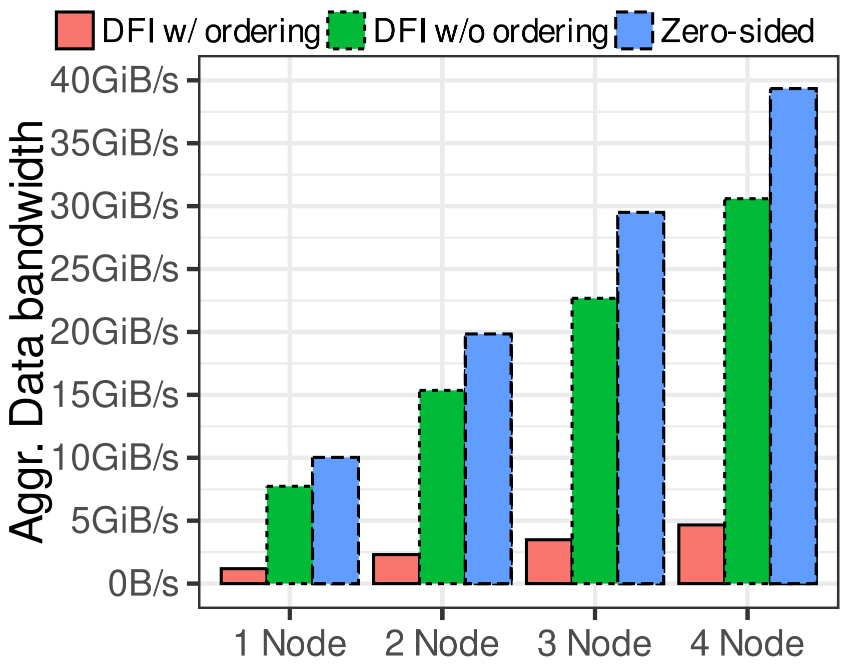


Figure 5.3: 1:N zero-sided replication compared to DFI [126] with or without global ordering. Zero-sided replication always ensures a global order observed by all consumers.

5.5 Evaluation & Methodology

In the following section, zero-sided RDMA is first evaluated in a replication scenario and subsequently, the benefit of switch-driven data transfers is investigated in the context of analytical distributed database systems.

Setup and implementation. Zero-sided RDMA is evaluated in a cluster of 4 nodes running Ubuntu 18.04 LTS with Linux kernel 4.15.0. Each node is equipped with an Intel(R) Xeon(R) Gold 5120 2.2GHz CPU, an Nvidia V100 GPU, and a Mellanox ConnectX-5 MT27800 that is connected to an Intel Tofino switch (BF2556X-1T) [93] via 100G RoCEv2. In addition, one node is equipped with a Xilinx Alveo U55C FPGA. The code is written in C++20 is compiled with gcc-12 and CUDA-12. The switch's control-plane logic is implemented in C++, and the switch's data-plane logic is implemented in P4 and compiled using Intel SDE-9.11.0 [53].

Replication with ordering. The first experiment centers on the effectiveness of zero-sided RDMA replication which by design ensures global ordering for each consumer. For the baseline, the replication flow of the Data Flow Interface (DFI) [126] (presented in Chapter 2) is used. The DFI replication flow enables replication using RDMA multicast

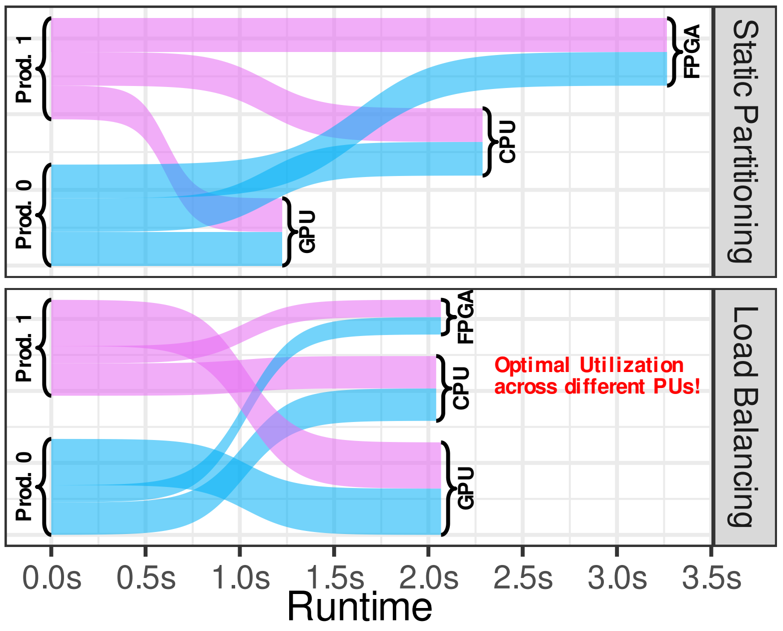


Figure 5.4: Static partitioning vs. load balancing for TPC-H Query 1 with SF 100 executed on CPU, GPU and FPGA.

with two-sided SEND/RECEIVE operations. The results for DFI are presented both with and without software-based ordering. For the experiment, CPUs are used as PUs.

The results in Figure 5.3 show the aggregated data bandwidth of all consumers for a different number of nodes that receive data from a single producer. Across all scenarios, zero-sided replication with ordering consistently demonstrates superior performance, nearly saturating the link bandwidth at each consumer node. In addition, zero-sided replication does not infer any overhead at the PUs. In stark contrast to native RDMA multicast that only supports two-sided SEND/RECEIVE operations and, as such, introduces communication overhead at not only the sender but also the receiver.

Load balancing on heterogeneous devices.

Scheduling processing jobs across a wide range of heterogeneous processing devices with different and varying throughputs is a challenging task that requires careful coordination between all participants. Therefore in this section, the load-balancing communication flow is evaluated in a setup that mimics a cloud data center with disaggregated storage and heterogeneous compute resources that consist of CPU, GPU, and FPGA.

To enable zero-sided RDMA on an FPGA, the FPGA local memory is exposed through the PCIe bar register which allows the RDMA NIC to issue DMA operations to it.

Important to note is that no RDMA stack needs to be implemented for the FPGA but only the PCIe bar memory mapping.

In this experiment, depicted in Figure 5.4, data producers on the storage are spread out on 2 (CPU) nodes. The workload is based on TPC-H Query 1. The system's performance is assessed under two distinct scenarios: static partitioning and load balancing.

The input TPC-H table *lineorder* is partitioned into two equal sizes at both producer nodes. In the scenario with static partitioning (Figure 5.4, upper part), the producers send the data in equal portions to three consumers: an FPGA, a GPU, and a CPU. The results show that the GPU completes its tasks more rapidly than the CPU, which is faster than the FPGA as the slowest device. Overall, the FPGA takes a total of 3.2 seconds, which dominates the end-to-end latency of the query.

In contrast, with load balancing (Figure 5.4, lower part), the *lineorder* table is dynamically distributed to each consumer based on each device's processing speed. This arrangement enables the CPU, FPGA, and GPU to finish their tasks simultaneously, thus reducing the total runtime of the query to 2.1 seconds and ensuring optimal utilization with no idle time for either processing unit.

These results underscore the potential of load balancing in zero-sided RDMA: it allows for optimal utilization of different processing units without the need for complex partitioning or work-stealing schemes, leading to more simple query execution code.

5.6 Related Work

Addressing a similar problem as this line of work is Lynx [128] and FpgaNIC [136]. These systems, however, take a different approach in that they rely on SmartNICs for driving the communication on behalf of the accelerator (in these cases, GPUs).

However, such a decentralized SmartNIC-driven approach has several downsides. First, realizing one-sided RDMA communication schemes is highly complex due to the distributed coordination for remote memory accesses. Adding to the fact that typical CPU-based SmartNICs are too weak to saturate line-rate throughput for many scenarios properly as seen in Chapter 3, SmartNIC-driven communication schemes are best realized on more performance-efficient compute architectures such as FPGAs. This, in turn further adds to the complexity when considering decentralized one-sided communication schemes. Second, since a dedicated SmartNIC is needed per server, the hardware cost is directly proportional to the number of servers. As the cost of a SmartNIC can be up to

10× more expensive than a normal RDMA-enabled NIC for the same link speeds, the added cost for a decentralized solution is non-negligible.

5.7 Summary

In this work, zero-sided RDMA was introduced as a way to enable direct RDMA-based accelerator-to-accelerator communication, which does not require CPUs to coordinate the communication because the communication scheme is executed by the network. Moreover, zero-sided RDMA facilitates data shuffling between heterogeneous hardware devices without the need to implement a complete RDMA stack on each heterogeneous device. The evaluation showed that zero-sided RDMA flows provide useful and efficient features such as replication and load balancing that do not introduce any complexity at the processing units.

6 Conclusion

In this thesis, multiple contributions were presented that aim to accelerate distributed databases by bringing the benefits of modern networks within reach. In the following, the thesis is summarized and reflected upon, and subsequently, an outlook on future research directions is presented.

6.1 Summary

The goal of this thesis is to bridge the gap between databases and modern networks in terms of understanding performance traits, proposing new accelerated distributed operators, and suggesting abstractions to allow fast and efficient networking without low-level complexities.

In Chapter 2, an interface was proposed that aimed to bring the benefits of fast networks and RDMA to data processing systems by abstracting away the low-level details and complexities of RDMA. The objective of the interface is to raise the level of abstraction without trading off the achievable network performance. Four key design principles were identified that are important for data processing systems, such as exposing declarative optimization goals to the application for adapting to the needs of various application demands, be it latency-sensitive or bandwidth-demanding workloads. The evaluation showed that the interface can achieve comparable performance to a low-level RDMA API while being able to heavily simplify the implementation of a distributed join use case.

Chapter 3 takes a step further into the network by evaluating the effectiveness of in-network processing for DBMS use cases. The goal of this chapter is to achieve a deeper understanding of where and when DBMSs can benefit from offloading processing into the network. The first half of the chapter evaluates the performance of a CPU-based SmartNIC in two DBMS-centric use cases; a remote B-tree and a remote sequencer. The findings show that the embedded CPU is substantially weaker than traditional server CPUs and as such two-sided RDMA degrades the overall system performance when too much workload is offloaded to the SmartNIC. However, in unison with the server, speedups can be detected which underlines the fact that hardware characteristics must be taken into close consideration when designing new in-network processing systems. In one-sided RDMA workloads, the SmartNIC achieves a lower latency compared to the main memory of the host which is due to its close proximity to the network. In the second half of the chapter, a distributed database join is realized on an FPGA-based switch to evaluate the potential performance benefits of memory-intensive operations in the network.

Subsequently, in Chapter 4, fast RDMA-capable networks are combined with GPU acceleration by implementing a distributed database join. An important discovery in this work is that the high processing power of GPUs can be leveraged without any additional data transfer cost as compared to CPU-based processing for data that already has to be sent over the network. This is enabled by GPUDirect RDMA, by allowing the NIC to do DMA directly to the GPU, and thus does not require relaying the network data over the main memory. A pipelined execution scheme is derived as it avoids fully persisting the database tables in the limited GPU memory and in addition reduces overall runtime by processing the join during network shuffling. The evaluation showed that a performance speedup can especially be seen for full queries where multiple operators can be pipelined and executed on the GPU, essentially hiding the processing cost under the unavoidable network shuffling.

Lastly, Chapter 5 suggests a network-driven communication scheme for various different accelerators. The work is motivated by the challenges of achieving direct accelerator-to-accelerator communication in heterogeneous systems without having to pay the cost of interleaving the CPU as a central part of the control-flow. By offloading the communication scheme into the network on a programmable switch, accelerators can be passive in the network communication since they do not have to rely on an RDMA-stack or implement the communication scheme directly on the device. The intuition behind this contribution is that by initiating one-sided RDMA from the switch, the accelerators are only on the target side of the one-sided communication and as such are oblivious to the network communication. Since both accelerator end-points of a network-driven communication channel are passive, this communication scheme is termed zero-sided RDMA. In addition to removing the communication burden from the accelerators, zero-sided RDMA also provides a set of features for many-to-many connection flows. These features entail replication, load balancing, and fine-grained quality of service which all are realized without introducing any complexity or overhead on the accelerators. The key takeaway is that by letting accelerators solely do data processing and driving the communication from the network, a separation of concerns can be achieved which both allows realizing novel communication flows but also simplify the processing on accelerators.

6.2 Future Research Directions

In the following, the future research directions are discussed. First, the immediate future directions for network-driven communication are presented, and afterward a general discussion of modern networks.

6.2.1 Network-driven Communication

As a continuation of the work presented for network-driven communication, a future direction is to build a full system capable of dynamically processing on different network-attached accelerators. To achieve this, managing the complexity of realizing efficient processing on many different types of compute devices is essential. Therefore, abstractions for heterogeneous processing are needed which target typical data processing use cases. An example of existing work is Tensor Query Processing [47] which leverages advances in tensor processing on modern hardware from the ML community for analytical query processing. In addition, Jungmair et al. [64] have proposed to break down data processing operators into a set of unified sub-operators that can be compiled and optimized to modern hardware. Therefore, a future direction is the integration of query processing on heterogeneous modern hardware with network-driven communication to achieve the vision of a distributed data processing system over heterogeneous hardware without inefficient reliance on CPUs.

Another future direction is to conduct experiments on a larger scale that goes beyond a single-rack with multiple programmable switches. Here, interesting challenges may arise such as the effects of increased latency between the switch and connected accelerators or the impact of network congestion. By incorporating multiple switches in network-driven communication there exist also new opportunities such as the ability to provide higher availability with a fail-over scheme in case of switch failures. This is possible as the programmable switch driving the communication only duplicates the state from the accelerator buffers. As such, when instructed, a new switch can take over the responsibility of driving the communication flow.

6.2.2 The Future of RDMA

As already stated, RDMA is on the rise in the cloud due to its many benefits such as zero-copy, kernel by-pass, and very low CPU overhead [5]. RDMA is however mainly only used by cloud providers for internal systems and is only exposed as a service in a lesser regard, typically in the form of expensive HPC instances. An open question for the future of RDMA networks in the cloud is in which capacity cloud tenant systems will be able to leverage RDMA. Related to this question, there exist a few current issues with InifiniBand-based RDMA (RoCE & InfiniBand) which might hinder the wider exposure of RDMA in the cloud. An example is in the domain of security where it has been shown that RDMA has several vulnerabilities [113]. In addition, Hoefler et al. argue that RDMA over Ethernet inhibits scalability problems due to simple assumptions made for load balancing, congestion control, and error handling [49].

To this end, other RDMA protocols have been developed to overcome some of the challenges inherent to InifiniBand-based RDMA. Examples are 1RDMA [121] and EFA [118] which both trade off ordering guarantees to support more flexible routing and only provide a reduced set of primitives. For example, no atomic operations or RDMA writes are supported directly by 1RDMA or EFA.

Whether a whole new high-performance networking technology is introduced or the issues of RDMA are addressed, the demand for fast and efficient networking in the cloud will only grow.

6.2.3 Hardware Evolution

The evolution of hardware continues to reshape the architecture and functionality of DBMSs. This thesis already delved into the integration and optimization of DBMSs with SmartNICs, programmable switches, GPUs, and FPGAs. However, as we look ahead, it is crucial to consider the potential advancements of hardware and the transformative influence they might have on DBMSs.

The Future of Networking Hardware. Networking hardware is consistently advancing, presenting opportunities for faster data transfers and optimized communication. With the current interconnect speeds of 400 Gbps and 800 Gbps around the corner, a major future challenge lies in designing networked systems. The goal is to fully utilize and take full advantage of these high-speed networks. This is, however, already a challenge with even 100 Gbps for many systems, e.g., CPU-driven data shuffling [85]. The reason behind this is that when computation and communication are overlapped, it becomes increasingly hard to issue data transfer requests at a rate fast enough to saturate the network. As such, moving forward, extra emphasis must be put on how to efficiently make use of the ever-increasing network speeds.

In addition, this thesis explored the opportunities for leveraging programmable networking devices for DBMS tasks and data structures. This field is still very young and there exist many open questions on how and where to best integrate modern networks for DBMSs. As we strive to remove existing network-related bottlenecks, benchmarking and new architectures will be integral to utilizing the full potential of the future's networking hardware.

The Role of the CPU in Future DBMSs. Traditionally, the CPU has been the heart of computation. But as specialized hardware units like GPUs and FPGAs demonstrate strong performance benefits, it raises a fundamental question: What is the future role of the CPU in the DBMS landscape?

The current development of CPUs is going in the direction of embedding accelerator-type processing capabilities into the CPU. We already see vectorized processing features such as SIMD (Single Instruction, Multiple Data) embedded into CPUs, but Intel aims to take this even further by introducing even more specialized accelerator components such as Intel IAA (In-Memory Analytics Accelerator) [52] and Intel DSA (Data Streaming Accelerator) [55]. With so many accelerators embedded into CPUs, it begs the question of whether the CPU can continue to be competitive against a set of interconnected discrete accelerators.

Recent research papers present another vision with entirely CPU-less servers [95, 129]. A radical approach that reimagines the roles and responsibilities of server hardware. While the feasibility of such designs remains to be seen, they undeniably present an interesting direction for future research.

In conclusion, the evolution of hardware promises transformative changes for DBMSs. This introduces many interesting future research directions for DBMSs for reinventing the traditional system design to pave the way for optimized, efficient, and versatile DBMS architectures.

Part II

Peer-Reviewed Publications

7 DFI: The Data Flow Interface for High-Speed Networks

Abstract

In this paper, we propose the Data Flow Interface (DFI) as a way to make it easier for data processing systems to exploit high-speed networks without the need to deal with the complexity of RDMA. By lifting the level of abstraction, DFI factors out much of the complexity of network communication and makes it easier for developers to declaratively express how data should be efficiently routed to accomplish a given distributed data processing task. As we show in our experiments, DFI is able to support a wide variety of data-centric applications with high performance at a low complexity for the applications.

Bibliographic Information

The content of this chapter was previously published in the peer-reviewed work: Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. "DFI: The Data Flow Interface for High-Speed Networks." In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. Best Paper Award. ACM, 2021, pp. 1825–1837. DOI: 10.1145/3448016.3452816. URL: https://doi.org/10.1145/3448016.3452816.

7.1 Introduction

Motivation. Scale-out data processing systems are the typical architecture used today by many systems to process large data volumes since they allow applications to increase compute and memory capacities by simply adding further processing nodes. However, a typical bottleneck in scale-out systems is the network which often slows down the speed of data processing if communication is in the critical path. For distributed in-memory systems this might lead to degraded performance when adding more nodes [110].

However, this changed with the advent of high-speed networks such as InfiniBand. Network bandwidth increased almost up to the speed of main memory and latencies dropped by orders of magnitude [11], making scale-out solutions more competitive. However, blindly upgrading to faster networks does often not directly translate into performance gains, as there is a plenitude of aspects to consider to achieve a good performance for distributed data processing systems.

One particular important aspect to efficiently use high-speed networks is to redesign data processing systems to leverage remote direct memory access (RDMA) as a low overhead communication protocol. RDMA provides kernel bypass and zero-copy making data transfers less expensive than classical network stacks such as TCP/IP [37]. In recent years, industry and academia have thus started to adapt scale-out data processing systems in order to make use of RDMA. As a result, significant speed-ups have been shown for a wide range of data processing systems ranging from key-value stores [69, 80, 91], over distributed DBMSs (for OLTP and OLAP) [11, 68, 138, 143, 147, 148] to Big Data systems and Distributed Machine Learning [62, 87, 142].

However, using RDMA is complicated because it provides only low-level abstractions (called RDMA verbs) for data processing [27]. Hence, redesigning data processing systems for RDMA often requires significant efforts to take care of many low-level detail choices [10, 11, 66, 68, 157] regarding remote memory and connection management as well as other decisions such as which RDMA verbs to use for which type of workload.

Contribution. In this paper, we propose the Data Flow Interface (DFI) as a way to make it easier for data processing systems to exploit high-speed networks. Accordingly, DFI defines abstractions and interfaces suited to a broad class of data-intensive applications, yet simple enough for practical implementation with predictable performance and low overhead relative to “hand-tuned”, ad hoc alternatives. In designing a high-level interface tailored to data processing, we adopt the approach taken by the high-performance community for MPI [42] to provide a simple yet effective interface for high-speed networks. However, since MPI has been designed for computation-intensive workloads such as large-scale simulations, it comes with many design choices that are not optimal for data-intensive workloads [70]. Consequently, MPI has seen only very limited adoption for data processing systems [9].

In brief, the main idea of DFI is that data movements are represented as *flows*. DFI flows are an abstraction providing primitives for efficient network communication. These primitives are intended to be used as a foundation for building data-intensive systems and provide many benefits over MPI (e.g., thread-centricity, pipelined communication). By lifting the level of abstraction, DFI flows not only hide much of the low-level complexity of network communication but also allow developers to declaratively express how data should be efficiently routed to accomplish a given distributed data processing task. Moreover, DFI flows allow developers to specify *optimization hints*; e.g., to maximize bandwidth-utilization or minimize network latency of transfers. By using flows as the main abstraction, DFI supports a wide variety of data-centric applications ranging from bandwidth-sensitive distributed OLAP to more latency-sensitive workloads such as distributed OLTP or replication with consensus protocols.

Recently, the need for better interfaces to high-speed networks has also been discussed in a vision paper [3]. We, however, are the first paper that provide a concrete suggestion and a full implementation for an interface that can enable a broad class of data-centric applications to make efficient use of modern networks. Moreover, there have also been several other attempts to build libraries for data processing over high-speed networks [18, 28, 35]. For example, FaRM [28] and GAM [18] provide a programming model based on a shared address space which focuses on supporting latency-sensitive workloads (e.g., such as distributed transactions). Another example is L5 [35], which target the communication between clients and servers to replace traditional client-centric communication libraries such as ODBC. Different from those libraries, as mentioned before, DFI flows aim to be a much more general abstraction that can support a broader class of data-centric applications.

Finally, like MPI and different from the approaches mentioned before (such as FaRM, GAM and L5), DFI only defines an interface for communication. Hence, different vendors can provide an efficient implementation of DFI for their network technology. As a result, this increases the compatibility and portability of data processing systems across different networking technologies including RDMA-capable networks (such as InfiniBand as well as RoCE).

In summary, this paper makes the following contributions:

- First, we present the design of DFI based on the general abstraction of flows that allow developers to declaratively specify the communication behavior of distributed systems by defining its topology (1:1, N:1, 1:N and N:M) as well as providing other properties for execution.
- Second, we provide a first implementation of DFI[1] for an InfiniBand-based networking stack and discuss how the high-level abstractions of DFI are being mapped to the low-level implementations using RDMA.
- Third, we provide an exhaustive evaluation of our DFI implementation and demonstrate that DFI does not only provide many benefits over MPI for data processing but also showcase that DFI can provide high performance for different data-centric applications.

Outline. The remainder of this paper is structured as follows: In Section 7.2, we first give an overview of two existing interfaces, RDMA verbs and MPI. Moreover, we analyze MPI as the direction taken by the high-performance community to provide a simple yet effective interface for high-speed networks and discuss the limitations of MPI for distributed data processing. Afterwards, in Section 7.3 we present an overview of DFI before we discuss details of the programming model in Section 7.4 as well as our implementation for InfiniBand in Section 7.5. Finally, we conclude with our evaluation in Section 7.6 and a summary in Section 7.7.

7.2 Existing Interfaces

In this section we aim to give an overview of existing interfaces namely the standard RDMA verbs interface native to the InfiniBand network stack and the Message Passing Interface (MPI), the de facto standard in the HPC community.

7.2.1 RDMA Verbs

The InfiniBand RDMA verb interface is a low-level interface providing low latency and high bandwidth communication. The interface exposes one-sided verbs (*write*, *read* & *atomics*) and two-sided verbs (*send* & *receive*) which refer to the involvement of end-points (i.e., one-sided verbs only involves the CPU of the sender). The high performance of RDMA is in general achieved by the asynchronous nature of RDMA, making it possible to pipeline computation and communication such that the CPU is not busy idling during

[1]`https://github.com/DataManagementLab/DFI-public`

network communication. To issue RDMA verbs (one- or two-sided), the application has to register a memory region in which the RNIC can directly access memory, leaving communication related memory-management a responsibility of the application. Moreover, due to the RDMA verb interface's very low abstraction level it provides also a huge design space. This requires, however, that applications need to carefully explore this design space and to optimally make use the available low-level options [35, 69, 156, 157].

7.2.2 Message Passing Interface

The Message Passing Interface (MPI) is widely used by the HPC community as a high-level abstraction for high-speed networks, and has through many years of development reached a mature and industrial-strength quality. One could now argue that MPI is already good enough for data-centric applications as well. In the following, we first aim to provide a better understanding of the programming and execution model of MPI and next we discuss the shortcomings of MPI for data-centric applications.

Programming Model. For programming distributed applications, MPI provides different primitives. These primitives can be categorized into point-to-point and collective communication.

- *Point-to-point communication:* The *MPI point-to-point* primitives provide communication operations to exchange data between a sender and a receiver. Point-to-point primitives, similar to native RDMA verbs, support two-sided communication with *send* and *receive*, as well as one-sided primitives with *put* and *get*. For the two-sided primitives, the *send* request has to be matched with a *receive* request, whereas the one-sided primitives transfer data without involving the remote side into the communication. While the point-to-point primitives offer high flexibly in how data is exchanged between two nodes, they still leave many low-level details to the application and thus only raise the level of abstraction minimally compared to native RDMA verbs (RDMA read/write or RDMA send/receive). For example, the remote memory management for the one-sided MPI primitives (*put* and *get*) is still up to the application to handle, which in a setup with many writers, involves considerable amounts of engineering efforts to coordinate the concurrent memory accesses as shown in [9].
- *Collective communication:* Different from the point-to-point primitives, the so-called *MPI collectives* targets many-to-many communication between multiple (sender and receiver) nodes and provide a higher-level abstraction to exchange data between nodes. Examples of MPI collectives are *scatter*, *gather*, *broadcast* or *reduce*

and *all to all* that transfer bulks of data (i.e., vectors of elements) between multiple nodes. Hence, collectives seem to be a perfect candidate for many data processing tasks such as data shuffling or even to implement replication protocols. However, while the collectives provide a convenient way to exchange data between multiple nodes, all these primitives use a bulk synchronous (i.e., blocking) communication model where all data needs to be available on the sender side before the collective is being executed. This limits the efficiency of MPI collectives for data processing [70] since it hinders overlapping of compute and communication.

Execution Model. MPI programs follow a process-centric execution model, where parallelism is achieved by running the multiple processes of the same program in parallel on multiple nodes. A new MPI program is started by running *mpirun* which launches the same program on all specified processes spread across the specified cluster nodes. Again, this process-centric parallelization model is not ideal for data processing systems as we discuss next.

7.2.3 Shortcomings of MPI

In the following, we give a brief overview of the main limitations of MPI for distributed data processing systems. Many of these shortcomings are evaluated further in Section 7.6.2.
Compute- and not Data-centric. MPI was designed towards supporting compute-intensive applications such as distributed simulations. However, the communication behavior in these types of distributed applications is very different from the needs of data-intensive applications. While distributed simulations exchange data in a bulk synchronous manner (i.e., after every iteration of a simulation), many data-centric applications are often dominated by data transfers (i.e., data shuffling).

Hence, for many data-centric applications it is important that applications can overlap computation and communication efficiently such that compute resources do not get idle [10]. This also holds for more latency sensitive operations such as distributed transactions. As shown in [137], overlapping does not only help to increase the overall throughput but it also reduces the end-to-end latency of distributed transactions.

Therefore, various types of data-centric applications would benefit from a pipelined (i.e., overlapping) communication model that provides a more loose coupling between senders and receivers. While a pipelined communication model is available for MPI's point-to-point primitives such as non-blocking send or one-sided put and get primitives, using these primitives in a non-blocking manner often results in more complex application code similar to using RDMA verbs directly [9]. In addition to point-to-point primitives, MPI collectives provide a higher-level of abstraction for communication between multiple nodes. However, as mentioned before, MPI collectives use a bulk-synchronous communication model. Extensions of MPI collectives to support pipelining [140] have unfortunately not made their way into today's MPI distributions. Hence collectives as they are available today do not only lack the the support for overlapping of computation and communication but also are thus sensitive to stragglers and skew which can both limit the performance in data processing systems significantly [21, 133].

Process-centric and not Thread-centric. As mentioned before, MPI has been designed for process-level parallelism. As such, the communication primitives of MPI (point-to-point and collectives) were designed for single-threaded usage; i.e., only one dedicated communication thread of an MPI process can call the communicating primitives. This, however, is very different from designs of modern data-centric systems for high-speed networks where multiple worker threads are often required to saturate the network [11, 147].

While recent papers [9] have shown that multi-process parallelism can be used in MPI to saturate the network, it comes with other downsides.For example, when using multi-process parallelism within a node, global data structures (e.g., an aggregation hash table) need to be accessed by different processes through shared memory.

Finally, in the recent years, many MPI distributions have added multi-threading support. However, as multi-threading support was only added as an afterthought, it lacks an efficient implementation in MPI as we show in our evaluation.

7.3 DFI Overview

In this section, we first highlight the central design goals of DFI before we discuss the flow-based programming model, as well as the high-level idea of the execution model behind flows.

7.3.1 Key Design Principles

The aim of DFI is to provide a high-level abstraction that provides efficient support for a broad set of data processing systems. In the following, we present the key design principles of DFI to ideally support the needs of these systems:

(1) Pipelining: Different from MPI, which targets compute-centric applications such as distributed simulations, many data-centric applications are often dominated by data

transfers (i.e., data shuffling). For this reason, it is shown to be crucial that computation and communication can be overlapped [10].

(2) Thread-centricity: Multi-threading is essential not only in achieving high degrees of parallelism in modern data-centric architectures but also to saturate the network as mentioned before. Hence, different from MPI, DFI should be designed from ground up to enable a thread-centric execution and communication model.

(3) Low-overhead synchronization: Another important aspect that goes along with thread-centricity is that DFI aims to provide low-overhead synchronization between sender and receiver threads as well as between sender threads that target the same receiver. By providing low-overhead synchronization, DFI thus should enable scalability to a high number of sender and receiver threads.

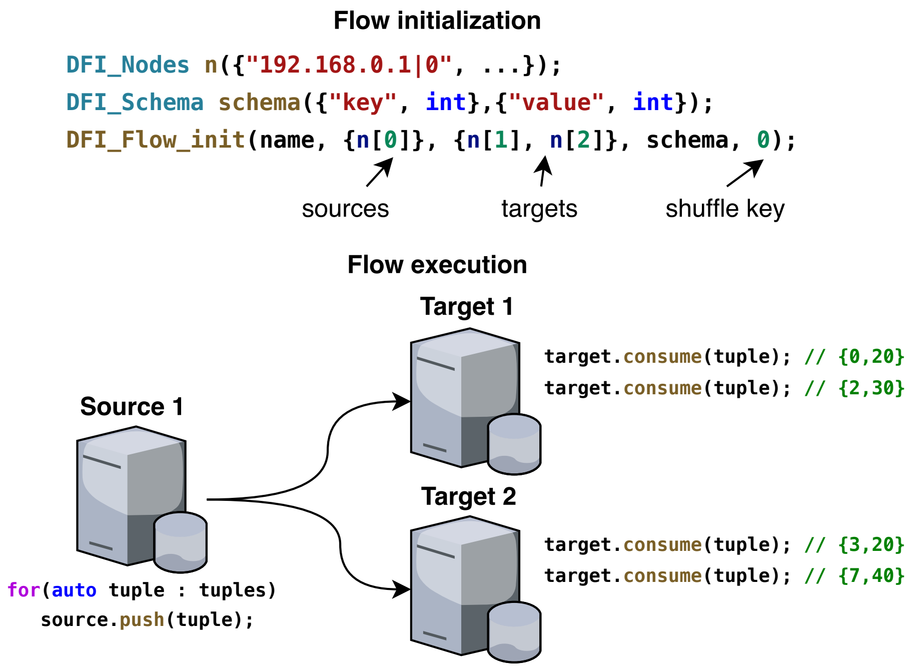
(4) Declarative optimization: A last important goal is that DFI exposes parameters as a handle for applications to declare what optimizations are desired. Examples of such optimizations are whether applications are bandwidth or latency sensitive, but also other guarantees such as global ordering of messages when data is send across flows (which is important, for example, for data replication protocols).

7.3.2 Flow-based Programming Model

At the center of the abstraction are DFI's flows. Flows encapsulate the movement of data between end-points in a distributed application, by exposing *sources* and *targets* as data entry and exit points on a per thread-level. This simple abstraction allows applications to compose potentially complex communication topologies, including both point-to-point, one-to-many, many-to-one and many-to-many communications between worker threads of multiple nodes. As we show later in this section, the flow abstraction is powerful enough to support a wide range of data processing use-cases such as distributed join algorithms, but also consensus protocols.

In the following, we provide an example of a concrete many-to-many flow type in DFI, which is one out of multiple other flow types as we discuss later. The most common many-to-many communication in data processing systems is arguably key-based shuffling of data across multiple sources and targets. An example of such a shuffle flow in DFI is illustrated in Figure 7.1.

As we see in the example, before a flow can be used it first has to be initialized by specifying a unique flow name identifier, location of source and target threads identified with the node address and a thread ID in DFI_Nodes, the schema of tuples that are transferred and on which key the tuples should be shuffled (see upper part of Figure 7.1).



```
DFI_Nodes n({"192.168.0.1|0", ...});
DFI_Schema schema({"key", int},{"value", int});
DFI_Flow_init(name, {n[0]}, {n[1], n[2]}, schema, 0);
```

```
target.consume(tuple); // {0,20}
target.consume(tuple); // {2,30}
```

```
target.consume(tuple); // {3,20}
target.consume(tuple); // {7,40}
```

```
for(auto tuple : tuples)
    source.push(tuple);
```

Figure 7.1: DFI's Programming and Execution Model. Example of flow initialization for setting up a shuffle based flow. The flow execution exemplifies the tuple-based push and consume primitives on DFI.

Note, it is also possible for applications to specify application-specific partition functions, but as default a simple key-based hash function is used to partition the tuples across receivers.

To make the flow available for other nodes, its metadata is published in a central registry upon initialization (e.g., a master node in a distributed system). For using a flow, sources and targets first need to retrieve the flow metadata from the central registry. The source nodes can then use a flow by pushing tuples into the flow and the target to consume tuples out of the flow by pulling from the flow (see lower part of Figure 7.1).

In addition to shuffle flows between N senders and M receivers, DFI provides many other flow types (i.e., a combiner and a replicate flow) and topologies (i.e., 1:1, N:1, 1:N and N:M) to support various data processing applications. More details about the full programming model of DFI will be explained in Section 7.4.

7.3.3 High-level Flow Execution

Key to the execution model of DFI's flows are the design principles discussed above. We achieve these design principles by implementing an execution model where each thread with a source or target has a private send/receive buffer that not only decouples sender from receiver threads but also uses a new memory layout for remote data transfer between sender/receiver threads with only minimal synchronization overhead as we discuss next.

In the following, we present the high level execution of flows by following the example of shuffling tuples shown in Figure 7.1. The push primitive on sources is asynchronous and returns immediately after the tuple to be transferred is copied into the internal send buffer. This non-blocking behavior allows applications to interleave the computation and communication, i.e., pipeline, and thus utilize both CPU and network resources. Moreover, internally the flow execution heavily uses the available one-sided RDMA primitives to reduce the CPU involvement of the targets, and thus decouples the sources and targets as much as possible. To enable one-sided network communication, as mentioned before, a receive buffer must be in place in which the tuples of one or multiple sending threads are written to. Details about the buffer design and their low-overhead synchronization model are discussed further in Section 7.5.

Once a tuple has been pushed into the flow, a routing decision will be made by the flow based on the provided shuffling key. Depending on the chosen optimization goal (bandwidth or latency), the execution of the flow will transport tuples across the network. For bandwidth optimization, flows batch tuples together destined for the same target in order to achieve a better bandwidth utilization through larger messages. On the other hand if a latency optimization is chosen, the flow execution will prioritize transferring the tuple as soon as possible. The details of these optimizations are discussed further in Section 7.5 as well.

7.4 Programming Model

In the following, we present a more detailed view on the programming model of DFI and its main abstractions by detailing the opportunities for setting up various communication flow types. In addition, the programming model will be demonstrated through a set of concrete use cases.

7.4.1 DFI Tuples

To allow processing of application-specific tuples between different end-points (i.e., threads) of DFI flows, DFI receives the tuple types through the passed schema on flow initialization. The schema can be constructed of various data types, that each mirrors the size of C++ types, specifically the LP64 data model (default data model in most

Flow type	*Communication topology*	*Flow options*
Shuffle flow	1:1, N:1, 1:N, N:M	Bandwidth/latency
Replicate flow	1:N, N:M	Bandwidth/latency + ordering guarantees
Combiner flow	N:1	Bandwidth/latency + various aggregations

Table 7.1: DFI flow types for a wide range of data-centric applications. Communication topologies and flow options further allow applications to adjust the behavior of flows based on application requirements.

Unix-based systems). The types, however, can be extended by the application to meet the need for other user-defined types.

DFI's type system enables efficient data processing: (1) Avoiding any type interpretation overhead is key to high-speed networks since every additional overhead can significantly reduce bandwidth or increase latency of the overall distributed algorithm [37, 112]. Tuple types are parameters of flows that are defined at flow initialization; i.e., no type interpretation happens at flow execution. Instead, efficient offset computation can be used to access attributes of a tuple (e.g., to make routing decisions). (2) The type system of DFI also allows applications to push down the processing to devices in the network, such that the interface is extensible towards leveraging the future generations of SmartNICs and programmable switches. For example, data aggregation of a DFI combiner flow (which is another DFI flow type) could be pushed into InfiniBand switches as we discuss below.

7.4.2 DFI Flows

So far we have only presented a concrete example of constructing a flow for shuffling tuples between a set of source and target threads. However, DFI defines flows with different characteristics to support the wide demands of data processing systems. Table 7.1 shows the three flow types in DFI, the communication topologies supported by the corresponding flows, as well as their declarative flow options.

The flow abstraction also offers easy adaptability of application algorithms, since different types of flows can be trivially exchanged to offer different behaviors. For instance, to change a symmetric re-partition join algorithm into a fragment-and-replicate join, instead of using a shuffle flow that routes tuples based on the join key, use a replicate flow to replicate the inner table. Performing such algorithmic changes on typical solutions leveraging the RDMA verb interface would infer a significant rewrite of the communication relevant parts of the solution.

In the following, we discuss the different flow types and their potential use in data processing systems.

7.4.2.1 Shuffle Flow:

The shuffle flow is a central abstraction of DFI, where various different communication patterns and routing options can be specified. The communication pattern is indirectly defined by declaring the participating sources and targets in the flow initialization, and can therefore follow 1:1, N:1, 1:N and N:M communication patterns between sending and receiving threads.

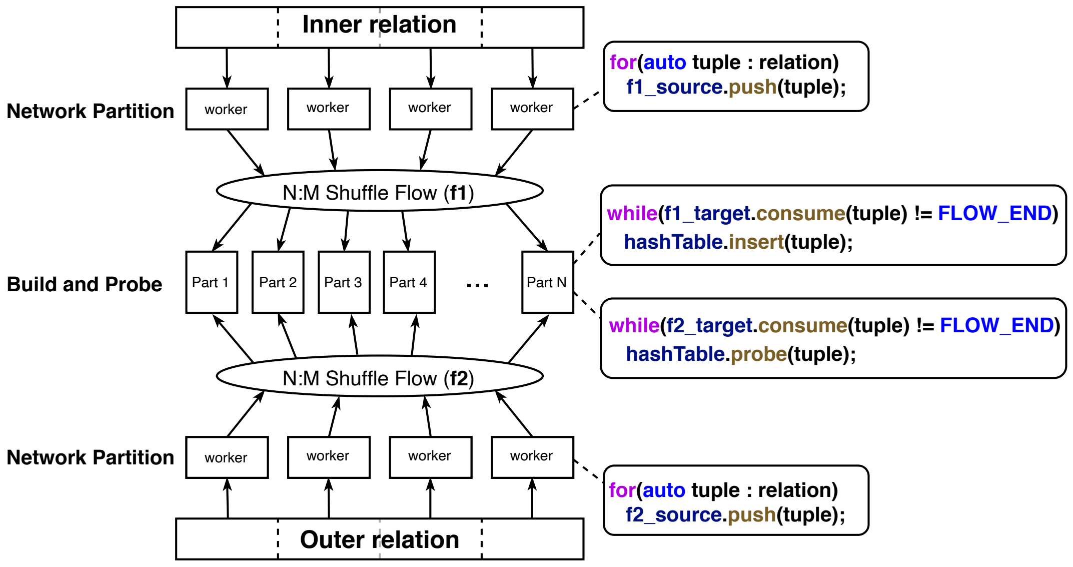
The routing of tuples from sources to targets can be defined in three ways in a shuffle flow: (1) The application specifies the shuffle key and let DFI handle the routing. (2) A routing function can be supplied for more control, e.g., to realize different partition functions such as range-partitioning or radix hash partitioning. (3) Lastly, it is also possible to directly specify the node identifier of a target thread on each push into the flow.

7.4.2.2 Replicate Flow:

Another flow type that DFI provides is a so called replicate flow, which targets data processing tasks involving data duplication, such as replicated state machines, fragment-and-replicate join operators or data duplication for stream processing.

The performance of a naive replication of tuples which uses multiple RDMA operations (i.e., one for each target), will quickly become limited by the outgoing link-speed of the source node; e.g., a replicate flow with 1 source and 8 targets, will have to divide the available network bandwidth at the source, if messages are replicated to all 8 targets on the source node. In DFI, we instead make use of RDMA multicast such that when enabled, messages are replicated in the network as to prevent the outgoing link of the source(s) from becoming a bottleneck.

For some applications using replication, ordering of messages plays an important role. An example of this is state machine replication, where the correctness depends on all replicas processing the incoming operations in the same order. Since many networks (including InfiniBand) do not provide this guarantee if multiple receivers are involved [82] (even not for simple networks with only one switch), replicate flows can be initialized to provide global ordering guarantees, such that all targets consume tuples out of the flow in the same order. Details on how global ordering is implemented for replicate flows in DFI are explained in Section 7.5.



```
for(auto tuple : relation)
  f1_source.push(tuple);
```

```
while(f1_target.consume(tuple) != FLOW_END)
  hashTable.insert(tuple);
```

```
while(f2_target.consume(tuple) != FLOW_END)
  hashTable.probe(tuple);
```

```
for(auto tuple : relation)
  f2_source.push(tuple);
```

Figure 7.2: Distributed Radix Hash Join with DFI flows. Two shuffle flows are used to partition tuples across network, one for each relation.

7.4.2.3 Combiner Flow:

The third flow type supported by DFI is the combiner flow. The focus of the combiner flow is many-to-one communication patterns which is typically used in aggregation scenarios, such as a SQL aggregation or a parameter server [83] for distributed machine learning. The combiner flow supports various different aggregations (e.g., SUM, COUNT, MIN, MAX) to be performed on the tuples.

Again while a naive implementation would implement the reduction at the target node, the network can be used to accelerate the reduction. For example, InfiniBand offers the SHARP protocol [41], that enables in-network aggregations for high-speed InfiniBand networks and thus could help to mitigate when the in-bound network of the receiver becomes a bottleneck.

7.4.3 Use Cases

In the following we present two distributed data processing use cases and how they are realized through DFI: First, we discuss distributed joins for OLAP where the aim is to reduce the runtime by making efficient use of the available network bandwidth. Second, we present a distributed consensus use case where the performance criteria is low latency and high message throughput.

7.4.3.1 Distributed Radix Join:

The distributed radix hash join is a popular join operator due to its dominating performance [9, 10]. The idea behind the radix hash join is to partition the input relations into such small partitions that the resulting hash tables fit into the CPU caches to reduce cache-misses.

In its original form the distributed radix join has a high level of complexity since multiple sender and receiver threads need to coordinate. For example, in [9, 10], histograms of buckets are pre-computed in a first pass on each input table to allocate private memory buffers for each thread on the receiver node and then use coordination-free one-sided communication in a second pass to shuffle the data of each input table.

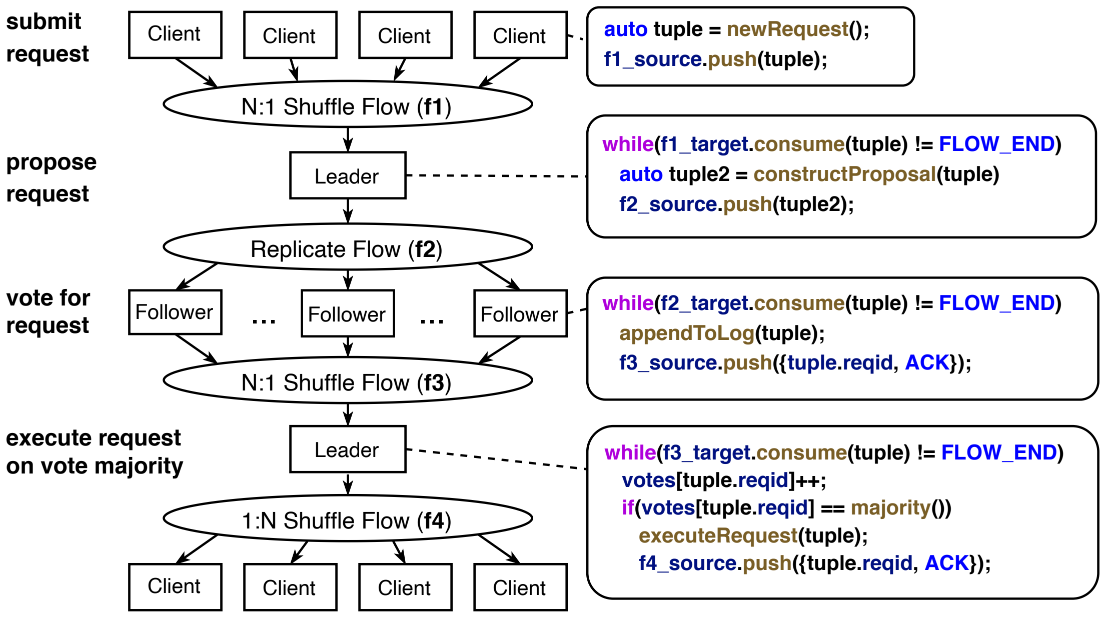
We argue that with DFI, the design of a distributed radix join is simpler while the performance is on par (and sometimes even better) with the latest distributed radix join implementations (as will be shown in Section 7.6.3). To realize the join with DFI, two bandwidth optimized shuffle flows are used as shown in Figure 7.2, one for shuffling each relation. Figure 7.2 also shows the pseudo-code how tuples can be pushed into the flows during network partitioning, and consumed at the target (i.e., receiver node) out of the flows for the relations to either build the hash table (for the inner relation) or probe the hash table (for the outer relation).

The shuffle flows for the join are initialized with one source per sender thread and one target per output partition. That way the flow can used for achieving the desired partition fan-out. The routing of tuples to the partition-specific *targets* is done on a per thread level by passing a radix hash function to DFI as the routing function. This also leads to a noticeable reduction of complexity of the DFI join compared to the original RDMA-based distributed radix join since the histogram computation can be completely omitted. Moreover, the memory management of local and remote buffers is handled in DFI.

7.4.3.2 Distributed Consensus:

Consensus in a distributed system describes the agreement of multiple (often asynchronous) participants on a single value, or a sequence of values, while tolerating the presence of faulty participants. It is a fundamental primitive in distributed computing which is needed, for example, for the reliable implementation of replicated state machines, leader election, or system reconfiguration.

Classical consensus protocols [74, 105] are centered around a centralized coordinator, called leader. The leader orders concurrently arriving requests of participants (i.e., clients)



```
auto tuple = newRequest();
f1_source.push(tuple);
```

```
while(f1_target.consume(tuple) != FLOW_END)
  auto tuple2 = constructProposal(tuple)
  f2_source.push(tuple2);
```

```
while(f2_target.consume(tuple) != FLOW_END)
  appendToLog(tuple);
  f3_source.push({tuple.reqid, ACK});
```

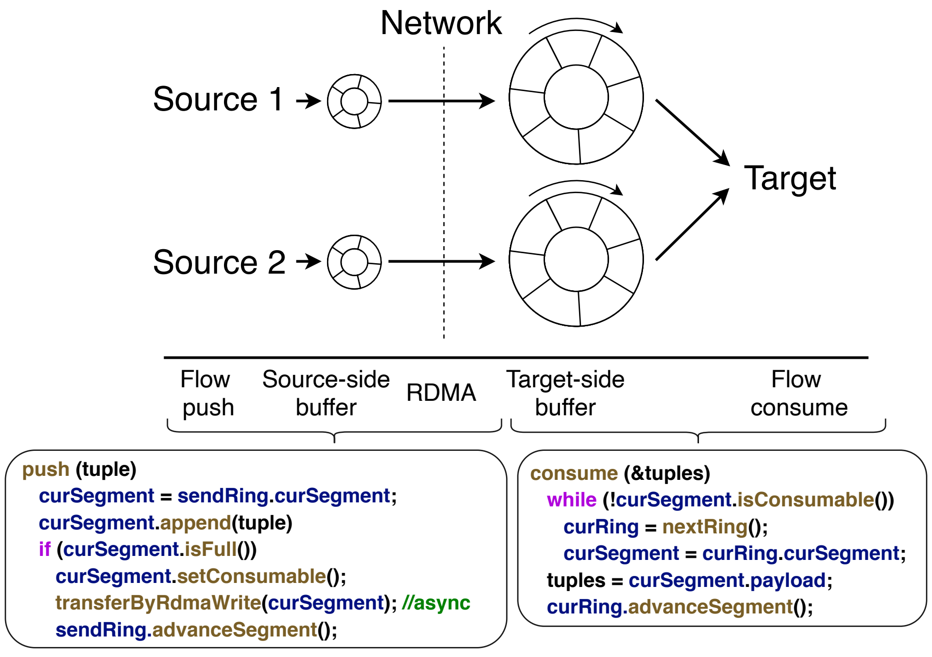
```
while(f3_target.consume(tuple) != FLOW_END)
  votes[tuple.reqid]++;
  if(votes[tuple.reqid] == majority())
    executeRequest(tuple);
    f4_source.push({tuple.reqid, ACK});
```

Figure 7.3: Leader-based consensus with DFI flows. Four flows are used to realize consensus between replicas.

and forwards them to a set of so called followers. The followers vote for requests that they receive from the leader. Once the leader has received a majority of votes (itself included), the leader can notify the corresponding client that its request was agreed-upon. The high-level message flow of a leader-based consensus implementation using DFI can be modeled directly with the flows provided by DFI and is depicted in Figure 7.3. Figure 7.3 additionally shows pseudo-code of how these flows are used for the communication which we explain in the following.

Clients initially send their vote with an N:1 shuffle flow to the leader. The replicate flow is ideal to handle the communication from the leader to its followers, as all followers receive identical messages. The use of the RDMA multicast verbs built into DFI alleviates load placed on the leader compared to the naive replication of messages. This is an interesting optimization, as the leader is typically a major bottleneck in consensus-based systems. Once followers received the request and voted for a result, they send the outcome back to the leader, again using a shuffle flow. In a last step the leader distributes the consensus-outcome to the client using the client IDs as the shuffle key.

An interesting optimization that DFI provides is to use the optimization option for global ordered multicast (also referred to as ordered unreliable multicast - OUM). In particular, Li et al. [82] propose a single round-trip consensus protocol based on OUM.



```
push (tuple)
  curSegment = sendRing.curSegment;
  curSegment.append(tuple)
  if (curSegment.isFull())
    curSegment.setConsumable();
    transferByRdmaWrite(curSegment); //async
    sendRing.advanceSegment();
```

```
consume (&tuples)
  while (!curSegment.isConsumable())
    curRing = nextRing();
    curSegment = curRing.curSegment;
  tuples = curSegment.payload;
  curRing.advanceSegment();
```

Figure 7.4: DFI flow implementation using ring buffers. In DFI flows, each source allocates a private target-side ring buffer to minimize coordination overhead.

While this work focuses on Ethernet-based systems, to our knowledge, DFI is the first system that can provide these semantics in the context of InfiniBand.

As we show in Section 7.6.3.2, using the ordered multicast significantly improves both throughput and latency compared to conventional consensus protocol designs using native RDMA that follow more classical consensus designs.

7.5 Flow Implementation

In this section, we discuss our implementation of DFI for a recent RDMA-capable InfiniBand EDR hardware stack to showcase the design choices of how to enable the key design principles discussed in Section 7.3. In future, we envision that different vendors can provide an efficient implementation of DFI for their network technology.

7.5.1 Flow Execution

The key design principles listed in Section 7.3 impose challenges for how the data transfer between the sources and targets is realized which are pivotal for distributed data processing. In the following, we give an overview of the flow execution (Section 7.5.1) and the buffer design (Section 7.5.2) for bandwidth optimized shuffle flows. Optimizations for latency-optimized flows and other flow types will be discussed at the end of this section.

On a high-level, to achieve the design goals listed in Section 7.3 for shuffle flows, DFI uses a private send/receive buffer for each pair of source and target threads as illustrated in Figure 7.4. The design of source- and target-side buffers follows a ring-based design where each ring is composed of a configurable number of segments and is allocated as one consecutive region in memory. The segment itself can be sized to contain a single tuple up to a batch of tuples. Therefore, the segment size is a tuning parameter that allows DFI to either optimize for bandwidth or latency independent of the tuple sizes used by the application.

One key question is how such a segmented ring design enables pipelining of tuples with low-overhead synchronization. In order to achieve pipelined data transfer between buffers (i.e., a decoupling of senders and receivers), one-sided RDMA writes are used to copy data asynchronously from sources to targets. This asynchronous data transfer using RDMA writes is implemented by the `transferByRdmaWrite` call in Figure 7.4. This method also implements the synchronization with the target buffer to not overwrite any segments that has not been consumed yet. The synchronization is based on the metadata of each segment as we discuss next in Section 7.5.2. For latency-optimized flows (see Section 7.5.3), we instead use a credit-based approach to further reduce the overhead.

In setups with a very high number of sources and targets, a design with private buffers for each source / target combination can by first sight lead to high memory consumption. In DFI, however, applications can effectively reduce the memory consumption by reducing the number of segments per ring, since only a few segments are needed to achieve good pipelining and source / target decoupling. As we show in our evaluation, this can efficiently reduce the memory overhead while only affecting performance minimally.

Other approaches also employ a circular buffer for communication over RDMA, e.g., such as FaRM [28, 29]. While our buffer design shares some similarities with the buffer design of FaRM (e.g., using one-sided writes for data transfer), there are noticeable differences: (1) The aim of the FaRM design is only latency sensitive message-passing and hence does not provide a bandwidth optimized communication primitive. (2) The buffer design of FaRM targets only shuffle flows but no other flow types such as replication flows or combiner flows, which require additional optimizations as we discuss in Section 7.5.4.

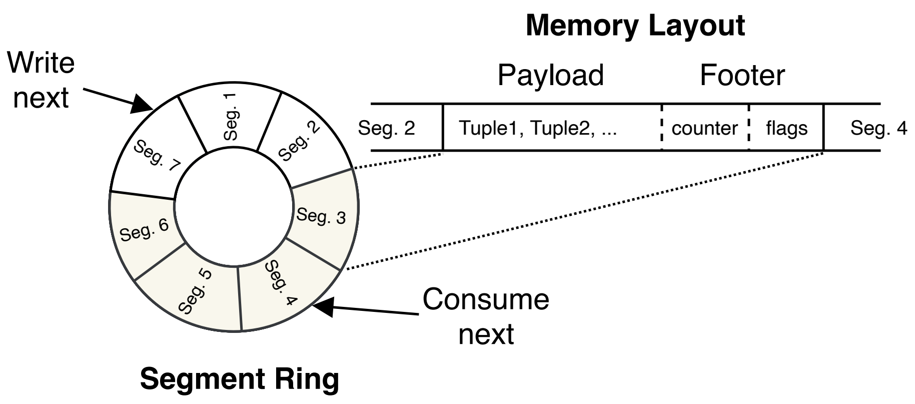


Figure 7.5: Target-side buffer structure. The segment ring data structure is a densely allocated memory region split up into segments. Segments are appended with small footers to handle coordination and fill grade of each segment.

7.5.2 Buffer Design

In the following we present further details of the buffer design for the bandwidth optimized setting. We first explain the design of the outgoing buffer on the source side before we then discuss the design of target-side buffers.

Source-side Buffer. The main difference is that source-side buffers use much fewer segments than target-side buffers to reduce the memory overhead of buffers. Smaller send buffers do not violate our goal to decouple sources and targets since data is sent directly once it is available. However, since data out of source buffers is transferred asynchronously we need more than one segment. For this reason, we need to ensure that an RDMA write of a segment has been carried out before the segment can be reused to avoid data loss. For achieving this, the source-side buffers use signaled RDMA writes, a technique to check if the asynchronous transfer is completed. However, in order to reduce the number of these checks, as they are quite expensive, the source only issues a signaled write once it wraps around the buffer.

Target-side Buffer. Target-side buffers use a slightly different design since sources and targets need to synchronize; i.e., targets need to decide which segments are ready to be read while sources must decide whether a remote segment was consumed already and thus can be reused by the source. For the synchronization each segment defines a footer which holds metadata about the segment as shown in Figure 7.5. The flags in the footer indicate whether a segment is *writable* or *consumable*. In a *writeable* state, a segment is free to be written by the source and *consumable* state indicates that the target can consume the tuples stored in the segment.

Unfortunately, an RDMA write of one segment is not guaranteed to be persisted atomically into remote memory since the segment might be split into multiple DMAs by the remote NIC. In order to avoid a checksum per segment, we make use of the fact that DMAs of the remote NIC are guaranteed to be written in an increasing memory order [28, 86]. Therefore, DFI places the metadata which indicates the state whether the segment is consumable or not *after* the payload of the segment. This ensures that when the target has detected a change of segment state, the payload of the segment is completely written.

From a target perspective, the footer is read when the target thread calls the consume function, which returns a pointer to the payload if the state is consumable and sets the state to writable on subsequent consume calls. Thereby allowing the application to process the returned tuples directly without memory copy. To write new segments to the target-side buffer the source first needs to verify if the current target segment is free and ready to reuse. Therefore, the source reads the footer of the remote target segment with RDMA reads to check if the state is writeable.

To efficiently read the remote segment states we pipeline the write of the current segment with the read of the footer of the next segment. In other words, while we transfer a source-side segment to the target-segment n we immediately read the footer of the target segment $n + 1$. Therefore, upon the next push call on the source, if the remote segment is detected to be writeable, the RDMA write can be executed directly without waiting. The RDMA write of the source includes the update of the footer for the target (i.e., setting its state back to consumable). If the remote segment is detected to be not writeable, the source periodically polls the segment footer that should be written next with a small random backoff, to avoid overloading the network with read requests.

Optimizations. For the buffer design, we use various optimizations for efficient RDMA. For instance, our design increases the chance to exploit DDIO which allows DMA data to be directly written to CPU caches. Additionally, we use common RDMA optimizations like inlining small messages and selective signaling. Moreover, in both buffers (source- and target-side), we additionally enable two performance-relevant settings in the InfiniBand stack: (1) We disable expensive spin locks in the RDMA library when using private buffers. (2) We use huge-pages for RDMA to avoid the high costs of TLB misses in the RDMA NIC.

7.5.3 Latency Optimization

The previously presented flow implementation is centered around maximizing bandwidth utilization. However, for some classes of distributed data processing systems (e.g., for OLTP or consensus) achieving low latency for data transfer is crucial. Hence DFI provides an optimization option for latency. This requires several changes in the buffer design and the overall execution flow.

A naive way to support low latency would be to simply reduce the segment size of buffers to the size of an individual tuple and rely on the same synchronization protocol between sources and targets. However, this implies that sources would need to check the footer of the next segment before every RDMA write to make sure that the segment is writable. In contrast to the bandwidth optimized version in which the cost of the additional read is amortized by the batch of tuples an additional read for each tuple transferred incurs high overhead. Hence, we use a different design to support low-latency data transfer with low overhead.

While the footer is still used by the target to decide if a segment is consumable, we use a credit system to decide how many segments a source can write without any synchronization. Therefore, a credit counter on the target side is used which is initialized to the number of segments in the ring, to reflect the number of tuples a source can write without overwriting tuples that haven't been consumed by the target yet. The credit counter is incremented by the target each time a tuple is consumed. The source thread holds a copy of the remote credit counter which is decremented on every RDMA write of a segment. Moreover, the remote credit is read once the local credit counter reaches a certain threshold.

7.5.4 Other Flow Types

The replicate and combiner flows employ similar buffer structures as the shuffle flow but introduce important optimizations.

Replicate Flow (no ordering). The replicate flow sends a tuple to a group of targets. In order to avoid that the outgoing network link of the source becomes a bottleneck, we exploit RDMA multicast which replicates tuples in the switch. However, RDMA multicast relies on two-sided communication over unreliable transport, which leads to changes in the buffer design and the control flow. Instead of detecting incoming messages on the target side by polling directly in main-memory, for two-sided RDMA primitives the target has to poll a completion queue. To avoid high overhead of coordinating for each RDMA send, we implemented a credit score for sources (similar as the one we used

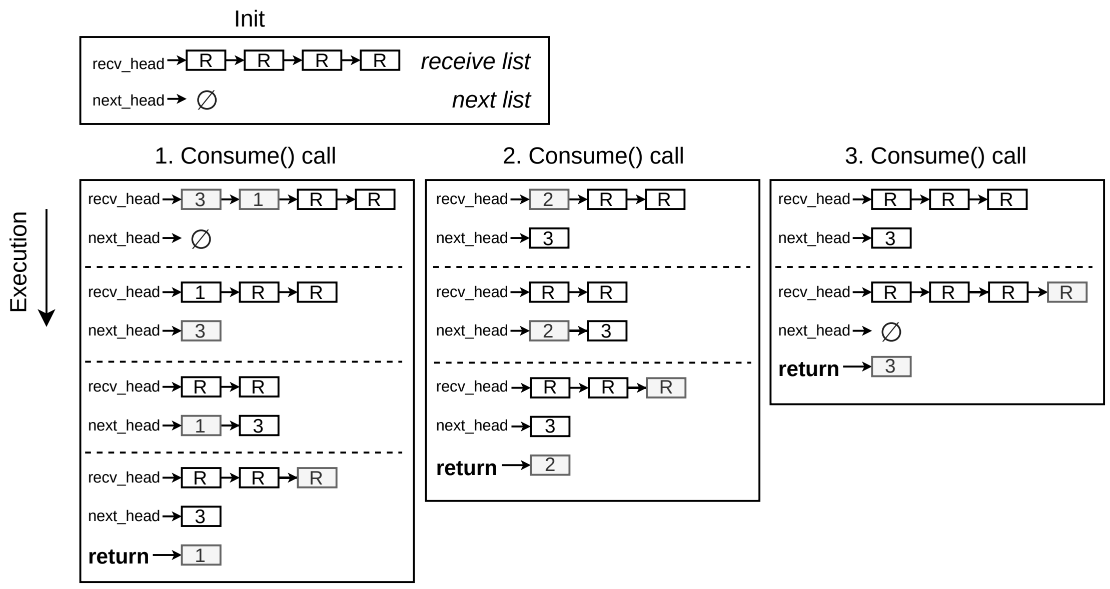


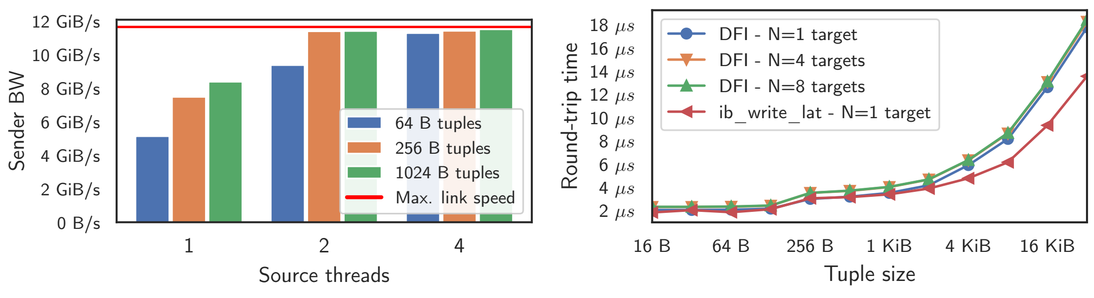
Figure 7.6: Ordering example for replicate flows: Receive requests (R) are initially posted and stored in the *receive list*. Subsequent consume calls detect incoming segments and insert segments into the *next list* in order. Messages are returned in order from *next list*.

for latency-sensitive flows) to know for *all* targets of a replicate flow, how many messages can be sent without coordination. For this, we initially pre-populate the receive queues on the target sides with as many receive requests as given by the credit score and as soon as a segment has been written into the target-side buffer a new receive request will be added to the receive queue once the *consume* call on the target-side buffer has returned the payload.

For coordinating credits at the source side we employ a back-flow from targets to source. This back-flow is used by targets to inform sources how many messages have been received which allows the sender to increase its credit score accordingly. Moreover, we add a sequence number to each segment such that targets can report missing segments to sources together with the credit back-flow. Lost segments are requested if a configurable timeout is reached. One important issue is that segments might arrive late (which were reported as lost by the target). In DFI this is handled by the target which filters out duplicate segments.

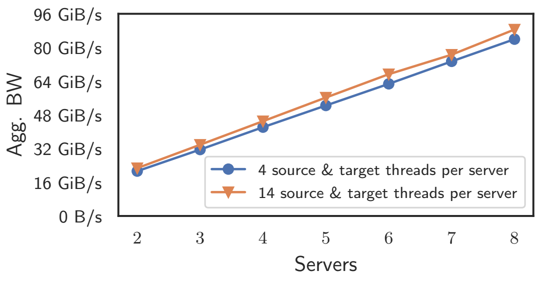
Replicate Flows (globally-ordered).

DFI can also provide global ordering guarantees on replicate flows. Global ordering is a common primitive for distributed systems which, however, often needs to be implemented



(a) Sender bandwidth (1:8)

(b) Median latency (1:N)



(c) Aggr. sender bandwidth (N:N)

Figure 7.7: Shuffle flow performance. DFI achieves max. bandwidth and low latency for various scenarios.

on the application layer. For instance, distributed consensus requires a global ordering as described in Section 7.4.3.2. DFI guarantees global ordering by implementing a so-called tuple sequencer, in which sources append sequence numbers to segments using an RDMA fetch-and-add on a global counter. With the advent of programmable switches, a tuple sequencer can instead be implemented in the network as shown in [82] to avoid the additional round-trip for the RDMA fetch-and-add. However, as we see in our experiments in Section 7.6.3, already the naive solution with a global counter can provide benefits for consensus over solutions which rely on flows without ordering guarantees.

While a tuple sequencer adds global sequence numbers to segments, they can still arrive out-of-order at the different targets. On the target-side we thus have to ensure that segments arrive in the same order by reordering the incoming potential out-of-order segments. Figure 7.6 exemplifies how reordering is implemented on the target side. For reordering, two linked lists are used: a *receive list* for storing incoming segments in the arrival order and a *next list* for ensuring ordering.

In the example in Figure 7.6, on the first consume call, segments with sequence number 3 and 1 have been received. The head of the *receive list* (i.e., 3) is "moved" to the *next* *list* which involves no data copy but only pointer updates and the segment with number 1 is returned. For the second consume call, in the example, the segment with number 2 has arrived and is directly returned while the segment 3 is kept in the *next list* which is then returned for the last consume call. Losses in this protocol are detected (as before) through gaps in sequence numbers (in case a configurable timeout is reached). Optionally, we only notify the application on a consume call of gaps and its left up to the application to handle re-transmission (which is a feature we use for our consensus implementation).

Combiner Flows. A last flow type supported in DFI is the combiner flow. The flow directly follows the design of a shuffle flow (using a N:1 topology) but adds functionality to aggregate tuples in the target buffer using an aggregate function/ group-by specification as explained in Section 7.4. An interesting optimization is to use in-network-processing capabilities such as the SHARP protocol that enables in-network-aggregation in a switch to avoid incast congestion on the in-going link to the target of a combiner flow. However, implementing this is an interesting avenue of future work.

7.6 Experimental Evaluation

We evaluated DFI on three different levels. First we look at the efficiency of DFI in terms of how well the high-level interface utilizes the network compared to low-level RDMA verbs. Next, we provide a detailed comparison of DFI and MPI and argue that MPI is the wrong abstraction for data processing systems. Lastly, we evaluate DFI for two typical use cases in data processing systems and compare the implementations to existing state-of-the-art solutions.

In all experiments we use the notation (N:M) to indicate the number of servers involved in a flow topology. The number of threads per server is reported separately per experiment.

Evaluation Environment. All experiments were conducted on an 8 node cluster where 6 of the nodes are equipped with two Intel(R) Xeon(R) Gold 5120 CPUs (14 cores) and 512 GB main-memory, and 2 nodes equipped with two Intel(R) Xeon(R) Gold 5220 CPUs (18 cores). Hyper-threading is disabled for all nodes. Each node is equipped with two Mellanox ConnectX-5 MT27800 NICs (InfiniBand EDR 4x NICs, 100 Gbps), connected to one SB7890 InfiniBand switch. The operating system is Ubuntu 18.04.1 LTS, with Linux 4.15.0-47 kernel on all nodes. DFI is implemented with C++17 and compiled with gcc-7.3.0.

7.6.1 Experiment 1: Efficiency of DFI

The first experiment shows the efficiency of DFI compared to low-level RDMA verbs.

7.6.1.1 Shuffle Flows:

In the following, shuffle flows are evaluated with bandwidth and latency optimization and lastly, a scale-out experiment is presented.

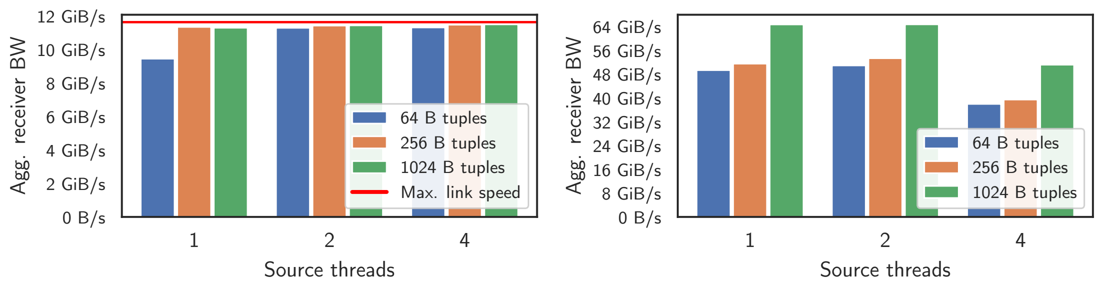
Bandwidth-Optimized. Our first experiment evaluates performance for the shuffle flow from 1 server to 8 servers with varying tuple sizes. Further, we vary the number of sources (threads) pushing tuples into the flow. The batch size for the bandwidth optimized version in our experiments is 8 KiB. We choose a batch size of 8 KiB as this offers a good tradeoff between network bandwidth and time until the batch is filled.

Figure 7.7a reports results for the bandwidth-optimized flow. As we see, in most settings we achieve the full network bandwidth. Only, the single-threaded scenario shows some overhead since batches must first be filled on the source side with individual tuples before they can be transferred to the target. This overhead can, however, be amortized by using more threads per server as shown in Figure 7.7a. Due to the efficient multi-threading support of DFI, we see that from two source threads on, the bandwidth is limited by the speed of the outgoing link (100 Gbps / 11,64 GiB/s - red line) for tuple sizes larger than 128 B. Moreover, when using 4 threads the maximal bandwidth is achieved independent of tuple sizes.

Latency-Optimized. We additionally evaluated the shuffle flow that implements latency optimizations. For measuring latency, two shuffle flows are used to implement a request and response pattern to measure the round-trip time between two nodes. To show that DFI's buffer design only adds minimal latency overhead, we compare the latency of DFI to *ib_write_lat*[2] which is a standard tool for performance testing that uses low-level verbs to implement a round-trip between a sender and a receiver node. For DFI, we additionally used a varying number of receiving servers (1, 4, and 8) to observe the effect on latency when shuffling to various destinations.

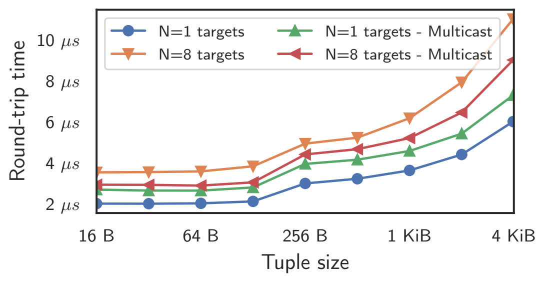
As we see in Figure 7.7b, the median latency of DFI for one full round-trip only adds minimal overhead when compared to *ib_write_lat* which is due to buffer. Moreover, keep in mind that DFI provides a high-level abstraction and thus not only reduces application complexity but also provides several optimizations to applications. This includes an efficient overlapping of compute and communication as well as many other optimizations such as efficient replication and ordering guarantees. As we show in Section 7.6.3, this

[2] `https://github.com/linux-rdma/perftest`.



(a) Aggregated receiver bandwidth (naive one-sided, 1:8)

(b) Aggregated receiver bandwidth (multicast, 1:8)



(c) Median latency (all targets, 1:N)

Figure 7.8: Replicate flow performance. DFI achieves max. bandwidth and low latency for various scenarios.

enables DFI to provide superior performance in different use cases when compared to existing approaches that are using other interfaces (low-level RDMA verbs or MPI).

Moreover, the advantage of DFI compared to plain RDMA is the encapsulated memory management, which allows applications to use RDMA transparently without hand-tuned memory management while still achieving optimal performance. The experiment shows that this abstraction hardly incurs any overhead compared to *ib_write_lat*. For multiple targets the latency of DFI is only slightly higher due to the internal routing in the shuffle flow. Multiple targets are not supported by *ib_write_lat* though (i.e., *ib_write_lat* uses only one target in this experiment).

Scale-out. Since data processing systems often need to scale out to many nodes, we conducted a scale-out experiment for the shuffle flow, increasing the number of source and target servers. Moreover, we use 14 sources and targets on all nodes which in total gives 12544 unique source/target connections for the maximal number of nodes used. As shown in Figure 7.7c, DFI scales linearly with the number of nodes (as indicated by the x-axis), effectively increasing the aggregated bandwidth with the link-speed of each added node.

7.6.1.2 Replicate Flows:

Next we benchmarked the replicate flow in terms of achievable bandwidth, with and without multicast and finally the latency behavior.

Bandwidth and Multicast Optimization. We tested the replicate flow bandwidth for two optimization settings, naive one-sided replication and multicast. The evaluation was conducted by replicating data from 1 node to 8 nodes. In Figure 7.8a, the reported bandwidth already achieves the practical network limit of the sender with 1 thread and tuples bigger than 64 B. In comparison to the shuffle flow bandwidth in Figure 7.7a, the replicate performance reaches max. bandwidth earlier due to network messages being replicated with one-sided writes that are issued in parallel by the NIC, reducing the per-tuple overhead.

As we see in Figure 7.8a, the naive replication is limited by the network speed of the sender. However, the replicate flow also provides a multicast optimization that replicates messages in the switch. With the multicast optimization (Figure 7.8b) the bandwidth goes beyond the 100 Gbps (11,64 GiB/s) limit of the outgoing sender link and reaches up to 64 GiB/s. Using more source-threads on the sender node, however, does not yield better performance as the NIC inhibits bad multi-thread scalability within same multicast group.

Latency Optimization. For evaluating the latency for the replicate flow, we conducted an experiment where a source replicates a request to 8 targets and measures the time for it to get replies from all. The reported latency for naive and multicast replication is shown in Figure 7.8c. The naive (one-sided) replication achieves the lowest latency with only 1 target, but increases with more targets. For multicast, though, the increase in latency from 1 to 8 targets is much smaller and outperforms the naive implementation.

7.6.1.3 Combiner Flows:

The last flow evaluated in this experiment is the combiner flow. Figure 7.9 reports the aggregated sender bandwidth for a combiner flow with a sum aggregation. As seen for 2 and 4 threads, the bandwidth becomes limited by the in-going link to the target node. As an avenue of future work, the advent of in-network processing such as SHARP [41] could be used to further speed up the aggregation beyond the limits of the in-going link.

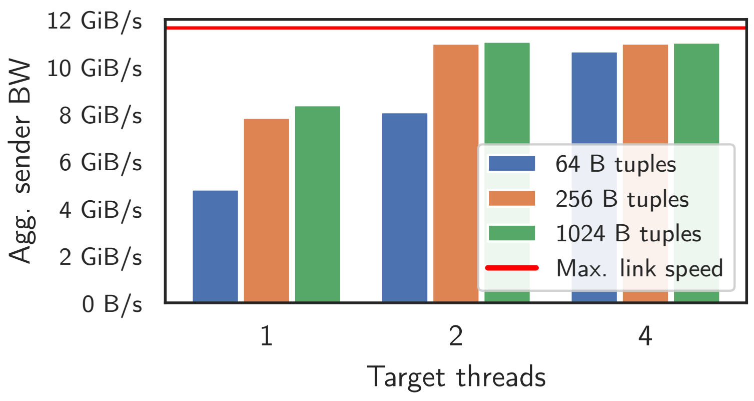


Figure 7.9: Combiner flow with sum aggregation (8:1). Aggregated sender bandwidth for various scenarios.

7.6.1.4 Memory Consumption:

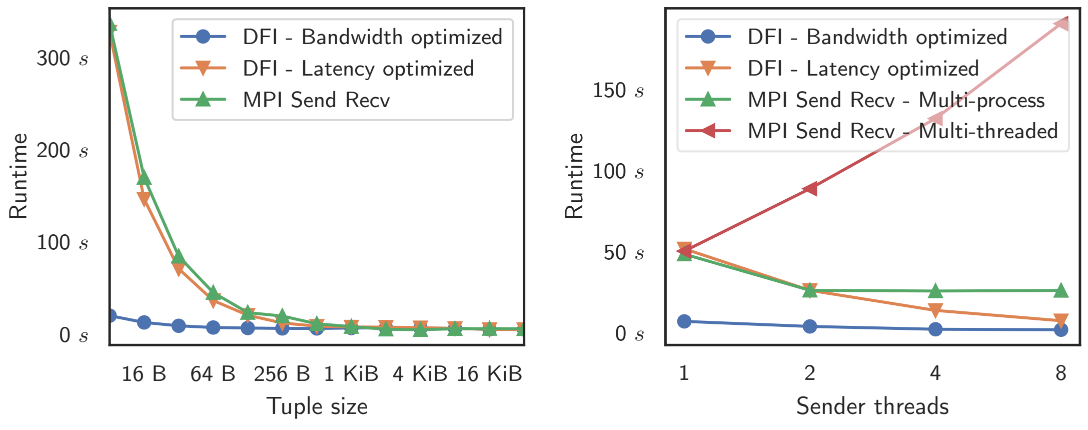
In order to provide insight to the degree of memory consumed through DFI, we observed the memory allocated in the scale-out experiment of the shuffle flow in Figure 7.7c. This setup is the most memory consuming one since each pair of source/target threads uses a private send/receive buffer.

For the smallest setup with only 4 source and 4 target threads per node in a cluster with two nodes in total, DFI consumes just 16 MiB per node[3]. Moreover, when increasing the setup to 8 nodes where each node has again 4 source and 4 target threads, the memory consumption grows only to 64 MiB. For the largest setup in Figure 7.7c (14 source & target threads and 8 servers in total) DFI consumes 785.5 MiB in total on each node.

However, as discussed in our experiments before, 4 source/target threads per node are sufficient at the moment to saturate the high bandwidth provided by the InfiniBand network in our setup. Moreover, the size of buffers in DFI are configurable. Hence, even smaller memory footprints can be achieved. As an example reducing the number of segments to 50% (i.e., 16 per buffer) the performance on 8 nodes just decreases by 2.7%, and further reducing the size to 25% (i.e., 8 per buffer) decreases performance by 8%.

Key Insights (Exp. 1): Our results show, that DFI flows can provide a high-level abstraction with no or only negligible overhead compared to low-level RDMA verbs.

[3]In DFI, each buffer uses 32 segments each having a 8 KiB size in its default configuration.



(a) Single-threaded (b) Multi-threaded - 64 B tuples

Figure 7.10: MPI vs. DFI - point-to-point runtime

7.6.2 Experiment 2: DFI vs. MPI

In the following experiment we evaluate the performance of MPI and DFI in various settings. First we will show point-to-point performance for single-threaded and multi-threaded setups. Next we look at the collective functions provided by MPI and compare their usage for a typical shuffle scenario. For MPI we use the latest version (4.0.3rc4) shipped with HPC-X (2.6.0) for our InfiniBand hardware. MPI is therefore highly optimized to make use of the RDMA primitives offered by the network.

7.6.2.1 Point-to-Point Primitives:

As described in Section 7.2, MPI is process-centric, meaning it achieves parallelism by executing parts of the program in multiple processes on the same server. We therefore first compare MPI and DFI in a single-threaded setup before we then study the multi-threaded extensions provided by the MPI version of our InfiniBand deployment.

Single-threaded. Figure 7.10a reports the runtime for transferring a fixed table size (16 GiB). The *MPI_Send* and *MPI_Recv* primitives are used for sending the various tuple sizes, thereby using both MPI and DFI on a tuple-basis. Since MPI does not support any bandwidth/batching optimizations, the runtime is high for lower tuple-sizes since the network is inefficiently used. The bandwidth optimization for DFI makes efficient use of the network and therefore achieves a small runtime already for small tuple sizes.

Multi-threaded. Data processing systems typically use multi-threading (and not multi-process) to achieve parallelism while being able to share data between threads within

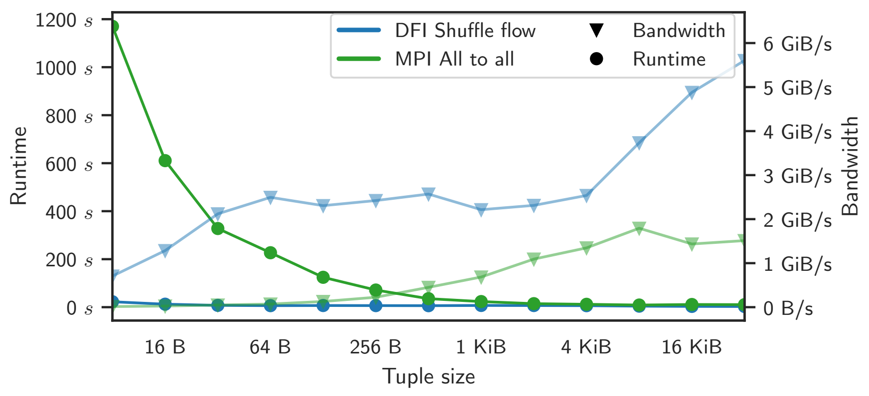


Figure 7.11: MPI vs DFI - collective shuffling (8:8)

the same virtual memory space. As such we evaluate the multi-threaded performance for DFI and MPI (using MPI_THREAD_MULTIPLE where multiple threads may call MPI primitives at once with no restrictions).

Figure 7.10b reports the runtime with an increasing number of threads. While DFI scales with the number of threads, the performance for multi-threaded MPI gets worse (red line). We analyzed this behavior and found that for data-heavy transfers (which MPI was not designed for), even a few threads lead to high internal contention on latches of MPI which causes the significant drop in performance.

The alternative in MPI for achieving parallelism per node is to use multiple processes instead of multiple threads per server. As we show in Figure 7.10b (green line) this leads to a better scalability than the multi-threaded MPI. However, multi-process solutions come at the cost that common data structures need to be accessed via (more expensive) shared memory.

7.6.2.2 Collective Primitives:

MPI also offers primitives that encapsulates communication between multiple nodes. Since MPI provides a rich library of collectives and we cannot provide an analysis in this paper which covers all collectives, we focus on the *MPI_Alltoall* collective since it resembles in a closest manner the semantics of an N:M shuffle flow of DFI.

Shuffle (Pipelined). We first look at the shuffle performance when using MPI in a streaming-based manner (i.e., we shuffle data in mini-batches with a size of 8 tuples - on average one tuple per target). In this experiment, only one thread per node is used (for MPI and DFI) which scans a table and shuffles the tuples based on their keys.

Multi-threading in MPI does not provide any benefit as we have seen before. For shuffling, we are using *MPI_Alltoall* (position-based) which uses a send buffer of the size of all nodes. As Figure 7.11 shows, the runtime of MPI for smaller tuple sizes is very high since the network is not utilized efficiently. However, as the tuple size increases, the bandwidth approximates that of DFI.

Shuffle (Batched).

To increase the network efficiency for MPI, we locally pre-shuffle the table on the shuffle key and invoke a *MPI_Alltoall* function for the complete batch. While this improves the bandwidth, collective functions are then susceptible to straggling behavior.

We evaluated the performance impact of MPI and DFI with one straggling node. To simulate a straggler, we decrease the CPU frequency of one of the nodes. The result is shown in Figure 7.12, where both the table sizes and straggling are varied. The increase of runtime for MPI with a straggling node (i.e., $s = 0.5$) comes from the fact that collective functions are blocking until all data is ready to be sent, and therefore limits the pipelining possibilities.

This is different for DFI. While DFI is also affected by straggling, it can constantly send data while the MPI implementation only starts the transfer once all data is available. Hence, DFI better overlaps the communication with the computation and therefore the straggling effect is less severe.

Key Insights (Exp. 2): In this experiment, we compared DFI to MPI. The experiments confirmed our speculations: (1) MPI neither provides efficient multi-threading, (2) nor does MPI allow to efficiently overlap compute and computation and hence support efficient pipelining.

7.6.3 Experiment 3: Use Cases

In the last experiments we evaluate DFI by implementing the two use cases we discussed in Section 7.4.3.

7.6.3.1 Distributed Joins:

Distributed joins are crucial operators in OLAP due to large amounts of data having to be transferred across the network, and therefore a good candidate to evaluate bandwidth-optimized flows of DFI.

Radix Join. We implemented a distributed radix hash join on DFI and compared its performance to a state-of-the-art implementation for RDMA using MPI [9]. Both implementations employ the same optimizations (e.g., write-combine buffer in partitioning

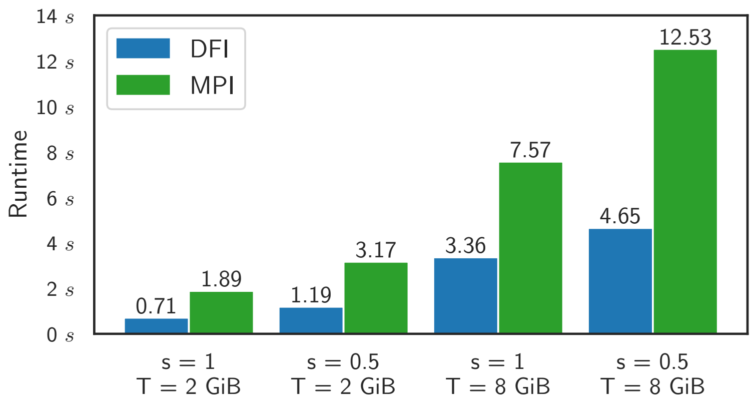


Figure 7.12: MPI vs DFI - collective shuffling (8:8) - One node straggling - s: straggling (s $*$ CPU freq.) - T: table size

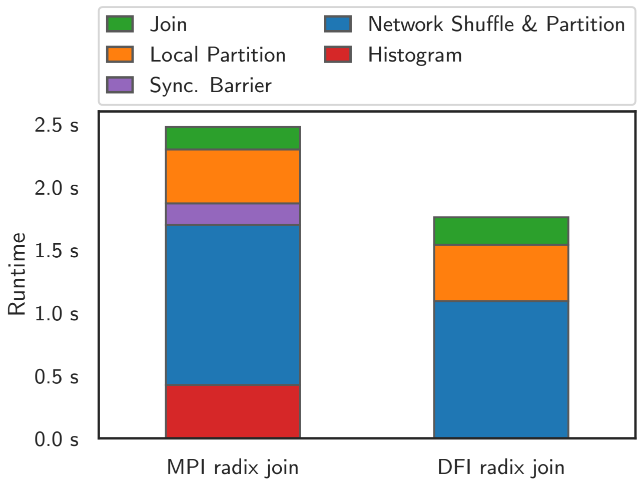


Figure 7.13: Distributed radix join - 8 nodes, 64 threads (DFI)/64 processes (MPI) in total. 2.56 B $\bowtie$ 2.56 B tuples.

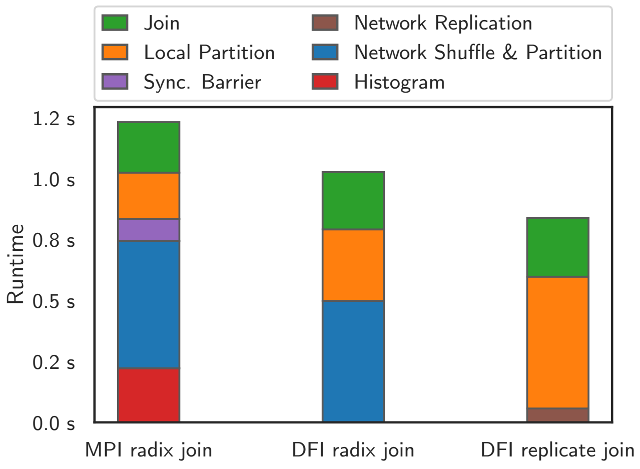


Figure 7.14: Distributed joins - 8 nodes, 64 threads (DFI)/64 processes (MPI) in total. 2.56 M ⋈ 2.56 B tuples.

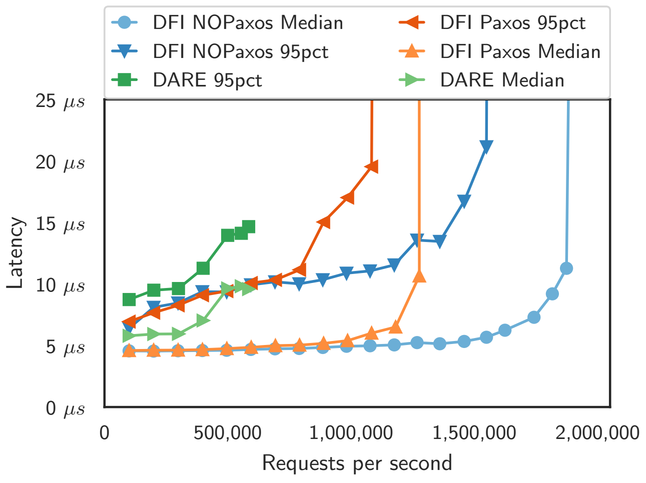


Figure 7.15: Performance comparison of DARE [108] with DFI-based implementations of Multi-Paxos and NOPaxos

phase and tuple compression). However, the MPI join of [9] uses multi-process parallelism while our join uses multi-threading instead. Figure 7.13 shows the average runtime of the two joins for all 8 nodes.

The DFI radix join achieves the best runtime mainly due to two design choices of DFI. At first, the DFI radix join does not need to first compute a global histogram of the partition buckets. The MPI radix join in [9] makes use of one-sided *MPI_Put* primitives. In order to achieve coordination free writes, it thus has to compute exclusive writing offsets for each partition using one additional pass. Different from this, DFI encapsulates the memory management through our buffer design which makes the additional pass superfluous.

The other reason for the runtime gap is due to the synchronization barrier needed in the MPI radix join after the network partition phase. Here, the join algorithm needs to make sure that all data has arrived before starting to process the local partitioning. While the data in this experiment is uniformly distributed, some runtime variance between multiple parallel workers still exists and is more pronounced in high-speed networks. This synchronization is not needed with DFI, since incoming tuples can already be processed when they arrive in a streaming-wise fashion.

Join Adaptability. Flows in DFI offer a high-level abstraction which encapsulates the data transfer of applications. As a result, it is trivial to adapt algorithms to use a different communication pattern. To demonstrate this, we adapted our radix hash join implementation to a fragment-and-replicate join variant which uses one replicate flow that replicates the inner table on all nodes. Figure 7.14 shows the runtimes of the three different join implementations with a smaller inner table (1000× smaller than the outer table). The replication of the small inner table is comparably cheap compared to shuffling the big outer table over the network. Overall, for this setup this helps to further reduce the overall runtime by another 20%,

7.6.3.2 State Machine Replication:

In this experiment, we implemented a simple key-value store that replicates data using a consensus protocol. For the experiment, we used two different consensus protocols, classical Multi-Paxos [74] and NOPaxos [82]. We modeled the normal, failure-free operation of Multi-Paxos as depicted in Figure 7.3. For NOPaxos, we implemented its *normal operation* protocol, which relies on the OUM primitive that can be provided by DFI's replicate flow, as well as its *gap agreement* protocol to detect lost messages. We compare both implementations with DARE [108], a state-of-the-art replicated key-value store that is based on a hand-crafted consensus protocol and heavily relies on one-sided RDMA.

We deployed all approaches with five replicas (a leader and four followers). Load was generated by six clients distributed across three separate nodes. Clients submitted 64 byte sized requests using YCSB's read-dominated workload [22] (95% reads and 5% writes). The results are shown in Figure 7.15.

The two DFI-based implementations consistently outperform DARE in our settings in both achieved throughput and latency. This is caused mainly by DARE's sequential design. First, each DARE client cannot submit a new request until it has received the result from its previous request, which limits its achievable throughput.

Second, DARE's write protocol serializes requests. While this limitation is mitigated by separately batching reads and writes, a mix of both request types frequently interrupts batches [134]. This is confirmed by DARE's own evaluation [108].

Our Multi-Paxos and NOPaxos implementation exhibit near-identical response latencies as long as they are not saturated. This appears counter-intuitive at first, as Multi-Paxos requires four message delays to respond to a client, whereas two messages delays suffices for NOPaxos as long as no messages are lost. However, fetching a global sequence number from the tuple sequencer of the ordered replicate flow incurs an additional two message delays.

For a load higher than 700k requests/s, we see benefits of our NOPaxos over our Multi-Paxos implementation. Under this load, the leader in Multi-Paxos becomes saturated as it has to repeatedly collect responses from a majority of replicas. In contrast, in NOPaxos the clients themselves collect these responses. This alleviates the burden placed on the leader in Multi-Paxos, which leads to stable response latencies in DFI's NOPaxos up to even higher request rates of almost 1.5M (95th percentile).

Key Insights (Exp. 3): In summary, DFI does not only achieve a better performance for distributed joins and consensus than state-of-the-art, but also offers an ease-of-use high-level abstraction to implement efficient solutions with a low code complexity.

7.7 Conclusions

In this paper, we presented DFI, a new data-centric interface for fast networks. With our implementation for InfiniBand we have shown that DFI adds only minor overhead compared to low-level abstractions such as RDMA verbs. Moreover, by implementing two use cases, we demonstrated that DFI can efficiently support data-centric applications with different requirements (high-bandwidth vs. low-latency) at high performance.

In future, we plan to integrate further useful extensions into DFI flows such as fault-tolerance as well as elasticity of flows to add/remove nodes at runtime. Furthermore, by open-sourcing our implementation, we hope to stimulate not only follow-up research but also allow that commercial vendors will provide a DFI implementation also for other high-speed network stacks.

7.8 Acknowledgements

This work was partially funded by the German Research Foundation (DFG) under the grants BI2011/1 & BI2011/2 (DFG priority program 2037), the DFG Collaborative Research Center 1053 (MAKI) as well as gifts from Mellanox and Huawei.

8 A DBMS-centric Evaluation of BlueField DPUs on Fast Networks

Abstract

Modern networks have evolved significantly in the last years. First, network speed has increased considerably and thus the use of low-overhead techniques such as RDMA has become more and more important to design efficient distributed DBMSs. Second, a recent trend in modern networks is that in addition to high-speed data transfer using RDMA, network components such as switches and NICs become programmable by providing additional computation on the device, such as DPUs (Data Processing Units). Such devices enable processing or manipulation of data as it is traversing the network and that way allow distributed systems to offload computation. While for the recent generation of RDMA-based DPU cards, there is no study that shows the offloading capabilities of DBMS tasks to such RDMA-enabled DPUs.

Therefore, in this paper, we aim to provide a first systematic study to evaluate the basic performance characteristics of the BlueField network cards in the context of typical DBMS operations. For the evaluation, we analyze the offload potential of using BlueField as a RDMA-enabled DPU for two important use cases: (1) a remote B-tree and (2) an end-host sequencer (i.e., remote counter). We chose these two use cases since they represent core tasks where RDMA has shown benefits. As a result, in our evaluation, we show that the recent generation of RDMA-based Bluefield DPUs can provide several benefits and can not only reduce access latencies but also improve the throughput. However, offloading computation to the DPU needs a careful design and naively offloading all computation to the DPU often leads to performance degradation.

Bibliographic Information

The content of this chapter was previously published in the peer-reviewed work: Lasse Thostrup, Daniel Failing, Tobias Ziegler, and Carsten Binnig. "A DBMS-centric Evaluation of BlueField DPUs on Fast Networks." In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2022, Sydney, Australia, September 5, 2022*. Ed. by Rajesh Bordawekar and Tirthankar Lahiri. 2022, pp. 1–10. URL: http://www.adms-conf.org/2022-camera-ready/ADMS22_thostrup.pdf.

8.1 Introduction

Motivation. In-memory DBMSs have become ubiquitous in academia and industry and many commercial offers are available today [13, 32, 34]. This is not surprising as the performance they offer is still unmatched compared to disk-based systems. However, a major challenge of in-memory DBMSs is that large data sets often do not fit into the memory of a single machine or the processing capabilities of one machine are insufficient. To that end, scale-out DBMSs are becoming the predominant solution to handle ever-increasing data set sizes and leverage more processing by utilizing more networked machines. Especially, DBMSs which are purpose-built for the cloud and can scale out or in on-demand are highly requested. As a core architecture for modern scale-out cloud DBMSs, disaggregated architectures have been crystallized which separate compute and storage. A prominent example of this architecture is Snowflake [24] but there are also other DBMS such as FoundationDB which take a more radical approach to separate not only storage and compute but also other database components in the pursuit to scale them independently [152].

However, with such disaggregated solutions, the network is increasingly on the hot path since data more often has to be fetched over the network. To keep up with this paradigm shift, networks are becoming faster with link speeds in the hundreds of gigabits, but also bringing rise to more powerful low-overhead networking technologies such as RDMA (Remote Direct Memory Access) which has influenced DBMS developments in academia and industry alike [69, 80, 87, 91]. The key driver for RDMA's success is that it offers low latencies in the single digit microsecond range as well as high throughput and thus has shown major gains for very different DBMS workloads.

A recent trend in modern networks is that in addition to high-speed data transfer using RDMA, network components such as switches and NICs become programmable by providing additional computation on the device, such as DPUs (Data Processing Units). Such devices thus enable processing or manipulation of data as it is traversing the network and that way allow distributed systems to offload computation [36, 60, 153]. The DPUs that are available today span far in terms of compute architectures, such as ASICs, FPGAs and general CPU cores, and therefore come with trade-offs in programmability and performance.

For the recent generation of RDMA-based network cards, DPUs are also becoming available. One of these devices which can be combined with RDMA is the BlueField Network Interface Card [102]. The BlueField card provides very flexible programmability due to its general-purpose ARM CPU cores that are available as compute resources on the

DPU. Moreover, specialized ASICs for tasks such as data encryption and decryption are available. While evaluations have shown that security-related tasks or tasks to provide tenant-isolation in data centers [20] can be provided in an efficient manner, there is no study that shows the offloading capabilities of DBMS tasks to such RDMA-enabled DPUs.

Contributions. Therefore, in this paper we aim to provide a first systematic study to evaluate the basic performance characteristics of the BlueField network cards in the context of typical DBMS operations. For the evaluation, we analyze the offload potential of using BlueField as an RDMA-enabled DPU for two important use cases: (1) a remote B-tree and (2) end-host sequencer (i.e., remote counter). We chose the remote B-tree because it is a frequently used data structure for DBMSs and it is also used in disaggregated architectures to avoid transferring all data across the network [135, 157]. On the other hand, the end-host sequencer is a commonly used building block for many distributed system tasks such as global ordering [82], for coordinating write access to shared memory [11], or to implement optimistic concurrency control [146].

For these two scenarios, we compare baselines that rely on existing one-sided and two-sided RDMA primitives with a solution that can use the DPU as an offload engine. For example, for an RDMA-based B-tree such as [157], the DPU could implement the tree traversal natively on the DPU. As such B-tree operations can be implemented in one round-trip from the compute layer to the storage layer without involving the CPU of the storage nodes. Moreover, in addition to compute resources DPUs often come with their own memory which allows them to store the B-tree in the DPU instead of using the CPU memory. Based on these observations, we thus aim to analyze whether a better performance can be observed for the two use cases above.

As a result, in our evaluation, we show that the recent generation of RDMA-based BlueField DPUs can provide several benefits. First, we show that we can reduce the network latency by avoiding the PCIe path from the NIC to the host which adds a non-negligible overhead to the overall latency [94]. Second, if DPUs are used as additional compute resources to the remote CPUs, we show that also overall gains in throughput can be achieved. However, offloading computation to the DPU needs a careful design and naively offloading all computation to the DPU often leads to performance degradations since the computational resources on the DPUs are often less powerful than the CPU in the remote host.

Outline. We first cover the background of the BlueField DPU and RDMA in Section 8.2 and next provide an overview of the experimental setup in Section 8.3. We then present the benchmarking of the before-mentioned use cases in Sections 8.4 to 8.6. Finally, we then present our conclusion on the evaluation to outline the pros and cons of offloading typical DBMS tasks to the BlueField DPU.

8.2 Background

In this section, we provide the relevant background on RDMA and the BlueField DPU.

8.2.1 Remote Direct Memory Access (RDMA)

RDMA has come to be an established state-of-the-art communication method for distributed data-processing systems [11, 68, 138, 146, 148, 157], since it overcomes the overhead of traditional kernel-space network stacks such as TCP/IP. To leverage RDMA, an application can make use of different communication schemes that can be categorized as one-sided (READ / WRITE) or two-sided (SEND / RECEIVE) operations, which refers to the involvement of the sender- & receiver-CPU in the communication. For one-sided operations, only the sender-CPU is actively involved, but as a consequence, the sender also has to decide where on the remote node the data should be written or read. With two-sided operations, the receiver-CPU is also actively involved in the communication since it needs to issue RECEIVE requests before SEND requests can be issued on the sender side, and it can thus also decide where to place data which simplifies the remote memory management.

Especially the one-sided RDMA operations have seen high adoption in distributed data processing systems since they allow sender nodes to write into remote memory directly without involving additional CPU cores of the receiving nodes. In a distributed DBMS, this reduces the overall CPU resources involved in data transfer and can thus lead to less resource consumption as well as overall lower latencies in many cases [35, 156]. Moreover, one-sided RDMA is thus, in particular, beneficial for disaggregated DBMS architectures in which the storage servers have limited or sometimes even near-zero computational resources [135, 149, 151]. On the other hand, the applications of two-sided RDMA is typically within RPC-style applications where the receiver node has to be actively involved in the communication.

8.2.2 Data Processing Units

We next provide the relevant background on the Data Processing Units (DPUs) that we use for our evaluation. Many of the major network hardware vendors are including DPUs

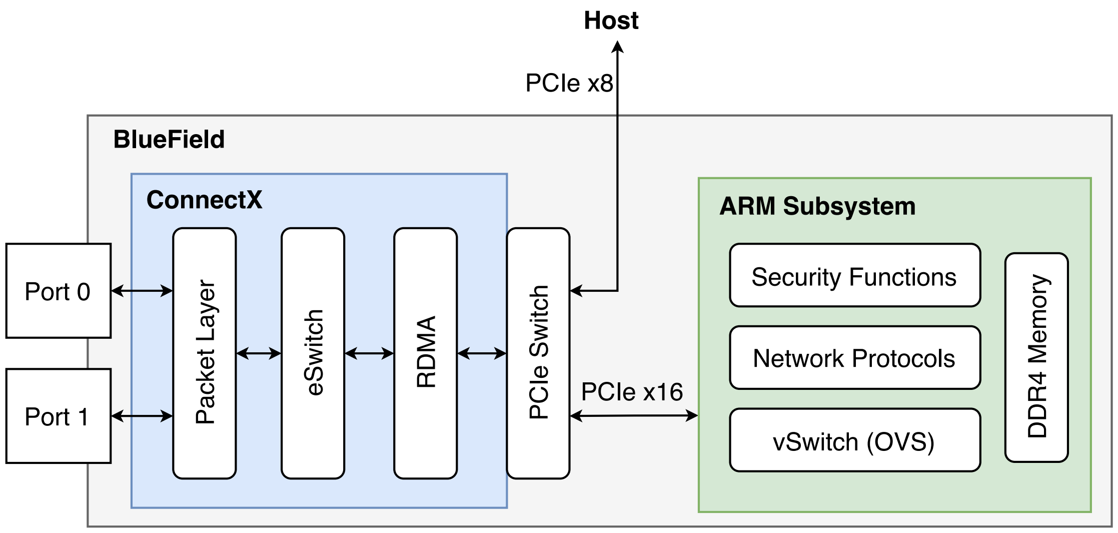


Figure 8.1: Functional diagram of the BlueField DPU [96].

to their product offerings, with examples like the Intel IPU, Broadcom Stingray, AMD Pensando or the Nvidia BlueField. DPUs come with different compute architectures ranging from P4 programmable ASICs to general-purpose CPU cores. In our evaluation, we focus on the BlueField cards from Nvidia. The BlueField cards are equipped with general-purpose ARM cores which are capable of executing any program logic in contrast to the more rigid ASIC architectures. The DPU runs its own OS (e.g., Ubuntu) and as such resembles another independent server with an added set of networking features. Internally, as illustrated in Figure 8.1, the DPU consists of the networking component (ConnectX) which provides hardware-offload of the network stack for more efficient (i.e., less CPU intensive) networking. Moreover, the ARM cores which act as computational resources on the DPU are equipped with DDR4 memory and are connected to the ConnectX over an internal PCIe switch. The BlueField (from the second generation on) is additionally equipped with hardware accelerators for compression/decompression, encryption and regex pattern matching.

The BlueField DPUs themselves are attached to the host CPU via PCIe. For controlling and routing the network traffic to the host, the DPUs have different modes of operation. There is a *Separated Host* mode, where the ARM subsystem will appear as an additional computer on the network. When the remote machine uses this setup, a client can decide where data should be transferred to, either the ARM subsystem or the host. Another mode is the *Embedded Mode*, where the ARM subsystem manages the network on behalf of the host and controls the physical ports of the network card. With that, the traffic from or to the host always goes through the ARM subsystem. The Embedded Mode can be further specialized, such that the host does not have access to the DPU directly. This mode allows cloud providers to provide isolation and offloading of cloud management tasks to the DPU.

To control the network communication in the Embedded Mode and forward traffic with low latency, the ConnectX subsystem of the BlueField offers a component called eSwitch. While the DPU controls the physical network ports in the Embedded Mode, the host as well as the DPU can have additional virtual interfaces which are then available in the DPU for further network flow control. By default, an OpenVSwitch process is running on the BlueField, which controls the communication between physical and virtual network ports. Flow rules controlling the flow of packets will be configured in the OpenVSwitch, and, if possible, offloaded to the eSwitch. In a typical network flow for RoCE (RDMA over Converged Ethernet), the network packets are processed by the eSwitch, which forwards the packet to the host or DPU based on the MAC addresses. However, the eSwitch is limited to specific network headers, like MAC or IP addresses, VLAN tags or TCP/UDP ports. As a result, processing or manipulation of RDMA packet headers is not possible in the eSwitch. Controlling traffic unsupported by the eSwitch means that the packets need to be processed by the BlueField's ARM cores.

Overall, the host CPUs, the BlueField DPU and the ConnectX subsystem are connected by a PCIe switch. In contrast to other vendors, BlueField DPUs do not have a dedicated DMA engine to access the memory of the host. However, RDMA operations can be used for the communication between DPU and host CPU. The RDMA API then generates normal requests through the ConnectX subsystem, which can then access both the host's as well as the DPU's memory. In this way, not only can the DPU make DMA requests to the host's memory, but the host can also access the DPU's memory.

8.3 Use Cases & Experimental Setup

Use Cases. As mentioned before, in our evaluation we consider two use cases to evaluate the performance of the BlueField cards. In the first use case, we focus on a remote B-tree, which has already been studied in the context of RDMA in previous work [157]. We choose to evaluate and implement two remote B-tree strategies, either with RPC over two-sided RDMA in Section 8.4 or completely one-sided RDMA (i.e., no RPC) in Section 8.5. In the last use case we evaluate an end-host sequencer, which is a common building block in distributed systems, to e.g., assert a global order or coordinate access to

	BlueField-1 BF1M332A	**BlueField-2** BF2H332A
CPU	8× ARMv8 A-72 model 0 @ 800MHz	8× ARMv8 A-72 model 3 @ 2.5GHz
Memory	1× 16 GB DDR4-2400	1× 16GB DDR4-3200
Caches	L1: 32 KB / core L2: 1 MB / 2 cores L3: 12 MB (shared)	L1: 32 KB / core L2: 1 MB / 2 cores L3: 6 MB (shared)
Network	ConnectX-5	ConnectX-6
Interfaces	2× 25 Gbps	2× 25Gbps
PCIe Host	PCIe 4.0 ×8	PCIe 4.0 ×8
PCIe ARM	PCIe 4.0 ×16	PCIe 4.0 ×16
OS	Ubuntu 18.04	Ubuntu 20.04

Table 8.1: Comparison of BlueField-1 and BlueField-2.

shared memory [11, 67]. We take a look at a sequencer implemented only with one-sided RDMA or with RPC two-sided RDMA.

Experiment Setup. We evaluate both available generations of the BlueField DPUs — the BlueField 1 and BlueField 2 — which mostly differ in their processing power; i.e., the BlueField 1 only runs at a clock frequency of 800 MHz whereas the BlueField 2 runs at 2.5 GHz (details outlined in Table 8.1). In our experimental setup, we are mirroring a typical disaggregated storage and compute setup. The storage node is equipped with a BlueField 1 as well as BlueField 2 card to compare both DPUs. The storage server is running with 2× Intel Xeon Gold 6326 CPUs, 512 GB DDR4 memory and PCIe 4.0. The compute node uses a non-programmable RDMA ConnectX-5 NIC and is equipped with 2× Intel Xeon Gold 5220 CPUs with 512 GB DDR4 memory (Table 8.2). For both servers, we only use one NUMA socket so as to not involve any cross-NUMA traffic effects. Both servers are connected with RDMA over Converged Ethernet (RoCE) v2. Note that the BlueField cards used in our setup only provide maximum 25 Gbps per network link. However, since the BlueFields NICs are equipped with two links, we can in total use a 50 Gbps connection between the compute and storage server by splitting the traffic over the two links.

We configure the BlueField cards in the *Embedded Mode*, and configure two virtual interfaces which route to either the ARM subsystem or the host. The routing is offloaded into the eSwitch and does therefore not introduce measurable overhead on the ARM subsystem.

	Storage node	**Compute node**
CPU	2× Intel Xeon Gold 6326 @ 2.9 - 3.5 GHz	2× Intel Xeon Gold 5220 @ 2.2 - 3.9 GHz
Caches	L1: 48 KB / core L2: 1.25 MB / core L3: 24 MB (shared)	L1: 32 KB / core L2: 1 MB / core L3: 24.75 MB (shared)
Network	BlueField-1 BlueField-2	ConnectX-5
Memory	16× 32GB DDR4-3200	8× 64GB DDR4-2666
OS	Ubuntu 20.04	Ubuntu 18.04

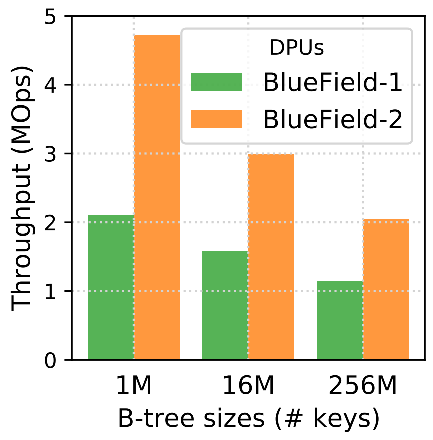
Table 8.2: Server nodes used in the experimental setup.

8.4 Use case 1: Remote B-Tree with RPC

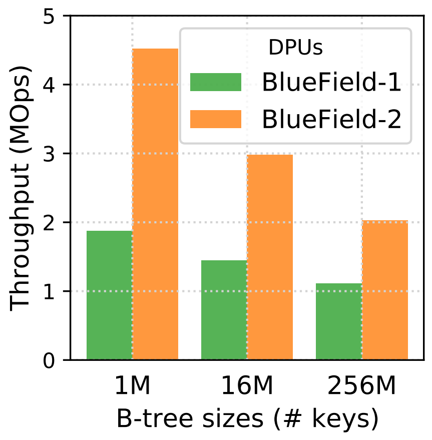
For the first use case we use a remote B-tree which is accessed via RPC calls. To realize the RPC framework, we use two-sided RDMA SEND/RECEIVE verbs with existing optimization such as door-bell batching on both client- and server-side and inlining for reducing PCIe overhead [67]. For our B-tree, we use an OLC (optimistic lock-coupling) synchronization protocol to allow scalable reads [78]. For keys and values, we use 8 byte integers. We use either a mix of 50/50 read-write or read-only workloads as evaluation.

For all experiments in this use case (unless otherwise stated) we use 8 threads (maximum number) on the BlueFields and on the CPUs of the storage server respectively to achieve a comparable setup. This is important for Section 8.4.1.2 when we show the offload potential by varying the percentage of requests handled by the NIC. Additionally, using 8 threads on the storage better reflects a typical storage node with weaker CPUs than compute nodes [151, 159]. On the compute server, we use 16 threads to attribute to the fact that compute servers are typically equipped with more computational resources and thus be able to generate sufficient workload in our evaluation.

The evaluation of this use case is structured as follows: we first report the throughput characteristics by first contrasting the two BlueField generations and next dive further into using the BlueField-2 in union with the host server. Subsequently, we evaluate the latency characteristics to determine whether any latency benefits can be observed for the BlueField DPU.



(a) Read-only



(b) 50/50 read-write

Figure 8.2: Remote B-tree on BlueField-1 or BlueField-2 with RPC.

8.4.1 Throughput Characteristics

8.4.1.1 BlueField-1 vs BlueField-2

The BlueField-1 was already launched in 2017, and has in recent years been preceded by the BlueField-2. We first aim to evaluate the performance improvement between the two generations and their individual ability to handle B-tree RPC requests. We initialize B-trees of different sizes in the local memory of either the BlueField-1 or 2. Due to the very low clock frequency of the BlueField-1 ARM cores (over 3× less than the BlueField-2) and the fact that the RPC handling is typically very CPU intensive, we expect to see a significant performance difference between the two generations.

In Figure 8.2 we observe the achieved throughput for a (a) read-only or (b) 50/50 read-write workload. For the smallest tree with 1 million (M) keys (tree size between 16 & 32 MB), the BlueField-2 outperforms the first generation by 2.4× for 50/50 read-write which can be attributed to the difference in clock frequency (800 MHz & 2.5 GHz). Coupled with the fact that most of the tree can reside in the CPU caches, this indicates they are both CPU bound. However, with larger tree sizes, the BlueField-2 becomes increasingly memory bound and the relative performance difference is decreasing.

Between the two workloads (read-only and 50/50) we can only observe a slight difference in performance for the small tree with 1 M keys. The reason why barely any difference in performance is observed for 16 M and 256 M is due to more sparse reads and writes, and the reduction in throughput (i.e., a read has a smaller chance of conflicting with a write).

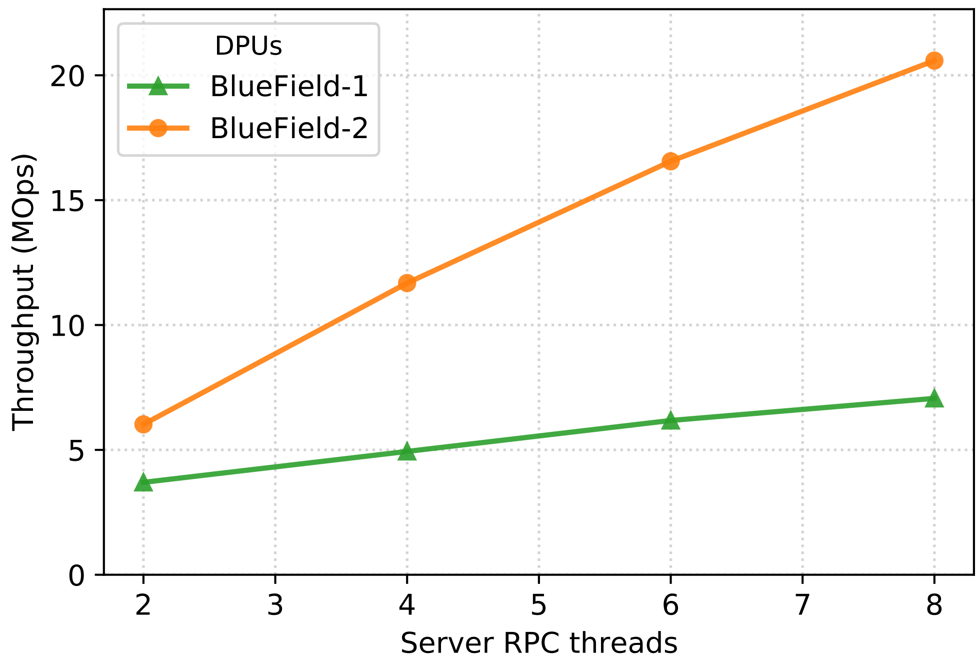


Figure 8.3: No-Op RPC BlueField-1 vs BlueField-2.

To isolate the performance difference between the two BlueField generations further, we now take a look at the pure RPC handling capabilities. We issue RPC calls from the compute node which does not contain any operation to be performed, i.e., a No-Op. In Figure 8.3 we scale the number of threads on the BlueFields that handle the RPC requests. We observe that the BlueField-2 has very good scalability, i.e., almost linear, whereas the BlueField-1 only increases the throughput with 1.9× from 2 to 8 threads. Note, since we use both available interfaces of the BlueFields, we run with a minimum of 2 threads.

Based on these findings it is clear that the second BlueField generation already provides a substantial performance boost over the first generation. The suboptimal performance of the BlueField-1 renders it hard to integrate into any performance-critical data-intensive use cases. We argue that the performance difference mainly stems from the very low clock frequency of the BlueField-1 ARM cores, as this is the main differentiator. In the remainder of the paper, we thus focus on the performance of the BlueField-2.

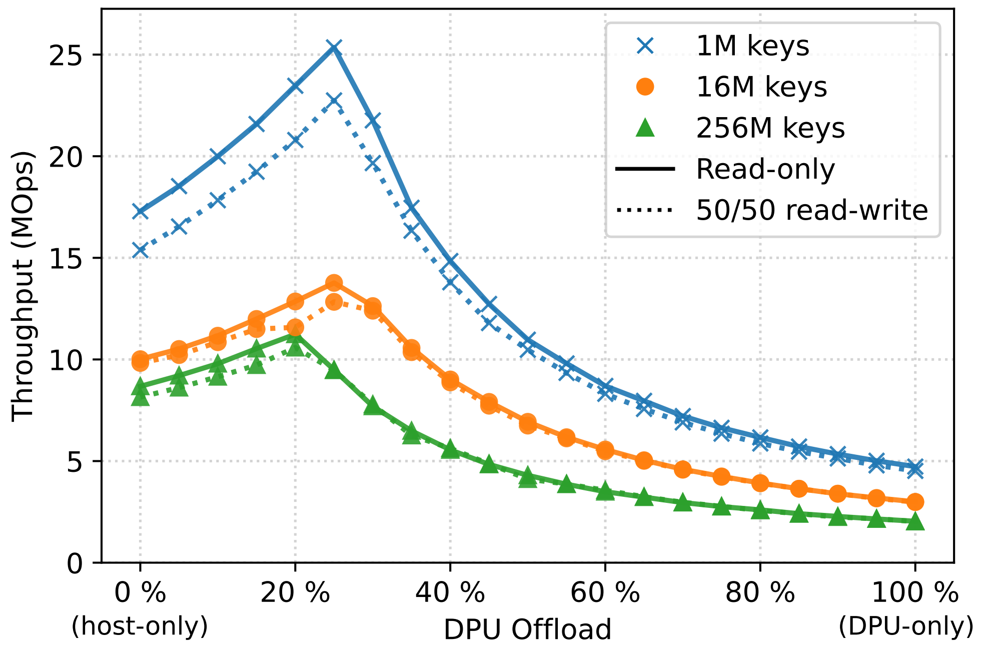


Figure 8.4: Remote B-tree with increasing offload on BlueField-2 for various B-tree sizes. RPC requests with read-only or 50/50 read-write.

8.4.1.2 BlueField-2 Offloading

We now evaluate the offloading potential of the BlueField-2 DPU by utilizing the DPU together with the host CPUs of the storage server. We use all 8 CPU cores on the DPU and also use 8 CPU cores of the host, which we keep fixed regardless of the partition sizes to the host or DPU. To use both compute resources, we range-partition the B-tree between the host memory and the DPU memory and observe the achieved overall throughput. For partitioning, we use different setups ranging from 0-100% of the B-tree being stored on the DPU. Moreover, we create the index requests uniformly in the whole key range such that the relative partition sizes of the tree also match the workload generated to each compute device (i.e., the host CPU or the BlueField-2 DPU).

In Figure 8.4, we see the results when we gradually increase the range partition of the B-tree on the DPU and decrease it on the host, indicated by the x-axis. With 0% DPU offload, the host contains the full B-tree and as such all requests are handled by the host and the DPU is not processing any requests. Instead, with 25% offload, 3/4 of the B-tree is on the host and 1/4 is on the DPU. Overall, in Figure 8.4 we initially see a steady increase in throughput as more requests are routed to the DPU up until around

25% whereas for the tree with 1M keys, the overall throughput increases by 47%. For larger tree sizes, the observed throughput increase is slightly less, with around 30% for the B-tree with 256 M keys.

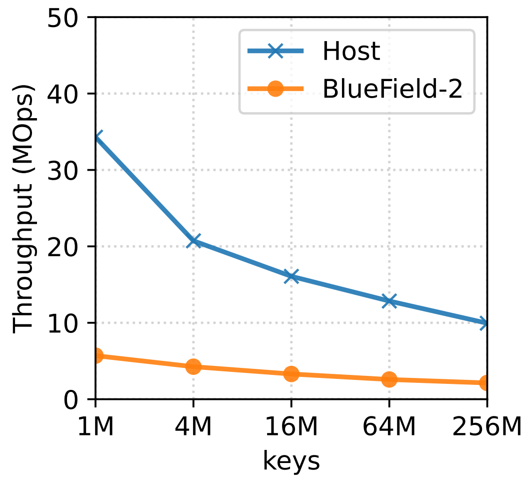
However, offloading more than about 25% of the B-tree to the DPU is detrimental to the throughput since the DPU is then overloaded and the performance degrades to DPU-only throughput as reported in Figure 8.2 already. This degradation of throughput is not surprising as both the CPU on the DPU is weaker and the main-memory is slower than that of the storage host as reported in Table 8.1 and Table 8.2. Moreover, the read-only and 50/50 read-write workload only differs slightly for the higher throughput and smaller B-tree cases for the same reasons as discussed before.

These results indicate that while the BlueField-2 is not powerful enough to achieve high throughput in comparison to the host, it yields a significant speedup by using the DPU resources in addition to the host CPU. However, this imposes challenges for real-world use cases, as the optimal partitioning of the B-tree is dependent on the workload (i.e., potential access skew) and the performance of the host server in relation to the DPU. As such, more sophisticated adaptive solutions could come into play, which re-balances and re-partitions the B-tree between host and DPU based on utilization metrics, to automatically adapt to the most optimal partitioning between the host and DPU. Such a design is, however, out of scope of this paper.

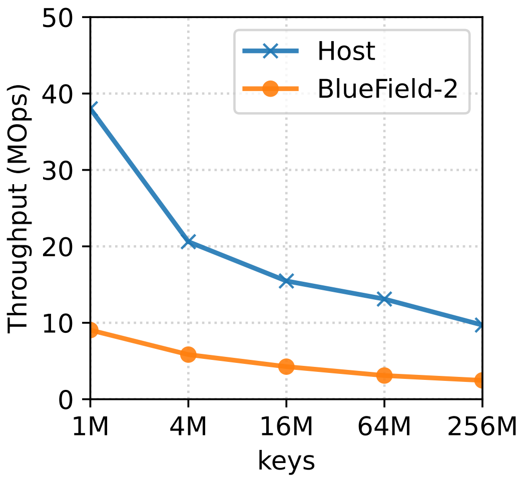
8.4.1.3 Local B-tree - Throughput

To compare the achieved performance for the remote B-tree with RPC, we locally execute lookup and update operations on the B-tree on either the storage host or BlueField-2, ultimately determining whether the RPC-handling or the B-tree is the bottleneck.

In Figure 8.5 we execute local updates (a) or lookups (b) on different sized trees. If we compare the lookup performance to the 100% offloaded B-tree scenario in Figure 8.4, for the smallest tree the RPC lookup (read-only) throughput is 4.7 MOps and 9.9 MOps for the local B-tree. This indicates that half of the throughput is lost to RPC networking overhead for the smaller tree. For the largest tree of 256M keys, the difference is much less pronounced where the RPC lookup throughput on the DPU is 2 MOps and 2.5 MOps without RPC overhead. The reason behind this is that for the largest tree, the bottleneck is increasingly the cache misses going to the main-memory. The CPU cache and instruction overhead introduced by the RPC handling does therefore not affect the performance as much relative to the smaller tree. The same trend is also observed for the host.



(a) Updates



(b) Lookups

Figure 8.5: Throughput of local B-tree update and lookup operations on 8 threads.

Overall, these findings also confirm our previous results in Figure 8.4 since the benefits of the offloading stem from the lookup requests that the DPU can provide additionally to the host CPUs.

8.4.2 B-tree Latency Characteristics

We now evaluate the latency characteristics of the B-tree with RPC.

8.4.2.1 BlueField-2 Offloading

Since the CPU cores of the DPU are co-located with the ConnectX networking chip on the same device, we want to evaluate whether any latency improvements can be observed by using the DPU in contrast to the host. We again evaluate the performance by offloading different partition sizes of the B-tree to the DPU. To not overload the host or DPU we let only 1 client thread on the compute node issue requests.

In Figure 8.6 we report the median latency for read-only RPC requests. We observe that the best latency is achieved with the complete B-tree located on the host (i.e., 0% offload). The latency increases slightly (between 14 and 20%) with more RPC requests being routed to the DPU. This is in contrast to our initial expectations, as having the CPU cores and memory situated on the same device should result in lower network latency. However, two factors might be at play here which render the achieved latency of the BlueField-2 DPU worse than the host; (a) since the DPU CPU cores are in fact

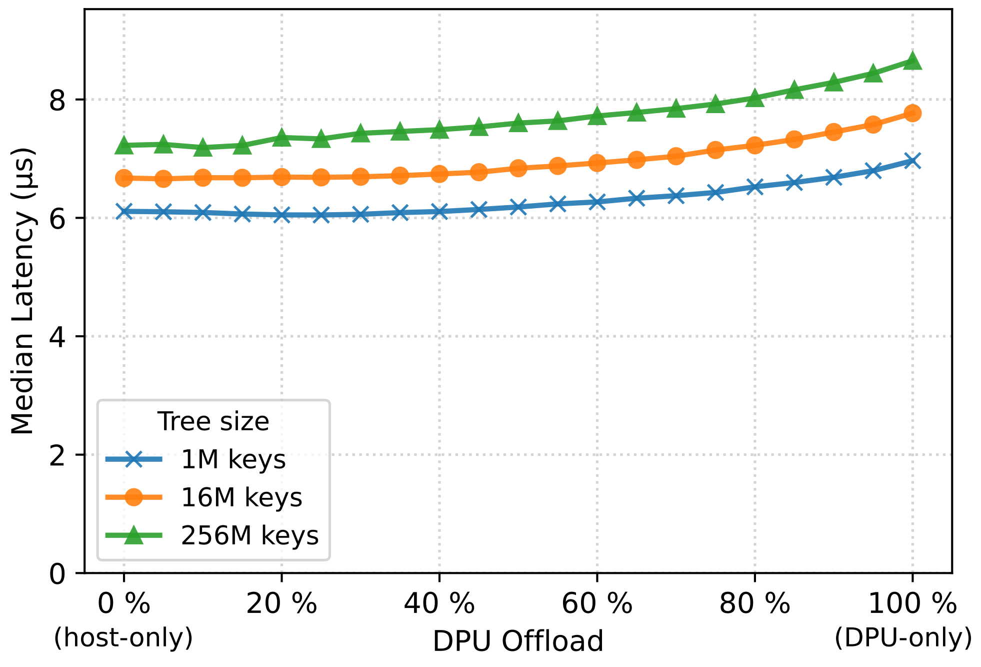


Figure 8.6: Remote B-tree latency with increasing offload on BlueField-2. RPC requests with read-only.

also separated from the networking hardware (ConnectX) over PCIe, this potentially diminishes the latency benefit and (b) memory access latencies might be worse on the DPU for B-tree lookup in comparison to the host.

In the following, we evaluate these two aspects independently to account for the latency increase observed in Figure 8.6.

8.4.2.2 RDMA Send Latency

To isolate the latency characteristics of the network we execute the PerfTest[1], which is a standard RDMA benchmarking tool.

In this experiment, the compute server issues SEND requests to either the storage host or the BlueField-2 DPU using PerfTest's *ib_send_lat*. It is important to note that the *ib_send_lat* test from PerfTest implements a ping-pong and reports half of the round-trip as latency. In this benchmark, the SEND performance of the storage server's host CPU or DPU is therefore also included. We execute the benchmark both with and without data inlining. If the payload of an RDMA send operation is less than the maximum

[1] https://github.com/linux-rdma/perftest

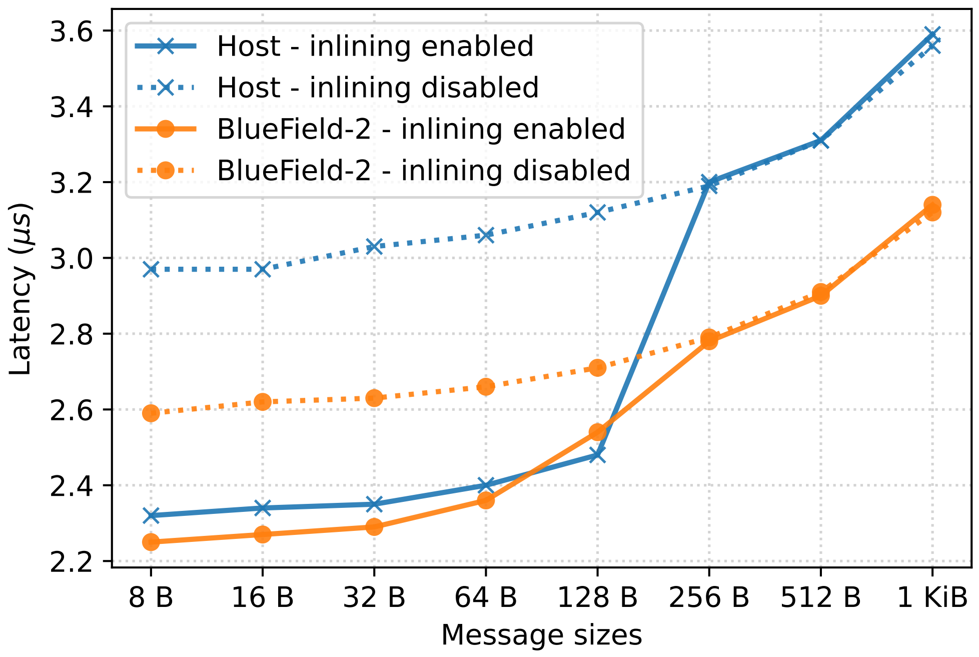


Figure 8.7: RDMA Send latency measured with *ib_send_lat* from the compute server to the storage server (host) or BlueField-2, with or without inlining of data up to 236 B.

possible inlinable size, a PCIe round-trip to the NIC can be saved. The inlining size limit for our hardware is 236 bytes and enabled by default.

Running the *ib_send_lat* benchmark for different message sizes in Figure 8.7 shows that with inlining enabled (up to 237 bytes), the latency is almost the same. However, without inlining (dotted lines or above 237 bytes) we see a latency improvement for the DPU around 0.4 μs. The experiment shows that while there can be a real latency improvement for the DPU, for small message sizes (e.g., 32 bytes as used in our RPC framework), inlining reduces the additional PCIe overhead present for the host versus the DPU.

8.4.2.3 Local Memory Latency

As such, we next speculate that the reason why the observed RPC B-tree latency is not lower on the DPU versus the storage host might be due to worse memory access latency. We test this by measuring the latency for random memory access over different memory block sizes. With smaller memory sizes, the CPU will be able to cache most requests, but with larger sizes, more cache-misses will occur. We compare the latencies of the

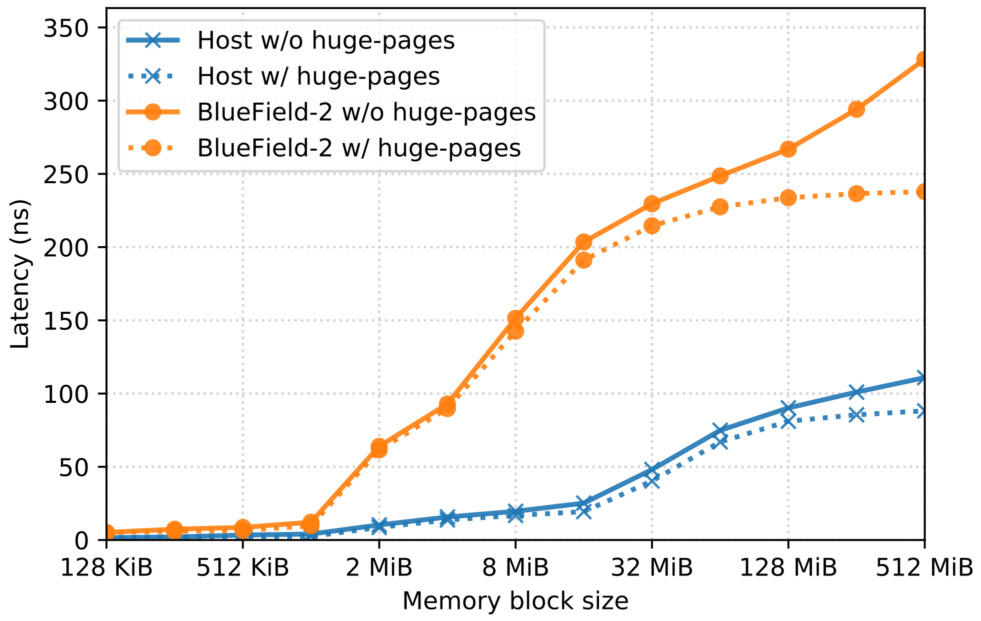


Figure 8.8: Memory read latencies for increasing memory block sizes. Data obtained through tinymembench[2].

storage host and the DPU to evaluate both their ability to cache reads and the cost of cache-misses.

In Figure 8.8, we report the local memory access latency for the host and BlueField-2. Already at around 1 MiB, the BlueField-2 memory accesses become relatively more expensive which can be attributed to the smaller cache sizes for the DPU. As the last-level cache of the BlueField-2 is only 6 MiB, the data spills out of the caches and memory accesses can to a smaller degree be cached. In comparison to the host with 48 MiB last-level cache, access latencies are much smaller and stable. We also report the effect of huge-pages where the increasing cost of TLB-misses already gets visible for the BlueField at around 64 MiB. For the B-tree RPC experiments, we do not utilize huge-pages.

Overall, the memory access latency is substantially higher on the DPU which contributes to the fact that we do not experience any latency benefit for the RPC B-tree use case.

[2] https://github.com/ssvb/tinymembench

8.4.3 Discussion

We now summarize the main findings for evaluating the BlueFields in a remote RPC B-tree use case. In conclusion, there is both potential for integrating the DPU, but also downsides.

A goal of our evaluation of the BlueField is to test whether any latency improvements can be observed. While there is in fact a detectable latency increase for networking messages on the BlueField, several factors render the remote B-tree access slightly slower than the host. These factors are slow local memory, smaller caches and smaller benefit of inlining small message sizes.

The CPU cores and the local memory on the DPU are substantially slower than a typical server-grade machine and as such blindly offloading data-intensive operations onto the DPU severely impacts the achievable performance. We instead argue that the DPU must be carefully integrated such that the workload offloaded to the DPU corresponds to the processing capabilities.

8.5 Use case 1: Remote B-Tree with One-sided RDMA

More and more designs are utilizing one-sided RDMA to access remote data structures in disaggregated memory setups [2, 135, 157, 159]. The reason for this is that one-sided operations help to remove the load on the (potentially weak) remote memory servers. Since we already saw that the BlueField-2 struggles to achieve good RPC performance due to the relatively slow memory and CPU cores, one-sided access is a promising use case as it does not incur any CPU overhead on the DPU.

In this use case, we therefore evaluate a remote B-tree accessed only over one-sided RDMA read operations. The way a remote B-tree lookup works for one-sided operations is that clients are first issuing a read on the root node and locally performing a binary search to determine the next child node. This is repeated until the leaf level. As such, an RDMA read request is issued for each level of the B-tree and since the reads are interdependent (i.e., the location of one read depends on the previous) they cannot be overlapped.

We first evaluate the throughput characteristics and subsequently look at the latency. Last, we discuss the findings holistically.

8.5.1 Throughput Characteristics

One-sided remote data structures often come with a lower throughput than their RPC counterparts since a network round-trip is necessary for each data-dependent read. However, a unique possibility with one-sided accesses is that the data structure can easily be distributed out on multiple storage nodes to spread out load and achieve higher throughput [135, 157]. This is possible since each read is anyway a remote access and can therefore be directed to any storage server.

In our B-tree benchmark, we however focus on just one storage server and compare the throughput while gradually offloading the B-tree to the DPU. We see in Figure 8.9 that the throughput stays stable with more of the B-tree being partitioned to the DPU. As such, there is no real difference in terms of throughput performance whether the B-tree is offloaded to the DPU or not. The DPU is therefore a good candidate for offloading one-sided accesses since it alleviates load on the main-memory of the host system of the storage server.

We also evaluate the impact on different node sizes of the tree, i.e., the size of each RDMA read, and observe that for the tested tree sizes, 2048 B node sizes provide the best throughput at 0.8 MOps. The reason behind this is that even though an RDMA read of 2048 B is more expensive in terms of throughput and latency than 512 B, larger nodes result in a bigger fanout and a shallower tree requiring fewer network round-trips. As previously mentioned, while the absolute throughput is much lower compared to an RPC solution, the contention on the remote NIC can be spread out to achieve higher throughput.

8.5.2 Latency Characteristics

While we did not see any latency improvement for offloading RPC B-tree lookups to the DPU due to the smaller caches and slower CPU, these factors are not as influential for one-sided accesses. We, therefore, expect to see a reduced latency as more lookups are offloaded to the DPU, due to the close proximity of the DPU cores to the network. In Figure 8.10 we execute the B-tree lookups with different partition sizes offloaded to the DPU. A clear trend is shown here that the DPU provides faster lookups than the host. For the different tree sizes and node sizes, the improvement is around 11-13%.

The difference in latency observed for the evaluated tree sizes and node sizes is due to the different latency of the RDMA read and the depth of the tree. As an example, a node size of 512 B has a fanout of $512B/16B = 32$ (with 16 B used as key and child pointer), so for a tree with 256 M keys, the depth will be $\lceil log_{32}(256M) \rceil = 6$, whereas

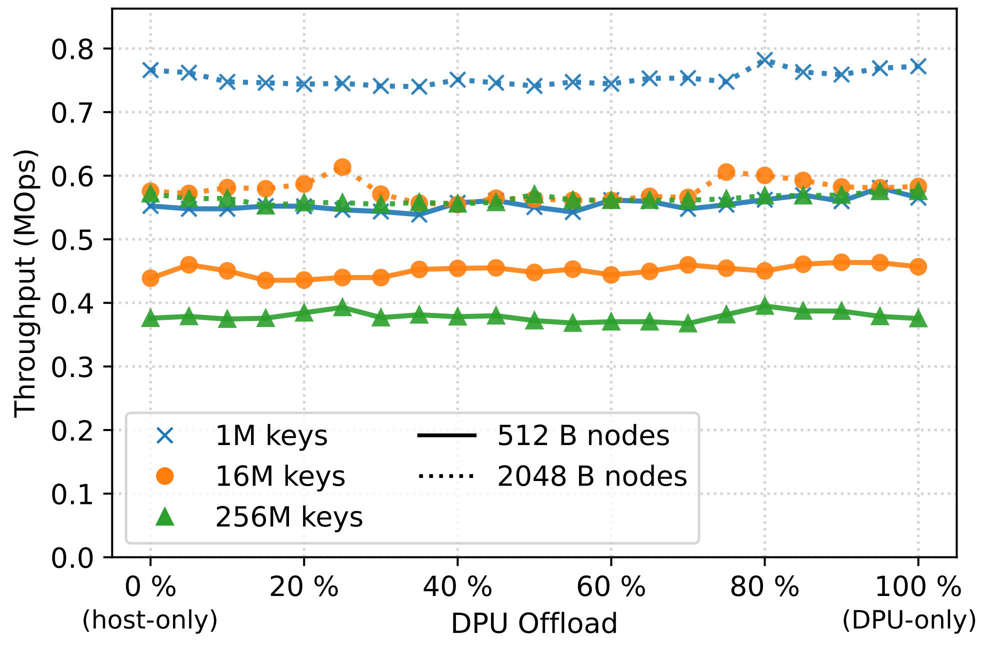


Figure 8.9: Remote B-tree throughput with increasing offload on BlueField-2. Read-only with one-sided RDMA.

a node size of 2048 B only has a depth of 5 and therefore one less RDMA read. The experiment shows that the lower latency of a 512 B read with respect to a 2048 B read does not amortize the cost of an extra read in the B-tree.

8.5.3 Discussion

In conclusion, since the relatively weak CPU cores of the BlueField-2 are not engaged with one-sided access, the DPU has better offloading potential. The strongest benefit comes with lower access latency due to the co-location of the CPU cores and the network on the same physical board.

Another interesting benefit given by offloading the one-sided accesses to the DPU is that read or write pressure on the local main-memory of the host is alleviated, which might benefit concurrent memory-intensive applications. This is even more noticeable with faster networks such as the BlueField-2 model with 200 Gbps.

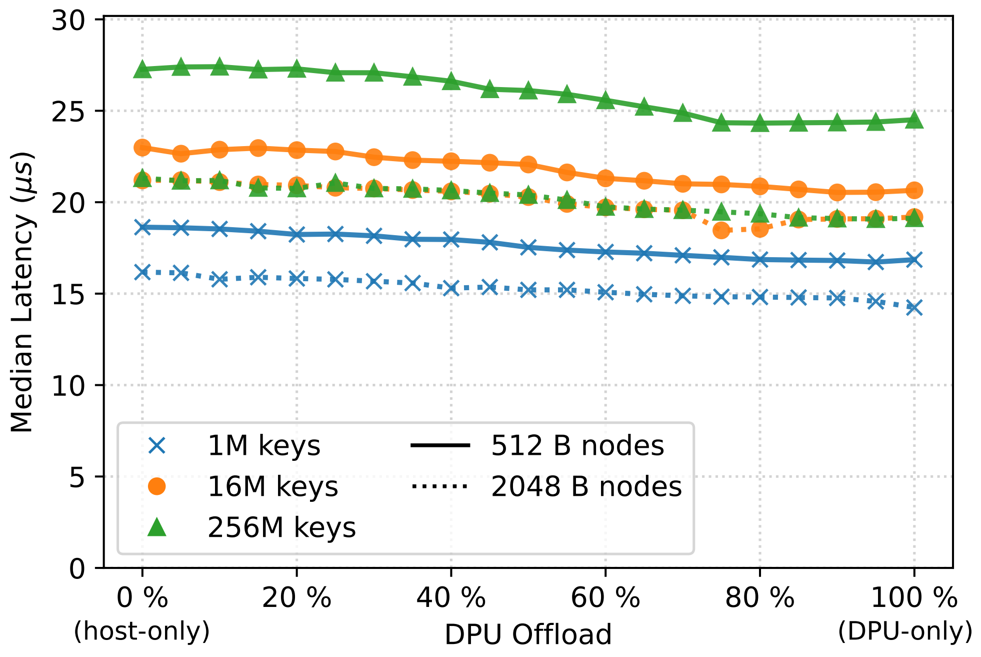


Figure 8.10: Remote B-tree latency with increasing offload on BlueField-2. Read-only with one-sided RDMA.

8.6 Use case 2: Remote Sequencer

A common building block in distributed systems is global counters. They are among others used for global timestamps, asserting message ordering or coordinating access to shared memory [67]. There has already been work that incorporates counters in the network such as a programmable switch [82] or directly in the SRAM of an RDMA-NIC [135], we, however, aim to evaluate the more traditional setup of placing a counter in remote DRAM main-memory. We evaluate the performance differences between the storage host and the BlueField-2 DPU with both one-sided RDMA atomic operations and with RPC with local atomic operations.

8.6.1 One-sided RDMA Atomics

Atomic operations are already provided in the collection of RDMA primitives such that multiple clients can perform fetch-and-add or compare-and-swap operations over the network without any additional coordination. In general, this can facilitate one-sided access to remote data structures without any locking. For this use case, we evaluate a remote counter (i.e., sequencer) accessed with one-sided RDMA fetch-and-add operations.

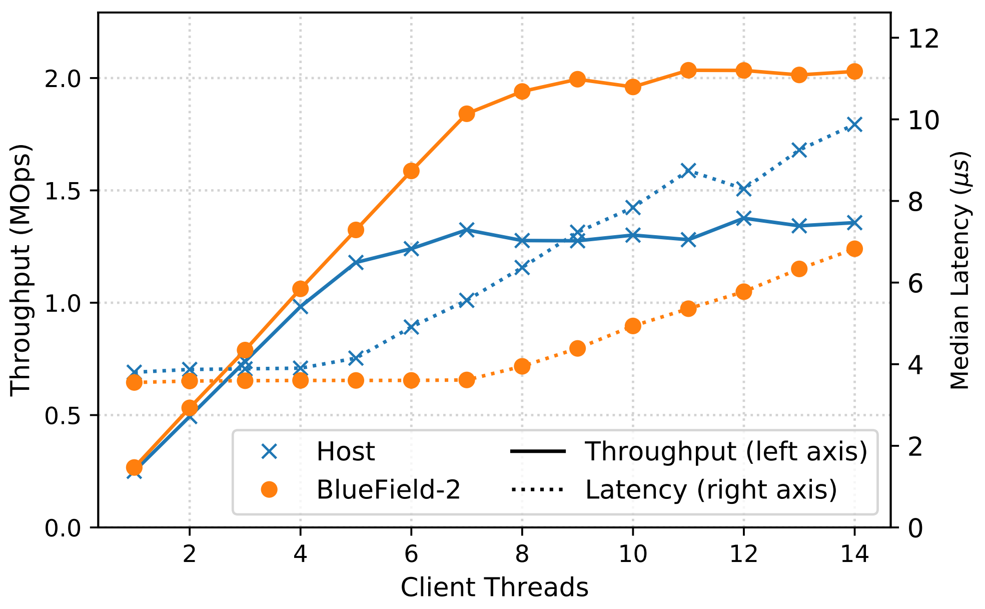


Figure 8.11: Throughput and latency of one-sided RDMA fetch-and-add on either the storage host or the BlueField-2 DPU.

In Figure 8.11 we report the throughput and latency with an increasing number of client threads on the compute node accessing the same counter. In this use case we see a substantial benefit of the DPU in terms of achieved throughput. Placing the atomic counter on the DPU achieves an almost 50% throughput speedup. This is also reflected in the number of clients needed to saturate the remote server where the throughput of the DPU is saturated by around 8 clients whereas the storage host only can scale up to around 6 clients.

The latency is almost identical with a slight benefit on the DPU. Beyond the saturation point, the latencies increase linearly with more clients added as contention is created and requests are increasingly queued.

8.6.2 RPC with Local Atomics

The alternative to realizing a global sequencer with one-sided primitives is to use a two-sided approach with RPC requests which access the counter with local atomic operations. Such a design involves the remote side and therefore the CPU and memory resources have a bigger impact on the performance as already established previously. For completeness and point-of-comparison we include the achieved performance with a global sequencer realized on either the host CPU or the DPU.

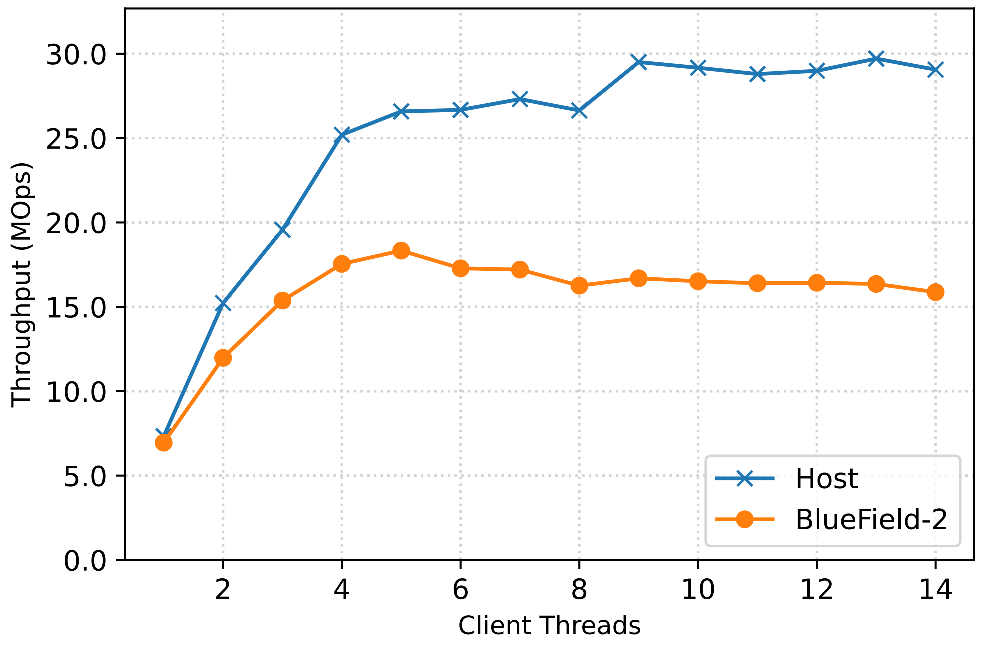


Figure 8.12: Throughput of RPC sequencer on either the storage host or the BlueField-2 DPU.

In Figure 8.12 we report the achieved throughput with an RPC implementation. Similar behavior as the B-tree RPC experiment can also be observed here, namely the relatively powerful CPU of the host results in substantially higher throughput than the DPU. The performance difference is however only up to 2× in this use case versus roughly 3× in the B-tree. The reason for this is that the counter is less memory-intensive and despite the smaller caches, the DPU can cache the atomic counter.

8.6.3 Discussion

Based on our experiments, the sequencer shows itself as an interesting use case to offload to the DPU, especially when utilizing one-sided atomic accesses. The closer proximity of the memory of the DPU to the network in comparison to the host results in a 50% throughput increase. While one-sided accesses achieve worse throughput than an RPC implementation, they are still often applied due to the fact that no CPU load is introduced on the storage server and that a throughput of 2 MOps is sufficient for many use cases [124].

8.7 Conclusion and Future Work

We now conclude our findings by summarizing in which scenarios the BlueField DPUs show an acceleration potential and in which scenarios they do not. Moreover, we discuss some avenues for future work.

Summary. The BlueField-2 DPU is naturally no replacement for a general server-grade CPU and memory. It is instead marketed as an accelerator for networking-related tasks, due to its set of network-oriented hardware accelerators. The majority of these accelerators are however not applicable to our evaluated use cases as they are mostly concerned with security tasks or virtualization.

In conclusion, we evaluated the BlueField DPUs with respect to typical DBMS tasks such as remote B-trees or a global sequencer. The main findings are that an acceleration potential exists for one-sided accesses both in terms of latency and throughput whereas two-sided accesses easily overload the DPU. However, using the DPU in combination with the host CPU of the storage server can yield promising performance benefits provided that the workload is carefully distributed with regard to the relative performance difference.

Future Work. We see that BlueField DPUs are under active development with a new generation BlueField-3 around the corner which shows even more and faster CPU cores and memory. We speculate that the BlueField-3 is therefore a promising platform for also accelerating two-sided data-intensive tasks.

Moreover, we did not cover tasks where BlueField cards provide ASIC-based accelerators such as engines for compression/decompression, encryption/decryption, regex pattern matching and NVMe. While we did not cover such engines in our evaluation, they are also interesting for DBMS workloads and are thus worth to be studied in future work.

Finally, while B-trees and global sequencers are important and interesting use cases for DBMS, there are many other use cases that are worth studying in the future. For example, if the DPUs become more powerful, we can think of implementing a full storage engine on the DPU including a concurrency scheme implementation that allows clients to access and modify data concurrently while coordinating these accesses.

8.8 Acknowledgements

We thank the reviewers for their feedback. This work was partially funded by the German Research Foundation (DFG) under the grants BI2011/1 & BI2011/2 (DFG priority program 2037) and the DFG Collaborative Research Center 1053 (MAKI). Finally, we want to thank hessian.AI at TU Darmstadt as well as DFKI Darmstadt for the support.

9 High-Performance In-Network Data Processing

Abstract

Recent research has shown the potential for using programmable network components such as switches for distributed data processing. Opportunities include in-network caching and the execution of distributed SQL operations such as joins or aggregations. However, a major weakness of the current generation of programmable switches is that the hardware still has many limitations not only with regard to what type of operations are supported in a switch (e.g., no loops), but also that the switches can often not sustain processing at line-rate.

As a first contribution of this paper, we propose a new switch architecture that can be employed as an in-network co-processor for analytical SQL workloads. Different from existing commercial switches, our switch architecture is based on an FPGA design and supports complex operations at line-rate. As a second contribution, we discuss how a typical distributed database architecture has to be changed to efficiently leverage the new switch architecture. In our evaluation we show that our new switch architecture can significantly speed-up distributed query processing by up to 7× compared to traditional shuffle-based approaches without in-network processing capabilities.

Bibliographic Information

The content of this chapter was previously published in the peer-reviewed work: Jaco A. Hofmann, Lasse Thostrup, Tobias Ziegler, Carsten Binnig, and Andreas Koch. "High-Performance In-Network Data Processing." In: *10th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage*

Architectures, ADMS@VLDB 2019, Los Angeles, California, USA, August 26, 2019. Ed. by Rajesh Bordawekar and Tirthankar Lahiri. 2019, pp. 64–73. URL: http://www.adms-conf.org/2019-camera-ready/hofmann_adms19.pdf.

9.1 Introduction

Motivation. Scalable database systems for analytical workloads such as Terradata, Microsoft Parallel Data Warehouse, or Amazon's Redshift are being used today for analyzing massive amounts of data in distributed setups. These systems exploit the sheer amount of nodes to leverage data parallelism by shuffling data back and forth. While this paradigm is quite successful, recent papers [111, 112] have shown that data parallelism in a distributed setup does not necessarily lead to improved performance especially in modern main memory databases due to high communication costs or inefficient utilization of the network.

Therefore, many recent papers have suggested to improve the performance of distributed databases by optimizing their network usage with the help of high-speed networks and RDMA [147, 155]. While RDMA allows to leverage high network bandwidth and low latency in database systems, it is not the only option of modern network technologies that can be leveraged by distributed data processing systems. An interesting direction is that network components such as switches are becoming programmable and thus allow to offload processing into the network itself.

This opens up many possibilities for tailoring the network stack to data processing, ranging from opportunities such as in-network caching to the execution of distributed SQL operations inside network components, i.e., in-network processing (INP) [12, 36, 116]. However, especially for in-network processing, a major weakness of the current generation of programmable switches is that the hardware still cannot sustain processing on line-rate and is not capable of many memory intensive operations, such as the computation of SQL joins or aggregations [12, 79] which need to keep an intermediate state.

Contribution. In order to address this challenge, we make the following contributions. First, we propose a new switch architecture that can be used as an in-network co-processor for analytical SQL workloads. The switch architecture relies on a system on a chip (SoC) design that leverages an FPGA and provides larger amount of DDR3 main memory in the switch to execute query pipelines, i.e., sequences of multiple SQL operators. By using this architecture we thus cannot only process data at line-rate, but also support more memory intensive operations inside the switch, such as hash-table building and probing.

Second, we show how a typical distributed database architecture can be adopted to efficiently use our new switch architecture. The main idea is that the database comes with a set of pre-compiled query pipelines that are installed in the FPGA. During query compilation the appropriate pipeline is chosen. Furthermore, the query optimizer

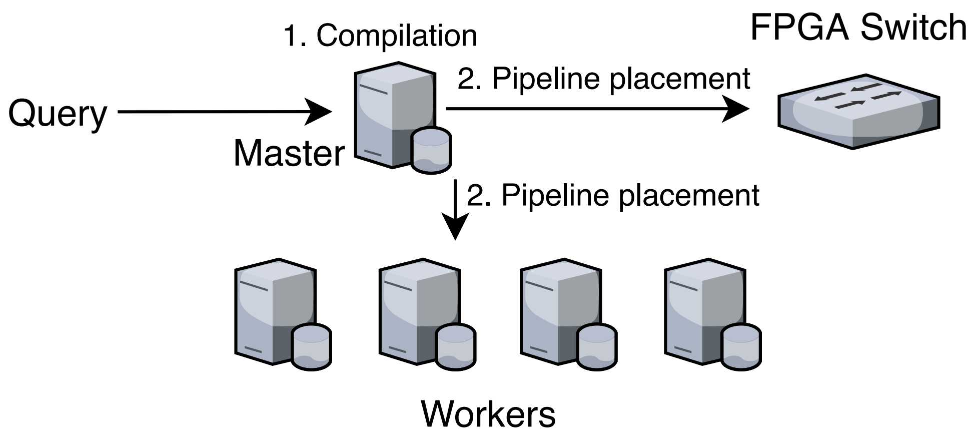


Figure 9.1: Overview of our Processing Scheme.

is extended to determine the best execution strategy of leveraging the in-network co-processor.

Finally, in our experimental evaluation we show that our optimized query processing scheme which utilizes the new switch architecture significantly speeds-up distributed join processing by up to 7×, compared to a traditional shuffle-based approach without INP. In contrast to traditional query processing, our INP-enabled scheme can eliminate the overhead of shuffling intermediate results and thus reduces the communication between nodes in a distributed database.

Outline. The remainder of this paper is structured as follows. In Section 9.2 we first give an overview of our in-network processing (INP) architecture for analytical SQL workloads and how it can be integrated into a distributed database. Afterwards, we present how query processing and optimization has to change to push SQL operations into the switch and then discuss the design details of our switch architecture in Section 9.4. Finally, we report initial experimental results in Section 9.5 and conclude with the limitations of our current prototype and a discussion of future directions in Section 9.6.

9.2 System Overview

This section provides an overview of our proposed INP-based query processing scheme and discusses its main differences compared to traditional distributed query processing.

Query Processing. To illustrate the main idea of our distributed query processing that utilizes our proposed switch design, we first review classical distributed query execution and then discuss the changes compared to our scheme. A typical setup of a shared-nothing database consists of one master and several compute nodes, as well as one switch connecting the nodes. As an example query, consider the execution plan in Figure 9.2 that could result from the SQL statement `SELECT * FROM A JOIN B JOIN C`.

In the classical distributed query processing, `A` and `B` are first shuffled according to the join key. Then, each node builds a hash table over `B` (assuming `B` is the smaller table) and uses tuples from `A` to probe in that hash table. For the subsequent join, the intermediate result of $A \bowtie B$ as well as relation `C` need to be shuffled again, such that the joins can be executed by building and probing into the hash table of `C`. Thus, each join (if data is not co-partitioned) typically requires one expensive shuffle operation.

In order to avoid the repeated shuffling of data, we propose a new execution scheme that utilizes our proposed switch for in-network processing. Figure 9.1 shows the two main steps of our scheme. As a first step, the master node determines and compiles an optimal execution plan when a new SQL query arrives (Figure 9.1 ①). Next, the master node places the different steps of the plan on the worker nodes as well as the switch (Figure 9.1 ②). How an optimal execution plan for our switch architecture is created and how the pipeline placement is performed will be described in more detail in the next section.

The equivalent query plan for INP execution is shown on Figure 9.3. For now, assume that the plan is the optimal one for our example query. As can be seen in the two figures, the classical - Figure 9.2 and the INP-based plan Figure 9.3 consist of two types of pipelines (probe-pipelines and build-pipelines), the INP-based plan splits the plan into multiple pipelines that can be placed on worker nodes or the switch respectively. As a consequence, different from the traditional plan many of the `Shuffle` operators can be completely avoided since the probe steps are executed all in the switch. In the following we will discuss the implications of the differences in more detail.

Discussion. As mentioned before, the main conceptual difference of our scheme is the elimination of shuffling, and in particular the re-shuffling of intermediate join results. This is beneficial, since shuffling comes with several challenges.

First, shuffling operations are so called pipeline breakers, since the streaming of tuples through an operator pipeline is stopped (i.e., the shuffle operation only starts once the previous intermediate result has been materialized completely). This however, limits the degree of parallelism of the execution since following phases of a query need to wait for the completion of previous ones. For instance, the second join of our example query can not be computed until the result of the first join has been materialized.

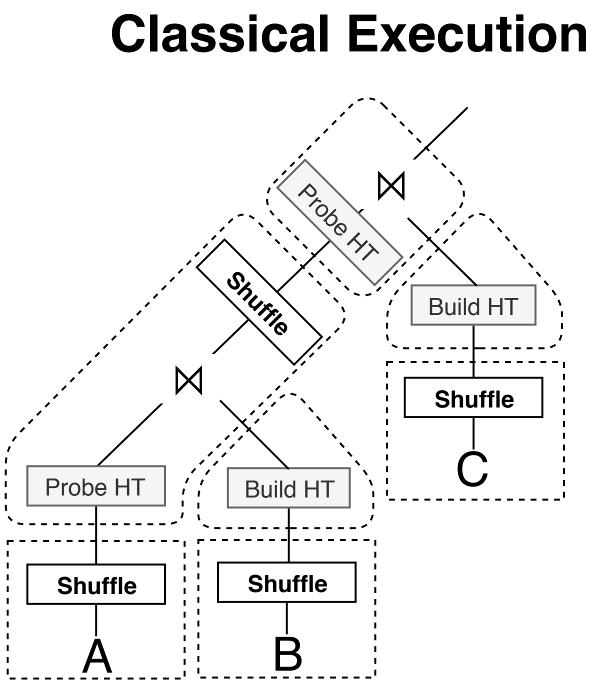


Figure 9.2: Example of Query Plan for Classical Execution.

Second, shuffling usually means that significant amounts of data need to be transferred via the network, since also the intermediate results need to be partitioned and sent to all workers. The cost of shuffling intermediate results is even higher in a data warehouse setup. This is because, in a star schema with one very big fact table and multiple smaller dimension tables that need to be joined, the cost of shuffling the fact table and the resulting intermediate results is dominating the overall query execution cost. Considering the example query plan shown in Figure 9.2, the fact table could be represented by relation `A` and the dimension tables by `B` and `C`.

Finally, we are less sensitive to skew, since typically one node receives more data than the others. When multiple nodes send to a single node, the network link of the node gets congested and slows down overall execution (also known as incast problem).

9.3 Query Processing

In this section we describe how the database architecture can be adapted to leverage the FPGA-based switch.

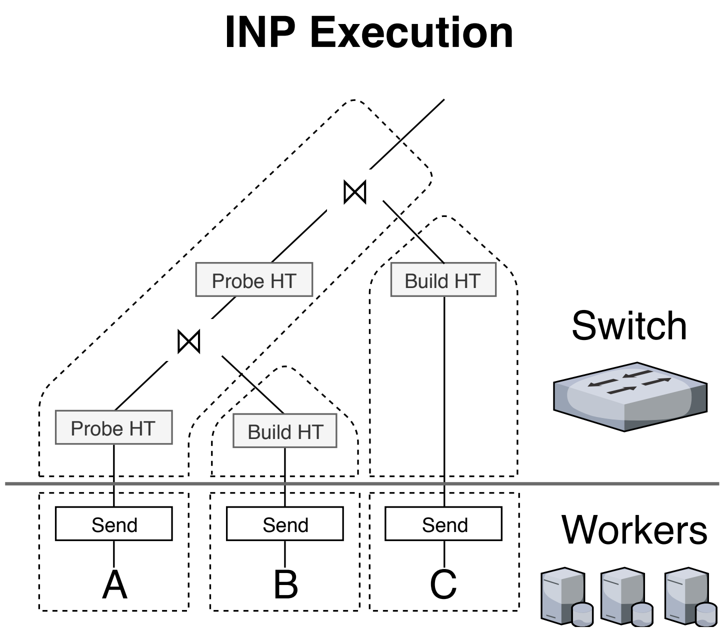


Figure 9.3: Example of Query Plan for INP Execution with Operator Placement.

9.3.1 Query Compilation

The query compilation resembles a physical execution plan for a given query. One important decision in distributed systems during optimization is *where* to execute pipelines optimally. Consequently, our adapted query compilation takes the FPGA switch as a processing unit into account. When employing an FPGA for query processing, it is not feasible to synthesize a complete configuration (a so called bitstream, i.e., the executable logic on the FPGA) on a query-to-query basis, as bitstream generation can take multiple hours. However, once the bitstream is generated and installed on the switch FPGA, re-configuring the switch to use a different pre-installed bitstream only takes a few milliseconds. Hence, our system allows to install a set of bitstreams for pre-generated pipelines to execute multiple different queries efficiently.

Figure 9.3 shows a physical operator plan for our example query. As shown, each worker is only responsible to send its part of the relation to the switch, which executes the main query pipelines, i.e., building and probing pipelines for executing joins. To support generic queries inside the switch, the pre-generated pipelines provide different signatures. For instance, the intermediate hash table for table `B` needs to store keys and values of 8 Bytes, whereas table `C` needs 4 Byte keys and 10 Byte values.

The master node thus tries to choose the best fitting signature, if there is no exact match it takes the next larger one. This clearly induces memory overhead, e.g., if the relation has a 64 Byte value, then the master chooses the 128 Byte pipelines. However, this should not be a common case, since optimal signatures can be generated as soon as the workload is known.

9.3.2 Query Optimization

In a traditional database system, the optimizer is responsible for finding the best plan. INP has a slightly different execution model and mandates an extension to the existing cost-based optimization. We therefore propose the following optimization objective: The optimizer should reduce the number of re-shuffle operations by offloading computation to the switch. These pipeline-breaking operations are especially expensive, since they require synchronization until the query execution proceeds. Moreover, the limited memory of the switch has to be taken into account. Therefore INP should be applied for the most beneficial joins and the optimization problem is in fact a constrained optimization problem.

In our prototypical implementation we only consider left-deep join trees with primary-foreign relations as in Figure 9.2 for INP, since this is a common join structure in analytical workloads.

In the following, we first explain our notation and then derive our cost model. We are considering a left-deep plan which consists of a left-deepest relation L (L might be an input relation or an intermediate result) and a set of tables T_i which are joined with L. The number of workers is defined by N. $|R|$ denotes the cardinality of any relation R and $ts(R)$ the size of a tuple in R. The subset of tables indexed by I which are qualified for INP is defined as $L \bigcup_{i=1}^{I} T_i$. For instance the query plan shown in Figure 9.2 consists of the tables A, B, C. However, due to high memory requirements, or lower costs of a shuffle-based join, the optimizer could define I such that only A and C are joined with INP. Consequently, the intermediate result of $A \bowtie C$ would be joined with B with the shuffle-based approach.

Cost Model for Classical Model. Based on the network cost of one relation, we describe the network cost for the classical approach used in distributed databases which is based on data shuffling. Later, we derive a new cost model for our INP-based query processing scheme. The following equation describes the cost for sending the qualifying tuples of one relation over the network:

$$c_{rel}(R) = |R| * ts(R) \tag{9.1}$$

In the classical cost model, the cost to create an intermediate join of the tables indexed by I is thus given by the following equation:

$$\begin{aligned} c_{shuffle}(I) = \frac{N-1}{N}\Big(&c_{rel}(A) + c_{rel}(A \bowtie T_{i1}) + \dots \\ &+ c_{rel}(A \bowtie T_{i1} \bowtie \dots \bowtie T_{im-1}) + \sum_{i \in I} c_{rel}(T_i)\Big) \end{aligned} \tag{9.2}$$

To explain the above equation we calculate the cost for the plan shown in Figure 9.2, with four workers, and the following parameters.

Relation	Size ($\|R\|$)	Tuple Size (ts(R))	$c_{rel}(R)$
A	100000	32	3200000
B	10000	10	100000
C	10000	10	100000
AB	100000	32	3200000

We consider the complete plan in our example - the intermediate result consists of A, B, C, therefore the costs of A, B, C can be included in the Equation (9.3). These describe the cost for shuffling A, B, C. Besides these input relations, we also shuffle the intermediate result of $A \bowtie B$. Hence, we include those costs as well, resulting in:

$$c_{shuffle}(I) = \frac{N-1}{N}(3.2e06 + 3.2e06 + 1e05 + 1e05) \tag{9.3}$$

Finally, we assume that only $\frac{3}{4}$ of the data is shuffled in the uniform case w/o skew and $\frac{1}{4}$ is kept locally. Consequently, the network cost for the given plan is $\frac{3}{4} * 6.6e06$.

Cost Model for INP. Now, the INP cost model can be derived from the classical model. The major change is to avoid re-shuffling of intermediates and thus they are removed from the equation. Hence, the following equation holds:

$$c_{INP}(I) = \left(c_{rel}(L) + \sum_{i \in I} c_{rel}(T_i))\right) \tag{9.4}$$

As shown in the equation, the network cost of the INP approach is determined only by the tables and tuples sizes and not by the intermediate results. We again apply this cost function to the example above.

$$c_{INP}(I) = (3.2e06 + 1e05 + 1e05) \tag{9.5}$$

This gives the following network costs for the INP approach $3.40e06$.

Hence, the cost improvement of the classical approach vs. the INP based approach is $c_{INP}(I) - c_{shuffle}(I)$ depending on the set of tables used to apply INP. The optimization problem is now to choose $I \subset \{1, 2, \ldots, n\}$ such that this improvement is maximized while the memory constraint in the switch $\sum_{i \in I} |T_i| \leq C_{RAM}$ is satisfied. Note that also $I = \emptyset$ is considered in this optimization problem, i.e. INP is not applied at all.

Theoretical Analysis:. This paragraph discusses the previously introduced cost-models and elaborates when INP is beneficial. Note, however INP is not a complete replacement of the traditional approach, but rather an optimization which can be used for some cases. We first show the effect of the network cost $c_{rel}(A)$ from A in relation to the other tables T_i. Based on Equation (9.1) the cardinality and the size of the tuples have an impact on the network cost of A. To analyze the effect of increasing costs of the left deepest relation we again use our query plan from Figure 9.2 with four workers.

Figure 9.4 shows on the x-axis the network cost of $c_{rel}(A)$ in relation to B, C, i.e., 0.1 means that the cost of the relation A is only 10 percent of B, C. To show the effect in isolation we thus fix B, C to the same size. The y-axis shows the outcome of the cost functions for $c_{shuffle}(A, B, C)$ and $c_{INP}(A, B, C)$.

The plot shows that if the costs for A is smaller than the cost for B, C, the classical approach is more suitable. Since the intermediate results will be cheaper, shuffling only a fraction of our relation ($\frac{N-1}{N}$) is cheaper than sending everything to the switch. However, INP is more efficient if the cost of A exceeds B, C, thus we avoid expensive reshuffling on intermediate results.

The best performing strategy is not only determined by the costs of A, B, C. The number of joins also influences the decision. Therefore, the next plot analyzes the effect of the number of joins when A and T_i have equal costs. Figure 9.5 shows on the x-axis the number of joins and the y-axis shows again the costs of the two strategies.

By avoiding the intermediate shuffle, the INP approach is clearly beneficial if the number of joins increases. In conclusion we have shown that the INP approach is beneficial if the cost of A is high compared to the other relations and further if the number of joins is high. Both of these circumstances are often met in data warehouse scenarios.

Cost-Model Extensions. This paragraph extends the proposed cost-model to support selections, co-partitioning, and skew. *Selections* can be modeled by multiplying $c_{rel}(R)$

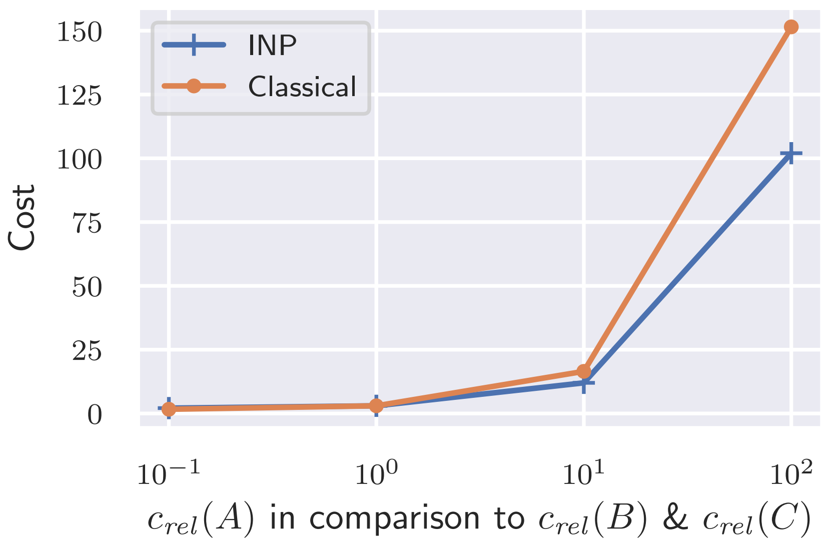


Figure 9.4: Cost analysis for different table ratios. With bigger relation `A` in comparison to `B` & `C`, the INP approach greatly reduces cost.

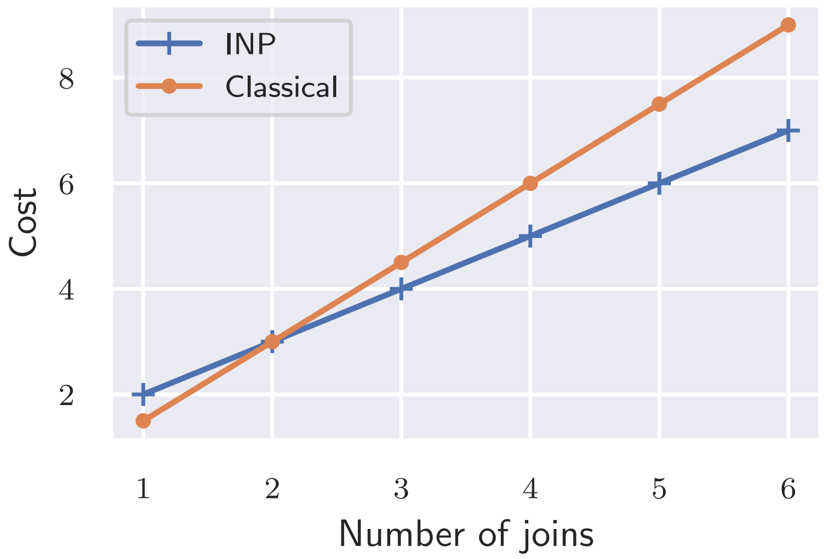


Figure 9.5: Cost Analysis for varying number of joins. Relation costs (c_{rel}) are kept the same for all joined relations.

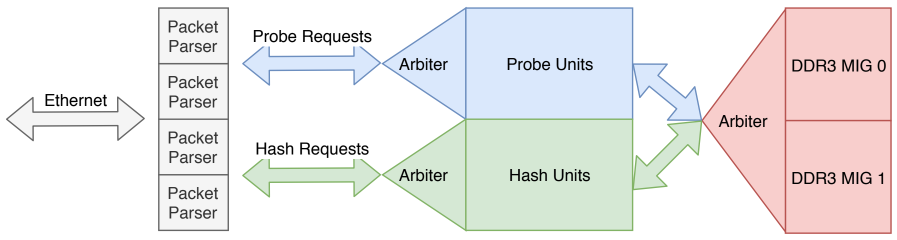


Figure 9.6: Overview of the proposed architecture on the NetFPGA SUME board. Data is processed as a stream of 64 bit words provided by the Xilinx 10G Ethernet Subsystem. The Ethernet packets are parsed using a Bluespec-generated packet parser. The extracted hashing and probing requests are forwarded to the hashing and probing infrastructure.

with the selectivity. This not only allows to support selection predicates but other types of joins as well. *Co-partitioning* is only relevant to the classical approach and is modeled trivially by removing the relations from the equation. Assuming that A, B are co-located in our example query, they can be joined locally and there is no need to shuffle A, B. However, it is still necessary to shuffle $A \bowtie B$ and C. In the INP approach, all tables need to be sent to the switch, and consequently co-partitioning has no effect on the costs. *Skew* is modeled by applying a write amplification factor to the costs of $c_{shuffle}(I)$. Furthermore, skew often leads to the incast problem; i.e., the ingoing link on the worker receiving the large amount of skewed data becomes congested and acts as a bottleneck. Hence, the write-amplification factor models that skew since it increases the shuffle cost. In contrast, the INP approach is not affected by skew since in the best case data does not need to be shuffled at all.

9.4 Switch Design

In the following, we describe the design of our switch architecture that can be optimally used as an in-network co-processor for typical analytical SQL workloads. We first explain the hardware platform our switch is based on before we explain how different query pipelines for hash table building and probing are supported inside our switch architecture.

9.4.1 Hardware Platform

The platform chosen for the demonstrator is the NetFPGA SUME [158], based around a Xilinx Virtex 7 FPGA, $8GB$ of DDR3-SDRAM memory and four SFP+ connectors. Those connectors can be used to interface the FPGA with off-the-shelf SFP+ solutions common in data centers, either via fiber optics or direct-attached cables. The bandwidth of all ports is $10Gbps$.

The switching hardware itself is described in Bluespec SystemVerilog, a Hardware Description Language (HDL) that combines high-level features of functional programming languages with the performance of hand-crafted low-level HDLs such as SystemVerilog or VHDL.

This section describes the different stages of packet processing in the proposed architecture: (1) Packet Parsing, (2) Hash Table Generation, and (3) Hash Table Probing.

For designing our switch architecture we leverage the TaPaSCo [72] tool chain. The tool chain provides all necessary steps to bring hardware acceleration to a variety of platforms, in many cases avoiding the need for explicit hardware development knowledge. The tools assist with all necessary steps such as bitstream generation, bitstream loading and interfacing to a host computer for control and monitoring tasks. TaPaSCo assists the designer finding the optimal working conditions for a given architecture. Furthermore, TaPaSCo has already been employed succesfully in other in-network-processing applications [25].

9.4.2 Ethernet Packet Parsing On FPGA

The interface to the SFP+ connectors is provided by Xilinx through their 10 Gigabit Ethernet Subsystem IP core. The packets received over SFP+ are provided by the core as a stream of 64 bit words at $156.25MHz$.

The streams of all four interfaces are collected in their corresponding packet parser infrastructure in the proposed architecture, as shown in Figure 9.6. These packet parser units can operate at line rate and are completely independent from each other. The packet stream is parsed using a custom parsing state machine generator. Common functionality such as dropping packets for the wrong destination MAC address or with the wrong protocol is done on the fly as soon as the relevant data is available.

Requests, in our case the probe or hashing modes, are immediately forwarded out of the parsing module for further processing. These taps into the parsing pipeline can occur at any processing step and provide very flexible protocol handling. Previous parts of the packet can also be examined in later steps by explicitly marking certain parts of the packet as relevant for later processing.

The extracted requests are then stored in FIFO buffers to be collected by the hashing and probing infrastructure for actual processing.

9.4.3 Hash Table Generation

The example application demands very high hashing performance from the system. Each of the 10Gb ports results in over 3.76e07 hash table inserts per second. The architecture should support this throughput for all pipelines in parallel. Latency, on the other hand, is not important as an insert does not result in any feedback to the sender. Accordingly, the sender will simply send out the pipelines without waiting for ACK signals or similar. The hash table generation architecture supports only insert operations, as deletes are not required, and probes are handled by a different part of the design.

Key Value Stores on FPGA are a well-researched field [127, 139]. These approaches usually differ from CPU based hash table implementations as FPGAs have different strength and weaknesses. For example, the FPGA can process wider words, such as 512 bit whereas a CPU based implementation has to consider the caching infrastructure of the given processor.

The design proposed here works in the following way:

1. Hash the key provided in the request by the parsing stage to calculate a bucket.
2. Retrieve the corresponding bucket from the main memory. All buckets are 512 bit in size, which corresponds to the data path width of the DDR3 controllers.
3. Place the key and value tuple in the first free position in the bucket and write the bucket back to main memory.

All of these stages are pipelined and multiple requests are processed at any time. The pipelining might result in Read-after-write hazards which are dealt with by the look-ahead buffers. These buffers inject the answer read from the memory, including the newly added tuple, whenever a succeeding insert request hashes to the same bucket.

To spread the hashing load over memory as much as possible an interleaved approach is used. Each of the hashing units uses all of the available memory but only every third entry belongs to a certain unit. For example address 0 stores bucket zero of hash table zero, address 64 stores bucket zero of hash table one and address 128 stores bucket zero of hash table two. This scheme allows for much better utilization of all the available memory resources compared to a simple block-wise arrangement.

The performance of this architecture is completely determined by the random access Read/Write performance of the DDR3 controllers and is typically around 5.63e07 inserts per second per memory controller. Higher performance can be reached by utilizing newer devices with memories such as HBM having higher random access speed.

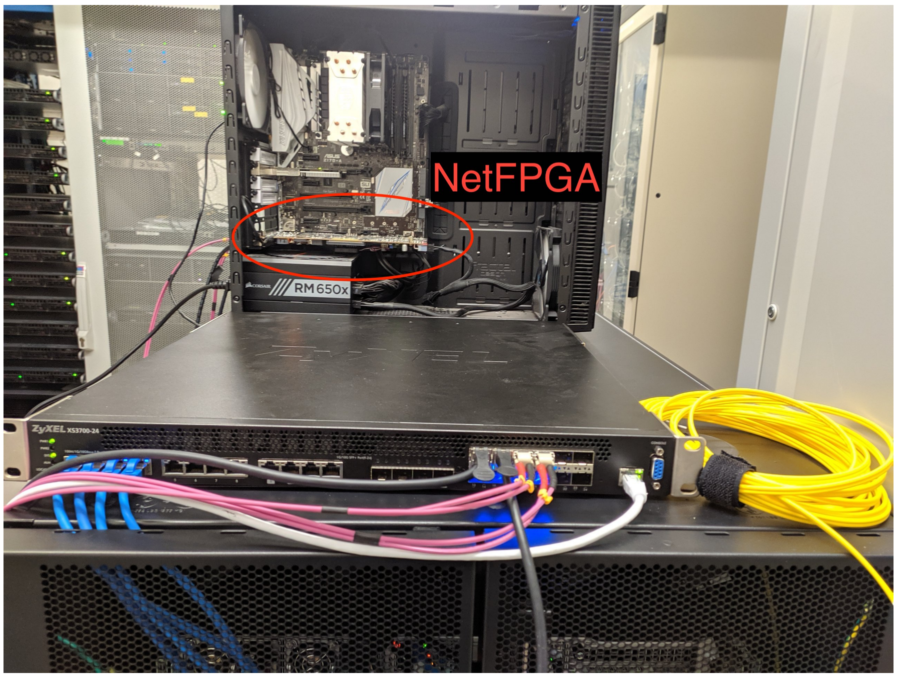


Figure 9.7: Setup used to evaluate the proposed architecture. Four nodes with Xeon 5120 CPUs are connected via 10*Gbps* links to a central Zyxel XS3700 switch. The NetFPGA SUME-based switch is connected with two SFP+ Fiber and two Direct Attached cables. The FPGA is installed in a host PC for simplified monitoring via PCIe. The host only performs management tasks, and thus does not process any packets on its own.

9.4.4 Hash Table Probing

Compared to the table generation, probing has to meet even higher performance requirements. For every probing request, all stored hash tables have to be probed in parallel. For one $10Gbps$ link and three hash tables this combines to about $1.20e08$ requests per second in total.

The architecture itself closely resembles the design of the insert units but without the write-back part. Every lookup requires three steps:

1. Hash the key provided in the request by the parsing stage to calculate a bucket.
2. Retrieve the corresponding bucket from the main memory.
3. Return the key contained in the bucket, or an invalid flag if the key is not found.

Again the performance is completely determined by the performance of the DDR3 controller. Considering that only reads, and no writes, are necessary the performance is about 75 % better at around $9.84e07$ probes per second per memory controller.

The results of the probes are forwarded to a combination unit which is responsible for answering the probe requests.

The retrieved tuples are combined with the original request and forwarded to the SFP+ parsing units which in turn send out the completed requests to their destination.

9.4.5 Performance

The architecture is shown to be able to handle packet processing at line rate and even multiple channels in parallel. The hashing mechanism is able to handle around $29.9Gbps$ of traffic and is only limited by the available random access performance of the memory controllers. The probe units are able to process up to $1.97e08$ tuples per second using two DDR3 controllers.

9.5 Initial Results

In the following, we show the initial results of our new switch design in a distributed database.

9.5.1 Setup and Workload

Our experiments were executed on a five node cluster – one master node and four compute nodes. Each server has an Intel(R) Xeon(R) Gold 5120 CPU @ 2,20GHz processor and

384GB RAM, running Ubuntu 18.04. All machines were equipped with a 10*Gbps* NIC. The four compute nodes are connected via CAT 6 RJ45 Ethernet Cables to a Zyxel XS3700 switch (without INP) and our FPGA-based switch (with INP capabilities). The FPGA switch is attached to our compute nodes using two SFP+ DAC cables from Digitus and two SFP+ fiber transceivers by *FLEXOPTIC*. A picture of our experimental setup is shown in Figure 9.7.

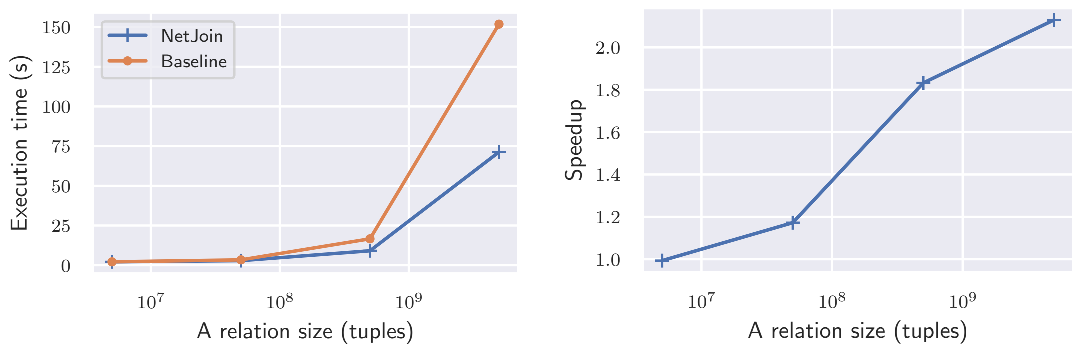
Based on this setup, multiple experiments were conducted to demonstrate the performance of our proposed architecture (referred to as *NetJoin*) over a baseline without INP. The experiment represents a shuffle-heavy scenario like described in Section 9.2. A table `A` is joined together with three other tables `B`, `C` & `D`. The join shows similarity to a data warehouse setup with `A` being the fact table with foreign keys to the dimension tables `B`, `C` & `D`.

In our setup, all tables are pre-partitioned such that no join partners can be found locally without the need of sending one of the tuples over the network. In data warehousing, typically one of the dimensional tables in a star schema can be co-partitioned with the fact table and thus shuffling can be avoided for the first join in a plan where the co-partitioned dimension table is joined. However, this optimization can be applied for both processing schemes; the traditional shuffle-based and our INP-based scheme and both would benefit equally. In our experiments, we thus show the performance of both schemes without this optimization.

Finally, a last important fact is that the *NetJoin* makes use of raw Ethernet frames, such that higher level protocols do not split up packets and no performance overhead is introduced. In our paper (as well as other INP papers such as [79]), we do not yet handle dropped frames. Instead, throughout the experiments we made sure that the amount of dropped tuples are monitored and limited to at most 2%. For providing reliability, a light protocol could be implemented on top of raw Ethernet which does not introduce a significant performance penalty.

9.5.2 Experiment 1: Uniform Join Keys

The first experiment shown in Figure 9.8 scales the size of the `A` relation in comparison to relations `B`, `C` & `D`. The left graph (a) shows the runtime of the distributed hash join over the varying sizes of the `A` relation. The `B`, `C` & `D` relations sizes are $5e07$ tuples, and with the `A` relation ranging from $5e06$ to $5e09$ tuples. Since the join keys are uniform, each of the four nodes receive the same amount of tuples when shuffling the relations.



(a) Join runtime.

(b) Speedup of *NetJoin* over baseline.

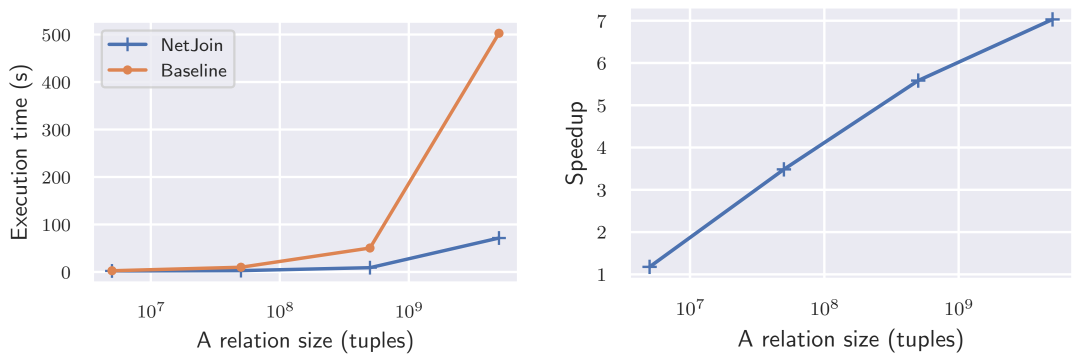
Figure 9.8: Experiment 1. Four nodes joining ranging `A` relation sizes (5*e*06 to 5*e*09 tuples) with fixed `B`, `C` & `D` relations (5*e*07 tuples). Link speed on each node at 5*Gbps*. For small sizes of A, the shuffling overhead is too small to make a significant impact on runtime. Larger sizes of A, compared to relations `B`, `C` & `D`, result in increased overhead due to the required shuffling. Accordingly, *NetJoin* is increasingly faster compared to the baseline as it does not need to shuffle the tables.

The results show that as the `A` relation size is small, the *NetJoin* does not perform better than the baseline since reshuffling the intermediate results is inexpensive due to `A`'s small size. As the `A` relation size grows, the *NetJoin* outperforms the baseline. Even though the nodes in the *NetJoin* have to completely send their local partitions of all relations to the switch, the reduced cost of reshuffling compensates for this. The speedup as shown in Figure 9.8b shows an over 2× performance gain against the baseline with the `A` relation 100× bigger than `B`, `C` & `D`.

9.5.3 Experiment 2: Skewed Join Keys

To show that our in-network execution scheme is more resilient against skew compared to a traditional query processing, we generated skewed join keys. The skew was such that when shuffling the relations on four nodes, 80% of all tuples go to Node 1, 13% to Node 2, 5% to Node 3 and the remaining 2% to Node 4.

Since the initial prototype only uses raw Ethernet frames, conducting this experiment for the baseline meant that Node 2, 3 & 4 needed to throttle the speed of outgoing packets to Node 1. This ensured packets were not being dropped due to congested in-going link to Node 1.



(a) Join runtime. (b) Speedup of *NetJoin* over baseline.

Figure 9.9: Experiment 2. Shuffle skew on four nodes joining ranging `A` relation sizes (5*e*06 to 5*e*09 tuples) with fixed `B`, `C` & `D` relations size (5*e*07 tuples). Link speed on each node at 5*Gbps*. *NetJoin* executes with same runtimes as compared to the unskewed scenario shown in Figure 9.8a. The baseline however is impacted by the bottleneck resulting from the non-uniformly distributed join keys.

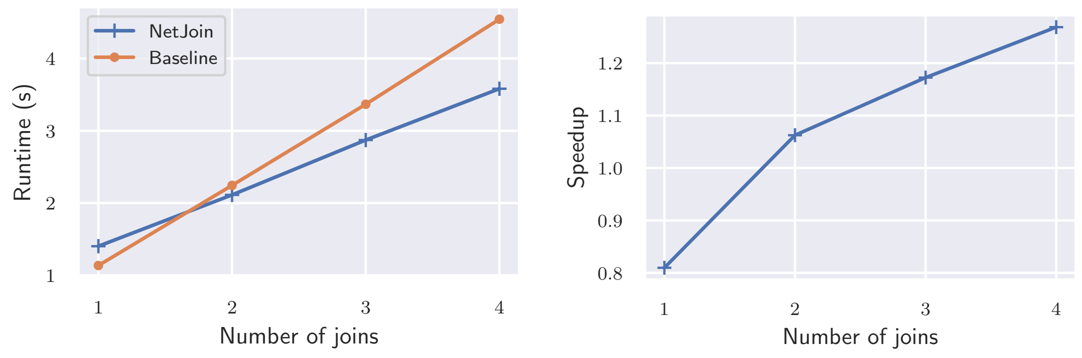
As shown in Figure 9.9a such skewed shuffling scenario heavily affects the performance of a distributed join, not only because the compute intensity and memory consumption are not equally distributed, but also because of incast congestion in the network switch. Since Node 2, 3 & 4 all need to send 80% of their local relation to Node 1, the in-going link is acting as a bottleneck and other nodes throttle down their sending rate.

However, with our *NetJoin*, skew on the join key does not play a role since no network shuffling is taking place. Figure 9.9a shows an identical runtime of the *NetJoin*, but with the baseline performance severely suffering in comparison to Figure 9.8a. The speedup shown on Figure 9.9b reports a speedup of 7× for the largest `A` relation size.

9.5.4 Experiment 3: Scaling Number of Joins

As discussed in Section 9.3, not only the relation size of the left deepest relation, but also the number of joined relations has an impact on the query runtime. In this experiment we show that our proposed approach is superior to the classical query processing when the number of joins increases. This is also true for the sub-optimal case when the size of the left deepest relation is not larger than the other relations (i.e., the fact table and dimension tables have the same sizes).

The experiment is executed by fixing all relation sizes to 5*e*07 tuples. Figure 9.10a shows the runtime of queries with 1 to 4 joins. As already shown in the cost analysis



(a) Join runtime. (b) Speedup of *NetJoin* over baseline.

Figure 9.10: Experiment 3. Scaling number of executed joins in query from 1 to 4. All relation sizes are fixed to 5e07 tuples. *NetJoin* is slower for 1 join since the complete relations are sent to switch. With more joins *NetJoin* outperform the baseline through not having to shuffle intermediate joined relations. With relation `A` bigger than joined relations, the speedup increases further as demonstrated in Experiment 1.

in Section 9.3 (Figure 9.5), the baseline cost increase with a higher gradient than the cost of *NetJoin* (INP). Moreover, we can see that only after two joins the INP-based approach outperforms the classical approach.

Additionally, we also conducted an experiment where the fact table is larger than the dimension tables. In this case, the INP-based approach again outperforms the classical shuffle-based approach by a higher factor.

9.6 Conclusion & Future Work

This work is motivated by the observation that existing programmable switches cannot process memory intensive operations and thus are not suited for distributed query processing.

To overcome this issue, we proposed a new FPGA-based switch architecture. By using an FPGA-based design we are flexible to process different incoming queries at line-rate. The high-level FPGA programming paradigm used here provides considerably more flexibility than existing solutions, such as P4's match+action stages.

Furthermore, we also discuss initial directions where to adapt distributed query processing to leverage the capabilities of INP. The main idea is to avoid expensive shuffling operations by offloading more complex query pipelines to the switch. We show that our proposed execution scheme can thus speed up query processing for left-deep join plans by up to 7×.

However, our prototype has also shown some limitations that we could not yet address. We leave these limitations for future work. One of the limitation of the current design is that the output of a probing pipeline cannot be larger than its input due to congestion. By using a more recent FPGA board to realize the switch, up to 256 GB of DDR4-SDRAM plus 8 GB of very fast HBM will be available, thus allowing the implementation of a better caching scheme. This cache could allow us to better control the congestion of the outgoing link by buffering/stalling some tuples before sending out.

Finally, while the initial results of our prototype are promising, there are many other open routes for future work that we have not been able to address yet. For example, in a real data-center setup multiple switches are involved. To that end, we could use parallelism in the network by using multiple of these switches. Furthermore, another direction would be fault-tolerance or isolation of multiple queries which are all not yet handled in our current design.

9.7 Acknowledgements

This work was partially funded by the German Research Foundation (DFG) under the grants BI2011/1 (DFG priority program 2037) as well as the DFG Collaborative Research Center 1053 (MAKI).

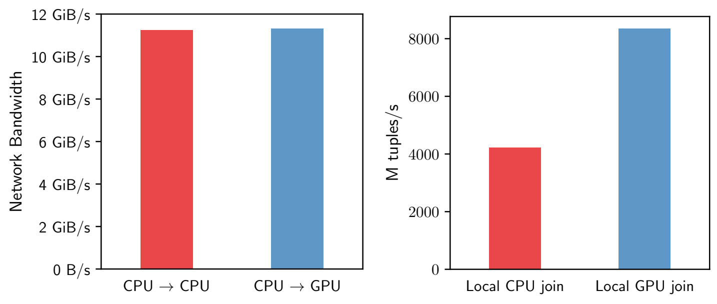
10 Distributed GPU Joins on Fast RDMA-capable Networks

Abstract

In this paper, we present a *novel pipelined GPU join* that accelerates the performance of distributed DBMSs by leveraging GPU resources on fast networks. A key insight is that we enable pipelined join execution by overlapping the network shuffling with the build and probe phases, thereby significantly reducing the GPU idle time. To demonstrate this, we propose novel algorithms for distributed pipelined GPU joins with RDMA and GPUDirect for both arbitrarily large probe- and build-side tables. In our evaluation, we show our pipelined distributed GPU join can reduce the overall runtime of a full query by up to 6× against a state-of-the-art CPU-only join.

Bibliographic Information

The content of this chapter was previously published in the peer-reviewed work: Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. "Distributed GPU Joins on Fast RDMA-capable Networks." In: *Proc. ACM Manag. Data* 1.1 (2023), 29:1–29:26. DOI: 10.1145/3588709. URL: https://doi.org/10.1145/3588709.



(a) Network bandwidth (b) Join throughput

Figure 10.1: GPUs represent an interesting accelerator for distributed joins in scale-out DBMSs because of the following reasons. (a) Network communication to remote CPUs and GPUs share the same characteristics (reported by *ib_write_bw*). (b) With the same overhead for data transfers, GPU joins can significantly outperform CPU joins based on their higher processing power.[2]

10.1 Introduction

Motivation. GPUs have turned into more general-purpose processing units beyond their use for just conventional graphics processing. In contrast to CPUs, GPUs provide a tremendous amount of processing power and often come with thousands of cores in a single unit of compute that can access GPU-internal memory with high bandwidth. As a result, GPUs have risen in popularity not only in graphics processing and machine learning but for a variety of different workloads. Moreover, with its wider use, GPUs have become a commodity and are today not only available in typical on-premise clusters but also in the cloud, where the GPU-equipped machines are only slightly more expensive compared to CPU-only machines.[1]

In the context of DBMS workload, most previous work on the integration of GPUs as accelerators has focused on the acceleration of OLAP workloads for single-node DBMSs [14, 15, 40, 122, 141, 144]. This is because OLAP queries map inherently well to the vectorized execution model of GPUs and hence are clearly an interesting workload for being accelerated by a GPU. This way, the DBMS is able to execute query operators such as joins or aggregations in a massively parallel manner on a single-node DBMS.

[1]E.g., the Azure cloud with the RDMA-capable instance Standard_NC24r with 4 K80 GPUs (with GPUDirect RDMA) is 10-25% more expensive in comparison to RDMA-capable CPU-only instances (Standard_HB60-15rs & Standard_HC44-16rs).

However, how to scale GPU joins and accelerate join queries in the context of distributed DBMSs is rather unexplored.

Hence, in this paper, we aim to show the potential of GPUs as accelerators in distributed DBMSs on clusters with fast RDMA-capable networks. Here, an important aspect is that using high-speed network cards with GPUDirect RDMA [97, 101], data shuffling over the network has the same cost independent of whether the target of the data transfer is remote CPU memory or remote GPU memory. To be more precise, a database node of a scale-out cluster can directly write data to the remote GPU memory (and also read data from the GPU) without the need to first write data to the remote host CPU memory and then copy it to the GPU memory. To bring this into perspective empirically, as shown in Figure 10.1a, RDMA writes with GPUDirect to remote GPU memory have the same network bandwidth as writing to the remote memory of a CPU. As such, in a distributed DBMS where data anyways need to be shuffled for executing the join operators, the higher speed of GPU joins compared to CPU joins, as shown in Figure 10.1b, can be leveraged without paying any higher cost for transferring data to the GPU.

However, when leveraging GPUs in a distributed setting, we argue that GPU joins need to be redesigned to work efficiently. To better understand why we first discuss the anatomy of a distributed join operator. A typical scheme for executing join operators in distributed DBMSs is that they first need to shuffle data across the network before the operator itself can be executed. For example, in a partitioned join, the data of the tables to be joined is first shuffled on the join keys over the network before the join is then executed in parallel on the resulting partitions. A key observation of this paper is that using the traditional sequential scheme described above, the GPUs remain *idle* when the CPU cores execute the shuffle operation. For the distributed partitioned join, this opens up an opportunity to better utilize GPUs during the shuffle operation in a *pipelined* manner by overlapping data transfer and actual join computation on the GPUs.

Contributions. In this paper, we thus propose a novel pipelined execution scheme for distributed query execution on GPUs to show the potential of using GPUs as accelerators in distributed DBMSs. We mainly focus on the effects of pipelining for distributed joins that are typically the most expensive operations when executing OLAP queries [30], particularly for large distributed DBMS. As a concrete contribution, we present two novel GPU-accelerated distributed join algorithms where the data partitions that result from shuffling are processed either in a *GPU-only* or in a *hybrid GPU/CPU* manner.

[2]Comparison of GPU join [122] on Tesla V100 without data-transfer overhead vs. CPU join [8] on 4 socket Intel Xeon 8268 (150M $\bowtie$ 300M - 8-byte tuples in random order).

The first join algorithm (GPU-only) introduces a pipelining execution scheme in a partitioned hash-join by using so-called active GPU kernels, which allow overlapping the network shuffling with the building and probing phases of the join. Moreover, it removes the need for materializing the data of the probing phase and can thus support arbitrarily large probe table inputs. As a second join algorithm, we present the hybrid GPU/CPU join that leverages both GPU and CPU for the join execution. That way, in contrast to the GPU-only join, the hybrid join can support building input tables larger than the aggregated GPU memory by materializing parts of the shuffled tables in CPU memory. In our evaluation, we show that these pipelined GPU joins can reduce the overall runtime by up to 6.8× over the state-of-the-art CPU baseline [125].

In summary, the main contributions of this paper are:

- A discussion and evaluation of the design space for integrating our *novel pipelined GPU execution* model into distributed DBMSs based on high-speed networks and GPUDirect over RDMA.

- A novel distributed GPU-accelerated join that can efficiently handle probe input tables larger than the available aggregated GPU memory.

- A hybrid GPU/CPU join that supports arbitrarily sized tables for both the build and probe inputs by transparently partitioning tuples across CPU and GPU memory.

- An extensive evaluation investigating acceleration potential over a range of different workloads as well as different settings such as available GPU memory capacities and in the context of complete queries.

Outline. In the remainder of this paper, we first give an overview of our GPU join algorithms. Next, we iterate over the design space for implementing a GPU-accelerated pipelined distributed join in Section 10.3. In Section 10.4, we present our detailed design of the GPU-accelerated pipelined join algorithm, followed by the hybrid algorithm in Section 10.5, which contains our solution for handling build-side tables larger than the collective GPU memory. Finally, we present our evaluation in Section 10.6, followed by related work in Section 10.7, and in the end, a conclusion in Section 10.8.

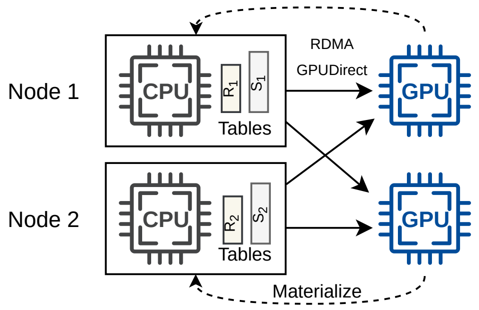
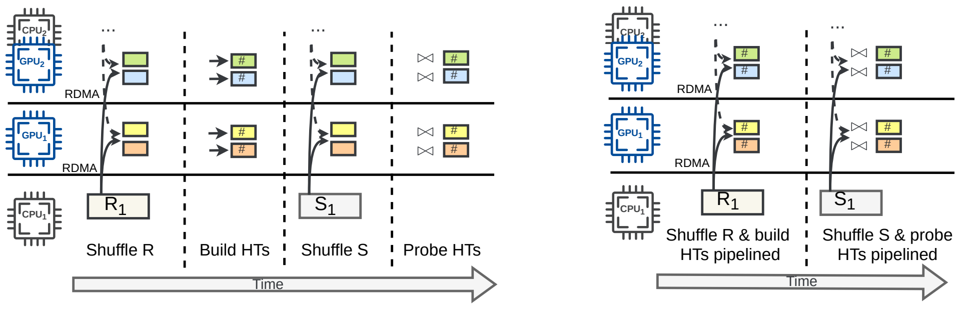


Figure 10.2: Distributed Join scheme of our Distributed GPU Joins where tables are stored in CPU memory and are shuffled to GPUs for join processing.



(a) Naive Blocking Execution Model (b) Pipelined Execution Model

Figure 10.3: For execution, we contrast two models: (a) a naive blocking execution where the building and probing phases are distinct from network shuffling of R and S versus (b) a pipelined execution where we overlap the GPU execution with the network shuffling.

10.2 Overview of Distributed GPU Join

In the following, we present an overview of our distributed GPU-accelerated join. We first discuss the overall scheme of the join before subsequently making a case for our pipelined join approach by contrasting it to a blocking variant.

10.2.1 Distributed GPU Join Scheme

In this paper, we target a scheme for the *distributed GPU join* where the input tables for the join are stored partitioned across the CPU memory of the different nodes, as shown in Figure 10.2.

We argue that this scheme provides significant benefits over a scheme that stores data to be joined in just GPU memory: First, storing data in CPU memory allows us to support joins over input tables (and intermediate results) that are larger than GPU memory. Second, distributed joins typically need to first shuffle the input tables based on the join key. Since GPUDirect RDMA allows data to be shuffled as fast from a CPU as well as from a GPU, the location of input tables does not have any effect on the performance as long as the tables are not pre-partitioned on their join keys. Finally, the scheme allows for several optimizations, such as chaining multiple joins on the GPU by caching small intermediates, as we discuss later, which helps to further improve the runtime.

For the join execution, we apply a partitioned hash-join where each GPU executes a build and a probe phase over the partitions resulting from shuffling. The main novelty is that the shuffling and join execution is pipelined, which has many benefits, such as support for much larger input tables and even arbitrarily large probe-side tables, together with overlapping of materialization of the join result back to local CPU memory.

10.2.2 The Case for Pipelining

As we discussed before, the core idea of our approach is to overlap and pipeline the execution of the shuffling phase (which is driven by the CPUs) with the build and probe phases of the join (executed on the GPUs). Overall, this allows us to avoid that GPU resources are idle during shuffling and, as such, reduces the runtime of the overall join execution. We illustrate this effect in Figure 10.3b in contrast to the blocking execution, as shown in Figure 10.3a. In the following, we contrast the details of blocking vs. our pipelining approach.

Naive Blocking GPU Join. Naively mapping the blocking execution scheme of the state-of-the-art distributed hash join approach [9] to a distributed GPU-accelerated join (as illustrated in Figure 10.3a) does not only come with severe limitations, but it also does not leverage the GPUs in the most optimal manner.

The main reason why a blocking execution scheme of the distributed join is not ideal for GPUs is that the GPU cores would stay idle until the network shuffling phase is finished. The same is true for the building and probing phases where the GPU would be active while the CPU cores would stay idle, resulting in an overall higher runtime as illustrated in Figure 10.3a (i.e., no work is executed on the CPUs in the build and probe phases of the GPU). Hence, in this paper, we argue for a pipelined execution that overlaps the CPU-driven network shuffling with the GPU join processing.

Moreover, another significant limitation of a blocking scheme for GPUs is that only tables of a certain limited size can be processed. The reason is that when executing the phases of a distributed join non-overlapped on the GPU, the GPUs need to be able to hold all intermediate data, e.g., the output of shuffling the build and probe tables in GPU memory. While recent generations, such as the Nvidia A100, has 40 GB or even 80 GB of internal GPU memory, the GPU memory sizes are still limited and cannot be extended as for CPUs. As such, when using a blocking model for executing a GPU-based join in distributed DBMSs, only joins where the input table sizes and intermediate data size do not exceed the aggregated memory of all available GPUs can be supported. In the following, we cover how we overcome these challenges with the pipelined join approach.

Pipelined GPU Join. The pipelined GPU join approach overlaps the execution of the shuffling of input tables with the building and probing on the GPUs. The conceptual reason why the pipelined scheme is more efficient is that such a scheme, as shown in Figure 10.3b, helps to efficiently hide the join processing on the GPUs under the data transfer by making use of CPU and GPU cores concurrently. Moreover, such a pipelined scheme clearly reduces the GPU memory consumption since only a chunk of data, instead of a full partition resulting from shuffling, need to be stored in GPU memory. For the probe phase, this means that arbitrarily large probe inputs can be supported and streamed through the GPUs. For the build phase, the pipelined model also has the same benefits since more GPU memory is available for the hash tables, effectively supporting larger build input tables.

However, clearly, the fixed size of GPU memory poses a strong limitation on the size of hash tables that can be kept in GPU memory. To mitigate this limitation and support input tables for the build phase larger than the aggregated GPU memory, in this paper, we propose a hybrid scheme (called hybrid GPU/CPU join) of our pipelined join that uses

GPU memory along with CPU memory (and the CPU cores) to execute a distributed join. The details of these two join algorithms: pipelined and hybrid schemes, are presented in Section 10.4 and Section 10.5, respectively.

Lastly, the pipelining approach allows to overlap not only the probing of the hash tables but also to hide additional processing given the efficient vectorized execution of GPUs by chaining multiple operations (e.g., multiple joins). Such *chained processing* can be used for typical OLAP queries with several joins between dimensions and the same fact table where we cache small (replicated) dimension tables on the GPU. We show in Section 10.6.3 how the pipelined approach can speed up a full query from the SSB benchmark with 4 joins up to 6.8× against a non-accelerated CPU query implementation.

Discussion. One might now argue that the pipelined scheme can also be used to overlap network shuffling and join execution to accelerate a distributed CPU join. A key aspect, however, of why pipelining for a distributed CPU join will not yield significant benefits is based on the anatomy of the distributed CPU join, which is very different from the GPU join. While for the CPU join, shuffling data over the network and executing the join by building or probing into a hash table utilizes the same resources, the GPU join can use distinct resources; i.e., when the CPU shuffles the data, the GPU is responsible for consuming the incoming data and building/probing hash tables.

As such, in the distributed GPU join, the GPU would remain idle if not overlapped with the CPU-driven shuffling. In a distributed CPU join, in contrast, this is different since the CPU resources need to be shared between shuffling and join execution if the two phases are overlapped. Moreover, with modern high-speed networks to saturate their high bandwidth, the CPU cores are already highly utilized by shuffling the data, which involves scanning the input tables of the join, partitioning the data for shuffling and sending the data [10]. In fact, for the hardware used in our experiments, all CPU cores are fully utilized during shuffling and thus, reserving dedicated CPU resources for joining would slow down the network shuffle, which anyway dominates the overall CPU join runtime.

Another aspect of GPU join is that GPUs provide additional memory resources for the read/write operations of building and probing hash tables. In contrast, for a CPU join, read/write operations of building and probing hash tables compete with the read/write operations of shuffling on the same memory resources, which hence limits the benefits of overlapping the shuffling with the join execution in the CPU-based join.

10.3 Design Space for Pipelined GPU Join

Designing a pipelined GPU join over fast RDMA-enabled networks is non-trivial and involves many design decisions. In the following, we thus first present the relevant background on RDMA and GPUDirect to enable the pipelined join and then discuss the design options for the join.

10.3.1 RDMA & GPUDirect

While RDMA has been used for distributed data-processing systems [9–11, 68, 110, 146, 157] over high-speed networks, there are some important aspects that need to be considered to directly transfer data from and to the GPUs.

In general, to leverage RDMA, an application can make use of different communication schemes that can be categorized as one-sided (READ / WRITE) or two-sided (SEND / RECEIVE) operations, which refers to the involvement of the sender- & receiver-CPU in the communication. For one-sided operations, only the sender node is actively involved, which typically brings along performance benefits since it avoids any additional overhead on the receiver. For two-sided operations, the receiver must actively be involved by posting RECEIVE requests to steer the placement of data.

To make use of RDMA on GPUs, GPUDirect RDMA provides a means of transferring data directly over the network from and to the GPU memory using the same RDMA primitives for one- or two-sided communication. With GPUDirect RDMA, data can be copied over the network at the same speed from and to the GPU memory as it can be copied from and to the CPU memory. Furthermore, data can be copied from the sender memory directly to the remote GPU memory without first being copied to the main memory of the remote CPU. This is enabled by exposing the virtual to physical address mapping of the GPU to third-party PCIe devices such as the NIC, thereby allowing the NIC to directly read and write to and from the GPU memory.

However, using GPUDirect RDMA (for distributed DBMSs) is not straightforward and comes with a few important challenges, in particular for implementing a pipelined execution on the GPU. First, for combining the data flow via RDMA and the GPU kernel execution, different design options exist compared to a pure CPU-based solution. Second, another challenge is that GPUs can potentially observe inconsistent data of incoming RDMA writes, such as partially written data or data that is not written sequentially (i.e., from lowest memory address to highest) or even observe RDMA writes arriving out-of-order. As such, we need to carefully design a pipelined execution scheme of a distributed join, as we discuss next.

10.3.2 Design Alternatives

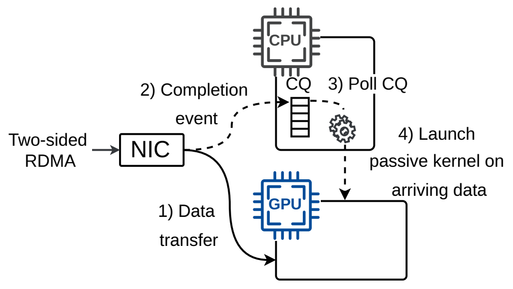
In the following, we discuss the different design options for our pipelined GPU join. We categorize the design options into two dimensions: *(i)* how to use the RDMA communication primitives for implementing the data flow for shuffling data over the network from CPU to GPU memory and *(ii)* the control flow of how the GPU kernel execution is triggered to consume incoming data for building hash tables and probing into them.

One- vs. Two-sided Data Flow. As previously mentioned, one-sided and two-sided RDMA operations exist to implement the data flow between machines. While these primitives work similarly for GPUs using GPUDirect as for CPUs, there are some important differences. Using one-sided primitives with GPUDirect works the same as for CPUs since GPUDirect allows that CPUs of the sending nodes can write in the remote GPU memory without involving the remote GPUs. When using two-sided operations instead, it is important to note that the RECEIVE requests are driven by the CPU (and not the GPU) in GPUDirect, since the CUDA library does not support calls to RDMA functions. Hence, in a two-sided communication, the CPU is always involved in the data flow even though the GPU is the target.

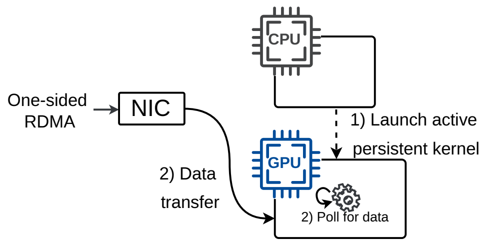
CPU- vs. GPU-driven Control Flow. Another aspect is whether the CPU or the GPU is driving the control flow, i.e., detecting when new data has arrived (on the GPU) and triggering the execution on the GPU for building/probing into the hash tables. In the following, we discuss two design options for the control flow: a CPU-driven approach and a GPU-driven approach.

(i) In a CPU-driven approach, the CPU actively detects that a new chunk of tuples has arrived (by polling for so-called RDMA completion events). Afterwards, the CPU then instructs the GPU for subsequent processing by launching a GPU kernel. Launching a GPU kernel *after* the write of a new chunk has been detected on the CPU ensures consistency and therefore overcomes the aforementioned challenge of inconsistent data with GPUDirect. However, the CPU-driven approach also introduces additional overhead and latencies for the pipelined join since, for each incoming batch of tuples, the GPU kernel needs to be called.

(ii) In the GPU-driven approach, the kernel launch overhead is removed by using persistent kernels where the GPU instead actively detects new incoming data by polling on a particular memory region for newly arrived data. This technique is also often applied in traditional CPU-based RDMA communication as memory polling has a smaller overhead in comparison to polling after RDMA completion events [28, 156]. However, having such



(a) CPU-driven Join / Two-sided RDMA



(b) GPU-driven Join / One-sided RDMA

Figure 10.4: Two design alternatives for implementing a pipelined GPU join over RDMA where either (a) the CPU is central in the control flow with two-sided RDMA or (b) the GPU controls when a new chunk of tuples can be processed without CPU involvement.

an active GPU kernel running concurrently with RDMA writes poses challenges to the consistency of the incoming data, which we later discuss how to overcome with minimal overhead in Section 10.4.2.

Join Design Space. By pairing the dimensions of the control flow and the data flow, different designs can be derived to enable a pipelined GPU join, as shown in Figure 10.4. The first design is a CPU-driven design that makes use of two-sided RDMA as illustrated in Figure 10.4a and GPU-driven with one-sided RDMA in Figure 10.4b. In the CPU-driven design (Figure 10.4a), the CPU is polling for completion events for incoming RDMA data and can subsequently launch a GPU kernel for the next chunk of incoming tuples to process the data written directly to the GPU. On the other hand, as a second design, a completely GPU-driven design (Figure 10.4b) can be used where a persistent GPU kernel is launched (once) initially, which itself actively detects and polls for incoming data. As can be seen, these two designs differ both in CPU overhead, the overall communication pattern, and how they ensure data consistency of incoming data to the GPU. Therefore, we next take a closer look at the performance of these two variants to find the best candidate for realizing the pipelined GPU join design.

10.3.3 Design-Space Evaluation

To guide the decision of which of the two alternatives has the most potential for a pipelined GPU join processing, we now evaluate the two designs (i.e., CPU-driven with two-sided RDMA vs. GPU-driven with one-sided RDMA).

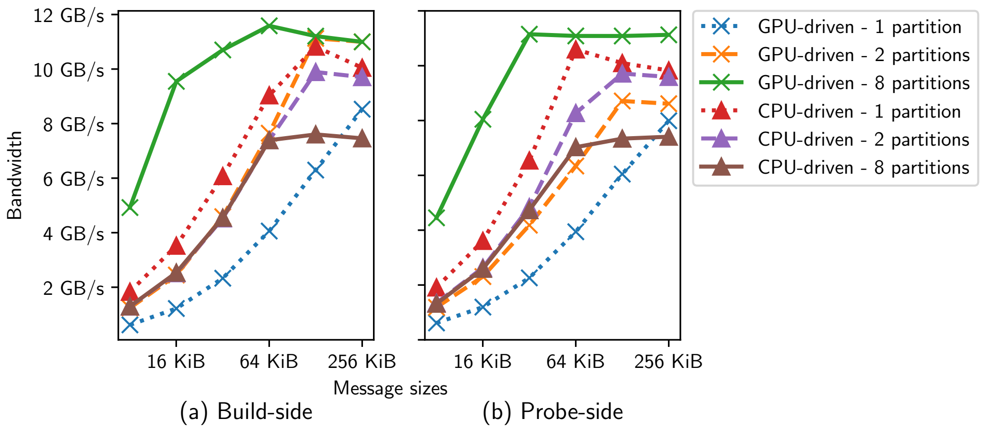


Figure 10.5: CPU-driven two-sided RDMA vs. GPU-driven one-sided RDMA for different degrees of partitions and message sizes on a 4 node cluster (1 GPU per node) using 2B $\bowtie$ 8B 16-byte tuples. The GPU-driven approach achieves a higher network utilization with more partitions, whereas CPU-driven does not scale with more partitions.

A key question that determines the overall performance of a distributed join is which design can better saturate the network bandwidth during shuffling, which is the limiting factor of a distributed join. We thus examine the designs by their ability to saturate the shuffling speed with different degrees of partition parallelism. Moreover, since the join is pipelined, it is paramount that the GPU kernel that executes the join steps (i.e., building or probing) can ingest the incoming data at the speed of the sender CPU cores. Otherwise, the GPU kernel for the join execution would slow down the join resulting in a reduced shuffling speed from senders since they can only send data at the speed at which the GPU can ingest the data.

For the design space evaluation, as shown in Figure 10.5, we hence execute a shuffling operation with a pipelined GPU kernel for building (a) and probing (b) hash tables that are the core components of our GPU join. We evaluate for varying message sizes, i.e., the size of each chunk of tuples we transfer during the shuffling phase from CPUs to the GPUs, against different number of partitions per GPU or CPU, as seen in the legend. A first observation is that using a higher number of partitions for the CPU-driven approach results in sub-optimal performance due to the CUDA library internally sequentializing the GPU kernel calls, even though a separate CUDA stream is used for each partition. Also, since each node is both sending and receiving (all-to-all shuffling), a higher partition fan-out results in fewer CPU resources available for sender threads in the CPU-driven approach since more receiver threads must be reserved for RDMA control flow and calling

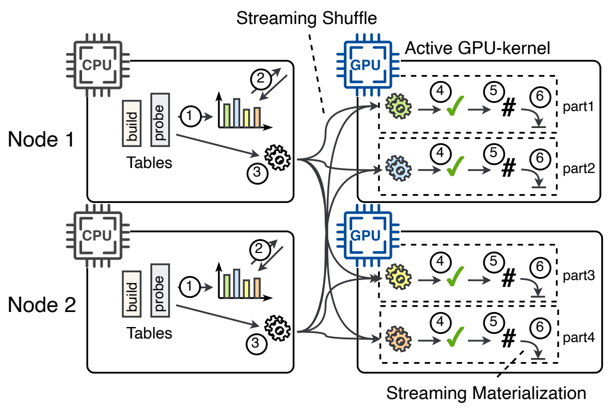


Figure 10.6: Pipelined GPU Join where tables are shuffled from CPU to (remote) GPU memory using a streaming shuffle operator that leverages GPUDirect RDMA. Hash tables are built and probed on the GPUs in a pipelined manner using active GPU kernels.

GPU kernels. This becomes visible for 8 partitions (brown line) where the performance plateaus at 64 KiB.

On the other hand, the GPU-driven approach does not inhibit the same limitation and thus scales better with the partition parallelism. For example, with 8 partitions per receiver node, the GPU-driven design can efficiently saturate the sender and network throughput. The reason is that the GPU kernels are not launched from the CPU on a per-message basis, as is the case for the CPU-driven approach. Instead, the GPU kernel is started once (at the beginning of the shuffling phase), and then the kernel actively polls in the GPU-local memory for new incoming data. As a result, the kernel launch contention of the CPU-driven approach where multiple CPU threads are launching GPU kernels in parallel is avoided. This allows the GPU-driven design to achieve a higher GPU utilization and thus higher processing speed through partition parallelism.

Summary. Based on these observations, the GPU-driven approach clearly dominates over the CPU-driven approach since it can better use partition parallelism and thus better utilize the network for a pipelined GPU join. In the remainder, we thus use the one-sided GPU-driven design for realizing our pipelined join.

10.4 Pipelined GPU Join Algorithm

In the following, we first give an overview of the one-sided GPU-driven design before we then explain the details of each step.

10.4.1 Overview of Execution Steps

In our pipelined GPU join, we aim to execute the shuffling of tuples for the join operation in a concurrent manner with the build and probe phase using so-called *active GPU kernels*. To better understand the overall execution phases on the CPU and GPU, we discuss the general steps for a two-node setup as shown in Figure 10.6:

Steps on CPUs (Sender-side). The CPUs on the sender-side are responsible for executing the data shuffling. For this, ① the CPUs first build the histograms in parallel on the build- and probe-side input tables that are to be joined. Afterwards ②, the CPUs exchange the local histograms (per database node) to compute a global histogram that allows all nodes to compute the total size per resulting partition before actually executing the shuffling. This global histogram is used by our pipelined GPU join to reserve adequate GPU resources and, in case of insufficient GPU memory, decide which of the partitions to place in GPU memory and which in CPU memory (as later discussed in Section 10.5). Once the global histograms are computed, the streaming network shuffle phase ③ starts that executes the re-partitioning based on join keys by transferring small chunks of data from the sender CPUs to the different GPUs.

Steps on GPUs (Receiver-side). The GPUs in our join are responsible for detecting incoming data and for executing the build and probe steps of the join, as shown in Figure 10.6 (right side). A novel aspect of our GPU join is that it executes the GPU steps ④–⑥ in an overlapped mode with the streaming shuffling using *active GPU kernels*. An active GPU kernel is started once at the beginning of a shuffle phase. Once started, an active kernel then polls for new incoming data chunks in GPU memory — first for the building and then for the probing phase. A challenge for the GPU-side execution with active kernels is to ensure data consistency as well as synchronization between CPU sender and GPU receivers. We accomplish this by proposing a parallelization strategy that aims to optimally map partitions resulting from shuffling to the vectorized execution model of the GPU. Details of this strategy, along with details about all the steps on the GPU, are discussed next.

10.4.2 Active GPU Kernel

Active GPU kernels not only poll for incoming data, but a core aspect is that they allow us to efficiently parallelize the join, which is composed of the consistency check step ④, build and probe steps ⑤ and successive operations ⑥ such as result materialization (illustrated in Figure 10.6). In the following, we focus on two aspects regarding the parallelization strategy of the active kernel: *(i)* how to determine the partition fan-out during shuffling and hence to determine the parallelism for an efficient GPU execution, and *(ii)* how to determine the amount of GPU resources for the different (potentially unequal sized) partitions. Moreover, we also briefly explain how we deal with data consistency for incoming RDMA writes as well as some relevant implementation details for the build and probe steps.

Parallelization Strategy. To answer the two aforementioned questions regarding the main aspects of parallelizing the join, we first need to explain some background on the vectorized execution model of GPUs. GPUs follow the SIMT (single instruction, multiple threads) execution model and typically come with thousands of threads; and thus, parallelizing the join processing on a GPU is very different from a CPU-based execution. The threads are grouped into a two-level hierarchy: *thread blocks* forming a group of threads and a *grid* comprising thread blocks. While all threads in a thread block are scheduled on a so-called Streaming Multiprocessor simultaneously, different thread blocks in a grid can be scheduled independently. We next discuss how the two aspects of the parallelization strategy work as discussed above.

(i) Determine Partition Fan-Out. First, we now look into the aspect of how to determine the partition fan-out during shuffling. For this, we use a scheme that over-partitions the data on the receiver GPUs, i.e., we use more partitions than the number of GPUs. As already shown in Section 10.3.3, this enables more efficient GPU join execution as each partition can be processed independently on the GPU without synchronization as the data arrives. When considering one partition, the maximum degree of parallelization for this partition depends on the size of the incoming message (chunks of tuples). For example, for a message size of 128 KiB with 16 B tuples, the maximum number of threads we can utilize is 128 KiB / 16 B = 8192 threads (since each tuple is processed by max. one thread). However, 8192 threads in this example might not be enough to saturate the message throughput considering high-speed networks, and as such more partitions might be required.

Yet, naively increasing the partition fan-out will negatively impact the speed at which the sender CPU threads can partition (and potentially could also exceed the TLB-cache

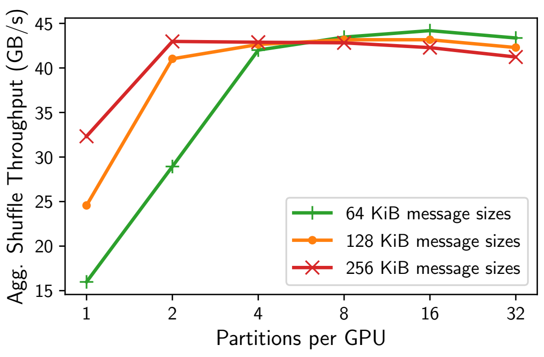


Figure 10.7: Aggregated network throughput on 4 nodes for the GPU-driven approach. From 4 partitions per GPU, the network is becoming saturated for the different message sizes.

[8]). Therefore, the optimal partition fan-out is achieved when the GPUs are able to saturate the tuple throughput, which in turn depends on CPU-processing speed and network throughput. For this reason, selecting the partition fan-out also highly depends on the specific GPU and CPU architecture and the network speed. So instead of fixing this parameter for our GPU join, we expose the degree of partition parallelism as a parameter one can configure.

In Figure 10.7, we show the effects of this parameter in a micro-benchmark. From 4 partitions per GPU, the network is becoming saturated for the different message sizes. While choosing a higher partition fan-out does not increase the aggregated network throughput, we empirically found that a fan-out of 16 partitions per GPU yields the most robust performance as more GPU resources can be allocated. Thus, in all our experiments, we are using a partition fan-out to get 16 partitions per GPU, which has shown to provide the best performance across all workloads we used in our evaluation.

(ii) GPU Resource Allocation. The second aspect of parallelizing the GPU join is how to determine the amount of GPU resources for the different (potentially unequal-sized) partitions. For this, we first discuss how we assign multiple thread blocks (grid) to each partition. Using multiple thread blocks per partition instead of limiting it to only a single thread block provides more flexibility in terms of allocating the necessary amount of resources needed to process each partition. However, this scheme raises a challenge of coordination across thread blocks when processing each chunk of tuples. We deal with this using so-called cooperative groups primitives of CUDA that enable us to provide synchronization across all threads assigned to process a specific partition.

Furthermore, for resource allocation, we need to decide how many threads (i.e., thread blocks) we assign to each partition. As the GPU join processing is pipelined, the size of each partition translates into an estimate of how frequently each partition will receive a chunk of tuples, assuming that the tuples of each partition are roughly spread out over the tables. Thus, an important question is how to determine the amount of GPU resources (threads) for the different partitions. One naive option is to assign a static number of thread blocks to each partition. While this performs well for uniformly sized partitions, this might result in suboptimal performance for unequally sized partitions as a fixed number of thread blocks might not be sufficient for the different partition sizes. Another option is to always allocate the maximum number of threads possible for each partition, but this, in turn, might exceed the number of threads available or result in wasted GPU resources and potentially block other kernels from running. To deal with this problem, we dynamically decide the size of the grids (i.e., the number of thread blocks) depending on the relative size of the partition that can be deduced from the histogram. As an example, for a fan-out of just two partitions, if one partition is twice the size, it will also get allocated twice the threads in comparison to the smaller partition. Deciding on the total amount of threads used on each GPU is dependent on the available hardware resources the join is running on. We show the ability to handle skewed workloads in Section 10.6.5.

Ensure Consistency. Another challenge we handle in implementing the GPU-driven/one-sided RDMA join is the issue of the GPU kernel observing potentially inconsistent data during concurrent incoming RDMA writes, as discussed in Section 10.3. These inconsistencies can come in the form of partially written data, data not written sequentially (i.e., from lowest memory address to highest), or out-of-order messages. While the general observations made in existing literature also apply to GPUs (such as the benefits of using one-sided over two-sided RDMA, inlining or door-bell batching), many techniques that are used to implement efficient RDMA-based communication schemes such as mailboxes or end-of-message polling [28, 35] cannot be directly applied. The reason for this is that all these techniques rely on the ordering guarantees between individual RDMA messages as well as a fixed write-order within one RDMA message, which is not guaranteed on the GPU.

Hence, a solution here is that the GPU kernel performs additional consistency checks on the incoming data chunks by appending a checksum per chunk of data (④ in Figure 10.6). In case data is only partially written on the GPU, the checksum verification that is executed by the GPU kernel in every iteration during busy polling will fail, and the GPU kernel can retry reading the data chunk and comparing the checksum in its next iteration.

As the type of error detection needed does not involve bit-flips, transmission errors, or data written to a wrong memory location (due to the underlying reliable network transport), we use the sum complement checksum that has a very small overhead while still being able to detect when the data has not fully been written.

We hope that NVIDIA will eventually provide memory fence primitives for RDMA, which would render the need for consistency checks obsolete.

Build and Probe Steps. After ensuring the consistency of the chunk of tuples, the tuples are inserted or probed into a partition-specific concurrent hash table (⑤ in Figure 10.6). For building the hash table, we use a custom design that leverages CUDA atomics to realize a lock-free hash table for our distributed GPU join algorithm. Logically, our custom hash table uses a chained hash table design. However, to avoid costly allocation and lock operations during the build phase, we use a design with two arrays — one array to implement the hash table and another dense array (called the chain array) that stores the chains for all hash buckets to handle collisions. Since we know the total number of tuples that we need to insert into the hash table from the histograms created during the shuffle phase, both arrays are pre-allocated to not incur any overhead for runtime memory allocations.

In the build phase, a batch of tuples for a given partition is then inserted in the hash table using these two arrays as follows: first, a GPU thread stores the new tuple into the chain array by atomically incrementing an offset into the chain array. Afterwards, this offset is written to the hash table array. To handle hash collisions, an atomic exchange operation is used. Specifically, the thread which inserts a new tuple will see the previous offset and use this to connect the tuples in the chain array accordingly. For linking the tuples in a chain, no atomic operation is needed. Finally, during the probing phase, the join key of each tuple is hashed and the chain of potentially colliding join tuples is traversed in the chain array. Contrary to the build phase, the probing does not incur any atomic operations since the hash table is static during probing. In summary, the hash table design is a good fit for the vectorized GPU execution as it does not incur any memory allocations, locking and only a few atomic operations for building.

10.4.3 Successive Operations

After finishing the probing, our join provides several options for successive operations in a query plan. One option is naturally to simply materialize the join result back to CPU memory (⑥ in Figure 10.6), as we discuss below. However, as analytical queries typically perform multiple joins and aggregate the join result, we designed the active

GPU kernel to be easily extendable to chain multiple GPU-local computations on the distributed join result before materializing results to the CPU. For example, instead of doing only a single probe, we could chain multiple probes that are still executed in a pipelined manner on the GPU. This chained scheme maps especially well to the typical abundance of processing power on the GPU and the pipelined execution model since it allows to hide even more computation under the network cost. We demonstrate this ability to chain multiple operations in our pipelined join to accelerate also complete queries in Section 10.6.3.

For materializing the result of the join either for subsequent operations on the CPU or delivering the result to the client, we take a streaming materialization approach which asynchronously writes data to the CPU memory, which nicely integrates with our pipelined execution model. This gives us several benefits over naively materializing the join result on the GPU and subsequently transferring the result back to the CPU. First, the GPU memory is limited, and thus avoiding result materialization on the GPU frees up resources to store hash tables of the join. Second, streaming the result directly to the CPU means that it comes with only a negligible overhead since the transfer can be overlapped with the pipelined execution of shuffling and GPU execution.

For realizing the streaming data transfer from the GPU to the CPU memory, there exists a range of transfer methods in CUDA: asynchronous memory copy, UM (Unified Memory) and zero-copy through UVA (Unified Virtual Addressing). The CUDA asynchronous memory copy from GPU to CPU memory can only be called from the CPU and, as such, is not applicable to our execution model with active GPU kernels where the CPU is not involved in the execution. Different from this, UM and zero-copy allow the GPU to directly access (read and write) CPU memory, where the CUDA abstraction takes care of utilizing the DMA hardware on the GPU to copy the data to CPU memory. We base our approach on UVA since we found that UM cannot fully pipeline the data migration of memory pages with the GPU processing. For leveraging UVA for pipelined materialization, we allocate the result relation using UVA in CPU memory. On the GPU, we first store the output tuples in intermediate output buffers and only copy a buffer (through CUDA memory copy device-to-device) to the result relation in UVA whenever the output buffer is full. To interleave GPU processing with the transfer to CPU memory, we use two output buffers per partition on the GPU.

10.4.4 Streaming Shuffle

Finally, we present how we realized efficient pipelined data transfers in the network shuffling phase (③ in Figure 10.6). To implement the streaming shuffling, we built our solution on top of the *Data Flow Interface* (DFI) [125], which is a high-level abstraction for fast networks that leverages one-sided RDMA communications for data transfers. The core abstraction of DFI is so-called flows that allow senders to push data into a flow and receivers to pull data out of the data flow. In a nutshell, flows support asynchronous communication between senders and receivers, which therefore allows for overlapping computation and communication.

However, DFI in its original design only supports CPU-to-CPU communication and does not come with GPU support. To realize the streaming shuffle operator with support for GPUs, we extended the flow abstraction of the original DFI code. As the main extension, we enabled the end-points of flows (called targets in DFI) to be located on GPUs by extending the buffer design and memory polling operations to allow the GPU to consume tuples out of the DFI flows. Additionally, we enabled GPUDirect RDMA by allocating the GPU-side DFI buffers through CUDA and then registering the memory region to the NIC.

Moreover, as previously mentioned, the GPU has a relaxed memory model that can result in memory inconsistencies when running a kernel concurrently with incoming RDMA data. For this, we extended DFI on the sender-side to compute a checksum over a batch of tuples if the target of a flow is on a GPU. On the target-side, when consuming this batch of tuples out of the DFI flow, we then re-compute the checksum in DFI and compare it with the appended checksum of the batch as discussed before in Section 10.4.2. Since the data is guaranteed to be eventually written consistently on the GPU [99], on the target side of a flow, we simply re-compute the checksum for the next batch until the checksum is correct and then hand the block to the active GPU kernel for either building or probing the hash tables.

10.5 Hybrid CPU/GPU Join Algorithm

In this section, we present our hybrid CPU/GPU join for supporting joins with arbitrary-sized build-side tables. One way to solve this problem is to pre-partition the build and probe input tables such that they fit in the GPU memory and use multiple rounds of the pipelined GPU approach presented before. However, the performance of this approach would significantly degrade since the pre-partitioning cannot be pipelined with network

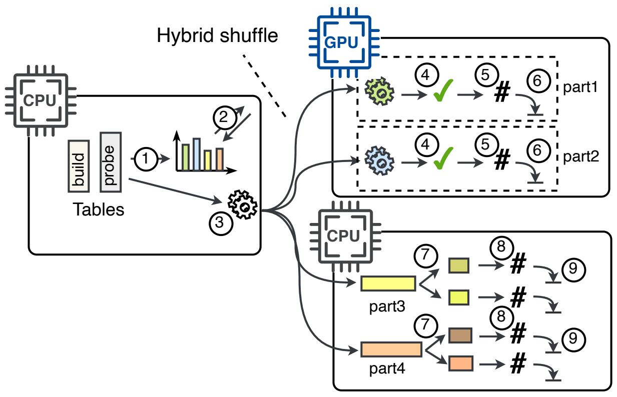


Figure 10.8: Overview of Hybrid Join. Tables reside in CPU memory and are shuffled to both GPUs and CPUs. Partitions going to CPUs are executed as a traditional radix hash join.

shuffling or GPU processing. As such, the goal of our hybrid pipelined join is to handle this scenario elegantly without, in the worst case, de-accelerating the join compared to a state-of-the-art CPU baseline.

The intuition behind our approach is to partition the input tables in the partitioning phase across remote GPU and CPU memory. While doing so, we leverage the GPU memory as a primary location for the partitions and only use the CPU memory for the remainder of the partitions that do not fit on the GPU. Figure 10.8 shows the idea of a hybrid join where a sender CPU is shuffling to both a GPU and CPU. For executing the join on the partitions shuffled to CPU memory, we make use of a state-of-the-art (blocking) CPU join implementation [9, 125]. This design comes with the benefit that its lower performance-bound is the performance of the CPU join algorithm, where any amount of available GPU memory can help to speed up the CPU baseline by using the additional GPU pipelined join. For implementing the hybrid join, we introduce a new hybrid shuffle operator that, in the shuffling step, over-partitions the input tables to a degree where we can maximize the benefit of the GPU pipelining by fully utilizing the available memory.

In order to determine the number of partitions that need to be created by the hybrid shuffling, we aim to place a fixed but configurable number of partitions on the GPU (we use 16 for our setup, as discussed before). However, as the build-side table exceeds the available collective GPU memory, setting the number of partitions to 16 per GPU would result in much fewer partitions actually assigned to the GPUs. We instead increase the fan-out such that the average size of 16 partitions roughly matches the available

GPU memory. Consider the following example: assume the build-side table has a size of 40 GB, and we can only store 32 GB when using two GPUs (with 16 GB per GPU). With 32 GB of GPU memory, the average partition size would be 1 GB. To decide the fan-out for a table with 40 GB, ideally, we would therefore create $40/1 = 40$ partitions. However, as the partitioning is done with radix-hashing, only the power of two partitions is applicable (2^n partitions for n bits), yielding 64 partitions. The average partition size for 64 partitions is $40/64 = 0.625$ GB and we can thus place $\lfloor 32/0.625 \rfloor = 51$ partitions on the GPUs (assuming a uniform distribution).

Moreover, often the resulting partitions after shuffling are not equally sized. To decide which partitions to allocate to the GPU, we thus draw on the knowledge of the histograms created in ① and ② as in the pipelined join (cf. Figure 10.8, left). Based on the global histograms, we can decide where to place each partition before starting the shuffling phase ③ to ideally allocate partitions to GPUs such that we leverage the full available GPU memory resources to store hash tables of the build phase. A naive random allocation instead would need to join unnecessarily many tuples on the CPU (which introduces additional runtime overhead) since GPU memory, and thus the computational resources are not fully utilized.

Finally, for the partitions assigned to the GPUs, steps ④–⑥ are executed pipelined on the GPU as earlier described in Section 10.4. For processing the partitions assigned to a CPU, we execute these steps in a sequential manner based on a CPU-based join (⑦ to ⑨) after the shuffling phase finishes. Therefore, we fully materialize the intermediate partitions for the CPU as the pipelined execution model does not map well to the CPU-based execution as discussed in Section 10.2.2.

10.6 Experimental Evaluation

In our evaluation, we analyze the GPU-acceleration potential using our distributed GPU join algorithms for different workloads and hardware resources. In the following, we first explain the setup before we discuss the results of the different experiments where we compare against various baselines.

10.6.1 Setup & Workloads

Setup. All experiments were conducted on a 5 node cluster, each node equipped with two Intel(R) Xeon(R) Gold 5120 CPUs (14 cores) and 512 GB main-memory split between both sockets. Each node has two Mellanox ConnectX-5 MT27800 NICs (InfiniBand

EDR 4x, 100 Gbps) and two Nvidia Tesla V100 GPUs with 16 GB memory, supporting GPUDirect RDMA. The operating system is Ubuntu 18.04.1 LTS with Linux 4.15.0 kernel on all nodes. All joins are implemented with C++17 and compiled with gcc-10.1.0 and nvcc-11.3 (CUDA 11.3).

Workloads. The workloads used in experiments 1 and $3 - 4$ follow the previous work on distributed CPU joins [9], where input tables are partitioned across the distributed nodes. The tables have randomized tuple order, and unless otherwise stated by selected experiments, the joins are evaluated with 16-byte tuples and without materialization of the join result for neither CPU baseline nor GPU-accelerated joins. In the second experiment, we evaluate two full queries comprising several joins and aggregation from the Star-Schema-Benchmark (SSB).

Join Variants.

In the evaluation, we compare the performance of the following join implementations:

- CRJ - *CPU Radix Join:* As a CPU baseline, we use state-of-the-art implementation of the distributed radix hash join [125].

- GPJ-B - *GPU Partitioned Blocking Join:* This is a distributed variant of the state-of-the-art single-node GPU partitioned join [122]. The shuffle phase (i.e., histogram creation & data shuffling from [9]) and the GPU execution phase are executed subsequently. Hence, we term this join *blocking*. Data shuffling is either realized with GPUDirect or without GPUDirect.

- GPJ - *GPU Partitioned Pipelined Join*: Our novel GPU-accelerated distributed join that supports pipelining of the network shuffling and the GPU join phases, as earlier explained in Section 10.4. This join already allows arbitrary-sized probe-side tables.

- GPJ-H - *GPU Partitioned Hybrid Join*: This is our hybrid algorithm where both GPU and CPU are used for join execution. This join allows arbitrary-sized build-side tables that go beyond the aggregated memory capacities of all GPUs, as earlier explained in Section 10.5.

All joins make use of established optimizations for efficient partitioning, such as software write combine buffers (SWWCBs), non-temporal streaming hints [6, 117], and one-sided RDMA writes [10].

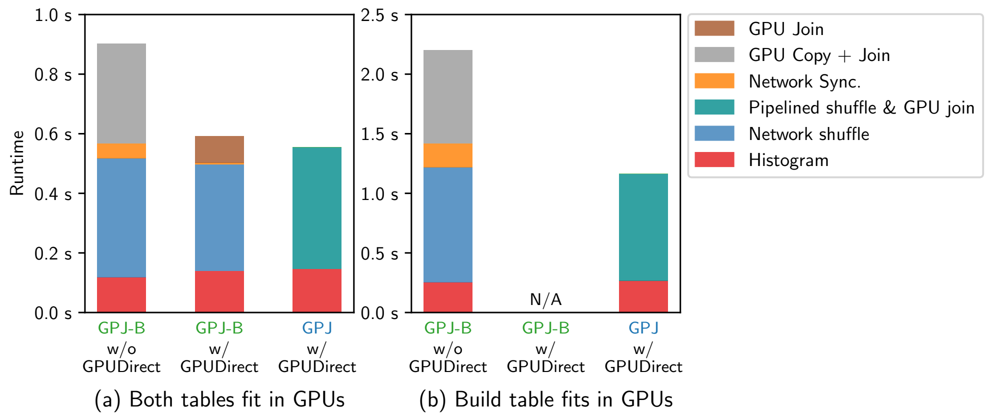


Figure 10.9: Blocking (GPJ-B) vs. pipelined (GPJ) GPU join with build-side of 600×10^6 tuples and probe-side of (a) 1.2×10^9 tuples and (b) 4×10^9 tuples. Our approach (GPJ) provides a speedup of approx. 2× while supporting arbitrarily sized tables.

10.6.2 Exp. 1 - Pipelined GPU Join

10.6.2.1 Comparison with a Blocking GPU Baseline:

In this section, we look at the benefits of the proposed pipelining model in comparison to a distributed blocking GPU join (GPJ-B). The blocking join takes a sequential approach where the two tables are only joined on the GPU once all data has been shuffled. We use the state-of-the-art single-node GPU join [122] to realize the blocking join on each node. The histograms and shuffling (re-partitioning) of the tables are based on [9] with minor modifications for enabling shuffling with GPUDirect, as we discuss next.

We execute GPJ-B both with and without GPUDirect to better show the effect of fast networks. As shown in Figure 10.9a, using GPUDirect greatly improves the acceleration potential of the GPU since we can leverage it without additional transfer costs. Moreover, we see that our pipelined GPU join, which also uses GPUDirect, can further improve over the blocking GPU join since with our pipelined join, we can overlap the join phases with data transfers.

Another significant advantage of our pipelined GPU join over the blocking GPU join is that for the blocking GPU join (GPJ-B), GPUDirect can only be used if both tables fit in the GPUs, as all input data must be accumulated before executing the join. In Figure 10.9b, we show this effect by increasing the probe-side table such that only the build-side table fits on the GPUs. As we can see, only GPJ-B without GPUDirect is supported in this case, whereas our pipelined GPU join can support arbitrary probe-side

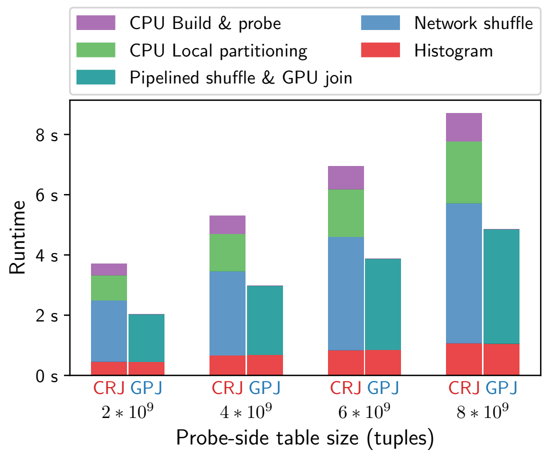


Figure 10.10: Varying probe-side table size and a fixed build-side table size of 2×10^9 tuples on 4 nodes. Our approach of GPU-accelerated pipelined join (GPJ) outperforms CPU radix join (CRJ) in all cases.

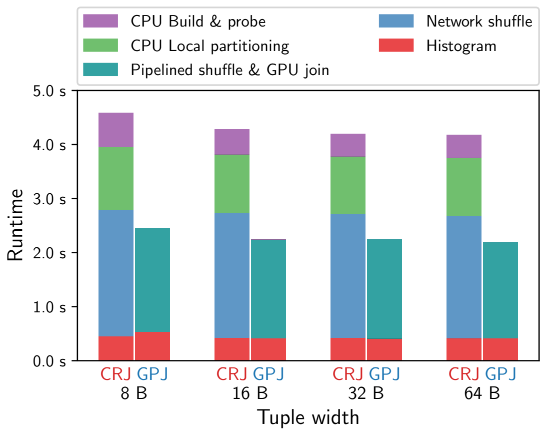


Figure 10.11: Our approach (GPJ) also outperforms CRJ for varying tuple widths. Build & probe tables are fixed at 16 GB & 64 GB respectively.

table sizes and, as such, provides a speedup of approximately 2×. Additionally, since GPJ-B performs multi-pass partitioning on the GPU to facilitate a faster join, it incurs a higher memory overhead, which further limits the table sizes that can be stored on the GPUs. In fact, for the workload in Figure 10.9b, GPJ-B consumes the complete GPU memory while GPJ only takes up half.

10.6.2.2 Comparison with CPU Baseline:

We now evaluate our pipelined GPU join (GPJ) against a non-accelerated CPU baseline (CRJ). Both joins share the execution steps performed on the CPUs and therefore highlighting the benefits of GPU acceleration and pipelining.

Varying Table Sizes (Probe-side). As both joins can support arbitrarily sized probe-side tables, we first compare both joins using a fixed-sized build input table that is joined with a probe-side table of different sizes. More precisely, we used a workload with a build-side table of 32 GB (2×10^9 tuples with 16 bytes), fitting into the 4 GPUs and with probe-side tables up to 128 GB (8×10^9 tuples), thereby exceeding the total collective

GPU memory that is 64 GB (disregarding the hash tables). Over the different probe-side table sizes, GPJ is up to 1.7× faster than CRJ.

The result can be seen in Figure 10.10, where we report the time each phase of the joins takes. As we can see, the GPU-accelerated join outperforms the CPU join in all cases. For the GPU-accelerated join, we do not report the time for the network shuffling, and GPU join execution separately due to the GPU pipelining; i.e., both phases are overlapped in this join. Interestingly, we can see that for GPJ, the shuffling and joining phases are shorter than only the shuffling phase of CRJ. The reason is that the RDMA writes performed by the sender CPU (one-sided) are going to the remote GPU and thus relieve pressure on the main memory. This is different for CRJ, where the CPU cores and the NIC are both writing to the main-memory during the shuffling phase.

Varying Tuple Width. Next, we evaluate the GPJ and CRJ join using the same table sizes but with varying tuple widths ranging from 8 to 64 byte tuples. For this experiment, we fixed the table sizes to 16 GB and 64 GB for the build- and probe-side tables, respectively. Our motivation for this experiment is to show the effect of different tuple widths since CPUs and GPUs use different execution and memory models and thus potentially affect the runtime differently.

The results of this experiment are shown in Figure 10.11. Similar to the above experiment, GPJ clearly outperforms the CPU join (CRJ) in all tuple widths, essentially due to the pipelining. Moreover, both join algorithms inhibit similar behavior with a small decrease in runtime for wider tuples. The reason is that as the tables with wider tuples contain fewer tuples, the overall per-tuple overhead decreases, which leads to a decrease in runtime.

Scalability of Joins. Finally, we show how the proposed pipelined GPU-accelerated join scales against the CPU baseline over an increasing number of database nodes. To show this, we simulate a cluster of up to 10 virtual nodes using 5 physical nodes where we use each of the two NUMA regions. Those regions are each equipped with their own GPU and RDMA NIC and therefore function as independent database nodes as all data shuffled between the logical nodes is transferred over the InfiniBand network (and not the cross-NUMA interconnect). For the workload, we used a build-side table of 300×10^6 tuples and a probe-side table of 1.2×10^9 tuples per node. As an example, for 10 nodes, a total of 48 GB $\bowtie$ 192 GB tables are joined.

Figure 10.12 shows the results with the tuple throughput as the main metric. As we can see, both joins have linear scalability up to the tested 10 nodes. For both cases, we see a performance increase of about 4.4× from 2 to 10 nodes.

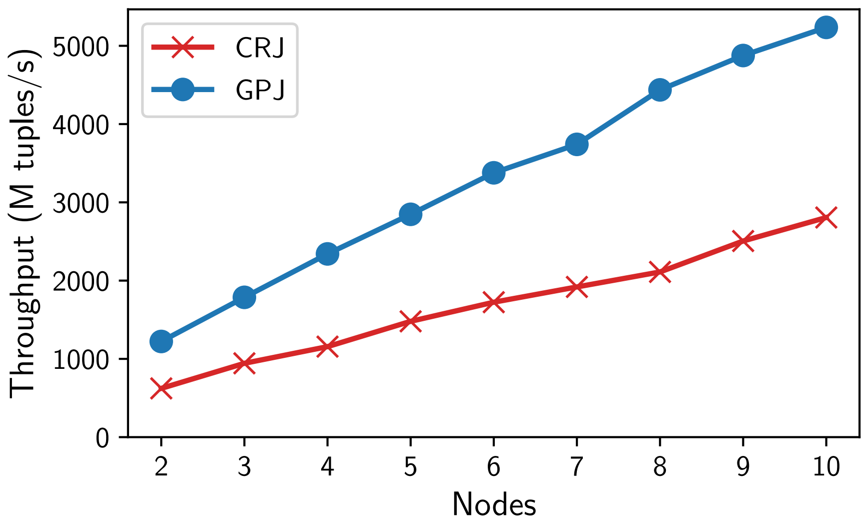
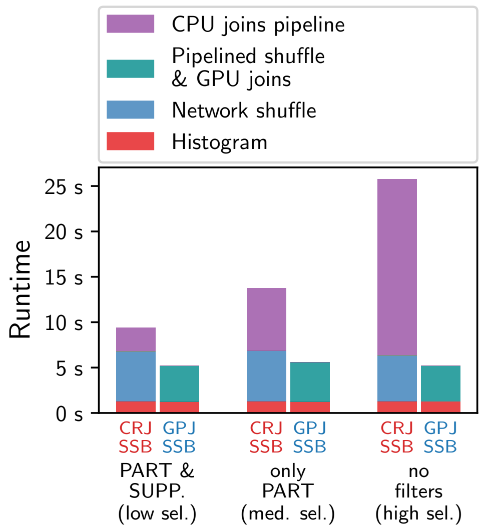


Figure 10.12: Scale out experiment (GPJ vs. CRJ) with build-side table of 300×10^6 and probe-side table of 1.2×10^9 tuples per node. Performance increase of $\sim 4.4\times$ is observed when nr. of nodes increases from 2-10.

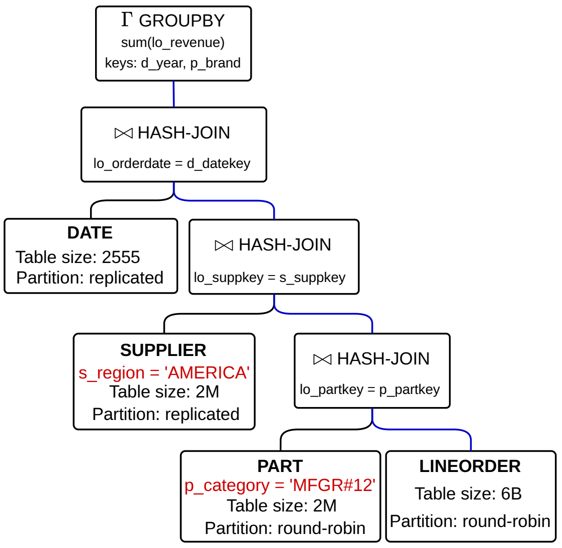
Why not pipelining a CPU Join?. Finally, we want to make the case why pipelining the network shuffling phase with the subsequent join is not an optimization to apply to a distributed CPU join as well. First, the nature of the shuffling operation is very memory intensive. Thus, all the CPU cores are reading the table and writing out the tuples to partitions, while the NIC is also reading the data to send over the network and writing incoming data from other nodes. As such, first, any additional memory operations (as in our pipelining approach) will further slow down the shuffling and thus the overall join execution since, as reported before, the shuffling is the limiting factor. Second, profiling the shuffling phase of CRJ supports this claim and reports not only over 10% of DRAM memory assesses are stalling but also 100% CPU core utilization which indicates that no idle (free) CPU resources would be available for a pipelined (overlapped) execution.

10.6.3 Exp. 2 - Complete Queries

In this experiment, we show the benefit of our pipelined join for a full query with multiple operations. As we explained before in Section 10.4.3, when running full queries, we allow multiple joins to be *chained* together (e.g., to chain multiple probe steps for a multi-way join in one pipeline on the GPU). The main intuition why chaining on GPUs is beneficial is that GPUs typically have an abundance of processing power in comparison to their i/o



(a) Query Runtime w/ Different Filters



(b) SSB Query 2.1 Plan

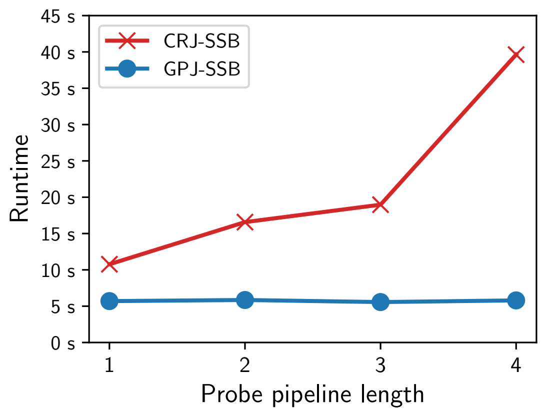
Figure 10.13: Execution (a) of SSB Query 2.1 (b) with SF 1000 on 4 nodes with different intermediate result sizes stemming from different dimension table filters. Probing of all hash tables and aggregation (blue edges) are chained both in CRJ-SSB and GPJ-SSB. Larger intermediate sizes incur extra runtime overhead for CRJ-SSB, while GPJ-SSB is unaffected due to pipelining on the GPU.

speeds. In fact, when executing a single join, as in the previous experiments, there are still untapped computational resources left for chaining multiple operators together.

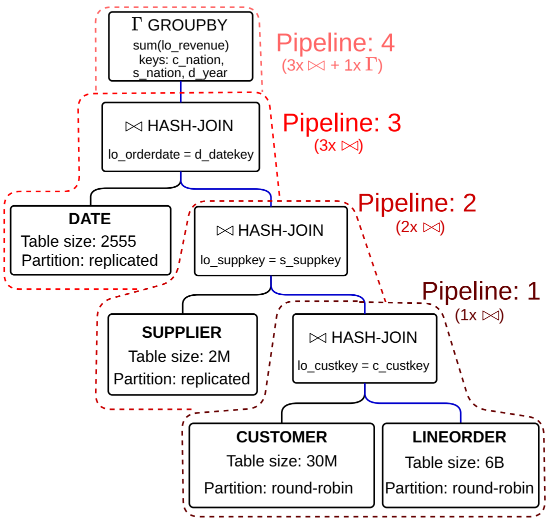
To show these effects of chaining, in this experiment, we use query 2.1 and 3.1 of the Star-Schema-Benchmark (SSB) that both involve three hash-joins and an aggregation. For comparison, we run two variants of the queries: one using only the pipelined join GPJ (GPJ-SSB) with the aggregation also on the GPU, and one that runs completely on CPUs using CRJ (CRJ-SSB). For both GPJ-SSB and CRJ-SSB, we use the same execution strategy, where we first build the hash tables on the dimensions tables and then chain together the probing of the LINEORDER tuples into these hash tables, followed by a final aggregation.[3]

Effect of Selectivities. We first execute the SSB query 2.1 with 3 different filter settings, which results in different intermediate result sizes. The query plan is shown in Figure 10.13b, along with which filters (marked with red) are changed to generate different

[3]PART and CUSTOMER are partitioned while DATE and SUPPLIER are replicated to enable chaining.



(a) Query Runtime



(b) SSB Query 3.1 Plan w/o Filters

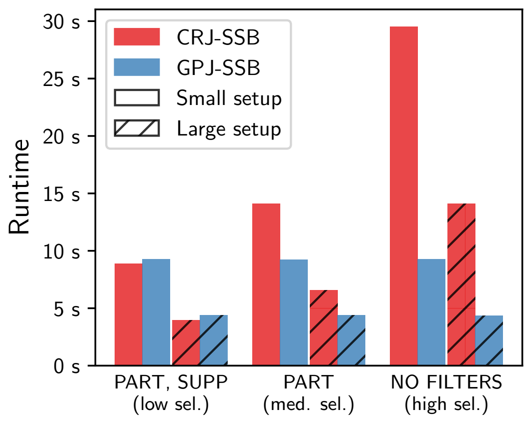
Figure 10.14: Execution (a) with a different number of joins of SSB Query 3.1 (b) with SF 1000, on 4 nodes. Probe-side joining is chained together both in CRJ-SSB and GPJ-SSB. With longer probe pipelines, the runtime of GPJ-SSB is unaffected due to its pipelined design resulting in a reduction in runtime by up to 6.8× against CRJ-SSB.

join selectivities and therefore different intermediate result sizes. We show the runtimes in Figure 10.13a. Here an interesting observation is that for our approach GPJ-SSB, a higher number of intermediate tuples does not introduce any runtime overhead due to the parallel execution and the higher processing power of the GPU, which completely hides the additional execution time under the network shuffle phase. Contrarily, we see for CRJ-SSB that larger intermediate sizes introduce higher runtime due to the sequential execution. For the largest intermediate results (i.e., no filters), our approach (GPJ-SSB) is, in fact, 5× faster than the non-accelerated CPU query execution.

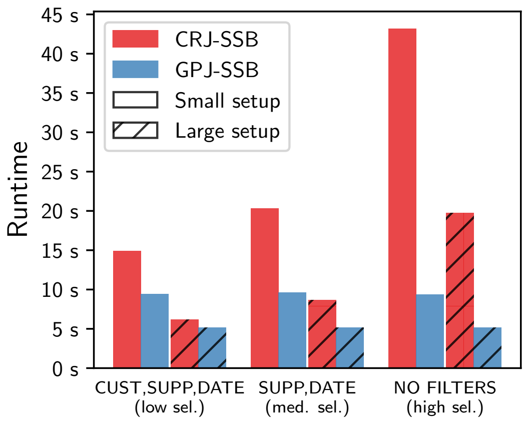
Effect of Number of Joins. Next, we evaluate the influence of the length of the (chained) probe pipeline, i.e., the number of joins in the query on the runtime. Similar to the effect of different intermediate sizes, the number of joins also varies the amount of processing needed during the probing stage of the LINEORDER table. We base the experiment on SSB query 3.1 as shown in Figure 10.14b, but vary the number of joins in the query (pipeline length). For instance, a probe pipeline length of 2 will join together the LINEORDER with CUSTOMER and SUPPLIER and a length of 4 contains the complete query. As can be seen in Figure 10.14a, the GPU-accelerated query execution (GPJ-SSB) is not affected by the query size since the additional probing is all hidden

Table 10.1: Cost-comparable clusters w/ and w/o GPUs: In both setups (small, large) the hardware costs are comparable. For example, a 4-node cluster with CPUs-only costs as much as 2 nodes with GPUs.

	Small setup	Large setup
GPU-accel (GPJ)	2× nodes w/ 2× GPUs	5× nodes w/ 5× GPUs
CPU-only (CRJ)	4× CPU nodes	10× CPU nodes



(a) SSB Query 2.1 Runtime w/ Different Filters



(b) SSB Query 3.1 Runtime w/ Different Filters

Figure 10.15: Query runtime with clusters that cause comparable cost as outlined in Table 10.1 for (a) SSB Query 2.1 and (b) SSB Query 3.1 (SF 1000). The advantages of the GPU join (GPJ) becomes more pronounced with larger intermediate results (i.e., higher selectivity).

under the network shuffling. This is in contrast to the CPU-only approach (CRJ-SSB) where more joins result in a higher runtime. For the full query without dimension table filters, a speedup of 6.8× can be observed over CRJ-SSB.

Comparable Resources. Choosing to accelerate a query on GPUs begs the question of whether the observed speedup outweighs the added cost of the GPUs. We thus now evaluate the same SSB queries as before using two hardware setups (with and without GPUs) that have approximately the same hardware cost. For our hardware, as detailed in Section 10.6.1, a machine with a GPU costs twice as much as the same machine without a GPU. To be more precise, the cost of one Tesla V100 GPU equals that of a server (including the CPU, memory, NIC as well as chassis without the GPU).[4]

In this experiment we hence use two different *cost-balanced* hardware configurations, as shown in Table 10.1, that reflects the aforementioned relationship in cost; i.e., for the cluster without GPUs, we use twice as many machines as for a cluster with GPUs. To be more precise, we use a *small* setup that uses 4 CPU machines (w/o GPUs) and compare it to a cluster of 2 machines w/ GPUs. For the *large* setup, we use 10 CPU machines (w/o GPUs) vs. 5 machines w/ GPUs. The results of running the SSB queries 2.1 and 3.1 on these two cluster setups (small and large) with a comparable set of resources can be seen in Figure 10.15.

Interestingly, even though the CPU-only cluster (for both the small and larger setups) can make use of twice as many nodes as the cost-comparable cluster with GPUs, we observe that the GPU join (GPJ-SSB) can still outperform the CPU join (CRJ-SSB) (or is at least competitive). Clearly, compared to Figure 10.13a, where the GPJ-SSB can use the same number of nodes as the CRJ-SSB, the benefits are less pronounced. For example, the runtime of using only 2 GPU-nodes in Figure 10.15a for query 2.1 is double as high compared to the GPJ-SSB in Figure 10.13a, which uses 4 GPU-nodes. However, we can still see in Figure 10.15a that the runtime of the CPU-only query (CRJ-SSB) linearly increases with larger intermediates while the GPJ-SSB can almost provide constant runtime since the GPU can make efficient use of its high degree of parallelism for larger intermediates. As such, for increasing intermediates the GPJ-SSB provides a significant speedup over the CRJ-SSB of up to 4× for Query 3.1 (no filters) even though GPJ-SSB is executed on a cluster with only half the nodes.

10.6.4 Exp. 3 - Hybrid Join Execution

In this experiment, we evaluate our hybrid GPU/CPU join algorithm (GPJ-H) on the effect of supporting arbitrarily larger tables than the GPU memory. With GPJ-H, different from GPJ, build-side tables that exceed the collective memory of the GPUs are supported by distributing the partitions across GPU and CPU memory.

Varying Table Sizes (Build & Probe-side). In this experiment, we increase the build- and probe-side tables from 32 GB to 128 GB. While using only 4 nodes with 1 GPU per node, this setup provides a total of 4×16 GB $= 64$ GB of GPU memory. With

[4]As hardware prices fluctuate, this experiment serves only as a rough comparison on balanced resources. Moreover, note that we use high-end Tesla V100 GPUs for the cost comparison and thus the ratio of CPU to GPU machines would be even more in favor of the GPU cluster in case we use less expensive GPU hardware.

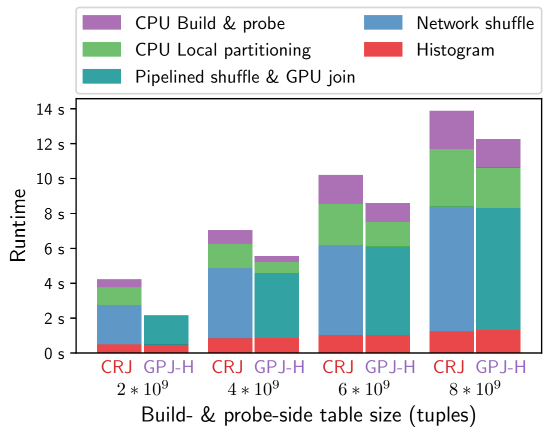


Figure 10.16: GPJ-H vs. CRJ when scaling input tables on 4 nodes. With larger build-side tables, more data spills out on CPUs.

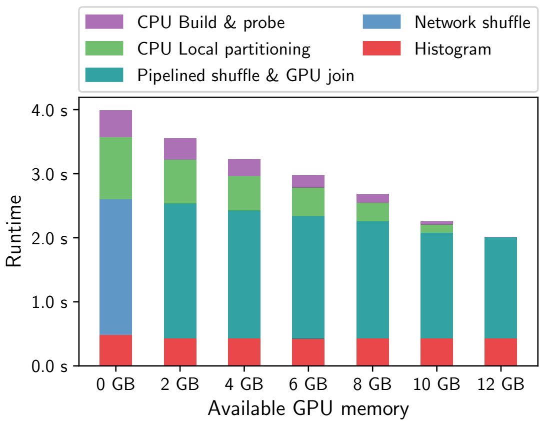


Figure 10.17: Hybrid join (GPU-H) adapts to increase in available per-GPU memory allowing reduction in runtime.

the smallest table size, the hash-tables for build-side tables can be fully stored in GPU memory, and thus GPJ-H is effectively the same as the GPJ.

Figure 10.16 shows the results of this experiment. With increasing table sizes, the runtime of both joins increases. For our hybrid join (GPJ-H), we see that with the increasing table sizes, more partitions will be assigned to CPUs resulting in an increased CPU runtime for GPJ-H as the partitions do not fit the GPU memory anymore. Still, with the hybrid execution, even for the largest tables size of 128 GB, we see a runtime benefit of around $2s$ in comparison to the CPU counterpart (CRJ). This is because, for the largest tables size, roughly a quarter of all tuples are joined on the GPUs and the rest on the CPUs. Thus, with the increasing table size and more partitions being assigned to CPUs, the benefit of pipelining diminishes, resulting in an increase in the runtime.

Varying GPU Memory. The hybrid execution of GPJ-H also allows a GPU-accelerated DBMS to adapt to different GPU memory sizes (and thus adapt to different available hardware). In this experiment, we analyze the effect of varying GPU memory sizes on the join runtime. As workload, we use fixed-size build- and probe-side tables of 2×10^9 tuples with 16 bytes and scale the available GPU memory from 0 B (pure CPU join) to the point where the hash tables (for the given workload) of the whole build-side table fit on the GPUs.

Figure 10.17 shows the result of this experiment of the GPJ-H over different GPU memory sizes. The main takeaway is that GPJ-H is able to efficiently adapt to different

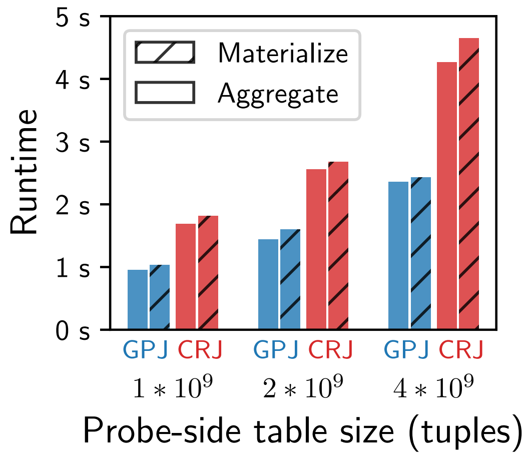


Figure 10.18: Cost of materialization on 4 nodes with a build-side table size of 1×10^9 tuples and 100% join selectivity.

GPU memory sizes and speed up the performance with the increasing available GPU memory. Moreover, even with a small amount of available GPU memory, the hybrid join is able to utilize the GPUs, and hence reduce the join runtime.

10.6.5 Exp. 4 - Microbenchmarks

In the following, we show the results of three microbenchmarks.

Streaming Materialization. In this experiment, we compare the runtime of GPJ and CRJ with and without materialization enabled. As a setup, we used 4 nodes and 4 GPUs with a build-side table size of 1×10^9 tuples of 16 bytes. In case materialization is used, CRJ writes the result directly to the main memory, while GPJ uses the streamed materialization from GPU to CPU memory as described in Section 10.4.3. In case no materialization is used, we execute a COUNT aggregation on the join result.

Figure 10.18 shows the results. What can be seen is that with materialization, the runtime increases slightly both for GPJ and CRJ, which stems from the added write-load on the CPU memory in both cases. Importantly, for GPJ we see that materialization of the result into CPU memory only has minimal overhead, which is in a similar range as for CRJ.

Multiple Concurrent Joins. For OLAP queries, it is common to schedule multiple queries at once on the same cluster. Thus, in this experiment, we evaluate whether there is any inherent disadvantage to accelerating the join with GPUs versus the traditional CPU-only execution when executing multiple joins concurrently. As workload, we execute

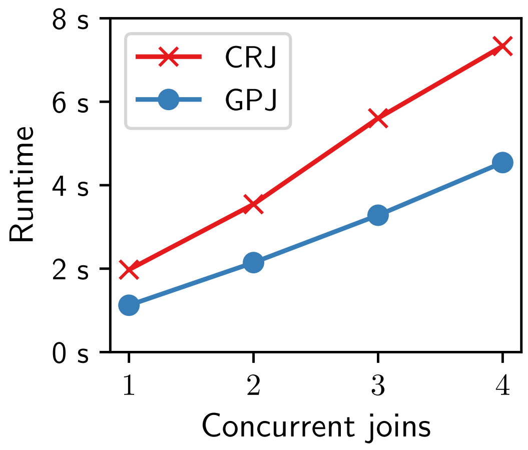


Figure 10.19: Runtime of concurrent joins on 4 nodes using 200×10^6 and 2×10^9 tuples for build- and probe-side tables.

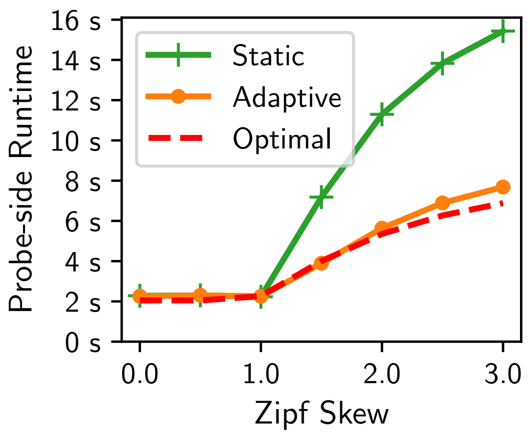


Figure 10.20: Runtime for probe-side shuffling with skew on 4 nodes with 6.4×10^9 tuples with adaptive or static grid sizes.

the same join operator using 200×10^6 and 2×10^9 tuples for build- and probe-side tables concurrently on a cluster of 4 nodes. Moreover, for each join, we allocate dedicated threads (of CPU or GPU cores); i.e., we split the resources depending on the number of concurrent queries that is a common scheme for OLAP queries.

As we can see in Figure 10.19, when increasing the number of concurrent joins, almost the same relative increase of runtime can be observed for CRJ and GPJ. However, since the absolute runtime of the GPJ is lower compared to the CRJ, as we already showed in the previous experiments, the gap in total gains increases with more concurrent queries (w.r.t. the elapsed runtime for all concurrent joins). For example, for 4 concurrent joins, the elapsed runtime is approx. 4 seconds for the GPJ, while the CRJ requires almost 8 seconds. As such, the total benefits of the GPJ are also significant under concurrent query execution and there is no strategic disadvantage for the GPU when running queries concurrently.

Effect of Skew. Skewness is often occurring in datasets and especially in the form of foreign-key skew where the probe-side table has many tuples which joins with a small subset of tuples from the build-side table. This skew can sometimes be beneficial since it increases the locality for the heavy-hitters and allows for better caching. However, in the context of a partitioned join, it will result in uneven-sized partitions, and for our pipelined execution algorithm, it means that the bigger partitions on the GPU will receive tuples at a faster rate than others. As discussed in Section 10.4.2, we handle this by adaptively sizing the allocated resources (grid-sizes) for each partition in relation to the size of the partition, such that the processing speed of the big partitions can be adapted with respect to their sizes.

To show this effect, we plot the observed probe-side runtimes for different skew factors in Figure 10.20. For a skew of range 0 to 1.0, the runtime is unaffected since we distribute the partitions in a round-robin manner to nodes by the size of the partitions to even out the network skew. Even when using statically allocated GPU resources, the GPUs have enough processing power to handle this level of skew. From zipf 1.0 and up, however, the static approach severely slows down the GPU join execution as the processing of the heavy partitions leads to a straggling behavior. However, with our adaptive optimization, the processing at each GPU can be done approximately at line rate (for the given distribution). The efficiency of our approach can be also seen when comparing the adaptive runtime to the dashed red line, which shows the theoretical optimal runtime given the overall network skew resulting from some nodes receiving more tuples than others. With high enough skew, the size of the largest partition dominates (i.e., it is larger than the uniform share per node) and thus the theoretical optimal runtime also increases since the skew cannot be mitigated completely.

10.7 Related Work

Single-node Joins: .

Acceleration of single-node joins with GPUs has been a well-studied topic over the last decade [45, 46, 48, 65, 88, 89, 114, 115]. Sioulas et al. [122] investigated how to best utilize the GPU hardware for a single-node join by implementing and evaluating a range of different joins while considering different scenarios for tables larger than GPU memory. However, when tables do not fit on the GPU, the slow PCIe interconnect limits the performance when compared to a CPU-only baseline.

More recent work evaluated larger-than-memory single-node joins on multi-GPU setups [114]. Lutz et al. [88, 89] evaluated the transfer bottleneck of GPU-accelerated single-node joins against the faster interconnect NVLink 2.0. They found that such interconnects can greatly increase the performance of the GPUs as co-processors both for joins smaller and larger than GPU memory. An interesting future route would be to combine GPUDirect and NVLink. However, currently RDMA NICs are only available with PCIe, and GPUDirect cannot be combined with Nvidia's interconnect NVLink.

Another approach by Shanbhag et al. [119] aims to remove the transfer overhead to the GPU by instead using the GPU as the main processor by storing the working set directly on the GPU. In this setup, analytical processing on the GPU greatly outperforms the CPU but still restricts the total working set sizes for main-memory DBMSs. Targeting a similar setup with GPUs as the main processor, Paul et al. [107] focuses on optimizing the communication paths of transfers between multiple GPUs through their multi-hop algorithm to maximize the cross-sectional bandwidth between GPUs.

Distributed Joins: . In the context of high-speed networks, join processing for scale-out distributed DBMSs has been studied by a few works [9, 10, 38, 39]. Barthels et al. [10] implemented a distributed radix hash join over RDMA networks by utilizing efficient one-sided RDMA primitives. While the authors do not explore GPU-acceleration, many findings of their work are still applicable, such as the efficiency of one-sided RDMA.

The approach by Guo et al. [43] is closest to our work, which also explores distributed GPU joins over RDMA. However, they cover only more naive blocking GPU joins. Moreover, they argue that remote transfer via GPUDirect from and to GPUs is less efficient than using RDMA with CPUs. Our findings are, however, different. Initially, we observed the same unbalanced bandwidth characteristics for RDMA between GPUs and CPUs. The reason for this is that when the NIC and GPU are not placed under the same PCIe switch, a suboptimal performance can be observed as also reported in [44].

10.8 Conclusion & Future Work

In this paper, we present two novel join algorithms for accelerating distributed joins on high-speed networks with GPUDirect RDMA. We show how our pipelined GPU join can speed up distributed joins up to 2× over a state-of-the-art CPU-based join while supporting arbitrary large probe-side tables. To accelerate joins with build-side tables larger than the collective GPU memory, we present a hybrid join that transparently leverages both GPUs and CPUs for joining the tables. In the context of complete queries, we show that pipelining and the support of successive operations in our distributed GPU join algorithm can additionally speed up the overall execution by up to 6× against a non-accelerated CPU-only version.

In the future, we aim to explore the combination of also involving SSDs for very data-heavy operations such as out-of-memory joins through GPUDirect Storage [97] which allows the GPU to directly access storage without relying on the main-memory.

10.9 Acknowledgements

This work was partially funded by the German Research Foundation (DFG) under the grants BI2011/1 & BI2011/2 (DFG priority program 2037), the DFG Collaborative Research Center 1053 (MAKI) as well as hessian.AI, 3AI and gifts from Mellanox. In addition, we gratefully acknowledge support from the Federal Ministry of Education and Research (BMBF) under Grant No. 01IS22091.

11 Zero-sided RDMA: Network-driven Data Shuffling

Abstract

In this paper, we present a novel communication scheme called zero-sided RDMA, enabling data exchange as a native network service using a programmable switch. In contrast to one- or two-sided RDMA, in zero-sided RDMA, neither the sender nor the receiver is actively involved in data exchange. Zero-sided RDMA thus enables efficient RDMA-based data shuffling between heterogeneous hardware devices in a disaggregated setup. In our initial evaluation, we show that zero-sided RDMA can outperform existing one-sided RDMA-based schemes due to offloading the coordination to the network and new optimizations that are only possible by coordinating the data exchange on the switch.

Bibliographic Information

The content of this chapter was previously published in the peer-reviewed work: Matthias Jasny, Lasse Thostrup, and Carsten Binnig. “Zero-sided RDMA: Network-driven Data Shuffling.” In: *Proceedings of the 19th International Workshop on Data Management on New Hardware, DaMoN 2023, Seattle, WA, USA, June 18-23, 2023.* Ed. by Norman May and Nesime Tatbul. ACM, 2023, pp. 82–85. DOI: 10.1145/3592980.3595302. URL: https://doi.org/10.1145/3592980.3595302.

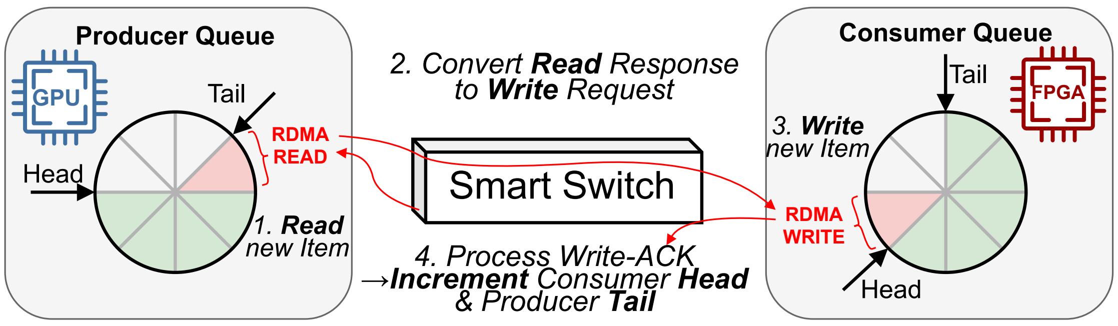


Figure 11.1: Zero-sided RDMA. The switch transfers items using RDMA from the producer to the consumer queue. Producers/consumers interact with their queues via local memory, without the need for any communication primitives.

11.1 Introduction

In the past decade, the landscape of DBMSs has undergone significant transformation. Notably, there has been a shift towards disaggregation, separating different resources such as compute and storage or accelerator pools from traditional compute [11, 28, 81, 131]. This approach offers improved resource utilization, as each resource can be scaled independently based on demand. However, it also means that the network has become more critical for data access, which has led to a significant focus on reducing the cost of data transfers. To this end, there has been considerable attention on developing fast network and kernel bypass libraries.

A technology that is thus often used in cloud data center networks is Remote Direct Memory Access (RDMA). RDMA has been shown to provide state-of-the-art performance for many different database components such as operators [10, 126], concurrency control [28] and storage layer [154]. One advantage of RDMA is the availability of one-sided primitives, namely READ and WRITE operations. These primitives solely involve the CPU of the initiator node, rather than the target node. As a result, nodes can read from and write to remote memory locations without requiring the remote CPU to be actively involved, as the remote NIC handles the read and write operations. This offloading of network overhead to the NIC enables efficient scaling and improved performance [156].

A prominent development in the field of databases is the incorporation of heterogeneous hardware accelerators such as GPUs [88] and FPGAs [75]. This trend is a response to the performance stagnation of CPUs due to the end of Moore's Law and Dennard Scaling. It is anticipated that CPUs will eventually be unsuitable for data processing as specialized hardware can provide superior performance and power efficiency [95].

Today, in distributed setups, accelerators such as GPUs and FPGAs typically need to be coupled with a CPU which is responsible for the control and communication over RDMA. This approach, however, limits the acceleration potential due to the CPU involvement in the control flow which can lead to under-utilized devices and sub-optimal performance [128]. Moreover, coupling CPUs and accelerators can also lead to underutilized CPUs if they only need to handle communication between devices.

However, moving the control flow and providing efficient communication between specialized hardware devices without involving the CPU as an intermediate step for the control path is a non-trivial task for two reasons: (1) The availability of the communication primitives (e.g., one-sided RDMA) might not be available for the given hardware, and (2) implementing the complete communication logic is intricate and likely varies for each hardware type. Moreover, this consumes compute resources that could otherwise be utilized for processing.

We argue that devices will continue to grow increasingly heterogeneous in the future, and the demand for device disaggregation and interconnection will increase. In this paper, we thus address the resulting challenge of enabling efficient communication between diverse specialized hardware devices without the complexity that arises from implementing RDMA communication schemes on these devices for a wide range of architectures. Instead, our approach fully offloads the communication logic to the network, making it unnecessary to port communication primitives to each specialized hardware device.

The main idea of offloading the communication logic to the network is that the switch acts as a coordinator of data transfers; i.e., it issues RDMA READs to a sender and rewrites the read response into an RDMA WRITE for the receiver using the programmable data plane of the switch. Since this does not incur any involvement of neither the sender nor receiver, we term this communication scheme *zero-sided RDMA*. A programmable switch presents an excellent choice for implementing a zero-sided communication scheme due to its central placement which allows it to optimally schedule data transfers. In addition, switches are able to process at the aggregated line-rate of all connected devices and thus provide a scalable solution to zero-sided RDMA, as we show in our evaluation.

11.2 Zero-sided RDMA

The overall flow of transferring a data item using zero-sided RDMA is illustrated in Figure 11.1. The core abstraction of zero-sided RDMA is a circular queue buffer implemented on both the producer- and consumer-side of a unidirectional communication channel. The design goal of the queues is to allow processing units (PUs) to push and pop items with only simple local memory operations while the switch transfers data fully asynchronously without any sender/receiver PU involvement. By handling communication centrally on the switch, communication patterns can be realized efficiently without coordination between PUs.

Communication Scheme & Adaptive Batching. For communication, we mirror the state of the producer and consumer queues (head and tail pointers) as state on the switch to know when a producer has an item to send and whether the consumer has free space. To transfer an item, (1) the switch first issues an RDMA READ on the producer. (2) The READ response is then converted on the switch into an RDMA WRITE and (3) written into the next free slot at the consumer. When converting a READ response into a WRITE request, the switch does not need to buffer or modify the data payload but changes the RDMA header on-the-fly in the data plane. (4) After the RDMA WRITE is acknowledged by the remote NIC, the head and tail pointers of the producer and consumer queues are updated accordingly. For transferring data, the switch polls the state of the producer and consumer queues at configurable intervals to detect whenever a producer has new items or the consumer has consumed items. Since the producer might have written multiple items to its queue between the reads to the head-pointer by the switch, we apply adaptive batching such that all available items that fit into the consumer queue are transferred in one big RDMA READ in step (1).

Reliable Transport & Congestion Control. An advantage of offloading the zero-sided communication scheme entirely into the switch's data plane is that the switch can handle line-rate data transfers between connected devices in a scalable manner [60, 61]. For coordination, the packet processing pipeline of the switch maintains the state in registers. We store the necessary state to adhere to the RDMA protocol and the state required to manage the queues of the PUs. During the initialization of the communication channel, the switch sets up reliable RDMA connections to each PU.

For ensuring reliable transport, RDMA uses sequence numbers and acknowledges each request to ensure the reliability of the transport and to be resilient to drops in the network. If a drop occurs, the switch detects it through a configurable timeout or gaps in the sequence numbers and subsequently reissues the data-transfer request (i.e., RDMA

READ on the producer device). However, if the drop happened in the last step (step 4 in Figure 11.1), the previously successful data transfer is not reissued and only the RDMA WRITEs of head and tail pointers are resent.

Furthermore, to handle incast scenarios (e.g. two senders with 2x bandwidth and a receiver with 1x bandwidth), we integrate congestion control into our zero-sided communication scheme on the switch. The switch emits explicit congestion notifications whenever a link becomes congested, similar to RoCEv2 [104]. When the NIC of the producer receives the congestion notification, it throttles down the rate of outgoing packets.

Hardware Requirements. Finally, zero-sided RDMA can easily be used by many heterogeneous devices since only a few requirements must hold for participating in zero-sided RDMA: (1) the device must have memory in which to store the queue data structure, (2) the memory must be accessible by an off-the-shelf RDMA-enabled NIC and (3) the memory consistency model must ensure that a write to an item and the subsequent update to the head pointer is executed in order (which is commonly the case).

Contrary to technologies like NVLink that are used for GPU communication, zero-sided RDMA communication can stretch across a variety of different processing units and is not limited to a vendor specific extension.

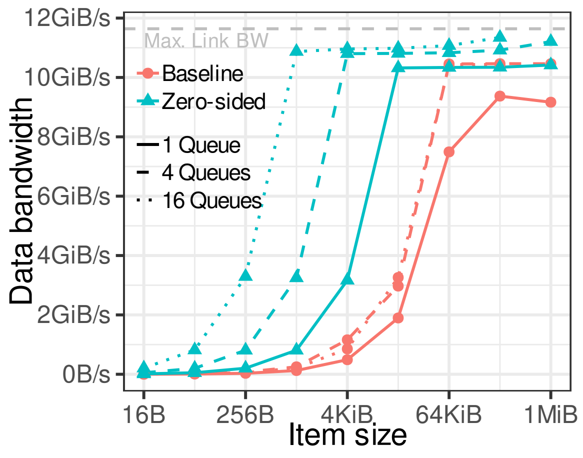
Integration with the DBMS. The DBMS query scheduler and optimizer are responsible for devising an execution plan and mapping the plan to different PUs. This entails sending the parameters of each queue, along with lists of producers and consumers, to the switch. The switch establishes direct RDMA connections with all producers and consumers. As soon as all connections are set up, the switch will initiate the data transfers.

When a producer completes its tasks, it tears down the RDMA connection. After the tear down of all producer connections for a given flow, the switch cleans up and tears down the consumer-side connection.

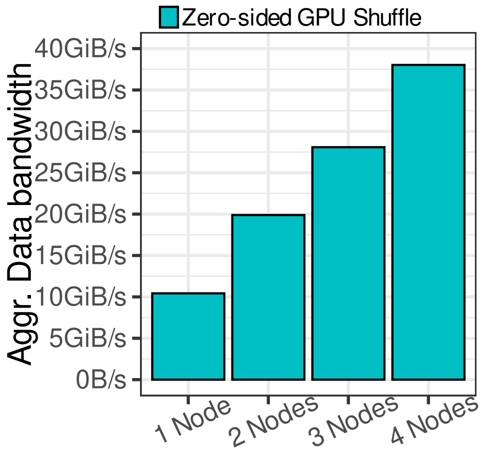
11.3 Initial Results

We evaluate our zero-sided RDMA communication scheme in a cluster of 4 nodes, each with an Nvidia V100 GPU and connected to an Intel Tofino 1 switch via 100G ConnectX-5 NICs with RoCEv2.

In Figure 11.2a, we compare the bandwidth of zero-sided data transfers directly from one GPU to another with a CPU-driven scheme where the CPU drives the communication between the GPUs using one-sided RDMA. This baseline is a common GPU



(a) 1:1 Communication



(b) N:N Shuffle

Figure 11.2: (a) GPU to GPU Shuffling. Zero-sided RDMA compared to a CPU-driven scheme. (b) Many-to-many Shuffling between N GPU nodes using zero-sided RDMA.

communication scheme where a CPU launches GPU kernels and subsequently initiates data transfers directly from local to remote GPU memory using GPUDirect [98]. To show the scalability of both setups, we increase the number of concurrent RDMA queues on the sender/receiver. We see that zero-sided RDMA is superior in performance. This is due to two factors: The producer GPU can process data without CPU involvement or synchronization. Secondly, our zero-sided scheme applies adaptive batching of multiple items into larger data transfers. Moreover, with additional queues, the zero-sided design achieves maximum performance earlier since the switch does not limit the scalability due to its processing model, unlike CPU-driven designs. In contrast, kernel launch and synchronization overhead limit the baseline's performance.

In Figure 11.2b, we report the data bandwidth for a GPU-to-GPU shuffle scenario while scaling out across multiple nodes. We use item sizes of $512KiB$ and execute 2 GPU kernels for producers and 2 for consumers on each GPU node. The producers shuffle data to all consumers on all nodes using zero-sided RDMA. After launching the GPU kernel no additional CPU or GPU cycles of the host system are spent on executing communication primitives. A benefit of the zero-sided approach is that each PU connects directly to the switch. This way, connection complexity is reduced because consumers can receive data from multiple producers through a single in-going connection (i.e., one RDMA queue) instead of separate ones. The observed bandwidth increases near-linearly for each added node because the switch handles all data transfers for each queue independently.

For shuffle scenarios with more queue end-points, our prototype implemented on the Intel Tofino 1 switch can handle thousands of queues in parallel due to its pipeline processing model without compromising performance.

11.4 Future Work

We envision a range of different extensions to our zero-sided communication scheme for supporting and offloading typical use cases. With the centralized position of the switch, a set of ample opportunities for DBMSs exist which we iterate next.

Load Balancing & Elasticity. Distributing work evenly across processing units in a distributed DBMS is non-trivial given unforeseen network congestion or processing contention. Multiple schemes have been devised to overcome this [31, 77, 155], which require additional coordination overhead, e.g., through work-stealing or a centralized server-side dispatcher. However, we envision directly supporting zero-sided load balancing by letting the switch initiate data transfers from multiple producers to multiple consumers, where each consumer processes and consumes items at its own speed without any need for coordination.

A natural extension to load balancing is elasticity, allowing to adjust computational resources by modifying the number of producers or consumers during runtime, thereby enabling the system to scale up or down as per demand. In zero-sided RDMA, the switch assumes the sole responsibility of driving data transfers with the capability to add or remove producers or consumers as needed without additional synchronization between the participants.

Quality of Service. Ensuring a certain quality of service or prioritization of parts of the network traffic (e.g., for a time-critical query) is hard as Priority-based Flow Control (PFC) requires network reconfiguration and only provides a means of prioritizing different classes of traffic. As such, PFC does not allow setting fine-grained prioritization (e.g., for each sending queue) and adapting these settings at runtime. Therefore, we envision extending our zero-sided communication scheme with fine-grained (per-queue) configurations to be able to ensure that a queue gets a certain slice of the network bandwidth. Specifically, we can achieve this by using explicit congestion notifications[1] in the switch to control the share of the network given to each flow.

Zero-sided Multicast. Multicast has multiple applications in distributed DBMS such as replicated joins [126] or state replication [51, 82] for providing availability. Our aim

[1]These notifications are supported e.g. by RoCEv2.

is to integrate zero-sided multicast capabilities, such that in a flow between multiple producers and consumers, all consumers will receive the same data generated by the producers. This approach would greatly improve on the existing capabilities of RDMA multicast which is only supported through unreliable transport and two-sided verbs which comes with the cost of higher CPU overhead at the communication endpoints due to the two-sided communication and the cost of ensuring reliability.

With the centralized switch approach, the switch can not only support multicast by replicating packets in the egress pipeline but also ensure that the data items replicated to all consumers will be globally ordered. Traditionally, effects like reordering of packets in the network can cause the received data at each consumer to observe a different order. However, as our zero-sided approach transfers data sequentially with separate acknowledgments, we are able to ensure globally ordered data transfers to all consumers without introducing any compute or coordination overhead on the producer or consumer side.

Integration of Heterogeneous Devices. Ultimately, we plan to build a CPU-less DBMS that uses zero-sided RDMA to span across diverse heterogeneous devices, showing the benefit of network-driven communication. As such, we aim to demonstrate the benefits of zero-sided communication with other hardware accelerators such as FPGAs. While existing RDMA libraries already exist for FPGA-based SmartNICs, the benefit of zero-sided RDMA communication is that only a small subset of the RDMA stack needs to be supported by the FPGA. In fact, if combining an FPGA with an RDMA-NIC, the RDMA stack can be completely avoided when using zero-sided RDMA. In addition, the communication and coordination logic is offloaded from the device which frees up valuable FPGA resources and significantly reduces engineering effort.

11.5 Acknowledgements

This work was partially funded by the German Research Foundation (DFG) under the grants BI2011/1 & BI2011/2 (DFG priority program 2037) and the DFG Collaborative Research Center 1053 (MAKI). We thank Intel for their valuable technical support.

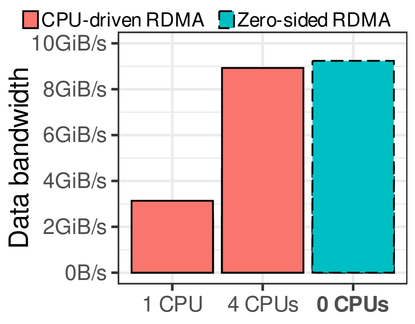
12 Zero-sided RDMA: Network-driven Data Shuffling for Disaggregated Heterogeneous Cloud DBMSs

Abstract

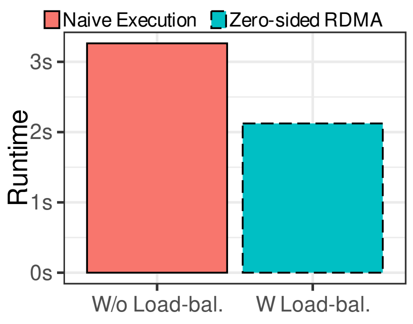
In this paper, we present a novel communication scheme called zero-sided RDMA, enabling data exchange as a native network service using a programmable switch. In contrast to one- or two-sided RDMA, in zero-sided RDMA, neither the sender nor the receiver is actively involved in data exchange. Zero-sided RDMA thus enables efficient RDMA-based data shuffling between heterogeneous hardware devices in a disaggregated setup without the need to implement a complete RDMA stack on each heterogeneous device or the need for a CPU that is co-located with the accelerator to coordinate the data transfer. As such, we think that zero-sided RDMA is a major building block to make efficient use of heterogeneous accelerators in future cloud DBMSs. In our evaluation, we show that zero-sided RDMA can outperform existing one-sided RDMA-based schemes for accelerator-to-accelerator communication and thus speed up typical distributed database operations such as joins.

Bibliographic Information

The content of this chapter was published in the peer-reviewed work: Matthias Jasny, Lasse Thostrup, Sajjad Tamimi, Andreas Koch, Zsolt István, and Carsten Binnig. “Zero-sided RDMA: Network-driven Data Shuffling for Disaggregated Heterogeneous Cloud DBMSs.” In: *Proc. ACM Manag. Data* 2.1 (Mar. 2024). DOI: 10.1145/3639291. URL: https://doi.org/10.1145/3639291.



(a) GPU-to-GPU transfers, CPU-driven vs. network-driven transfers.



(b) Perfect utilization in heterogeneous execution environments.

Figure 12.1: Motivating examples of using zero-sided RDMA for a GPU-to-GPU shuffle of 4KiB data items. (a) Zero-sided RDMA provides equal or better performance than traditional CPU-based schemes without the need of co-locating a CPU per accelerator. (b) Zero-sided RDMA enables seamless load balancing of data transfers in case accelerators consume data at different speeds.

12.1 Introduction

Disaggregation and the need for RDMA. In the past decade, cloud computing and the landscape of database systems have undergone a significant transformation. Notably, there has been a shift towards disaggregation, separating compute and storage or even accelerator pools from traditional compute [11, 28, 81, 131] because disaggregation offers improved resource utilization, as each resource can be scaled independently based on demand. This trend has been transformative for cloud-native DBMSs, which all build on top of such disaggregated architectures. Since, in disaggregated architectures, the rate at which data can be moved across nodes is one of the most important factors to performance, there has been considerable attention on developing fast network solutions. Remote Direct Memory Access (RDMA) is deployed by major cloud vendors to enable efficient resource disaggregation [5, 118]. In database use cases, RDMA has successfully improved the performance for distributed join operators [10, 126], reduced the cost of concurrency control [28], and made storage access [154] possible at unprecedented speeds.

The need for heterogeneous compute. Another significant trend in cloud data centers is that, in response to the performance stagnation of CPUs due to the end of Moore's Law and Dennard Scaling, heterogeneous accelerators are becoming commonplace as an alternative to CPU-based compute resources [95]. Accelerators have been shown to provide significant speed-ups for DBMS workloads [75, 88]. However, looking ahead, a major question for future cloud DBMSs is how to efficiently use disaggregated heterogeneous compute resources such as GPUs or FPGAs. Similar to disaggregated CPU-based compute resources, these accelerators should be deployed as network-attached accelerator pools, but today this is not the case. Accelerators depend on the presence of a host CPU for both control and moving data in and out of the accelerator. As we show in this work, if we continue to keep the CPU on the critical path of data movement, we will not be able to take advantage of fast networks and RDMA-based communication. Instead, we propose zero-sided RDMA as a solution to remove the CPU from the critical path and enable efficient accelerator-to-accelerator data movement powered by a smart switch.

The pitfalls of RDMA today for accelerators. To explain how zero-sided RDMA works, it is important to see how RDMA involves CPUs today. In addition to two-sided operation, where the CPUs of both the sender and receiver nodes are involved, it already offers one-sided primitives, namely READ and WRITE operations. These primitives solely involve the CPU of the initiator node rather than the target node. As a result, nodes can read from and write to remote memory locations without requiring the remote CPU to be actively involved, as the remote NIC handles the read and write operations. This offloading of network overhead to the NIC enables efficient scaling and improved performance [156]. However, if we want to fully utilize accelerators in the cloud, the involvement of CPUs, even if just on the sender side, will lead to bottlenecks (as seen in Figure 12.1a, multiple CPU cores need to be dedicated to coordinating data transfers).

Accelerator-driven RDMA as an alternative? One way to remove the CPU would be to implement RDMA-based operations directly on the accelerators. Even though this is technically possible [73], it is challenging for the following reasons: (1) RDMA primitives, such as one-sided RDMA verbs, might not be available for a given accelerator, requiring a full RDMA stack to be implemented per accelerator type. Beyond the engineering cost, the additional problem is that executing communication logic on specialized hardware devices consumes compute resources that could otherwise be utilized for processing. (2) Implementing advanced RDMA-based communication schemes, such as many-to-many data shuffling or multicast, on top of the RDMA stack on each accelerator type requires re-implementing features and has different challenges and limitations on each hardware type.

The case for network-driven RDMA. To remove the CPU from the critical path of performance and to reduce the engineering effort in disaggregated accelerators, in this work, we propose a novel network-driven scheme where neither a CPU co-located with an accelerator is actively needed nor the GPU/FPGA has to implement the RDMA stack. Our approach fully offloads the RDMA stack and the RDMA-based communication logic between devices to the network, particularly to a programmable switch. The main idea of offloading the communication logic to the network is that the switch acts as a coordinator of data transfers; i.e., it issues an RDMA READ to a sender and rewrites the read response into an RDMA WRITE for the receiver using the programmable data plane of the switch. Since this does not involve the sender or receiver actively, we term this communication scheme *zero-sided RDMA*.

Why use a programmable switch? To implement zero-sided RDMA, a programmable switch presents an excellent choice due to its placement as an intermediate in the network, which allows it to support complex data transfer operations, such as distributed data shuffling used in databases, and optimally schedule data transfers. Important is that zero-sided RDMA is not limited to point-to-point communication (1:1) and, as we show later in this paper, we can support various communication flows for common distributed data management use cases, namely N:M shuffles, as well as complex distributed operations such as network-driven reliable multicast or seamless load-balancing. From a complexity perspective, this is much more efficient than implementing the RDMA stack and shuffling logic in each instance of an accelerator (i.e., FPGA, GPU). Finally, switches can process at the aggregated line-rate of all connected devices [60] and thus provide a scalable solution to zero-sided RDMA, as we show in our evaluation.

Motivating examples. Figure 12.1 shows the main benefits of our novel switch-driven scheme that uses zero-sided RDMA for data transfers. First, compared to the CPU-driven scheme (Figure 12.1a), zero-sided RDMA can achieve close to the maximum network bandwidth without requiring a co-located CPU per accelerator. In fact, the CPU-driven scheme requires multiple CPU cores to achieve the same bandwidth for transferring messages. Moreover, zero-sided RDMA provides many other benefits, such as seamless load-balancing (Figure 12.1b). Load-balancing is important if multiple (heterogenous) receivers consume data while one of the receivers is slower than the others. As we later discuss, our switch-driven scheme can detect such cases by monitoring the progress of a data shuffle and seamlessly redistributing data.

Relevance for DBMSs & contributions. The database community continuously seeks ways to adapt to the complexities of disaggregation and the integration of RDMA with diverse accelerators. In this context, zero-sided RDMA presents a practical solution by enabling network-driven data transfers between heterogeneous accelerators in DBMSs. We evaluate our idea on specific DBMS use cases focusing on OLAP scenarios, including data shuffling (i.e., 1:1, N:M), a core task in distributed query processing. In addition to the chosen shuffle scenarios, zero-sided RDMA can be applied to many more, including data replication or efficient data transfers in disaggregated databases to and from compute.

As our experimental results on latency indicate, it will be possible in the future to extend zero-sided RDMA to also support latency-critical distributed workloads beyond OLAP, such as OLTP or data streaming.

In summary, our work on zero-sided RDMA brings the following contributions: (1) First, we introduce our novel network-driven communication scheme — zero-sided RDMA — which allows efficient accelerator-to-accelerator communication without CPU involvement. (2) We use zero-sided RDMA to implement a network-driven data shuffling operator for distributed DBMS use cases. We demonstrate that it achieves line-rate performance for different communication patterns (i.e., 1:1, N:M shuffles). (3) We showcase how complex communications schemes, such as globally-ordered replication and load-balancing for multi-producer to multi-consumer, can be implemented efficiently in zero-sided RDMA. (4) Finally, we evaluate zero-sided RDMA through a set of DBMS-centric scenarios, such as a distributed TPC-H query, shuffling, and a distributed join, as well as using zero-sided RDMA for two different types of accelerators: GPUs and FPGAs.

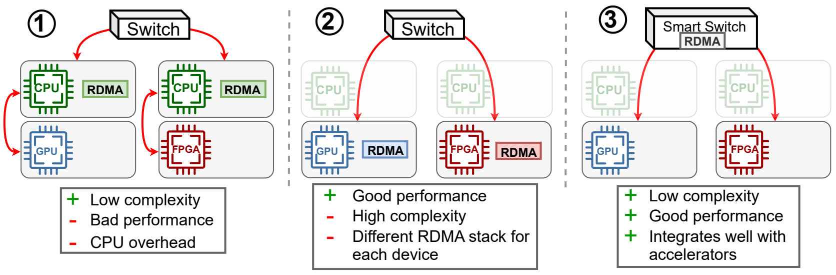
12.2 Background

12.2.1 Remote Direct Memory Access (RDMA)

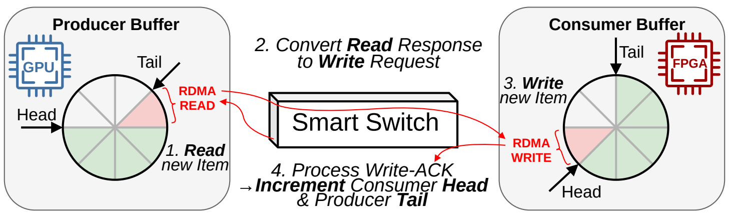
RDMA has become the state-of-the-art communication method for distributed data-processing systems over high-speed networks [11, 138, 147, 148, 157]. Its main benefit is that it removes the overhead of traditional kernel-space network stacks such as TCP/IP. Major cloud vendors have already adopted RDMA in their pursuit of faster networking with little CPU overhead. An example of this is Microsoft Azure which reports that already around 70% of internal ToR traffic is RDMA [5].

Communication schemes in RDMA can be categorized as one-sided (READ / WRITE) or two-sided (SEND / RECEIVE) operations, which refer to the involvement of the sender and receiver in the communication. For one-sided operations, only the sender is actively involved and thus has to decide where the data should be placed on the remote node. With two-sided operations, also the receiver is actively involved in the communication and decides where to place data by issuing RECEIVE requests before SEND requests can be issued on the sender side. This simplifies remote memory management.

Especially the one-sided RDMA operations have seen high adoption in distributed data processing systems since they allow sender nodes to write into remote memory directly fully bypassing CPU cores of the receiving nodes [10, 156]. However, even with one-sided



(a) CPU- vs. Accelerator- vs. Network-driven (zero-sided RDMA).



(b) 1:1 Flow with zero-sided RDMA.

Figure 12.2: Network-driven communication with zero-sided RDMA vs. alternative communication schemes. (a) In a CPU-driven scheme ①, the CPU is responsible for carrying out the communication. In the accelerator-driven scheme ②, the RDMA stack is realized directly on the accelerator. With zero-sided RDMA ③, the RDMA stack and communication scheme are fully offloaded into the network, removing the need for CPUs, while only one RDMA stack (e.g., on the switch) is needed. (b) In zero-sided RDMA, producers/consumers interact only with their buffers in local memory. The switch reads items from the sender buffer (step 1) and moves them to the receiver buffer (step 3) by converting RDMA READS into WRITES (step 2). The switch coordinates the communication using head/tail pointers (step 4).

RDMA, achieving direct accelerator-to-accelerator communication is often not possible or infeasible. As such, the communication control flow must be relayed over the CPU [128].

Finally, to understand our zero-sided RDMA contribution, which directly processes RDMA traffic in the network layer, we want to mention that RDMA communication can run over different transport types, such as Reliable Connection or Unreliable Datagram. The most commonly used transport is the Reliable Connection which requires stateful connections between any end-points. The reliability is provided through acknowledgments and re-transmissions. While Unreliable Datagram has the lowest complexity and overhead, most systems require reliable data transfers and need to implement it manually in the application with added CPU overhead [66, 68]. In addition, Unreliable Datagram only supports two-sided operations. In this paper, we exclusively use Reliable RDMA Connections, as one-sided operations are essential to achieving network-initiated communication, and reliability is vital for most systems.

12.2.2 Programmable Switches

Programmable switches have gained traction in data centers for many years. Unlike traditional fixed-function switches, they offer flexible, line-rate packet processing capabilities of up to billions of packets per second. These are primarily attributed to the programmable packet processing pipeline. In networking, the control plane and data plane are two fundamental components that handle different responsibilities. The control plane, usually running on a CPU, is the brain of the network device, making decisions about where traffic should be directed. In contrast, the data plane is responsible for to carry out actual operations on network packets and routing traffic to different destinations.

The programmability of the data plane is given by utilizing a reconfigurable architecture based on match-action tables, extending the utility of switches beyond data routing to include the offloading of application logic via customized match-action rules. The programming of this data plane predominantly employs P4 (Programming Protocol-Independent Packet Processors) [16], a high-level language that allows users to draft match-action rules to define packet processing in the data plane. Initially conceived for programmable network switches, P4 is now applicable to numerous systems that process packets, including SmartNICs and FPGAs. P4 programs, in general, manipulate packet headers and specify their rewriting while adopting a C-like syntax without allowing complex constructs such as pointers, floating-point numbers, or loops for line-rate processing. Besides these constraints, however, many new opportunities can be used when engineering new algorithms for this platform. The most important thing to note is that every compiled P4 program can run at line-rate in the switch, which is an important aspect of implementing zero-sided RDMA in the switch.

12.3 Overview of Zero-sided RDMA

12.3.1 Why Network-driven Communication?

Hardware accelerators are increasingly used for data-intensive processing tasks as they provide very high processing power compared to the stagnating performance of CPUs. As such, a lot of work is put into incorporating accelerators such as GPUs and FPGAs into DBMS [33, 119, 123]. However, when considering the trend of disaggregation and the adoption of fast RDMA-capable networks in data centers, the challenge of bringing these two trends together is of growing importance.

CPU-driven data shuffling. When looking at how distributed accelerator communication is typically done today, we see that it is predominantly CPU-driven. We illustrate this in ① in Figure 12.2a. Here, the CPU handles communication control flow while processing tasks are offloaded to the accelerators. This design makes sense because accelerators are typically built with specific processing tasks in mind and do not cope well with the control-flow-heavy type of communication handling, and in addition, the RDMA library stack is readily available and built for the CPU. However, this configuration causes a tighter coupling between the CPU and accelerator, which can hinder accelerator utilization due to stalling and CPU-to-accelerator communication overhead. At the same time, it increases the CPU load, which can be substantial considering the task of saturating fast networks and accelerator processing.

Accelerator-driven data shuffling. A common solution in related work to overcome the limitation of design ① is to implement the RDMA stack (i.e., `libibverbs` library) directly in the accelerators as illustrated in ② in Figure 12.2a. This allows the accelerators to directly execute RDMA and communicate over the network without CPU involvement [1, 26, 73]. While this approach has strong merits as it removes the strong coupling between the CPU and accelerator, we argue that in many cases, such a configuration might be suboptimal or even impossible for the following reasons: (1) As previously mentioned, accelerators are built with a specific processing architecture, e.g., the massively parallel SIMT execution model by GPUs. Therefore, they might perform very poorly on the control-heavy and often non-parallelizable task of coordinating the communication. (2) In many cases, the RDMA stack might not even be available for a particular type of accelerator. Developing a new RDMA stack requires high efforts that not only slow down the adoption of accelerators in data centers but might also consume a significant fraction of the available hardware resources on accelerators. This is in particular problematic for FPGAs since this limits how much resources are available for the actual application. (3) Realizing distributed operations, such as shuffling, with one-sided RDMA on an accelerator is non-trivial and requires the re-implementation of the same logic for each accelerator type leading to increased system complexity.

Network-driven data shuffling. For these reasons, it is a highly complex task to develop a communication system that relies on implementing the communication logic directly in the heterogeneous accelerators. We take a different approach by separating the data processing and communication logic by offloading the whole communication scheme into the network. This overcomes the challenges mentioned above by taking away the burden on the accelerator side and providing a clean separation of concerns. Moreover, with programmability in the network, CPUs can be removed from the critical path of communication between accelerators. Programmable network components typically come in the form of SmartNICs or programmable switches. While both types can be used to drive communication on behalf of the accelerator, a programmable switch has many benefits over SmartNICs. Among these reasons is guaranteed line-rate processing for all connected nodes and the possibility of realizing coordination-free communication schemes due to the centralized position of the switch. We further discuss this in Section 12.6.3.

12.3.2 How Does Zero-sided RDMA Work?

Our approach to achieving zero-sided RDMA builds on the ability of the centralized programmable switch to initiate and direct data traffic directly in the data-plane at the aggregated line rate of all connected accelerators — called processing units (PUs) in the sequel. By placing the communication scheme in the switch data-plane, the switch can initiate one-sided RDMA operations (READ & WRITE) to PUs. At the same time, the PUs are completely oblivious to the network communication scheme, which not only has the benefit that the PUs do not have to issue communication primitives actively but also removes the distributed coordination for communication flows such as data shuffling or even replication.

Communication abstractions. On the side of the PUs (i.e., accelerators), a circular buffer is the core abstraction to participate in zero-sided RDMA data transfers. The design goal of the buffer is to allow PUs to push and pop items with only simple local memory operations while the switch transfers data fully asynchronously without any sender/receiver PU involvement. The coordination between the switch and PU is handled

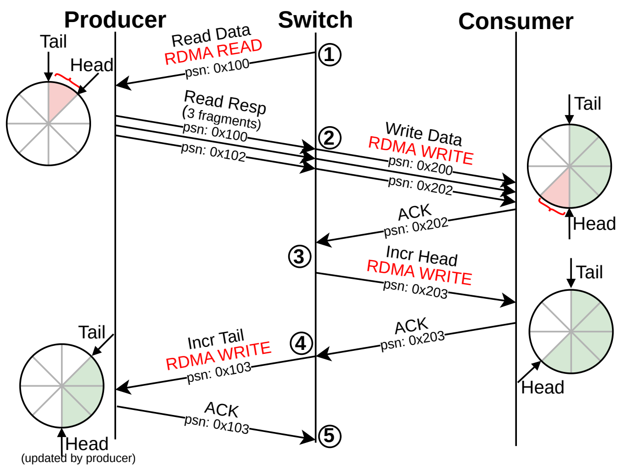


Figure 12.3: Sequence of network packets for a switch-driven 1:1 data flow. The switch transfers data by reading from the producer and converting the READ response into a WRITE to persist the data in the consumer memory.

with two pointers to the buffer, a head and tail pointer. The head pointer indicates where the next data item can be written, and the tail pointer indicates where the next data item can be read.

Flow of data transfers. The overall flow of transferring a data item between one producer and one consumer using zero-sided RDMA is illustrated in Figure 12.2b. The switch mirrors the state of the producer and consumer buffers (head and tail pointers) to know when a producer has an item to send and whether the consumer has free space. To transfer an item, (1) the switch first issues an RDMA READ on data in the producer buffer. (2) The READ response is then converted on the switch into an RDMA WRITE and (3) written into the next free slot at the consumer. When converting a READ response into a WRITE request, the switch does not need to buffer or modify the data payload. (4) After the remote NIC acknowledges the RDMA WRITE, the head pointer of the consumer is incremented to indicate the new item. The tail pointer is incremented on the producer, which frees up the item in the buffer for reuse. We cover the detailed design in Section 12.4 to enable zero-sided data transfers.

Hardware requirements. Zero-sided RDMA can easily be used by many heterogeneous devices since only a few requirements must hold for participating in zero-sided RDMA: (1) the device must have memory in which to store the buffer data structure (2x 4 bytes for head/tail pointers and memory for data items), (2) the memory must be accessible by an off-the-shelf RDMA-enabled NIC and (3) the memory consistency model must ensure that a write to an item and the subsequent update to the head pointer is executed in order. In the case of weaker consistency models (like GPUs and ARM CPUs), memory fences can be used to ensure the ordering of writes. (4) In addition, to avoid stale reads, accelerators with caches must ensure that local reads and writes to the head and tail pointers of the buffer are directed to the RDMA-registered memory. For Nvidia GPUs this is achievable with the `volatile` keyword. We support accelerators with both little and big endianness. The switch processing pipeline operates in big-endian and as such to support little-endian systems, we perform endian conversion (if needed) on the switch in order to process the head and tail pointer values.

Software requirements. In order to enable zero-sided RDMA from the software side, it must be possible to register the accelerator memory of the zero-sided buffers to the RDMA-enabled NIC and establish an RDMA connection between the switch and PU. This is not a performance-critical task and only has to be done once in the setup phase. Hence, a rather small CPU is sufficient to carry out these management-related tasks. Important is that this CPU is not involved in the actual data transfers.

As an example, we discuss the details to enable zero-sided RDMA for an FPGA (or any other PCIe device): The memory of an FPGA can be exposed through a PCIe bar register and registered to the RDMA NIC. Registering external memory of PCIe devices as RDMA-enabled memory is supported by the `libibverbs` library using PeerDirect [100]. This requires a kernel driver that implements `peer_memory_client` [90] for the given device. With that, the RDMA NIC can directly access the FPGA's memory through PCIe without the involvement of the CPU or operating system using peer-to-peer DMA.

12.3.3 Integration into a DBMS

We now cover the basic steps needed to integrate zero-sided RDMA into DBMSs. The most important question is when a zero-sided communication flow between accelerators is set up and torn down in a DBMS. Moreover, other important aspects of cloud DBMS are the support for elasticity of flows as well as the use of zero-sided RDMA in different DBMS architectures.

Setting up and tearing down zero-sided flows. Here, different variants are possible. One variant is that the communication flows are instantiated when a DBMS cluster is launched, and different buffers are reserved, allowing a cluster node to send data to any other node. However, we also support variants where flows are instantiated and deployed ad hoc. In such a scheme, before query execution, a coordinator node (e.g., using a CPU) has to initialize the communication flow by setting up the buffers on the processing units. Subsequently, it initializes the communication flow by sending the buffer locations to the switch. During execution, we stress that no involvement is needed by the coordinator (i.e., no CPU is involved anymore). The switch handles the tear-down of flows. Once it detects that all producers are finished, it propagates the information to consumers and closes the RDMA connections.

Supporting elasticity in zero-sided flows. In order to support elasticity, we additionally allow producers and consumers to seamlessly be added or removed anytime from the communication flow during execution. We show an experiment for this in our evaluation in Section 12.7. The benefit of switch-driven communication is that changes to the communication flow will not incur any connection changes to already connected processing units. In fact, in e.g., a load-balancing communication flow adding another consumer at runtime will be completely oblivious to the producers.

Support for different DBMS architectures. Zero-sided RDMA integrates well into common DBMS architectures such as shared-nothing and shared-storage. For shared-nothing architectures, data is predominantly shuffled between nodes storing partitioned tables. Zero-sided RDMA can effortlessly interconnect processing units through the provided communication schemes while saving communication-related overhead and complexity at the connected nodes. Shared-storage architectures are gaining traction in the cloud due to their flexibility to scale in and out by placing tables on specific storage devices. This architecture can also benefit a lot from zero-sided RDMA in the way that it facilitates direct storage-to-accelerator communication.

12.4 Switch-driven Data Transfers

To realize zero-sided RDMA, we have to enable a switch-driven data transfer scheme that transfers data from producers to consumers without their active involvement.

Core challenges of realizing zero-sided RDMA. A core challenge of realizing zero-sided RDMA is to map the data transfer logic to the pipelined execution model of a switch, which provides a limited set of instructions and memory per stage. In addition, since all connected PUs are not directly connected to each other but are connected to the switch, the switch must adhere to the exact protocol to be compliant with off-the-shelf RDMA NICs. This includes managing stateful RDMA connections with reliability and correctly propagating congestion- and flow-control information from the consumers to the producers within the switch's data-plane to guarantee execution at line-rate.

The following explains how zero-sided RDMA can be implemented on the switch for a 1:1 data transfer. In subsequent sections, we will explain how to extend zero-sided RDMA to more complex flows (e.g., N:M flows or load balancing).

Overview of a 1:1 flow. The overall sequence of network messages for a data transfer from a single producer to a single consumer is depicted in Figure 12.3. The switch logic can be grouped into five steps: in ①, the switch evaluates whether it is possible to initiate a data transfer, next ②, the response of the read, which can be in multiple fragments, is rewritten into an RDMA WRITE to the consumer, in ③ the ACK from the data write triggers a new RDMA WRITE to increment the head pointer of the consumer buffer. In ④, the tail pointer of the producer is incremented, and the subsequent ACK packet ⑤ is recirculated back to the head of the switch processing pipeline to step ①. With that, the switch continuously evaluates whether it is possible to issue a data transfer for a given producer-consumer pair. Recirculation is a technique that virtually connects an output port in the switch with an input port, such that packets can pass through the pipeline multiple times. This resembles a looping construct. We initiate this main control loop per flow upon successful completion of the connection setup of the PUs. In the following, we present how we initialize the data transfer in step ① and rewrite the RDMA READ into an RDMA WRITE in step ②. Furthermore, we discuss which state is needed on the switch for achieving RDMA communication.

① Initiating data transfers. As a first step (① in Figure 12.3), the switch evaluates whether it is possible to initiate a data transfer. The control flow for this step is illustrated in Figure 12.4. The switch uses a pipelined processing model which operates on a per-packet basis and executes logic in a sequence of stages. By accessing registers and applying simple match-action-rules (i.e., actions can be arithmetic operations in each stage), we can express the logic needed to decide whether a data transfer can be initiated by letting a packet traverse the pipeline.

In the ingress pipeline, the switch first checks whether the producer has items in the buffer to send and whether the consumer has space. If neither the producer has data nor the consumer space available, the packet is recirculated. Recirculating the packet back to the ingress is necessary because while-loops are not supported in the pipelined processing model. We use unique indexes for all producers and consumers to identify a connection between a specific producer & consumer and to access their respective registers, e.g., pointer values and packet-sequence numbers (PSNs). PSNs are used to detect missing or duplicate packets. With that information, we can construct communication schemes between multiple producers and consumers. We apply adaptive batching such that if a producer has generated multiple items that fit into the consumer, all the items will be

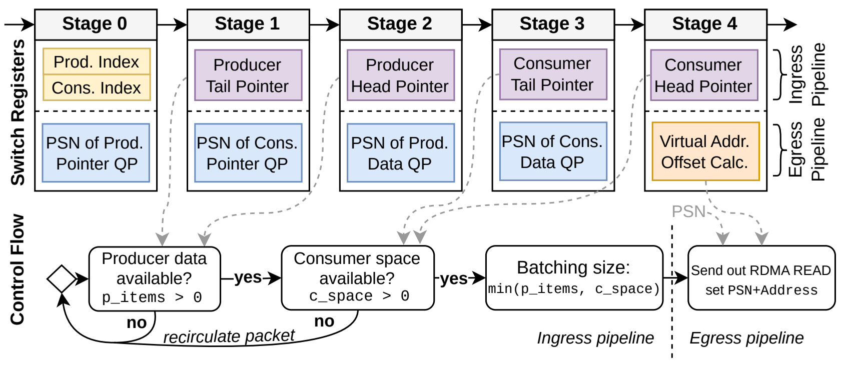


Figure 12.4: Switch state needed to initialize a data transfer using a pipelined execution model in the switch. The steps of reading head/tail pointers of the producer and consumer, deciding on the batch size of elements to transfer, and sending the data need to be mapped to a sequence of stages in the switch. All steps are executed sequentially.

transferred in one big RDMA READ. This optimization yields better efficiency as the overhead from control messages is relatively reduced. If these checks pass, the packet is forwarded to the egress pipeline, where necessary header fields are set to create an RDMA READ request which adheres to the RDMA protocol.

② Data payload rewrite. In the second step (② in Figure 12.3), the read response, containing the data payload to be transferred to the consumer, is streamed through the switch. Here the switch rewrites the response of the RDMA READ into an RDMA WRITE. Rewriting the RDMA operation type entails updating the relevant header fields and making it adhere to the correct packet sequence numbers and virtual address expected by the consumer. In terms of bandwidth consumption of the payload transfer from the producer to the consumer, rewriting the RDMA READ into an RDMA WRITE consumes the exact same bandwidth as a CPU-driven transfer would. Figure 12.3 step ② shows this rewrite by the switch for three fragment transfers of a payload.

The read response might be fragmented into several packets if the payload exceeds 1024 bytes, as this is the default maximum payload size (MTU) for RoCEv2. The challenge here is that each packet from the read response must be assigned a new packet sequence number (PSN) according to the consumer connection. Instead of naively assigning new PSNs to the RDMA WRITE packets in the order they arrive, we calculate an offset between the producer-side and consumer-side sequence numbers such that our protocol is resilient to any network reordering of fragments (e.g., a fragment with PSN `0x101` arrives on the switch after fragment `0x102`). As such, the assigned PSNs on the consumer side will correctly map to the right fragments, allowing the target consumer-side NIC to assert the right order for the memory write.

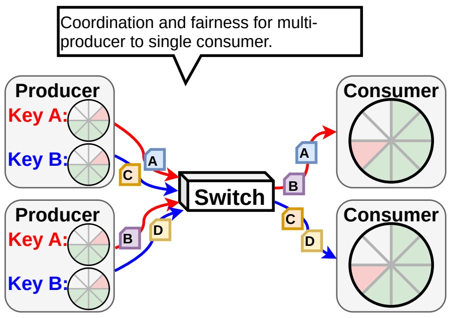
Memory requirements on the switch. The programmable switch in zero-sided RDMA does not need to buffer data items in its memory. This is because data transfers are only initiated when there is enough free space in the consumer buffer in step ①. In fact, the switch only rewrites the headers of incoming packets in a streaming manner (at line rate) such that data items are immediately transferred to the consumer's buffer.

To enable zero-sided RDMA on the switch, we need to maintain the necessary metadata information in the switch. We can group the state needed on the switch into the static connection state which does not change during communication and the dynamically changing state. For RDMA, the static connection state includes a remote key and destination queue-pair number to identify the remote RDMA queue-pair, along with its IP and MAC addresses. Since none of these header fields change during the course of communication, we store this information in the static switch tables upon connection setup. However, for the dynamically changing state, such as packet sequence numbers (PSNs) and virtual remote addresses, we need to change these values on a per-packet basis, and they need to be stored directly in registers on the data-plane. Since the switch is in the middle of the communication channel between the PUs, the switch must maintain PSNs and addresses for all connected end-points. In addition to the RDMA-specific state, we also store the buffers' head and tail pointer values in switch registers. This sums up to around 50 bytes in register memory and around 700 bytes of static table data for one producer-consumer pair connection. Concretely, more than 1500 concurrent flows can fit on the first-generation Tofino switch which is typically sufficient for most single-rack use cases. We estimate that with more recent programmable switches with longer pipelines and bigger state, the number of supported concurrent flows will be substantially higher.

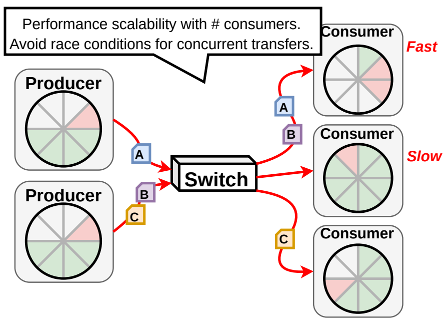
12.5 Complex Flows & Use Cases

12.5.1 N:M Data Shuffling

Data shuffling between multiple senders and receivers (i.e., an N:M shuffle) is very common in typical distributed DBMSs, due to the need for re-partitioning of tables in, e.g., distributed joins. However, these types of communication flows can be challenging due to the connection complexity of many sender/receiver combinations, which typically can even result in RDMA queue-pair cache thrashing on the NIC [28].

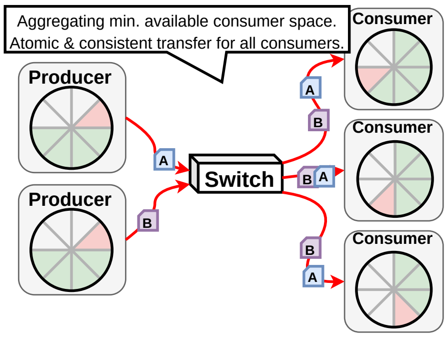


(a) N:M Shuffle



Performance scalability with # consumers.
Avoid race conditions for concurrent transfers.
Producer
Producer
Switch
Consumer
Fast
Consumer
Slow
Consumer
A
B
C

(b) N:M Load Balancing



Aggregating min. available consumer space.
Atomic & consistent transfer for all consumers.
Producer
Producer
Switch
Consumer
Consumer
Consumer
A
B

(c) N:M Replication with Global Order

Figure 12.5: Communication flows supported with zero-sided RDMA. All flows are completely switch-driven and require no processing or coordination by connected processing units. The core challenges per flow are highlighted in the text boxes.

With our centralized switch design, instead, we can reduce the connection complexity significantly as each communication end-point is only connected to the switch. Specifically, the switch can connect multiple producers to one consumer by handling the coordination of multiple writers into the same consumer-side memory directly in the data-plane of the switch. As such, at the consumer side, only one connection is needed for all incoming data, which both reduces memory overhead and also has the benefit that the PU only has to poll one memory location for incoming messages for multiple producers. In a key-based shuffle flow with N producer end-points and M consumer end-points, where each producer needs to send to multiple destinations based on, e.g., a join-key, we need $N \times M$ producer-side buffers and connections. In contrast, on the consumer-side, only M buffers are needed. The composition of client-side buffers and connections is shown in Figure 12.5a.

To handle this coordination between multiple producers connected to one consumer, we implement switch-side logic, which initiates data transfers for each producer separately in a round-robin manner. While a round-robin scheme might seem inefficient at first glance, the switch can iterate over a single producer in a matter of nanoseconds. As such, the latency overhead of iterating each producer round-robin to detect when a data transfer can be initiated is negligible, considering normal RDMA microsecond network latency. Moreover, this scheme has the benefit that producers are treated fairly and ensures a fair bandwidth share among producers.

12.5.2 Advanced Flows & Features

Switch-driven data transfers can also provide more advanced flows (i.e., load balancing, replication) and features (i.e., fine-grained quality of service guarantees), which open up interesting applications in cloud DBMSs.

Load balancing. Distributing work evenly across processing units in a distributed DBMS is non-trivial, given unforeseen network congestion or processing contention. Multiple schemes have been devised to overcome this [31, 77, 155], which require additional coordination overhead, e.g., through work-stealing or a centralized server-side dispatcher.

With zero-sided RDMA, we provide a communication flow between N producers and M consumers, providing automatic load balancing across all consumers. This flow is realized without any form of producer- or consumer-side coordination since the switch will transparently initiate the data transfers between producers and consumers. The overall buffers and connections needed for this are only $N + M$ as illustrated in Figure 12.5b.

The key to achieving this communication flow is introducing consumer-specific thread-like processing on the switch, where it is evaluated when a data transfer can be initiated from any of the producers, essentially adapting the data transfer rate from all producers to each consumer independently. We provide more details on how we realized this in Section 12.6.1.

Replication with ordering. Lastly, zero-sided RDMA also provides a replication-based communication flow in which items from each producer are multicasted out to all consumers.

Multicast has many applications in distributed DBMSs, such as replicated joins [126], or state replication [51, 82] for providing availability. While RDMA already has multicast capabilities, it is only supported through the Unreliable Transport and two-sided verbs. As such, it comes with the cost of higher CPU overhead at the communication endpoints due to the two-sided communication and the cost of ensuring reliability. With zero-sided RDMA, we realize replication without any coordination or computational involvement on the processing units by initializing, steering, and multicasting the data directly in the data-plane of the switch.

Traditionally, effects like reordering of packets in the network can cause the received data at each consumer to observe a different order. However, as our zero-sided approach transfers data sequentially with separate acknowledgments from each consumer, we can ensure globally ordered data transfers to all consumers without introducing any overhead at the processing units.

Fine-grained quality of service. Finally, another important aspect is that many DBMSs wish to prioritize certain processing jobs or queries over unimportant background jobs, e.g., to provide certain SLAs. However, ensuring a certain quality of service (QoS) or prioritization of parts of the network traffic (e.g., for a time-critical query) is hard to achieve as the available Priority-based Flow Control (PFC) requires network reconfiguration and only provides a means of prioritizing different classes of traffic. As such, PFC does not allow setting fine-grained prioritization (e.g., for each zero-sided RDMA flow) and adapting these settings at runtime.

To address this, we allow systems to adjust the QoS on per-flow granularity. The key to achieving this is to throttle down any zero-sided data transfers to a desired rate through congestion control primitives native to RDMA over Converged Ethernet (RoCE). With this feature, splitting up the available bandwidth for contending flows into any ratio desired is trivial. We show the ability to do fine-grained flow prioritization later in Section 12.7.2.

12.5.3 Use Cases

Zero-sided RDMA offers DBMSs with easy-to-use flow abstractions that integrate well into typical DBMSs use cases. We iterate a set of common use cases that can be trivially realized with zero-sided RDMA over various hardware accelerators.

Distributed joins (OLAP). In analytical data processing, distributed joins account for a substantial amount of the runtime as they have to shuffle big tables over the network. Due to their popularity, they have seen a lot of work to accelerate with hardware devices such as GPUs and FPGAs in the last decade [19, 88, 123]. Through our key-based shuffle communication flow, we simplify the join implementation by abstracting away all communication complexity from the processing unit. As such, the only task for the processing unit is to execute the actual join logic, as the producer and consumer buffer abstraction only has a few simple reads and writes into local memory for pushing or popping items.

Scan operators (OLAP). Table scans with filters are an interesting operator to accelerate due to their computational simplicity but high bandwidth demands, making them a great fit for specialized accelerators that meet these demands. In disaggregated compute and storage setups, it is often desirable to push down the table scan as close to the data to reduce the amount of data to ship over the network. We argue that zero-sided RDMA fits this use case well since the CPU can be completely avoided on storage nodes, making it possible to directly interface with Near Data Processing devices which carry out the table scan on the storage servers [132].

Replication (OLTP & OLAP). Lastly, with the zero-sided replication flow, we trivially support use cases involving data replication. These entail database replication [71], state-machine replication [51, 76, 82, 134], or even joins that rely on replication of tables.

12.6 Implementation Details

In this section, we take a closer look at how we realized the more complex communication flows involving load balancing or replication in N:M scenarios. We conclude with advanced challenges and a discussion of alternative zero-sided implementations.

12.6.1 Load Balancing and Replication

In the following, we take a look at a set of unique technical challenges when realizing load balancing and replication on the switch.

Many-to-many load-balancing. First, we present how we realize load-balancing between multiple producers to multiple consumers. This communication design aims to enable an even distribution of data to the consumers based on their individual processing speeds, without requiring any additional coordination or overhead at the processing units. We cannot simply extend the 1:1 scheme by alternating between different consumers because a single data transfer done by a switch thread can only saturate the bandwidth of a single consumer. Therefore we identify two challenges associated with this flow from the switch's perspective: *(1) ensuring efficient scaling in line with the number of processing units*, and *(2) preventing race conditions that could occur due to simultaneous data transfers.*

To provide efficient scaling, we extend the switch threading mechanism (i.e., continuous packet recirculation after data transfers) to not only transfer data between a specific producer-consumer combination but instead we spawn a switch thread packet per consumer which each iterate over all producers for the communication flow. With this, we can concurrently issue data transfers for each consumer. However, as previously mentioned, it comes with the challenge of avoiding race conditions in the producer-side switch registers. The reason for this is that the switch threads per consumer might try to issue a data transfer from the same producer at the same time, which would result in a race condition and inconsistent register state. To avoid this, we introduce a lock per producer such that only one switch thread can initiate a data transfer from a producer at any given time. We introduce this lock in a register of the switch pipeline and let each switch thread test-and-set the producer lock. If a switch thread fails, it will recirculate and try the next producer. If it succeeds, the lock will be taken, and a data transfer will be initiated for the producer and consumer pair.

By employing this design, each switch thread, specific to a consumer, will aim to read from all producers in a round-robin fashion. This ensures a fair distribution of data from all producers to each consumer. Additionally, it allows each consumer to process and remove data from its buffer at its own pace, independent of the other consumers.

Many-to-many replication. For the replication communication flow, the goal is to do switch-driven multicast to all consumers without any overhead or coordination at the processing units. The challenge here is twofold: *(1) how to accumulate the global minimum space available at all consumers* and *(2) how to ensure atomic & consistent* *transfer for all consumers.* In our implementation, the replication of packets to be multicasted happens between the ingress and egress of packets in the programmable switch. Therefore any logic and state needed to modify packets specifically for each consumer have to be placed in the stages of the egress pipeline.

Computing the global minimum buffer space available for all consumers cannot be trivially implemented using a for-loop over the head and tail registers because a packet can only access a specific register (e.g., all consumer head values) once per pipeline. To work around this constraint, we recirculate a packet $N - 1$ times to accumulate the minimum space over all consumers in the communication flow. The recirculating packet contains a temporary header that stores the current consumer index and the minimum space value between the pipeline passes. Once the head and tail values for the last consumer have been read, the switch asserts that the current producer has something to send and sends out the RDMA READ request. Each response fragment of the RDMA READ is then multicasted N times (step ② in Figure 12.3) and rewritten to RDMA WRITEs that target the appropriate consumer buffer memory. These N write requests generate N acknowledgments coming from each consumer connection. The switch then counts all acknowledgments before it increases the consumer-heads. We accumulate all acknowledgments to ensure that either all consumers see a new item or none of them in case a drop occurred for a data transfer. Upon receiving the last ACK from write data, the switch multicasts N times the Incr-Head RDMA WRITE to each consumer. After receiving N acknowledgments as before, the switch increases the tail from the current producer and starts the sequence again.

12.6.2 Further Challenges

The key to achieving zero-sided RDMA is that the PUs do not directly communicate; instead, all communicate with the switch. This introduces unique challenges such as handling of queue-pair state, ensuring reliability, and congestion control, which we will address in this section.

Buffer-state updates. One challenge of zero-sided RDMA is how to propagate the local buffer changes of PUs (i.e., when a producer pushes a new item or a consumer frees an item) to the switch. The switch uses this information to determine when new data transfers can be initiated. Here, there are two different mechanisms to consider: poll-based or doorbell-based switch state updates. In the poll-based mechanism, the switch polls the head and tail pointers of the PUs with RDMA READ operations issued by the switch. This has the benefit that neither the producer nor the consumer has to execute any RDMA operations actively. However, it naturally comes with the downside of added latency. To this end, we expose the polling interval as a configurable parameter as it is very application specific what the requirements for latency or throughput are. However, for PUs that already have the possibility of issuing RDMA WRITEs themselves (e.g. a network-enabled FPGA or CPU), the doorbell-based mechanism allows PUs to directly send out an update to the switch to indicate changes in their buffer state (i.e., new items or free space). One could contend that now the PUs could potentially handle all communication, taking charge of the communication scheme and data transfers. However, doing so would negate the advantages of network-driven data transfers, which include high-level communication flows such as load-balancing, replication, and the distinct separation between control and data paths. While the introduction of an additional doorbell mechanism might increase the complexity of the PUs it can significantly decrease communication latency, as demonstrated in Section 12.7.1.

Ensuring reliability. While the RDMA Reliable Connection transport already provides network reliability, since the switch is operating on the Data Link Layer (L2), any packet can be dropped in the network, and it is up to the switch to adhere to the RoCEv2 protocol [104] when this happens. In essence, any of the network packets illustrated in Figure 12.3 can be dropped, which, without any further action, would cause the data transfer to halt. If a drop occurs, the switch either detects it through a configurable timeout or gaps in the sequence numbers. If the drop happened during data transfers (i.e., step ②), the switch reissues the data transfer request (i.e., RDMA READ on the producer device). However, if the drop happened after receiving the ACK from the data transfer from the consumer, the transfer was successful, and it only reissues the subsequent RDMA WRITEs for the head and tail pointers.

Congestion control. Furthermore, to handle incast scenarios (e.g., two producer nodes with 2x bandwidth and a consumer node with 1x bandwidth), we integrate congestion control into our zero-sided communication scheme on the switch. The switch emits explicit congestion notifications whenever a link becomes congested, similar to RoCEv2. When the NIC of the producer receives the congestion notification, it throttles down the rate of outgoing packets, removing the congestion bottleneck.

12.6.3 Discussion

Offloading the network communication from the hardware processing units can be realized in different ways. We now discuss the centralized or decentralized approaches, compare their strengths and weaknesses, and subsequently touch on the possibility and challenges of single vs. multi-rack setups.

Decentralized SmartNIC-driven approach. Instead of using the main CPU to carry out the communication, an approach is to use a SmartNIC to issue RDMA operations on behalf of the accelerator, such as the CPU-based Nvidia BlueField or FPGA-based Mellanox Innova SmartNICs, or even FPGA-based SmartNICs that implement the RDMA stack directly in the FPGA [73, 136].

The benefit is that a specialized compute unit embedded in the NIC carries out the communication instead of the main CPU. Concrete examples of this approach are Lynx [128] and FpgaNIC [136]. Lynx and FpgaNIC aim to enable direct GPU communication by letting the GPU communicate without any CPU involvement or need for RDMA operations directly on the GPU.

However, a decentralized SmartNIC-driven approach has several downsides. First, realizing one-sided RDMA communication schemes is highly complex due to the distributed coordination for remote memory accesses. Adding to the fact that typical CPU-based SmartNICs are too weak to saturate line-rate throughput for many scenarios properly [124], SmartNIC-driven communication schemes are best realized on more performance-efficient compute architectures such as FPGAs. This, in turn further adds to the complexity when considering decentralized one-sided communication schemes. Second, since a dedicated SmartNIC is needed per server, the hardware cost is directly proportional to the number of servers. As the cost of a SmartNIC can be up to 10× more expensive than a normal RDMA-enabled NIC for the same link speeds, the added cost for a disaggregated solution is non-negligible.

Centralized programmable switch-driven approach. We argue that a switch-driven design of direct accelerator communication is preferable. The reasons for this are: First, a programmable switch provides the unique possibility of doing line-rate processing for all connected nodes. In terms of realizing a zero-sided RDMA communication scheme, the switch can natively scale out to full link throughput for all connected devices without hitting any performance bottlenecks. Switch-driven communication is different from, e.g., CPU-driven communication, where with increasingly faster networks, saturating the throughput of just one link becomes increasingly difficult. Second, the centralized position of the switch in the network allows us to realize communication flows (as covered in Section 12.5) without expensive distributed coordination. Lastly, offloading the communication onto a programmable switch with many ports results in a cheaper overall hardware cost for typical scale-out solutions compared to decentralized per-server solutions.

Data center deployments. In many cloud deployments, the jobs for query processing are scheduled on compute nodes within a single rack connected by a Top-of-the-Rack (ToR) switch to increase locality [56, 145]. This is crucial for distributed operators (e.g., joins) to have high all-to-all shuffle bandwidth.

In this paper we focus on a single-switch setup to mirror a typical rack setup but cross-rack communication is supported by zero-sided RDMA out of the box. Regular RDMA communication between end hosts (and initiated by end hosts) can be used across racks, traversing multiple switches. This is the same for switch-initiated RDMA traffic. Nontheless, there are several aspects left for future investigation: e.g., the added latency between the switch and connected accelerator devices, asymmetric bandwidth, and possible implications to congestion control. Moreover, with multi-switch setups, we envision the possibility of graceful failover between switches in case of failures. This is possible since the runtime state stored on each switch, e.g. fill-levels of the buffers and connection state, can be recovered by reading them out from participating PUs.

Finally, a limiting factor for many in-network processing ideas is that data streams are often encrypted in data centers, making payloads opaque to the processing element. This is however not an issue for zero-sided RDMA: the switch does not need to read or modify the payload of the transferred data items, but merely modifies the header and forwards the packets.

12.7 Experimental Evaluation

With our zero-sided RDMA contribution, we mainly target distributed analytical workloads, which is the focus of the evaluation. We first evaluate the efficiency of zero-sided RDMA through a set of microbenchmarks that vary in different performance-related parameters. Subsequently, we investigate the benefit of switch-driven data transfers in the context of distributed database systems by evaluating two DBMS use cases (i.e., a distributed join and a full TPC-H query) using a disaggregated accelerator setup.

Setup and implementation. We evaluate our zero-sided RDMA communication scheme in a cluster of 4 nodes running Ubuntu 18.04 LTS with Linux kernel 4.15.0. Each node is equipped with an Intel(R) Xeon(R) Gold 5120 2.2GHz CPU, an Nvidia V100 GPU, and a Mellanox ConnectX-5 that is connected to an Intel Tofino switch [93] via 100G RoCEv2. Our code written in C++20 is compiled with gcc-12 and CUDA-12. The switch's control-plane logic is implemented in C++, and the switch's data-plane logic is implemented in P4 and compiled using Intel SDE-9.11.0 [53]. For all experiments, unless

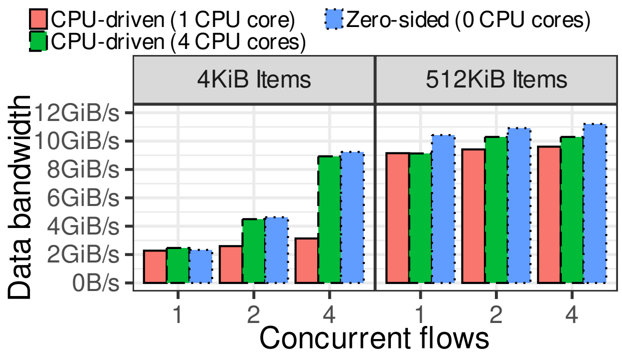


Figure 12.6: GPU-to-GPU transfer: Zero-sided RDMA vs. a CPU-driven one-sided RDMA baselines for 1:1 GPU data transfer with persistent GPU kernels. The CPU-driven baselines use 1 or 4 CPU cores; our approach does not use the CPU.

otherwise stated, the polling-based mechanism is used for detecting new data items on producers and fill-grade on consumers. The source-code is available at [57].

12.7.1 Efficiency of Zero-sided RDMA

Zero-sided vs. CPU-driven transfer (GPU-to-GPU). In this experiment, we focus on accelerator-to-accelerator communication across two nodes. As accelerators in this experiment, we use a homogeneous setup (GPU-to-GPU) to provide the same execution speeds on all senders and receivers. At the end of the evaluation, we also show heterogeneous setups with GPUs and FPGAs.

The data transfers for all variants in this experiment (i.e., CPU-driven and zero-sided RDMA data transfers) are all directly from GPU-to-GPU without intermediate copies over the main memory. Moreover, in both the CPU-driven baseline and our zero-sided communication, we execute the exact same GPU kernels and use the same buffer abstraction on the GPUs. However, for the CPU-driven baseline, we let the CPU detect new items in the buffer and issue the RDMA data transfers on behalf of the GPU. The GPU kernels are executed as persistent kernels to minimize any kernel launch overhead for both variants.

Figure 12.6 shows the bandwidth between CPU-driven and network-driven communication in our experiment. The CPU-driven baseline issues asynchronous RDMA data transfers and utilizes persistent GPU kernels to reduce synchronization overhead between the CPU and the GPU. We report the data bandwidth for varying item sizes. In addition, we test with a different number of parallel producer and consumer communication flows to evaluate different communication loads of realistic scenarios (e.g., multiple concurrent

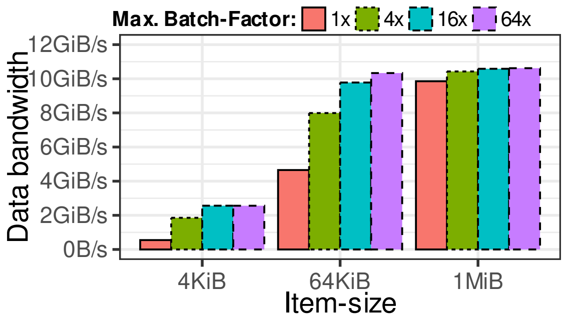


Figure 12.7: GPU-to-GPU transfer: 1:1 GPU data transfer with adaptive batching for different tuple sizes. Bigger batch sizes yield more efficient data transfers.

queries running at the same time or one query using multiple flows for intra-query parallelization).

For small item sizes (4 KiB), we can see that multiple CPU cores are necessary to achieve comparable performance to the zero-sided approach when increasing the concurrent flows. In fact, with small item sizes, one CPU core is not sufficient to handle all flows (see the red bar in Figure 12.6, left-hand-side). This indicates that even for persistent GPU kernels that overcome the GPU kernel launch overhead data transfers are inherently CPU bound. In our zero-sided approach, the switch instead handles all flows completely independently and as such scales perfectly with the number of flows without the need to use a dedicated CPU per flow as in the CPU-driven scheme (red and green bars). Moreover, for the CPU-driven scheme and zero-sided RDMA, another effect is that the communication overhead is reduced for larger item sizes, which enables higher transfer bandwidths with fewer concurrent transfers.

Adaptive batching. In the next experiment, the main objective is to understand how adaptive batching implemented in the switch can improve the performance of zero-sided RDMA. This optimization mitigates the overhead of smaller transfers by seamlessly grouping multiple items into a single transfer. We measured throughput across a variety of batching factors and three distinct item sizes: 4 KiB, 64 KiB, and 1 MiB. The results are depicted in Figure 12.7 and show how the data bandwidth is influenced by item-size and batch-size. The results reveal that the use of adaptive batching can minimize the overhead of smaller transfers. However, the effect of batching becomes less pronounced as the item-size increases. This is because a data transfer only with one large item is enough to almost saturate the network. The performance still increases slightly with larger batches as the overhead of buffer head and tail pointer updates are amortized further. However, choosing a too-large batch size can have detrimental effects on data transfer latency, especially in scenarios with network congestion. The reason for this is

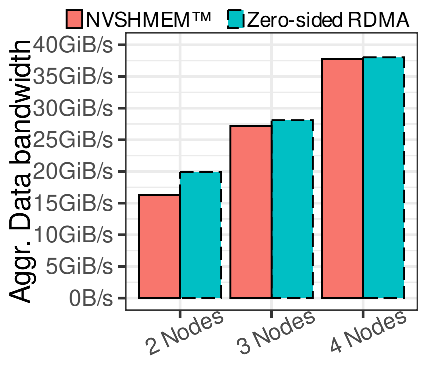


Figure 12.8: GPU-to-GPU shuffle: Comparison of zero-sided RDMA GPU-to-GPU shuffle bandwidth with NVSHMEM [103], that only allows GPU-to-GPU communication and requires custom coordination.

that larger data transfers will consist of a higher number of fragments, increasing the probability that one fragment is lost and the data transfer has to be reissued. We found that a batching size no larger than 4 MiB can help to saturate the available network bandwidth even for just one producer and consumer, but at the same time performs very stable under congestion.

Comparison with CPU-less shuffling (GPU-to-GPU). Shuffling data between nodes is crucial for many distributed database operators such as joins or aggregation. In Figure 12.8, we report the data bandwidth for a GPU-to-GPU shuffle scenario where we compare against NVSHMEM (version 4.0.2) [103]. NVSHMEM is the only commercial library that supports GPU-initiated data transfers to remove the CPU from the control-path and is as such a comparable baseline to zero-sided RDMA. For the experiment, we use the non-blocking PUT primitive of NVSHMEM for efficient one-sided data transfers. We use item sizes of $512KiB$ and execute 2 shuffle kernels on each GPU node. After launching the GPU kernels, no CPU cycles of the host system are spent on communication primitives for either zero-sided RDMA or NVSHMEM. Both setups use RDMA (RoCE). For simplicity, we compare against NVSHMEM without any distributed coordination and use a non-skewed workload and thus report the best-case bandwidth for NVSHMEM.

As can be seen in Figure 12.8, the reported bandwidth is almost equal since both NVSHMEM and zero-sided RDMA manage to fully utilize the link bandwidth. However, an important difference between zero-sided RDMA and NVSHMEM is that while zero-sided RDMA can integrate many different types of heterogeneous accelerators, NVSHMEM works exclusively for GPU-to-GPU transfers. Furthermore using NVSHMEM requires a decentralized protocol between GPUs to coordinate data transfers which is provided

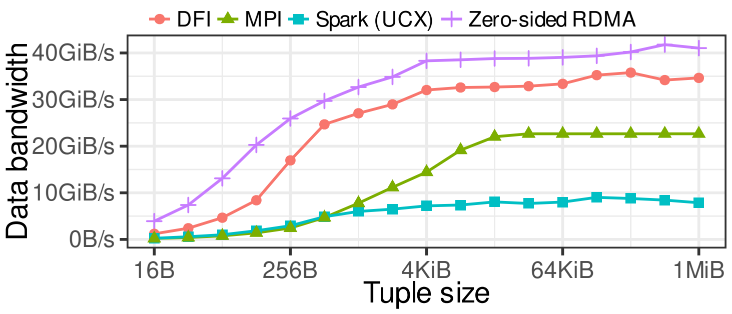


Figure 12.9: CPU-to-CPU shuffle: Comparison of RDMA-based data shuffling with zero-sided RDMA vs. CPU baselines (DFI, MPI, Spark UCX) for various tuple sizes.

in zero-sided RDMA by the switch along with other functions for load balancing not available in NVSHMEM.

Comparison with existing CPU baselines (CPU-to-CPU). In this experiment, we shift our attention from a GPU-to-GPU to a CPU-to-CPU shuffle (i.e., data resides in CPU memory), in order to show that zero-sided RDMA can also be beneficial in such a setting. In particular, we compare zero-sided RDMA against three CPU-based baselines for CPU-to-CPU data shuffling: (1) Spark (UCX) [106] which is a baseline for RDMA-based data shuffling in a production-ready system (Apache Spark v3.0 with UCX Plugin), and two isolated CPU-based data shuffling baselines - (2) a DFI-based shuffle [126] as well as (3) an MPI-based shuffle that uses MPI collectives (Nvidia HPC-X v2.17) for shuffling [42, 92]. DFI and MPI are both state of the art open-source libraries that enable data shuffling using RDMA. In this experiment, we report the bandwidth of a data shuffle between 4 nodes for varying tuple sizes. On each node, we use four worker threads (cores) as data producers/consumers for the shuffle for zero-sided RDMA and all baselines.

As shown in Figure 12.9, zero-sided RDMA achieves constantly higher throughput than the others. The reason for this is twofold: First, zero-sided RDMA frees up sender/receiver threads from issuing RDMA primitives and thus CPUs can use their cycles solely for producing/consuming data. Second, zero-sided RDMA applies adaptive batching on the network level leading to transfers that are efficiently larger than the tuple sizes, which further improves the bandwidth usage.

Ping-pong latency (polling vs. doorbell). Finally, in the last experiment of this section, we focus on transfer latency. We first analyze on the impact of switch polling intervals in the context of buffer updates (i.e., new items pushed or popped at the PUs). As these updates are detected on the switch via polling the buffers at the

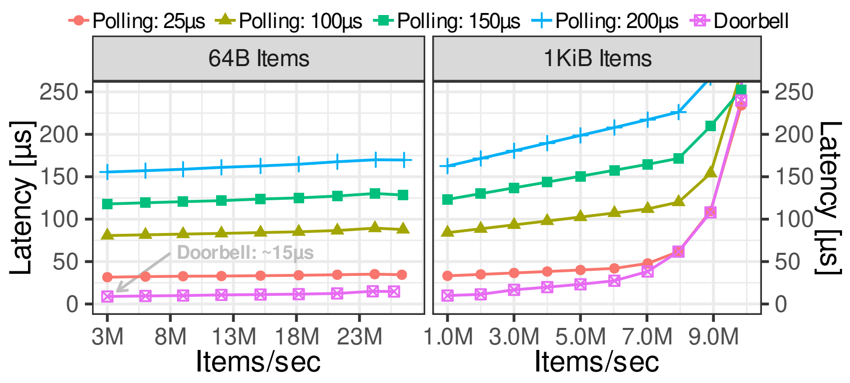


Figure 12.10: One-way median latency over different throughput and polling intervals. In "Doorbell" mode PUs actively signal the switch for available data.

PUs, the duration of polling intervals can potentially influence the transfer latency of items. Understanding this relationship can assist in determining the optimal polling intervals for various applications with different requirements for throughput and latency. In the experimental setup, we report half round-trip time on the y-axis across various polling intervals, denoted by different line symbols. We use two CPU nodes as PUs, each with both a producer and consumer, which the switch connects to form a bi-directional communication flow to measure the round-trip time.

Additionally, we analyze the effect of the doorbell mechanism in zero-sided RDMA. Clearly, polling for state updates using the switch enables data transfers without any involvement from the PUs. However, this potentially comes with the cost of increased latency. Zero-sided RDMA thus also allows a PU to actively signal the switch when it has data available for transfer, referred to as doorbell mode. To send out a doorbell message, the PU simply sends out a network packet, containing the new buffer state, signaling an update to the switch. If the PU lacks this capability, a small co-located FPGA could also be used to perform this task.

The experimental results of our polling and the doorbell mechanism are illustrated in Figure 12.10 for 64B and 1KiB item sizes. The transfer rate, specified in items per second, is fixed to certain values denoted on the x-axis. As we see in Figure 12.10 for the polling scheme, the median latency is slightly less than the polling interval. This is because, on average, an update to the producer buffer is detected in half of the polling interval plus the latency of the data transfer. Enabling the doorbell mechanism, the median latency is reduced to $\sim 15 \mu s$ since the producer can immediately signal the switch that an item can be transferred. This optimization is, however, only possible on PUs with RDMA capabilities as discussed before in Section 12.6.2.

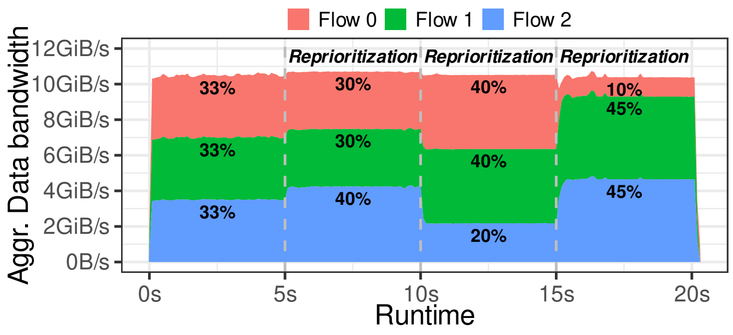


Figure 12.11: QoS flow prioritization between two PUs with 3 concurrent flows. Prioritization changes every 5 seconds.

Important is that latency remains stable across various throughputs, demonstrating consistency in performance irrespective of the transfer rate. For 64 B item sizes, the throughput scales up to around 27 million items per second due to the adaptive batching optimization. However, for the larger 1 KiB sizes, the latency stays stable up to around 10 million items per second which almost matches the link's capacity resulting in full buffers and high latency. Interestingly, for the slower polling intervals, e.g., 200μs, the higher rate of pushing items to the producer buffer results in more items being batched for each data transfer resulting in a slightly higher latency.

12.7.2 Benefits of Switch-driven Data-transfers

Utilizing network-driven data transfers from a centralized switch provides significant advantages. Given its overarching view of all network traffic, the switch allows for streamlined management of data transfers. In particular, we enable fine-grained Quality of Service (QoS) via flow prioritization, facilitate load-balancing, and promote elasticity for efficient resource scaling. These benefits, challenging to achieve in decentralized data-flow solutions, underscore the value of centralized, network-driven data transfers.

Fine-grained flow prioritization. In this experiment, we investigate the fine-grained (per flow) prioritization capabilities provided by our zero-sided communication scheme. Figure 12.11 shows the individual data bandwidth for three active concurrent flows between two nodes over 20 seconds, with the ratio altered every 5 seconds. Our system demonstrates a swift and efficient response to changes in the prioritization of different flows, with the processing units remaining oblivious to the adjustments. Centralized initiated data transfers allow fine-grained prioritization of different flows, which is not supported natively within RDMA.

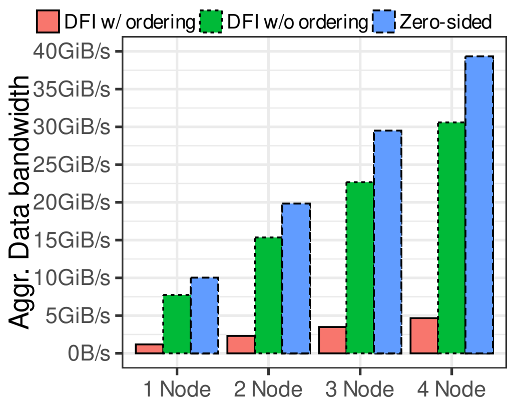


Figure 12.12: 1:N zero-sided replication compared to DFI [126] with or without global ordering. Zero-sided replication always ensures a global order observed by all consumers.

Replication with ordering. The subsequent experiment, presented in Figure 12.12, centers on the effectiveness of zero-sided RDMA replication which by design ensures global ordering for each consumer. For the baseline, we use the Data Flow Interface (DFI) [126], a state-of-the-art library that enables replication using RDMA multicast with two-sided SEND/RECEIVE operations. We present results for DFI both with and without software-based ordering. For the experiment, we use CPUs as PUs to be able to compare against the CPU-only DFI baseline.

The results show the aggregated data bandwidth of each consumer for a different number of nodes that receive data from a single producer. Across all scenarios, zero-sided replication with ordering consistently demonstrates superior performance, nearly saturating the link bandwidth at each consumer node. In addition, zero-sided replication does not infer any overhead at the PUs. In stark contrast to native RDMA multicast that only supports two-sided SEND/RECEIVE operations and, as such, introduces communication overhead at not only the sender but also the receiver.

Elasticity of flows (TPC-H Query 1). In this experiment, we demonstrate the elasticity of our system. For this experiment, we execute TPC-H Query 1 in a disaggregated setup. In particular, storage nodes (using CPUs) stream data to GPUs as processing units in the compute layer. We scale up the compute layer from one to four nodes (each having one GPU).

Figure 12.13 represents the aggregated bandwidth of each consumer in an area plot, with each consumer distinguished by different colors. This experiment shows that the processing capability can dynamically be scaled up from a single node to four nodes and scaled down when required. Important is that the producer nodes (i.e., the storage)

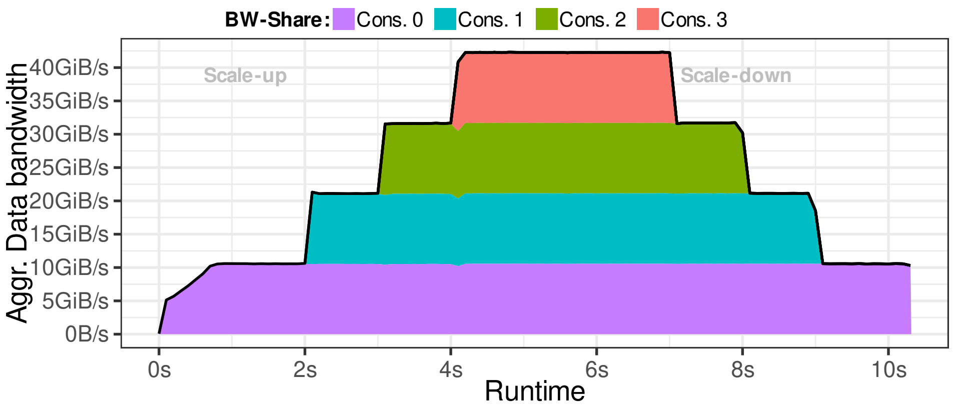


Figure 12.13: Elasticity for 4:N Load balancing flow - TPC-H Query 1 with SF 100 executed on GPUs. Each producer PU is unaware of the elasticity.

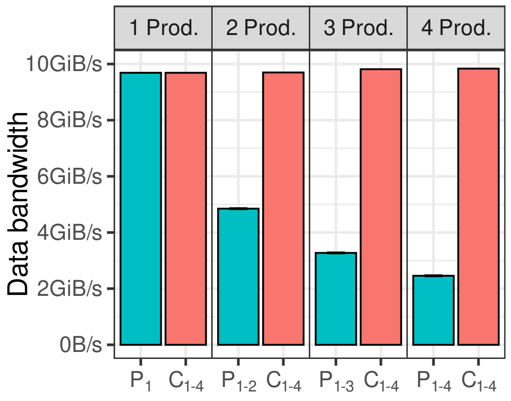


Figure 12.14: Per producer & consumer bandwidth for multicast with an increasing number of producers (from left-to-right) and four consumers (C1-4). P1-2 refers to a scenario with two producers.

remain completely unaware of the compute-layer elasticity. The entire coordination of this scaling operation is managed by the switch. This experiment illustrates the ability of our system to efficiently adjust and manage resource allocation dynamically, offering significant potential for enhanced scalability and efficiency in distributed accelerator environments.

Fairness between producers. In this next experiment, we explore the fairness of bandwidth distribution among multiple producers, each producing data to the same multicast flow. Producer fairness is crucial to ensure producers are given an equal share of the consumer-side available link bandwidth. In Figure 12.14, we report the bandwidth separately for producers and consumers. Important to see is that in every scenario (1-4 producers from left to right), all consumers receive a fair share of the bandwidth. When

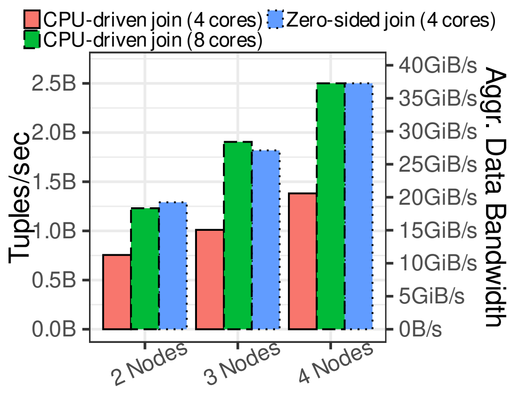


Figure 12.15: Distributed Join. Tables are shuffled from CPU-based storage nodes to GPU nodes for joining. Zero-sided RDMA achieves better performance with fewer CPU cores compared to [123].

more producers are added to the system, the bandwidth is divided evenly among them. The barely noticeable error bars in the figure for the producers denote the low degree of unfairness, emphasizing the fairness between multiple producers.

Distributed join. Distributed joins represent a fundamental and often performance-critical operation in DBMSs. In light of this, we evaluate the performance of a distributed join using zero-sided RDMA. We mirror a typical disaggregated setup, where tables are residing on CPU-based storage nodes and are shuffled on the join key to GPUs for join processing. As a baseline, we use a state-of-the-art pipelined GPU join implementation where the shuffling is CPU-driven [123]. Both the zero-sided join and the baseline implement the same optimizations for shuffling such as software write combine buffers (SWWCBs) and non-temporal streaming hints [7, 117], We use two synthetic and uniformly distributed relations with 16-byte tuples of sizes 10M and 10B for the workload.

We apply an N:N shuffle flow and measure the throughput in tuples/sec for configurations with up to 4 nodes. The experiment includes both the build phase for the hash table and the probe phase. Similar to the baseline implementation, we overlap the shuffling with the building or probing such that the join is executed overlapped on the GPUs [123]. For the zero-sided join, we use 4 producers and 4 consumers per node, totaling up to 256 producer buffers and 16 consumer buffers for 4 nodes. Each producer pushes tuples based on the join key into separate buffers from which they are transferred to the corresponding consumer using zero-sided RDMA. In the CPU-driven GPU join baseline we report numbers for 4 and 8 active CPU cores shuffling to 4 GPU buffers per node.

Our results in Figure 12.15 showcase a near-linear speed-up for the zero-sided RDMA join as the number of nodes increases and performance that is on par with the baseline.

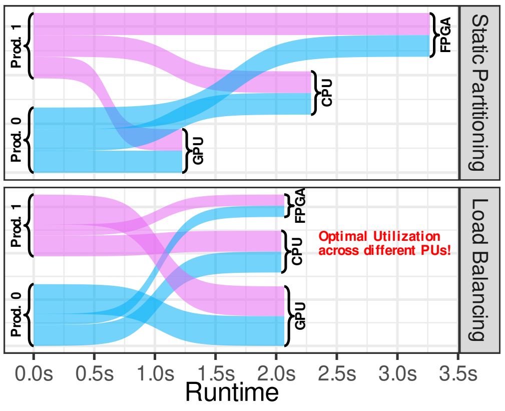


Figure 12.16: Static partitioning vs. load balancing for TPC-H Query 1 with SF 100 executed on CPU, GPU and FPGA. The thickness of the lines indicates the speed of the consumers.

As an important effect, we additionally see the benefit of offloading the communication scheme into the network and thereby freeing up CPU resources by letting the CPU-based storage nodes focus solely on table scanning and shuffling. This is apparent since the zero-sided RDMA join only needs 4 CPU cores to achieve comparable performance to the CPU-driven join baseline.

12.7.3 Heterogeneous Communication

Scheduling processing jobs across a wide range of heterogeneous processing devices with different and varying throughputs is a challenging task that requires careful coordination between all participants. Therefore in this section, we evaluate the load-balancing communication flow in a setup that mimics a cloud data center with disaggregated storage and heterogeneous compute resources that consist of CPU, GPU, and FPGA.

Load balancing (TPC-H Query 1). In this experiment, depicted in Figure 12.16, we explore load-balancing for a disaggregated setup using a heterogeneous computing environment comprising a CPU, a GPU, and an Xilinx Alveo U55C FPGA as consumers. We expose the FPGA's local HBM memory to the RDMA NIC through the PCIe bar for peer-to-peer DMA (see Section 12.3.2). Data producers on the storage are spread out on 2 (CPU) nodes. The workload is based on TPC-H Query 1, as in the elasticity experiment before. The system's performance is assessed under two distinct scenarios: static partitioning and load balancing.

In the scenario with static partitioning, the input TPC-H table *lineorder* is partitioned into two equal sizes at both producers which send the data in equal portions of the data to three consumers: an FPGA, a GPU, and a CPU. Without load-balancing (Figure 12.16, upper part) in the switch, the GPU completes its tasks more rapidly than the CPU, which is faster than the FPGA as the slowest device. Overall, the FPGA takes in total 3.2 seconds, which dominates the end-to-end latency of the query.

In contrast, with load balancing (Figure 12.16, lower part), the *lineorder* table is dynamically divided to each consumer based on each device's processing speed. This arrangement enables the CPU, FPGA, and GPU to finish their tasks simultaneously, thus reducing the total runtime of the query to 2.1 seconds and ensuring optimal utilization with no idle time for either processing unit.

These results underscore the potential of load balancing in zero-sided RDMA: it allows for optimal utilization of different processing units without the need for complex partitioning or work-stealing schemes, leading to more simple query execution code.

12.8 Conclusions

In this paper, we presented zero-sided RDMA as a way to enable direct RDMA-based accelerator-to-accelerator communication, which does not require CPUs to coordinate the communication because the communication scheme is driven from the network. Moreover, with zero-sided RDMA, we enable efficient RDMA-based data shuffling between heterogeneous hardware devices without the need to implement a complete RDMA stack on each heterogeneous device. Our evaluation showed that zero-sided RDMA can outperform CPU-driven one-sided RDMA schemes for accelerators and in addition, provide a set of useful and efficient communication flows targeting disaggregated cloud DBMSs.

12.9 Acknowledgements

This work was partially funded by the German Research Foundation (DFG) under the grants BI2011/1 & BI2011/2 (DFG priority program 2037) and the DFG Collaborative Research Center 1053 (MAKI). We also thank hessian.AI, 3AI, DFKI as well as Intel for their support.

Bibliography

[1] Elena Agostini, Davide Rossetti, and Sreeram Potluri. "GPUDirect Async: Exploring GPU synchronous communication techniques for InfiniBand clusters." In: *J. Parallel Distributed Comput.* 114 (2018), pp. 28–45. DOI: 10.1016/j.jpdc.2017.12.007. URL: https://doi.org/10.1016/j.jpdc.2017.12.007.

[2] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. "Designing Far Memory Data Structures: Think Outside the Box." In: *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019.* ACM, 2019, pp. 120–126. DOI: 10.1145/3317550.3321433. URL: https://doi.org/10.1145/3317550.3321433.

[3] Gustavo Alonso, Carsten Binnig, Ippokratis Pandis, Kenneth Salem, Jan Skrzypczak, Ryan Stutsman, Lasse Thostrup, Tianzheng Wang, Zeke Wang, and Tobias Ziegler. "DPI: The Data Processing Interface for Modern Networks." In: *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings.* www.cidrdb.org, 2019. URL: http://cidrdb.org/cidr2019/papers/p11-alonso-cidr19.pdf.

[4] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. "Socrates: The New SQL Server in the Cloud." In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* Ed. by Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska. ACM, 2019, pp. 1743–1756. DOI: 10.1145/3299869.3314047. URL: https://doi.org/10.1145/3299869.3314047.

[5] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema,

Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert G. Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. "Empowering Azure Storage with RDMA." In: *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023.* Ed. by Mahesh Balakrishnan and Manya Ghobadi. USENIX Association, 2023, pp. 49–67. URL: https://www.usenix.org/conference/nsdi23/presentation/bai.

[6] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. "Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited." In: *Proc. VLDB Endow.* 7.1 (2013), pp. 85–96. DOI: 10.14778/2732219.2732227. URL: http://www.vldb.org/pvldb/vol7/p85-balkesen.pdf.

[7] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. "Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited." In: *Proc. VLDB Endow.* 7.1 (Sept. 2013), pp. 85–96. ISSN: 2150-8097. DOI: 10.14778/2732219.2732227. URL: https://doi.org/10.14778/2732219.2732227.

[8] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware." In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013.* Ed. by Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou. IEEE Computer Society, 2013, pp. 362–373. DOI: 10.1109/ICDE.2013.6544839. URL: https://doi.org/10.1109/ICDE.2013.6544839.

[9] Claude Barthels, Gustavo Alonso, Torsten Hoefler, Timo Schneider, and Ingo Müller. "Distributed Join Algorithms on Thousands of Cores." In: *Proc. VLDB Endow.* 10.5 (2017), pp. 517–528. DOI: 10.14778/3055540.3055545. URL: http://www.vldb.org/pvldb/vol10/p517-barthels.pdf.

[10] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. "Rack-Scale In-Memory Join Processing using RDMA." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 1463–1475. DOI: 10.1145/2723372.2750547. URL: https://doi.org/10.1145/2723372.2750547.

[11] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. "The End of Slow Networks: It's Time for a Redesign." In: *Proc. VLDB Endow.* 9.7 (2016), pp. 528–539. DOI: 10.14778/2904483.2904485. URL: http://www.vldb.org/pvldb/vol9/p528-binnig.pdf.

[12] Marcel Blöcher, Tobias Ziegler, Carsten Binnig, and Patrick Eugster. "Boosting scalable data analytics with modern programmable networks." In: *Proceedings of the 14th International Workshop on Data Management on New Hardware, Houston, TX, USA, June 11, 2018*. Ed. by Wolfgang Lehner and Kenneth Salem. ACM, 2018, 1:1–1:3. DOI: 10.1145/3211922.3211923. URL: https://doi.org/10.1145/3211922.3211923.

[13] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. "Breaking the memory wall in MonetDB." In: *Commun. ACM* 51.12 (2008), pp. 77–85. DOI: 10.1145/1409360.1409380. URL: https://doi.org/10.1145/1409360.1409380.

[14] Sebastian Breß. "Why it is time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS." In: *Proc. VLDB Endow.* 6.12 (2013), pp. 1398–1403. DOI: 10.14778/2536274.2536325. URL: http://www.vldb.org/pvldb/vol6/p1398-bress.pdf.

[15] Sebastian Breß, Henning Funke, and Jens Teubner. "Robust Query Processing in Co-Processor-accelerated Databases." In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Fatma Özcan, Georgia Koutrika, and Sam Madden. ACM, 2016, pp. 1891–1906. DOI: 10.1145/2882903.2882936. URL: https://doi.org/10.1145/2882903.2882936.

[16] Mihai Budiu and Chris Dodd. "The P416 Programming Language." In: *ACM SIGOPS Oper. Syst. Rev.* 51.1 (2017), pp. 5–14. DOI: 10.1145/3139645.3139648. URL: https://doi.org/10.1145/3139645.3139648.

[17] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. "PRISM: Rethinking the RDMA Interface for Distributed Systems." In: *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. Ed. by Robbert van Renesse and Nickolai Zeldovich. ACM, 2021, pp. 228–242. DOI: 10.1145/3477132.3483587. URL: https://doi.org/10.1145/3477132.3483587.

[18] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. "Efficient Distributed Memory Management with RDMA and Caching." In: *Proc. VLDB Endow.* 11.11 (2018), pp. 1604–1617. DOI: 10.14778/3236187.3236209. URL: http://www.vldb.org/pvldb/vol11/p1604-cai.pdf.

[19] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. "Is FPGA Useful for Hash Joins?" In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: http://cidrdb.org/cidr2020/papers/p27-chen-cidr20.pdf.

[20] Scott Ciccone and John F. Kim. *NVIDIA Introduces BlueField DPU as a Platform for Zero Trust Security with DOCA 1.2*. NVIDIA. June 2022. URL: https://developer.nvidia.com/blog/nvidia-introduces-bluefield-dpu-as-a-platform-for-zero-trust-security-with-doca-1-2/.

[21] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric P. Xing. "Solving the Straggler Problem with Bounded Staleness." In: *14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013*. Ed. by Petros Maniatis. USENIX Association, 2013. URL: https://www.usenix.org/conference/hotos13/session/cipar.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking cloud serving systems with YCSB." In: *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*. Ed. by Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum. ACM, 2010, pp. 143–154. DOI: 10.1145/1807128.1807152. URL: https://doi.org/10.1145/1807128.1807152.

[23] Benoıt Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. "The Snowflake Elastic Data Warehouse." In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Fatma Özcan, Georgia Koutrika, and Sam Madden. ACM, 2016, pp. 215–226. DOI: 10.1145/2882903.2903741. URL: https://doi.org/10.1145/2882903.2903741.

[24] Benoıt Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. "The Snowflake Elastic Data Warehouse." In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Fatma Özcan, Georgia Koutrika, and Sam Madden. ACM, 2016, pp. 215–226. DOI: 10.1145/2882903.2903741. URL: https://doi.org/10.1145/2882903.2903741.

[25] Huynh Tu Dang, Jaco Hofmann, Yang Liu, Marjan Radi, Dejan Vucinic, Robert Soulé, and Fernando Pedone. "Consensus for Non-volatile Main Memory." In: *2018 IEEE 26th International Conference on Network Protocols, ICNP 2018, Cambridge, UK, September 25-27, 2018*. IEEE Computer Society, 2018, pp. 406–411. DOI: 10.1109/ICNP.2018.00056. URL: https://doi.org/10.1109/ICNP.2018.00056.

[26] Feras Daoud, Amir Wated, and Mark Silberstein. "GPUrdma: GPU-side library for high performance networking from GPU kernels." In: *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers, Kyoto, Japan, June 1, 2016*. Ed. by Kamil Iskra and Torsten Hoefler. ACM, 2016, 6:1–6:8. DOI: 10.1145/2931088.2931091. URL: https://doi.org/10.1145/2931088.2931091.

[27] Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. "RDMA Reads: To Use or Not to Use?" In: *IEEE Data Eng. Bull.* 40.1 (2017), pp. 3–14. URL: http://sites.computer.org/debull/A17mar/p3.pdf.

[28] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. "FaRM: Fast Remote Memory." In: *Proceedings of the 11th USENIX Symposium*

on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014. Ed. by Ratul Mahajan and Ion Stoica. USENIX Association, 2014, pp. 401–414. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%5C%5C%C4%5C%5C%87.

[29] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. "No compromises: distributed transactions with consistency, availability, and performance." In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. Ed. by Ethan L. Miller and Steven Hand. ACM, 2015, pp. 54–70. DOI: 10.1145/2815400.2815425. URL: https://doi.org/10.1145/2815400.2815425.

[30] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. "Quantifying TPC-H Choke Points and Their Optimizations." In: *Proc. VLDB Endow.* 13.8 (2020), pp. 1206–1220. DOI: 10.14778/3389133.3389138. URL: http://www.vldb.org/pvldb/vol13/p1206-dreseler.pdf.

[31] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, and Weiwei Gong. "A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores." In: *Proc. VLDB Endow.* 12.12 (2019), pp. 2218–2229. DOI: 10.14778/3352063.3352137. URL: http://www.vldb.org/pvldb/vol12/p2218-dursun.pdf.

[32] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin J. Levandoski, Thomas Neumann, and Andrew Pavlo. "Main Memory Database Systems." In: *Found. Trends Databases* 8.1-2 (2017), pp. 1–130. DOI: 10.1561/1900000058. URL: https://doi.org/10.1561/1900000058.

[33] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. "In-memory database acceleration on FPGAs: a survey." In: *VLDB J.* 29.1 (2020), pp. 33–59. DOI: 10.1007/s00778-019-00581-w. URL: https://doi.org/10.1007/s00778-019-00581-w.

[34] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. "SAP HANA database: data management for modern business applications." In: *SIGMOD Rec.* 40.4 (2011), pp. 45–51. DOI: 10.1145/2094114.2094126. URL: https://doi.org/10.1145/2094114.2094126.

[35] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory." In: *36th IEEE International Conference on Data Engineering,*

ICDE 2020, Dallas, TX, USA, April 20-24, 2020. IEEE, 2020, pp. 1477–1488. DOI: 10.1109/ICDE48307.2020.00131. URL: https://doi.org/10.1109/ICDE48307.2020.00131.

[36] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. "Azure Accelerated Networking: SmartNICs in the Public Cloud." In: *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018.* Ed. by Sujata Banerjee and Srinivasan Seshan. USENIX Association, 2018, pp. 51–66. URL: https://www.usenix.org/conference/nsdi18/presentation/firestone.

[37] Philip Werner Frey and Gustavo Alonso. "Minimizing the Hidden Cost of RDMA." In: *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada.* IEEE Computer Society, 2009, pp. 553–560. DOI: 10.1109/ICDCS.2009.32. URL: https://doi.org/10.1109/ICDCS.2009.32.

[38] Philip Werner Frey, Romulo Goncalves, Martin L. Kersten, and Jens Teubner. "A Spinning Join That Does Not Get Dizzy." In: *2010 International Conference on Distributed Computing Systems, ICDCS 2010, Genova, Italy, June 21-25, 2010.* IEEE Computer Society, 2010, pp. 283–292. DOI: 10.1109/ICDCS.2010.23. URL: https://doi.org/10.1109/ICDCS.2010.23.

[39] Philip Werner Frey, Romulo Goncalves, Martin L. Kersten, and Jens Teubner. "Spinning relations: high-speed networks for distributed join processing." In: *Proceedings of the Fifth International Workshop on Data Management on New Hardware, DaMoN 2009, Providence, Rhode Island, USA, June 28, 2009.* Ed. by Peter A. Boncz and Kenneth A. Ross. ACM, 2009, pp. 27–33. DOI: 10.1145/1565694.1565701. URL: https://doi.org/10.1145/1565694.1565701.

[40] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. "Pipelined Query Processing in Coprocessor Environments." In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018.* Ed. by Gautam Das, Christopher M.

Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 1603–1618. DOI: 10.1145/3183713.3183734. URL: https://doi.org/10.1145/3183713.3183734.

[41] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenberg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim, and Eitan Zahavi. "Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction." In: *First International Workshop on Communication Optimizations in HPC, COMHPC@SC 2016, Salt Lake City, UT, USA, November 18, 2016*. IEEE, 2016, pp. 1–10. DOI: 10.1109/COMHPC.2016.006. URL: https://doi.org/10.1109/COMHPC.2016.006.

[42] William Gropp et al. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, 2014. ISBN: 0262527634, 9780262527637.

[43] Chengxin Guo, Hong Chen, Feng Zhang, and Cuiping Li. "Distributed Join Algorithms on Multi-CPU Clusters with GPUDirect RDMA." In: *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019*. ACM, 2019, 65:1–65:10. DOI: 10.1145/3337821.3337862. URL: https://doi.org/10.1145/3337821.3337862.

[44] Khaled Hamidouche, Akshay Venkatesh, Ammar Ahmad Awan, Hari Subramoni, Ching-Hsiang Chu, and Dhabaleswar K. Panda. "Exploiting GPUDirect RDMA in Designing High Performance OpenSHMEM for NVIDIA GPU Clusters." In: *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*. IEEE Computer Society, 2015, pp. 78–87. DOI: 10.1109/CLUSTER.2015.21. URL: https://doi.org/10.1109/CLUSTER.2015.21.

[45] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. "Relational query coprocessing on graphics processors." In: *ACM Trans. Database Syst.* 34.4 (2009), 21:1–21:39. DOI: 10.1145/1620585.1620588. URL: https://doi.org/10.1145/1620585.1620588.

[46] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. "Relational joins on graphics processors." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. Ed. by Jason Tsong-Li Wang. ACM, 2008, pp. 511–524. DOI: 10.1145/1376616.1376670. URL: https://doi.org/10.1145/1376616.1376670.

[47] Dong He, Supun Chathuranga Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodrıguez, Konstantinos Karanasos, and Matteo Interlandi. "Query Processing on Tensor Computation Runtimes." In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2811–2825. URL: https://www.vldb.org/pvldb/vol15/p2811-he.pdf.

[48] Jiong He, Mian Lu, and Bingsheng He. "Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture." In: *Proc. VLDB Endow.* 6.10 (2013), pp. 889–900. DOI: 10.14778/2536206.2536216. URL: http://www.vldb.org/pvldb/vol6/p889-he.pdf.

[49] Torsten Hoefler, Duncan Roweth, Keith D. Underwood, Bob Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyuan Shen, Abdul Kabbani, Moray McLaren, and Steve Scott. "Datacenter Ethernet and RDMA: Issues at Hyperscale." In: *CoRR* abs/2302.03337 (2023). DOI: 10.48550/arXiv.2302.03337. arXiv: 2302.03337. URL: https://doi.org/10.48550/arXiv.2302.03337.

[50] Jaco A. Hofmann, Lasse Thostrup, Tobias Ziegler, Carsten Binnig, and Andreas Koch. "High-Performance In-Network Data Processing." In: *10th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2019, Los Angeles, California, USA, August 26, 2019.* Ed. by Rajesh Bordawekar and Tirthankar Lahiri. 2019, pp. 64–73. URL: http://www.adms-conf.org/2019-camera-ready/hofmann_adms19.pdf.

[51] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. "The Performance of Database Replication with Group Multicast." In: *Digest of Papers: FTCS-29, The Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, Madison, Wisconsin, USA, June 15-18, 1999.* IEEE Computer Society, 1999, pp. 158–165. DOI: 10.1109/FTCS.1999.781046. URL: https://doi.org/10.1109/FTCS.1999.781046.

[52] Intel. *Intel In-Memory Analytics Accelerator Architecture Specification.* https://www.intel.com/content/www/us/en/content-details/721858/intel-in-memory-analytics-accelerator-architecture-specification.html. Intel, Sept. 2023.

[53] Intel. *Intel P4 Studio*. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-studio.html. 2023.

[54] Intel. *Intel® Tofino™ Series*. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html. 2023.

[55] Intel. *Scalable I/O Between Accelerators and Host Processors*. https://www.intel.com/content/www/us/en/developer/articles/technical/scalable-io-between-accelerators-host-processors.html. Intel, Sept. 2023.

[56] Virajith Jalaparti, Peter Bodık, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. "Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can." In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*. Ed. by Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye. ACM, 2015, pp. 407–420. DOI: 10.1145/2785956.2787488. URL: https://doi.org/10.1145/2785956.2787488.

[57] Matthias Jasny and Lasse Thostrup. *Zerosided RDMA Code*. https://github.com/DataManagementLab/zerosided_rdma. 2023.

[58] Matthias Jasny, Lasse Thostrup, and Carsten Binnig. "Zero-sided RDMA: Network-driven Data Shuffling." In: *Proceedings of the 19th International Workshop on Data Management on New Hardware, DaMoN 2023, Seattle, WA, USA, June 18-23, 2023*. Ed. by Norman May and Nesime Tatbul. ACM, 2023, pp. 82–85. DOI: 10.1145/3592980.3595302. URL: https://doi.org/10.1145/3592980.3595302.

[59] Matthias Jasny, Lasse Thostrup, Sajjad Tamimi, Andreas Koch, Zsolt István, and Carsten Binnig. "Zero-sided RDMA: Network-driven Data Shuffling for Disaggregated Heterogeneous Cloud DBMSs." In: *Proc. ACM Manag. Data* 2.1 (Mar. 2024). DOI: 10.1145/3639291. URL: https://doi.org/10.1145/3639291.

[60] Matthias Jasny, Lasse Thostrup, Tobias Ziegler, and Carsten Binnig. "P4DB - The Case for In-Network OLTP." In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Zachary G. Ives, Angela Bonifati, and Amr El Abbadi. ACM, 2022, pp. 1375–1389. DOI: 10.1145/3514221.3517825. URL: https://doi.org/10.1145/3514221.3517825.

[61] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. "Life in the Fast Lane: A Line-Rate Linear Road." In: *Proceedings of the Symposium on SDN Research, SOSR 2018, Los Angeles, CA, USA, March 28-29, 2018*. ACM, 2018, 10:1–10:7. DOI: 10.1145/3185467.3185494. URL: https://doi.org/10.1145/3185467.3185494.

[62] Chengfan Jia, Junnan Liu, Xu Jin, Han Lin, Hong An, Wenting Han, Zheng Wu, and Mengxian Chi. "Improving the Performance of Distributed TensorFlow with RDMA." In: *Int. J. Parallel Program.* 46.4 (2018), pp. 674–685. DOI: 10.1007/s10766-017-0520-3. URL: https://doi.org/10.1007/s10766-017-0520-3.

[63] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. "NetChain: Scale-Free Sub-RTT Coordination." In: *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*. Ed. by Sujata Banerjee and Srinivasan Seshan. USENIX Association, 2018, pp. 35–49. URL: https://www.usenix.org/conference/nsdi18/presentation/jin.

[64] Michael Jungmair and Jana Giceva. "Declarative Sub-Operators for Universal Data Processing." In: *Proc. VLDB Endow.* 16.11 (Aug. 2023), pp. 3461–3474. ISSN: 2150-8097. DOI: 10.14778/3611479.3611539. URL: https://doi.org/10.14778/3611479.3611539.

[65] Tim Kaldewey, Guy M. Lohman, René Müller, and Peter Benjamin Volk. "GPU join processing revisited." In: *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012*. Ed. by Shimin Chen and Stavros Harizopoulos. ACM, 2012, pp. 55–62. DOI: 10.1145/2236584.2236592. URL: https://doi.org/10.1145/2236584.2236592.

[66] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Datacenter RPCs Can Be General and Fast." In: *login Usenix Mag.* 44.2 (2019). URL: https://www.usenix.org/publications/login/summer2019/kalia.

[67] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Design Guidelines for High Performance RDMA Systems." In: *login Usenix Mag.* 41.3 (2016). URL: https://www.usenix.org/publications/login/fall2016/kalia.

[68] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs." In: *12th USENIX Symposium on Operating Systems Design and Implementation,*

OSDI 2016, Savannah, GA, USA, November 2-4, 2016. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 185–201. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia.

[69] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Using RDMA efficiently for key-value services." In: *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*. Ed. by Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy. ACM, 2014, pp. 295–306. DOI: 10.1145/2619239.2626299. URL: https://doi.org/10.1145/2619239.2626299.

[70] Sol Ji Kang, Sang-Hoon Lee, and Keon-Myung Lee. "Performance Comparison of OpenMP, MPI, and MapReduce in Practical Problems." In: *Adv. Multim.* 2015 (2015), 575687:1–575687:9. DOI: 10.1155/2015/575687. URL: https://doi.org/10.1155/2015/575687.

[71] Bettina Kemme and Gustavo Alonso. "Database Replication: a Tale of Research across Communities." In: *Proc. VLDB Endow.* 3.1 (2010), pp. 5–12. DOI: 10.14778/1920841.1920847. URL: http://www.vldb.org/pvldb/vldb2010/pvldb%5C%5C_vol3/TY02.pdf.

[72] Jens Korinth, Jaco A. Hofmann, Carsten Heinz, and Andreas Koch. "The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems." In: *Applied Reconfigurable Computing - 15th International Symposium, ARC 2019, Darmstadt, Germany, April 9-11, 2019, Proceedings*. Ed. by Christian Hochberger, Brent Nelson, Andreas Koch, Roger F. Woods, and Pedro C. Diniz. Vol. 11444. Lecture Notes in Computer Science. Springer, 2019, pp. 214–229. DOI: 10.1007/978-3-030-17227-5\\_16. URL: https://doi.org/10.1007/978-3-030-17227-5%5C%5C_16.

[73] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. "Do OS abstractions make sense on FPGAs?" In: *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020, pp. 991–1010. URL: https://www.usenix.org/conference/osdi20/presentation/roscoe.

[74] Leslie Lamport et al. "Paxos made simple." In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[75] Robert Lasch, Mehdi Moghaddamfar, Norman May, Süleyman Sirri Demirsoy, Christian Färber, and Kai-Uwe Sattler. "Bandwidth-optimal Relational Joins on FPGAs." In: *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*. Ed. by Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang. OpenProceedings.org, 2022, 1:27–1:39. DOI: 10.5441/002/edbt.2022.03. URL: https://doi.org/10.5441/002/edbt.2022.03.

[76] Long Hoang Le, Mojtaba Eslahi-Kelorazi, Paulo R. Coelho, and Fernando Pedone. "RamCast: RDMA-based atomic multicast." In: *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*. Ed. by Kaiwen Zhang, Abdelouahed Gherbi, Nalini Venkatasubramanian, and Luıs Veiga. ACM, 2021, pp. 172–184. DOI: 10.1145/3464298.3493393. URL: https://doi.org/10.1145/3464298.3493393.

[77] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age." In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. Ed. by Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu. ACM, 2014, pp. 743–754. DOI: 10.1145/2588555.2610507. URL: https://doi.org/10.1145/2588555.2610507.

[78] Viktor Leis, Michael Haubenschild, and Thomas Neumann. "Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method." In: *IEEE Data Eng. Bull.* 42.1 (2019), pp. 73–84. URL: http://sites.computer.org/debull/A19mar/p73.pdf.

[79] Alberto Lerner, Rana Hussein, and Philippe Cudré-Mauroux. "The Case for Network Accelerated Query Processing." In: *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. URL: http://cidrdb.org/cidr2019/papers/p142-lerner-cidr19.pdf.

[80] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC." In: *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 137–152. DOI: 10.1145/3132747.3132756. URL: https://doi.org/10.1145/3132747.3132756.

[81] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. “Accelerating Relational Databases by Leveraging Remote Memory and RDMA.” In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Fatma Özcan, Georgia Koutrika, and Sam Madden. ACM, 2016, pp. 355–370. DOI: 10.1145/2882903.2882949. URL: https://doi.org/10.1145/2882903.2882949.

[82] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. “Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering.” In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 467–483. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li.

[83] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. “Scaling Distributed Machine Learning with the Parameter Server.” In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014*. Ed. by Jason Flinn and Hank Levy. USENIX Association, 2014, pp. 583–598. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li%5C%5C_mu.

[84] Feilong Liu, Claude Barthels, Spyros Blanas, Hideaki Kimura, and Garret Swart. “Beyond MPI: New Communication Interfaces for Database Systems and Data-Intensive Applications.” In: *SIGMOD Rec.* 49.4 (2020), pp. 12–17. DOI: 10.1145/3456859.3456862. URL: https://doi.org/10.1145/3456859.3456862.

[85] Feilong Liu, Lingyan Yin, and Spyros Blanas. “Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems.” In: *ACM Trans. Database Syst.* 44.4 (2019), 17:1–17:45. DOI: 10.1145/3360900. URL: https://doi.org/10.1145/3360900.

[86] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. “High Performance RDMA-Based MPI Implementation over InfiniBand.” In: *Int. J. Parallel Program.* 32.3 (2004), pp. 167–198. DOI: 10.1023/B:IJPP.0000029272.69895.c1. URL: https://doi.org/10.1023/B:IJPP.0000029272.69895.c1.

[87] Xiaoyi Lu, Dipti Shankar, Shashank Gugnani, and Dhabaleswar K. Panda. “High-performance design of apache spark with RDMA and its benefits on various workloads.” In: *2016 IEEE International Conference on Big Data (IEEE BigData*

2016), Washington DC, USA, December 5-8, 2016. Ed. by James Joshi, George Karypis, Ling Liu, Xiaohua Hu, Ronay Ak, Yinglong Xia, Weijia Xu, Aki-Hiro Sato, Sudarsan Rachuri, Lyle H. Ungar, Philip S. Yu, Rama Govindaraju, and Toyotaro Suzumura. IEEE Computer Society, 2016, pp. 253–262. DOI: 10.1109/BigData.2016.7840611. URL: https://doi.org/10.1109/BigData.2016.7840611.

[88] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. "Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 1633–1649. DOI: 10.1145/3318464.3389705. URL: https://doi.org/10.1145/3318464.3389705.

[89] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. "Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects." In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Zachary G. Ives, Angela Bonifati, and Amr El Abbadi. ACM, 2022, pp. 1017–1032. DOI: 10.1145/3514221.3517911. URL: https://doi.org/10.1145/3514221.3517911.

[90] Mailinglist. *[RFC 6/7] IB/core: Peer memory client for IO memory*. https://www.spinics.net/lists/linux-rdma/msg33298.html. 2023.

[91] Christopher Mitchell, Yifeng Geng, and Jinyang Li. "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store." In: *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*. Ed. by Andrew Birrell and Emin Gün Sirer. USENIX Association, 2013, pp. 103–114. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell.

[92] MPICH. *Manpage: MPI Alltoall*. https://www.mpich.org/static/docs/latest/www3/MPI_Alltoall.html. 2023.

[93] APS Networks. *Intel Tofino APS Networks BF2556X-1T-A1F*. https://www.opencompute.org/documents/210216-bf2556x-1t-switch-specifications-v2-pdf-1. 2019.

[94] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. "Understanding PCIe performance for end host networking." In: *Proceedings of the 2018 Conference of the ACM Special*

Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018. Ed. by Sergey Gorinsky and János Tapolcai. ACM, 2018, pp. 327–341. DOI: 10.1145/3230543.3230560. URL: https://doi.org/10.1145/3230543.3230560.

[95] Joel Nider and Alexandra (Sasha) Fedorova. "The last CPU." In: *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*. Ed. by Sebastian Angel, Baris Kasikci, and Eddie Kohler. ACM, 2021, pp. 1–8. DOI: 10.1145/3458336.3465291. URL: https://doi.org/10.1145/3458336.3465291.

[96] NVIDIA. *BlueField DPU OS - Functional Diagram*. NVIDIA. July 2022. URL: https://docs.nvidia.com/networking/display/BlueFieldDPUOSLatest/Functional+Diagram.

[97] NVIDIA. *GPUDirect RDMA*. NVIDIA. Mar. 2023. URL: https://developer.nvidia.com/gpudirect.

[98] NVIDIA. *GPUDirect RDMA*. NVIDIA, Mar. 2023. URL: https://developer.nvidia.com/gpudirect.

[99] NVIDIA. *GPUDirect RDMA Design Considerations - Synchronization and Memory Ordering*. NVIDIA. June 2021. URL: https://docs.nvidia.com/cuda/gpudirect-rdma/index.html%5C#sync-behavior.

[100] NVIDIA. *HowTo Implement PeerDirect Client using MLNX OFED*. https://enterprise-support.nvidia.com/s/article/howto-implement-peerdirect-client-using-mlnx-ofed. 2023.

[101] NVIDIA. *Mellanox OFED GPUDirect RDMA*. NVIDIA. June 2021. URL: https://www.mellanox.com/products/GPUDirect-RDMA.

[102] NVIDIA. *NVIDIA BLUEFIELD DATA PROCESSING UNITS*. NVIDIA. June 2022. URL: https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[103] NVIDIA. *Nvidia NVSHMEM*. https://developer.nvidia.com/nvshmem. NVIDIA, 2023.

[104] NVIDIA. *RDMA Over Converged Ethernet (RoCE)*. https://docs.nvidia.com/networking/m/view-rendered-page.action?abstractPageId=56986516. NVIDIA, Mar. 2023.

[105] Diego Ongaro and John K. Ousterhout. “In Search of an Understandable Consensus Algorithm.” In: *2014 USENIX Annual Technical Conference, USENIX ATC ’14, Philadelphia, PA, USA, June 19-20, 2014*. Ed. by Garth Gibson and Nickolai Zeldovich. USENIX Association, 2014, pp. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[106] OpenUCX. *SparkUCX ShuffleManager Plugin*. https://github.com/openucx/sparkucx. 2023.

[107] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. “MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures.” In: *SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2021, pp. 1413–1425. DOI: 10.1145/3448016.3457254. URL: https://doi.org/10.1145/3448016.3457254.

[108] Marius Poke and Torsten Hoefler. “DARE: High-Performance State Machine Replication on RDMA Networks.” In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015, Portland, OR, USA, June 15-19, 2015*. Ed. by Thilo Kielmann, Dean Hildebrand, and Michela Taufer. ACM, 2015, pp. 107–118. DOI: 10.1145/2749246.2749267. URL: https://doi.org/10.1145/2749246.2749267.

[109] Dan R. K. Ports and Jacob Nelson. “When Should The Network Be The Computer?” In: *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*. ACM, 2019, pp. 209–215. DOI: 10.1145/3317550.3321439. URL: https://doi.org/10.1145/3317550.3321439.

[110] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. “Flow-Join: Adaptive skew handling for distributed joins over high-speed networks.” In: *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 2016, pp. 1194–1205. DOI: 10.1109/ICDE.2016.7498324. URL: https://doi.org/10.1109/ICDE.2016.7498324.

[111] *Flow-Join: Adaptive skew handling for distributed joins over high-speed networks.* IEEE Computer Society, 2016, pp. 1194–1205. DOI: 10.1109/ICDE.2016.7498324. URL: https://doi.org/10.1109/ICDE.2016.7498324.

[112] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. "High-Speed Query Processing over High-Speed Networks." In: *Proc. VLDB Endow.* 9.4 (2015), pp. 228–239. DOI: 10.14778/2856318.2856319. URL: http://www.vldb.org/pvldb/vol9/p228-roediger.pdf.

[113] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. "ReDMArk: Bypassing RDMA Security Mechanisms." In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021.* Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 4277–4292. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/rothenberger.

[114] Ran Rui, Hao Li, and Yi-Cheng Tu. "Efficient Join Algorithms For Large Database Tables in a Multi-GPU Environment." In: *Proc. VLDB Endow.* 14.4 (2020), pp. 708–720. DOI: 10.14778/3436905.3436927. URL: http://www.vldb.org/pvldb/vol14/p708-rui.pdf.

[115] Ran Rui, Hao Li, and Yi-Cheng Tu. "Join algorithms on GPUs: A revisit after seven years." In: *2015 IEEE International Conference on Big Data (IEEE BigData 2015), Santa Clara, CA, USA, October 29 - November 1, 2015.* IEEE Computer Society, 2015, pp. 2541–2550. DOI: 10.1109/BigData.2015.7364051. URL: https://doi.org/10.1109/BigData.2015.7364051.

[116] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. "In-Network Computation is a Dumb Idea Whose Time Has Come." In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017.* Ed. by Sujata Banerjee, Brad Karp, and Michael Walfish. ACM, 2017, pp. 150–156. DOI: 10.1145/3152434.3152461. URL: https://doi.org/10.1145/3152434.3152461.

[117] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. "Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010.* Ed. by Ahmed K. Elmagarmid and Divyakant Agrawal. ACM, 2010, pp. 351–362. DOI: 10.1145/1807167.1807207. URL: https://doi.org/10.1145/1807167.1807207.

[118] Amazon Web Services. *Elastic Fabric Adapter.* https://aws.amazon.com/hpc/efa/. Amazon, Sept. 2023.

[119] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. “A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics.” In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 1617–1632. DOI: 10.1145/3318464.3380595. URL: https://doi.org/10.1145/3318464.3380595.

[120] David Sidler, Zsolt István, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. “doppioDB: A Hardware Accelerated Database.” In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu. ACM, 2017, pp. 1659–1662. DOI: 10.1145/3035918.3058746. URL: https://doi.org/10.1145/3035918.3058746.

[121] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. “1RMA: Re-envisioning Remote Memory Access for Multi-tenant Datacenters.” In: *SIGCOMM ’20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*. Ed. by Henning Schulzrinne and Vishal Misra. ACM, 2020, pp. 708–721. DOI: 10.1145/3387514.3405897. URL: https://doi.org/10.1145/3387514.3405897.

[122] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. “Hardware-Conscious Hash-Joins on GPUs.” In: *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 2019, pp. 698–709. DOI: 10.1109/ICDE.2019.00068. URL: https://doi.org/10.1109/ICDE.2019.00068.

[123] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. “Distributed GPU Joins on Fast RDMA-capable Networks.” In: *Proc. ACM Manag. Data* 1.1 (2023), 29:1–29:26. DOI: 10.1145/3588709. URL: https://doi.org/10.1145/3588709.

[124] Lasse Thostrup, Daniel Failing, Tobias Ziegler, and Carsten Binnig. “A DBMS-centric Evaluation of BlueField DPUs on Fast Networks.” In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Pro-*

cessor and Storage Architectures, ADMS@VLDB 2022, Sydney, Australia, September 5, 2022. Ed. by Rajesh Bordawekar and Tirthankar Lahiri. 2022, pp. 1–10. URL: http://www.adms-conf.org/2022-camera-ready/ADMS22_thostrup.pdf.

[125] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. "DFI: The Data Flow Interface for High-Speed Networks." In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. Best Paper Award. ACM, 2021, pp. 1825–1837. DOI: 10.1145/3448016.3452816. URL: https://doi.org/10.1145/3448016.3452816.

[126] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. "DFI: The Data Flow Interface for High-Speed Networks." In: *SIGMOD Rec.* 51.1 (2022). Research Highlight Award, pp. 15–22. DOI: 10.1145/3542700.3542705. URL: https://doi.org/10.1145/3542700.3542705.

[127] Da Tong, Shijie Zhou, and Viktor K. Prasanna. "High-Throughput Online Hash Table on FPGA." In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*. IEEE Computer Society, 2015, pp. 105–112. DOI: 10.1109/IPDPSW.2015.149. URL: https://doi.org/10.1109/IPDPSW.2015.149.

[128] Maroun Tork, Lina Maudlej, and Mark Silberstein. "Lynx: A SmartNIC-driven Accelerator-centric Architecture for Network Servers." In: *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. Ed. by James R. Larus, Luis Ceze, and Karin Strauss. ACM, 2020, pp. 117–131. DOI: 10.1145/3373376.3378528. URL: https://doi.org/10.1145/3373376.3378528.

[129] Animesh Trivedi and Marco Spaziani Brunella. "CPU-free Computing: A Vision with a Blueprint." In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS 2023, Providence, RI, USA, June 22-24, 2023*. Ed. by Malte Schwarzkopf, Andrew Baumann, and Natacha Crooks. ACM, 2023, pp. 1–14. DOI: 10.1145/3593856.3595906. URL: https://doi.org/10.1145/3593856.3595906.

[130] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases." In: *Proceedings of the 2017*

ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017. Ed. by Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu. ACM, 2017, pp. 1041–1052. DOI: 10.1145/3035918.3056101. URL: https://doi.org/10.1145/3035918.3056101.

[131] Lluıs Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. "Slashing the disaggregation tax in heterogeneous data centers with FractOS." In: *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022.* Ed. by Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis. ACM, 2022, pp. 352–367. DOI: 10.1145/3492321.3519569. URL: https://doi.org/10.1145/3492321.3519569.

[132] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. "Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads." In: *Proc. VLDB Endow.* 15.10 (2022), pp. 1991–2004. URL: https://www.vldb.org/pvldb/vol15/p1991-petrov.pdf.

[133] Christopher B. Walton, Alfred G. Dale, and Roy M. Jenevein. "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins." In: *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings.* Ed. by Guy M. Lohman, Amılcar Sernadas, and Rafael Camps. Morgan Kaufmann, 1991, pp. 537–548. URL: http://www.vldb.org/conf/1991/P537.PDF.

[134] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. "APUS: fast and scalable paxos on RDMA." In: *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017.* ACM, 2017, pp. 94–107. DOI: 10.1145/3127479.3128609. URL: https://doi.org/10.1145/3127479.3128609.

[135] Qing Wang, Youyou Lu, and Jiwu Shu. "Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory." In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022.* Ed. by Zachary Ives, Angela Bonifati, and Amr El Abbadi. ACM, 2022, pp. 1033–1048. DOI: 10.1145/3514221.3517824. URL: https://doi.org/10.1145/3514221.3517824.

[136] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. “FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs.” In: *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. Ed. by Jiri Schindler and Noa Zilberman. USENIX Association, 2022, pp. 967–986. URL: https://www.usenix.org/conference/atc22/presentation/wang-zeke.

[137] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. “Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!” In: *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. Ed. by Andrea C. Arpaci-Dusseau and Geoff Voelker. USENIX Association, 2018, pp. 233–251. URL: https://www.usenix.org/conference/osdi18/presentation/wei.

[138] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. “Fast in-memory transaction processing using RDMA and HTM.” In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. Ed. by Ethan L. Miller and Steven Hand. ACM, 2015, pp. 87–104. DOI: 10.1145/2815400.2815419. URL: https://doi.org/10.1145/2815400.2815419.

[139] Wei Liang, Wenbo Yin, Ping Kang, and Lingli Wang. “Memory efficient and high performance key-value store on FPGA using Cuckoo hashing.” In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016, pp. 1–4. DOI: 10.1109/FPL.2016.7577355.

[140] Joachim Worringen. “Pipelining and Overlapping for MPI Collective Operations.” In: *28th Annual IEEE Conference on Local Computer Networks (LCN 2003), The Conference on Leading Edge and Practical Computer Networking, 20-24 October 2003, Bonn/Königswinter, Germany, Proceedings*. IEEE Computer Society, 2003, pp. 548–557. DOI: 10.1109/LCN.2003.1243181. URL: https://doi.org/10.1109/LCN.2003.1243181.

[141] Haicheng Wu, Gregory F. Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. “Red Fox: An Execution Environment for Relational Query Processing on GPUs.” In: *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*. Ed. by David R. Kaeli and Tipp Moseley. ACM, 2014, p. 44. URL: https://dl.acm.org/citation.cfm?id=2544166.

[142] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. “Fast Distributed Deep Learning over RDMA.” In: *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*. Ed. by George Candea, Robbert van Renesse, and Christof Fetzer. ACM, 2019, 44:1–44:14. DOI: 10.1145/3302424.3303975. URL: https://doi.org/10.1145/3302424.3303975.

[143] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. “Distributed Lock Management with RDMA: Decentralization without Starvation.” In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 1571–1586. DOI: 10.1145/3183713.3196890. URL: https://doi.org/10.1145/3183713.3196890.

[144] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. “The Yin and Yang of Processing Data Warehousing Queries on GPU Devices.” In: *Proc. VLDB Endow.* 6.10 (2013), pp. 817–828. DOI: 10.14778/2536206.2536210. URL: http://www.vldb.org/pvldb/vol6/p817-yuan.pdf.

[145] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. “Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling.” In: *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*. Ed. by Christine Morin and Gilles Muller. ACM, 2010, pp. 265–278. DOI: 10.1145/1755913.1755940. URL: https://doi.org/10.1145/1755913.1755940.

[146] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. “The End of a Myth: Distributed Transactions Can Scale.” In: *Proc. VLDB Endow.* 10.6 (Feb. 2017), pp. 685–696. ISSN: 2150-8097. DOI: 10.14778/3055330.3055335. URL: https://doi.org/10.14778/3055330.3055335.

[147] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. “The End of a Myth: Distributed Transaction Can Scale.” In: *Proc. VLDB Endow.* 10.6 (2017), pp. 685–696. DOI: 10.14778/3055330.3055335. URL: http://www.vldb.org/pvldb/vol10/p685-zamanian.pdf.

[148] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. “Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks.” In:

SIGMOD Rec. 50.1 (2021), pp. 15–22. DOI: 10.1145/3471485.3471490. URL: https://doi.org/10.1145/3471485.3471490.

[149] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. “FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory.” In: *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*. Ed. by Dean Hildebrand and Donald E. Porter. USENIX Association, 2022, pp. 51–68. URL: https://www.usenix.org/conference/fast22/presentation/zhang-ming.

[150] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. “Rethinking Data Management Systems for Disaggregated Data Centers.” In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: http://cidrdb.org/cidr2020/papers/p6-zhang-cidr20.pdf.

[151] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. “Optimizing Data-intensive Systems in Disaggregated Data Centers with TELEPORT.” In: *SIGMOD ’22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Zachary G. Ives, Angela Bonifati, and Amr El Abbadi. ACM, 2022, pp. 1345–1359. DOI: 10.1145/3514221.3517856. URL: https://doi.org/10.1145/3514221.3517856.

[152] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. “FoundationDB: A Distributed Unbundled Transactional Key Value Store.” In: *SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2021, pp. 2653–2666. DOI: 10.1145/3448016.3457559. URL: https://doi.org/10.1145/3448016.3457559.

[153] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. “Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection.” In: *Proc. VLDB Endow.* 13.3 (2019), pp. 376–389. DOI: 10.14778/3368289.3368301. URL: http://www.vldb.org/pvldb/vol13/p376-zhu.pdf.

[154] Tobias Ziegler, Carsten Binnig, and Viktor Leis. “ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA.” In: *SIGMOD ’22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Zachary G. Ives, Angela Bonifati, and Amr El Abbadi. ACM, 2022, pp. 685–699. DOI: 10.1145/3514221.3526187. URL: https://doi.org/10.1145/3514221.3526187.

[155] Tobias Ziegler, Carsten Binnig, and Uwe Röhm. “Skew-resilient Query Processing for Fast Networks.” In: *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Workshopband*. Ed. by Holger Meyer, Norbert Ritter, Andreas Thor, Daniela Nicklas, Andreas Heuer, and Meike Klettke. Vol. P-290. LNI. Gesellschaft für Informatik, Bonn, 2019, pp. 81–85. DOI: 10.18420/btw2019-ws-06. URL: https://doi.org/10.18420/btw2019-ws-06.

[156] Tobias Ziegler, Viktor Leis, and Carsten Binnig. “RDMA Communciation Patterns.” In: *Datenbank-Spektrum* 20.3 (2020), pp. 199–210. DOI: 10.1007/s13222-020-00355-7. URL: https://doi.org/10.1007/s13222-020-00355-7.

[157] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. “Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks.” In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska. ACM, 2019, pp. 741–758. DOI: 10.1145/3299869.3300081. URL: https://doi.org/10.1145/3299869.3300081.

[158] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. “NetFPGA SUME: Toward 100 Gbps as Research Commodity.” In: *IEEE Micro* 34.5 (2014), pp. 32–41. DOI: 10.1109/MM.2014.61. URL: https://doi.org/10.1109/MM.2014.61.

[159] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. “One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory.” In: *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. Ed. by Irina Calciu and Geoff Kuenning. USENIX Association, 2021, pp. 15–29. URL: https://www.usenix.org/conference/atc21/presentation/zuo.