Towards Scalable OLTP Over Fast Networks

Tobias Ziegler

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Towards Scalable OLTP Over Fast Networks

Doctoral thesis by
Tobias Ziegler, M.Sc.

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

Reviewers
Prof. Dr. rer. nat. Carsten Binnig
Prof. Spyros Blanas, Ph.D.

Department of Computer Science
Technical University of Darmstadt

Darmstadt, 2023

Tobias Ziegler: *Towards Scalable OLTP Over Fast Networks*
Darmstadt, Technical University of Darmstadt
Year thesis published in TUprints: 2023
URN: urn:nbn:de:tuda-tuprints-247162
URL: https://tuprints.ulb.tu-darmstadt.de/24716
Date of the viva voce: 22.08.2023

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, June 27, 2023

Tobias Ziegler

Abstract

Online Transaction Processing (OLTP) underpins real-time data processing in many mission-critical applications, from banking to e-commerce. These applications typically issue short-duration, latency-sensitive transactions that demand immediate processing [72]. High-volume applications, such as Alibaba's e-commerce platform [87], achieve peak transaction rates as high as 70 million transactions per second, exceeding the capacity of a single machine. Instead, distributed OLTP database management systems (DBMS) are deployed across multiple powerful machines. Historically, such distributed OLTP DBMSs have been primarily designed to avoid network communication, a paradigm largely unchanged since the 1980s.

However, fast networks challenge the conventional belief that network communication is the main bottleneck. In particular, emerging network technologies, like Remote Direct Memory Access (RDMA), radically alter how data can be accessed over a network. RDMA's primitives allow direct access to the memory of a remote machine within an order of magnitude of local memory access [20]. This development invalidates the notion that network communication is the primary bottleneck. Given that traditional distributed database systems have been designed with the premise that the network is slow, they cannot efficiently exploit these fast network primitives, which requires us to reconsider how we design distributed OLTP systems.

This thesis focuses on the challenges RDMA presents and its implications on the design of distributed OLTP systems. First, we examine distributed architectures to understand data access patterns and scalability in modern OLTP systems. Drawing on these insights, we advocate a distributed storage engine optimized for high-speed networks. The storage engine serves as the foundation of a database, ensuring efficient data access through three central components: *indexes, synchronization primitives, and buffer management (caching)*. With the introduction of RDMA, the landscape of data access has undergone a significant transformation. This requires a comprehensive redesign of the storage engine components to exploit the potential of RDMA and similar high-speed network technologies. Thus, as the second contribution, we design RDMA-optimized tree-based indexes especially applicable for disaggregated databases to access remote data efficiently. We then turn our attention to the unique challenges of RDMA. One-sided RDMA, one of the network primitives introduced by RDMA, presents a performance advantage in enabling remote memory access while bypassing the remote CPU and the operating system. This allows the remote CPU to process transactions uninterrupted, with no requirement to be on hand for network communication. However, that way, specialized one-sided

RDMA synchronization primitives are required since traditional CPU-driven primitives are bypassed. We found that existing RDMA one-sided synchronization schemes are unscalable or, even worse, fail to synchronize correctly, leading to hard-to-detect data corruption. As our third contribution, we address this issue by offering guidelines to build scalable and correct one-sided RDMA synchronization primitives. Finally, recognizing that maintaining all data in memory becomes economically unattractive, we propose a distributed buffer manager design that efficiently utilizes cost-effective NVMe flash storage. By leveraging low-latency RDMA messages, our buffer manager provides a transparent memory abstraction, accessing the aggregated DRAM and NVMe storage across nodes. Central to our approach is a distributed caching protocol that dynamically caches data. With this approach, our system can outperform RDMA-enabled in-memory distributed databases while managing larger-than-memory datasets efficiently.

Zusammenfassung

Die Bedeutung von Online-Transaktionsverarbeitung (OLTP) im Echtzeitbetrieb, beispielsweise in der Banken- und E-Commerce-Branche, ist immens. Anwendungen mit sehr hohem Datenaufkommen, wie beispielsweise Alibabas Onlinehandel, die bis zu 70 Millionen Transaktionen pro Sekunde aufweisen [87], übersteigen die Kapazitäten eines Einzelrechners. Um diese hohen Datendurchsätze bewältigen zu können, wird ein für das OLTP-Datenbankmanagementsystem (DBMS) verteilter Ansatz, welcher auf einer Vielzahl von Rechnern läuft, unerlässlich. Historisch gesehen wurde die Entwicklung solcher verteilter OLTP-Datenbanken vor allem mit dem Ziel vorangetrieben, die Netzwerkkommunikation zu reduzieren - eine Herangehensweise, die seit den 1980er-Jahren weitgehend konstant geblieben ist. Mit dem Aufkommen schneller Netzwerke wird jedoch eben diese Ansicht, dass die Netzwerkkommunikation der größte Engpass ist in Frage gestellt. Insbesondere neuartige Netzwerktechnologien wie Remote Direct Memory Access (RDMA) revolutionieren die Methode des Datenzugriffs über Netzwerke. Sie bieten Netzwerkprimitive, die einen direkten und schnellen Zugriff auf entfernten Arbeitsspeicher über das Netzwerk ermöglichen. Allerdings sind die für langsame Netzwerke konzipierten verteilten Datenbanksysteme nicht in der Lage, die Vorteile dieser Netzwerkprimitive vollständig zu nutzen. Diese Arbeit fokussiert sich auf die mit RDMA verbundenen Herausforderungen und die dadurch bedingten Auswirkungen auf die Gestaltung von verteilten OLTP-Systemen. Wir untersuchen zunächst verteilte Architekturen, um das Verständnis von Datenzugriffsmustern und Skalierbarkeit in modernen OLTP-Systemen zu vertiefen. Basierend auf diesen Erkenntnissen schlagen wir eine Speicher-Engine vor, die speziell für ein skalierbares, verteiltes OLTP-System optimiert ist.

Die Speicher-Engine, eine Kernkomponente jeder Datenbank, hat drei Hauptfunktionen: Bereitstellung von Indizes, Synchronisationsmechanismen und Puffermanagement (Caching). Angesichts der neuen Netzwerkprimitive, die RDMA zur Manipulation und zum Lesen von Daten über das Netzwerk bereitstellt, wird eine Überarbeitung dieser drei Kernfunktionen notwendig. Daher schlagen wir in unserem zweiten Beitrag die Entwicklung von baumstrukturierten Indizes vor, die speziell für den Einsatz mit RDMA und insbesondere für disaggregierte Datenbanken optimiert sind. Dies ermöglicht das effiziente Speichern, Finden und Zugreifen auf entfernte Daten über das Netzwerk. Ein entscheidender Vorteil von One-Sided RDMA (eine der neuen Netzwerkprimitive von RDMA) besteht darin, dass Daten über das Netzwerk manipuliert werden können, ohne dass die entfernte CPU involviert wird. Dies ermöglicht es der entfernten CPU, weiterhin Transaktionen zu bearbeiten, ohne dass sie für die Netzwerkkommunikation zur Verfügung stehen muss.

Allerdings erfordert dieser Ansatz spezielle Synchronisationsmechanismen, die mit Hilfe von One-Sided RDMA Primitiven implementiert werden. Unsere Untersuchungen haben ergeben, dass bestehende One-Sided RDMA-Synchronisationsprimitiven entweder nicht skaliert werden können oder fehlerhaft synchronisieren, was zu Datenbeschädigungen führen kann. Um dieses Problem zu lösen, schlagen wir Richtlinien vor, die zur korrekten und skalierbaren Gestaltung dieser One-Sided RDMA-Synchronisationsprimitiven beitragen. Da die Speicherung aller Daten im Arbeitsspeicher aus Kostengründen nicht mehr praktikabel ist, schlagen wir als abschließenden Schritt ein Design für einen Puffermanager vor, der den kostengünstigeren NVMe-Flash-Speicher nutzt. Unser Puffermanager nutzt RDMA mit niedriger Latenz und bietet eine transparente Speicherabstraktion, sodass auf den aggregierten DRAM- und NVMe-Speicher über mehrere Server hinweg zugreifen werden kann. Im Gegensatz zu vorherigen verteilten In-Memory RDMA-Designs kann unser Puffermanager Datenmengen bewältigen, die den verfügbaren Arbeitspeicher übersteigen. Im Zentrum unseres Ansatzes befindet sich ein verteiltes Caching-Protokoll, das Daten dynamisch zwischenspeichert. Unser System kann dabei mit RDMA-fähigen verteilten Datenbanken, die ausschließlich im Arbeitsspeicher arbeiten, konkurrieren oder diese sogar übertreffen, während es gleichzeitig größere Datensätze performant und kosteneffizient verwaltet.

Publications

The following peer-reviewed publications are part of this cumulative dissertation. Their content is printed in Part II, Chapters 8 to 13.

[1] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. "Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks." In: *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.

[2] Tobias Ziegler, Dwarakanandan Bindiganavile Mohan, Viktor Leis, and Carsten Binnig. "EFA: A Viable Alternative to RDMA over InfiniBand for DBMSs?" In: *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022*. Ed. by Spyros Blanas and Norman May. ACM, 2022, 10:1–10:5. DOI: 10.1145/3533737.3538506. URL: https://doi.org/10.1145/3533737.3538506.

[3] 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 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.

[4] 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.

[5] Tobias Ziegler, Philip A. Bernstein, Viktor Leis, and Carsten Binnig. "Is Scalable OLTP in the Cloud a Solved Problem?" In: *13th Annual Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023, Online Proceedings*. www.cidrdb.org, 2023. URL: https://www.cidrdb.org/cidr2023/papers/p50-ziegler.pdf.

[6] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. "Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA." In: *SIGMOD '23: International Conference on Management of Data, Seattle, WA, USA, June 18 - 23, 2023*. ACM, 2023.

Further co-authored peer-reviewed publications are:

[1] Arnd Christian König, Yi Shan, Tobias Ziegler, Aarati Kakaraparthy, Willis Lang, Justin Moeller, Ajay Kalhan, and Vivek Narasayya. "Tenant Placement in Over-subscribed Database-as-a-Service Clusters." In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2559–2571. URL: https://www.vldb.org/pvldb/vol15/p2559-k%5C%5C%f6nig.pdf.

[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), pp. 15–22. DOI: 10.1145/3542700.3542705. URL: https://doi.org/10.1145/3542700.3542705.

[3] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. "Benchmarking the Second Generation of Intel SGX Hardware." In: *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022*. Ed. by Spyros Blanas and Norman May. ACM, 2022, 5:1–5:8. DOI: 10.1145/3533737.3535098. URL: https://doi.org/10.1145/3533737.3535098.

[4] 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 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.

[5] Matthias Jasny, Lasse Thostrup, Tobias Ziegler, and Carsten Binnig. "P4DB - The Case for In-Network OLTP (Extended Technical Report)." In: *CoRR* (2022). DOI: 10.48550/arXiv.2206.00623. arXiv: 2206.00623. URL: https://doi.org/10.48550/arXiv.2206.00623.

[6] 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. ACM, 2021, pp. 1825–1837. DOI: 10.1145/3448016.3452816. URL: https://doi.org/10.1145/3448016.3452816.

[7] Benjamin Hilprecht, Carsten Binnig, Tiemo Bang, Muhammad El-Hindi, Benjamin Hättasch, Aditya Khanna, Robin Rehrmann, Uwe Röhm, Andreas Schmidt, Lasse Thostrup, and Tobias Ziegler. "DBMS Fitting: Why should we learn what we already know?" 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/p34-hilprecht-cidr20.pdf.

[8] Matthias Jasny, Tobias Ziegler, Tim Kraska, Uwe Röhm, and Carsten Binnig. "DB4ML - An In-Memory Database Kernel with Machine Learning Support." 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. 159–173. DOI: 10.1145/3318464.3380575. URL: https://doi.org/10.1145/3318464.3380575.

[9] Lukas Berg, Tobias Ziegler, Carsten Binnig, and Uwe Röhm. "ProgressiveDB - Progressive Data Analytics as a Middleware." In: *Proc. VLDB Endow.* 12.12 (2019), pp. 1814–1817. DOI: 10.14778/3352063.3352073. URL: http://www.vldb.org/pvldb/vol12/p1814-berg.pdf.

[10] 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%5C%5C_adms19.pdf.

[11] Tobias Ziegler, Carsten Binnig, and Uwe Röhm. "Skew-resilient Query Processing for Fast Networks." In: *NoDMS BTW*. 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.

[12] 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.

[13] 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.

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

First and foremost, I would like to express my deepest gratitude to Prof. Dr. Carsten Binnig for his support and guidance throughout my doctoral journey. His incredible mentorship has consistently pushed me to grow both personally and academically. No matter the topic, whether it be academic or personal, his door has always been open for me. He also got me into research in the first place by inspiring me with his wisdom, technical knowledge, and visions. Without him, I would have never pursued my Ph.D. I am truly fortunate to have had him as my advisor all these years.

The same can be said for Prof. Dr. Viktor Leis, whose mentorship was incredibly valuable throughout my journey. He always provided insightful ideas, participated in brainstorming sessions and answered the many questions I had regarding building databases, some of which I sent in the middle of the night. He gave me additional tools to be a better researcher, engineer, and writer. I couldn't have asked for a better team to guide me than Carsten and Viktor.

I also sincerely thank Prof. Spyros Blanas for his valuable time and effort in reviewing this dissertation.

I am deeply grateful to my colleagues at the Systems Group at TU Darmstadt for making the past years enjoyable and productive. The supportive atmosphere, the exchange of ideas, the feedback on my work, and the engaging discussions have always been fun and (often) productive. I thank every one of you. In particular, thanks to the best office buddies I could have wished for: Muhammad, Matthias, Lasse, and Nils. Special thanks go to Mona. Her attention to detail ensured that I always met the administrative deadlines I tended to push away while focusing on research.

I am fortunate to have collaborated with exceptional individuals on various projects. Working with them has been a privilege, and their expertise and guidance have significantly contributed to my academic growth. In particular, I express my gratitude to Philip A. Bernstein, who is one of the greatest database researchers, for his willingness to share his expertise has been invaluable.

I am deeply grateful to my parents for their unwavering support and encouragement during my life. Lastly, I would like to express my appreciation to Maren. She made me laugh during challenging times and celebrated my achievements with me. Her encouragement has made this journey way easier. I am truly thankful to all the individuals who have played a part, directly or indirectly, in my academic and personal development.

Contents

I Synopsis 1

1 Introduction 3
- 1.1 Traditional Distributed Database Design . . . 3
- 1.2 Fast Networks . . . 4
- 1.3 State of the Art . . . 7
- 1.4 Storage Engine: The Missing Component . . . 8
- 1.5 Contributions . . . 12
- 1.6 Outline . . . 14

2 RDMA Primer 15
- 2.1 Capabilities . . . 15
- 2.2 Primitives . . . 15
  - 2.2.1 One-Sided Verbs . . . 16
  - 2.2.2 Two-Sided Verbs . . . 17

3 Scalable OLTP Architectures 19
- 3.1 Towards Scalable Online Transaction Processing . . . 19
  - 3.1.1 Revisiting the Old Taxonomy . . . 20
  - 3.1.2 A New Taxonomy Based on the Data Access Path . . . 21
  - 3.1.3 Data Access Archetypes . . . 23
  - 3.1.4 Key Findings: A Road Towards Scalable OLTP . . . 28
- 3.2 Fast Networks in the Cloud . . . 28
  - 3.2.1 A Detailed Comparison of EFA and RDMA . . . 29
  - 3.2.2 Benchmarking EFA and RDMA . . . 32
  - 3.2.3 Key Findings: RDMA Is Faster . . . 34
- 3.3 Summary . . . 35

4 RDMA-Enabled Tree-Based Index Structures 37
- 4.1 The Need for Tree-Based Indexes . . . 37

4.2 Mapping the Design Space . . . 38
4.3 Design 1: Coarse-Grained/Two-Sided . . . 41
4.3.1 Index Structure . . . 41
4.3.2 RDMA-Based Accesses . . . 42
4.4 Design 2: Fine-Grained/One-Sided . . . 43
4.4.1 Index Structure . . . 43
4.4.2 RDMA-Based Accesses . . . 43
4.5 Design 3: Hybrid Scheme . . . 46
4.5.1 Index Structure . . . 46
4.5.2 RDMA-Based Accesses . . . 46
4.6 Experimental Evaluation . . . 47
4.7 Key Findings: Non-Trivial Trade-Offs . . . 49
4.8 Summary . . . 49

5 One-Sided RDMA Synchronization 51
5.1 The Need for Guidelines . . . 51
5.2 Methodology . . . 54
5.3 Pessimistic Synchronization . . . 55
5.3.1 Basic Pessimistic Latch Implementation . . . 55
5.3.2 Performance of RDMA Atomics . . . 56
5.3.3 Effect on a Disaggregated DBMS . . . 60
5.4 Optimistic Synchronization . . . 61
5.5 Key Findings: One-Sided Synchronization Is Hard . . . 62
5.6 Summary . . . 63

6 Buffer-Managed Storage Engine 65
6.1 ScaleStore: Design Overview . . . 66
6.1.1 Design Considerations From Previous Work . . . 66
6.1.2 Main Building Blocks for ScaleStore . . . 67
6.1.3 A Motivating Example . . . 69
6.1.4 Main Components . . . 70
6.2 Low-Latency RDMA Messaging . . . 70
6.2.1 Efficient Message Handling in RDMA . . . 70
6.2.2 Evaluation . . . 72
6.3 Invalidation-Based Page Coherence Protocol . . . 74
6.3.1 Protocol Overview . . . 74

6.3.2 Local Hot Path . . . 75
6.3.3 Remote Invocation . . . 76
6.4 High Performance Page Eviction . . . 78
6.4.1 Epoch-Based LRU Approximation . . . 78
6.4.2 Page Provider . . . 79
6.5 Synchronization Primitives & Indexes . . . 80
6.5.1 Example: B-Tree Lookup . . . 80
6.6 Experimental Evaluation . . . 82
6.6.1 Scale-Out . . . 82
6.6.2 Data Scalability . . . 83
6.6.3 Distributed In-Memory DBMS Comparison . . . 84
6.7 Summary . . . 86

7 Concluding Remarks and Future Work 89
7.1 Reflection . . . 89
7.2 Future Research Directions . . . 91
7.2.1 Reconsidering Eviction & Admission . . . 91
7.2.2 Concurrency Control . . . 92
7.2.3 Durability & Recovery . . . 93
7.2.4 Evolution of Hardware . . . 93

II Peer-Reviewed Publications 95

8 Is Scalable OLTP in the Cloud a Solved Problem? 97
8.1 Introduction . . . 98
8.2 OLTP Data Access Archetypes . . . 100
8.2.1 Archetype: Single-Writer . . . 101
8.2.2 Archetype: Partitioned-Writer . . . 102
8.2.3 Archetype: Shared-Writer . . . 104
8.2.4 Categorizing Systems With Archetypes . . . 106
8.2.5 The Importance of Latency . . . 108
8.3 Towards a Scalable Cloud OLTP DBMS . . . 110
8.3.1 A Blueprint for a Shared-Caching DBMS . . . 110
8.3.2 Caching and Eviction Go Hand in Hand . . . 113
8.3.3 Elasticity . . . 113
8.3.4 ACID Guarantees . . . 115

8.3.5 Cloud Infrastructure and Services . . . 116
8.4 Summary . . . 118
8.5 Acknowledgments . . . 118

9 EFA: A Viable Alternative to RDMA Over InfiniBand for DBMSs? 119
9.1 Introduction . . . 120
9.2 Elastic Fabric Adapter . . . 121
9.2.1 SRD: A Reliable and Unordered Protocol . . . 121
9.2.2 Ibverbs: Low Level Interface . . . 122
9.2.3 Libfabric: EFA's Programming Interface . . . 123
9.3 Experimental Evaluation . . . 124
9.3.1 Latency Comparison . . . 124
9.3.2 Synchronous Bandwidth . . . 125
9.3.3 Asynchronous Networking . . . 126
9.3.4 NIC Parallelism . . . 127
9.3.5 EFA Interface Evaluation . . . 128
9.4 Related Work . . . 129
9.5 Lessons Learned and Summary . . . 129
9.6 Acknowledgments . . . 130

10 Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks 131
10.1 Introduction . . . 133
10.2 Overview . . . 135
10.2.1 RDMA Basics . . . 135
10.2.2 Design Space for RDMA-Based Indexes . . . 136
10.2.3 Scalability Analysis . . . 140
10.3 Design 1: Coarse-Grained/Two-Sided . . . 143
10.3.1 Index Structure . . . 143
10.3.2 RDMA-Based Accesses . . . 145
10.4 Design 2: Fine-Grained/One-Sided . . . 146
10.4.1 Index Structure . . . 146
10.4.2 RDMA-Based Accesses . . . 147
10.4.3 Optimization of Index Structure . . . 149
10.5 Design 3: Hybrid Scheme . . . 150
10.5.1 Index Structure . . . 150

10.5.2 RDMA-Based Accesses . . . 151
10.6 Experimental Evaluation . . . 151
10.6.1 Exp.1: Throughput . . . 154
10.6.2 Exp.2: Scalability . . . 157
10.6.3 Exp.3: Workloads With Inserts . . . 158
10.7 Other Architectures . . . 159
10.8 Related Work . . . 160
10.9 Conclusions . . . 162
10.10 Appendix . . . 164
10.10.1 Additional Index Operations . . . 164
10.10.2 Latency of Index Designs . . . 164
10.10.3 Effect of Co-location . . . 165
10.10.4 Opportunities and Challenges of Caching . . . 166

11 Design Guidelines for Correct, Efficient, and Scalable Synchronization Using One-Sided RDMA 175
11.1 Introduction . . . 176
11.2 Background and Methodology . . . 178
11.2.1 Remote Direct Memory Access . . . 178
11.2.2 RDMA Hardware Stack . . . 180
11.2.3 Existing Synchronization Techniques . . . 181
11.2.4 Evaluation Methodology . . . 182
11.3 Pessimistic Synchronization . . . 183
11.3.1 Running Example of a Pessimistic Latch . . . 184
11.3.2 Performance of RDMA Atomics . . . 184
11.3.3 Optimized Pessimistic Latches . . . 188
11.3.4 Ablation Study of Latch Optimizations . . . 191
11.3.5 Effect on a Disaggregated DBMS . . . 192
11.4 Optimistic Synchronization . . . 194
11.4.1 Intuition for Optimistic Synchronization . . . 194
11.4.2 PCIe's Ordering Guarantees . . . 195
11.4.3 Correct Optimistic Synchronization . . . 197
11.4.4 Single-Threaded Performance . . . 199
11.4.5 Scalability of Optimistic Techniques . . . 200
11.4.6 Effect on a Disaggregated DBMS . . . 202
11.5 Discussion of Other Approaches . . . 204

11.6 Lessons Learned and Conclusion . . . 204

12 RDMA Communciation Patterns 209
12.1 Introduction . . . 210
12.2 RDMA Background . . . 212
12.3 Related Work . . . 213
12.4 Methodology . . . 214
12.5 Evaluation: One-to-One . . . 218
12.6 Evaluation: N-to-One . . . 220
12.7 Evaluation: N-to-M . . . 225
12.8 RDMA Optimizations . . . 229
12.9 Discussion and Conclusions . . . 231
12.10 Acknowledgements . . . 231

13 ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA 233
13.1 Introduction . . . 235
13.2 System Overview . . . 237
13.2.1 A Motivating Example . . . 237
13.2.2 Main Components . . . 239
13.3 Distributed Page Coherence . . . 239
13.3.1 Protocol Overview . . . 240
13.3.2 Local Hot Path . . . 240
13.3.3 Remote Invocation . . . 242
13.3.4 Robustness and Fairness . . . 244
13.3.5 Low-Latency RDMA Messaging . . . 248
13.4 High Performance Page Eviction . . . 251
13.4.1 Epoch-Based LRU Approximation . . . 251
13.4.2 Page Provider . . . 252
13.4.3 RDMA and NVMe Optimizations . . . 253
13.5 Programming Abstraction . . . 253
13.5.1 Interface . . . 254
13.5.2 Example: B-Tree Lookup . . . 254
13.6 Evaluation . . . 256
13.6.1 Experimental Setup . . . 256
13.6.2 Scale-Out . . . 257

13.6.3 Data Scalability . . . 259
13.6.4 ScaleStore vs. GAM . . . 259
13.6.5 Workload Change . . . 260
13.6.6 Elasticity . . . 261
13.6.7 System Comparison . . . 261
13.7 Related Work . . . 266
13.8 Conclusion and Future Work . . . 268
13.9 Acknowledgments . . . 269

Acronyms

2PC Two-Phase Commit

DBMS Database Management System

DMA Direct Memory Access

EFA Elastic Fabric Adapter

SRD Scalable Reliable Datagram

IPoIB IP over InfiniBand

NIC Network Interface Card

OLTP Online Transaction Processing

RDMA Remote Direct Memory Access

RNIC RDMA-enabled NIC

SQ Send Queue

CQ Completion Queue

RQ Receive Queue

RoCE RDMA over Converged Ethernet

MVCC Multiversion Concurrency Control

TPU Tensor Processing Units

GPU Graphics Processing Units

SSD Solid State Drives

FPGA Field Programmable Gate Array

Part I

Synopsis

1 Introduction

Online Transaction Processing (OLTP) forms the foundation of real-time data processing across various applications, such as banking transactions, credit card payments, or e-commerce. These applications typically issue short-duration, latency-sensitive transactions that demand immediate processing [72]. Alibaba recently reported that their mission-critical workloads reach peak transaction rates of up to 70 million transactions per second [87].

These high transaction volumes surpass the capacity of a single machine and require *distributed OLTP Database Management System (DBMS)*, which are deployed across multiple powerful machines[1]. The basic architecture of these systems, originally designed to minimize network communication, has seen little alteration since the 1980s. At that time, high network latency was the primary concern for efficient OLTP.

However, with the emergence of modern high-speed network protocols, the landscape of distributed DBMSs is undergoing significant transformations. These modern networks challenge the conventional belief that network communication is the bottleneck, questioning if the design decisions based on such a notion are still valid.

The subsequent sections explore this observation in more detail. We first examine how distributed DBMSs were historically designed and why modern high-speed networks challenge the existing design paradigm. This examination sets the stage to explore the unresolved challenges this thesis intends to address.

1.1 Traditional Distributed Database Design

The *shared-nothing architecture* [202] has been the foundation of distributed databases since its proposal in the 1980s, underpinning numerous commercial systems [45, 90, 115].

As shown in Figure 1.1, its core idea is to *partition (or shard)* the database across multiple nodes with node-private storage and compute resources. In this design, every node hosts an autonomous database where only local data can be accessed. When a

[1] In this dissertation, we focus on databases that run in a single datacenter, i.e., cluster.

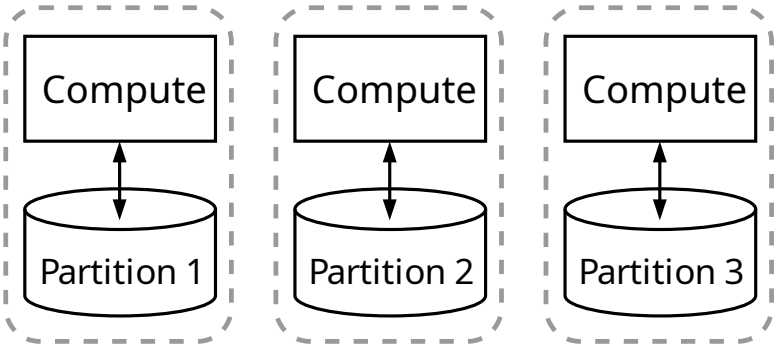


Figure 1.1: Shared-nothing architecture.

transaction is confined to one partition, i.e., only requiring data from a single node, it can be processed locally without network communication. However, OLTP workloads can be complex and hard to partition [156], resulting in transactions that access data on multiple partitions. As resource sharing is avoided and every node operates independently, distributed databases require intricate techniques such as Two-Phase Commit (2PC) [70] to coordinate these cross-partition transactions. This limits their scalability for more complex workloads containing cross-partition transactions [46, 239] (due to the need for inter-node communication and coordination). Furthermore, this design struggles to handle workload shifts, balance load across all machines, and is susceptible to hot spots, e.g., in cases of read skew [170, 178, 202]. Moreover, tightly coupling storage to the compute resources makes it impossible to scale them independently, a need increasingly driven by modern cloud databases [6, 217, 218]. Nonetheless, in 1982, the shared-nothing architecture offered a reasonable trade-off since the network latency for sending a 2 kB packet was about 13000× more expensive than a simple memory reference [169]. Consequently, reducing network access through data partitioning and only accessing local data was a sensible design choice.

1.2 Fast Networks

In recent years, advancements in networking technology have invalidated the long-standing assumption that networks are fundamentally slow. Notably, the increase in bandwidth has been so dramatic that it now closely rivals the bandwidth of main memory [20]. Modern servers, such as the NVIDIA DGX A100, come equipped with as many as eight network cards yielding an aggregated bandwidth of up to 200 gigabytes per second. This

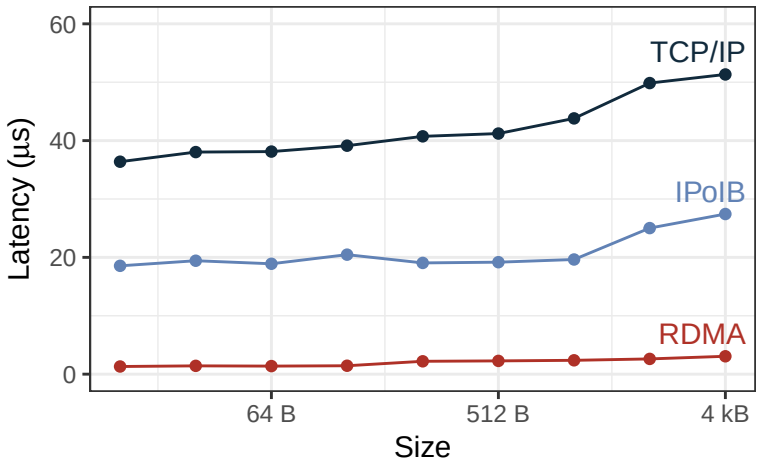


Figure 1.2: Latency comparison TCP/IP over Ethernet, IPoIB, and RDMA. Measured on ConnectX5 with Mellanox benchmark tools `sockperf` for TCP/IP and IPoIB and `ib_write_lat` for RDMA)

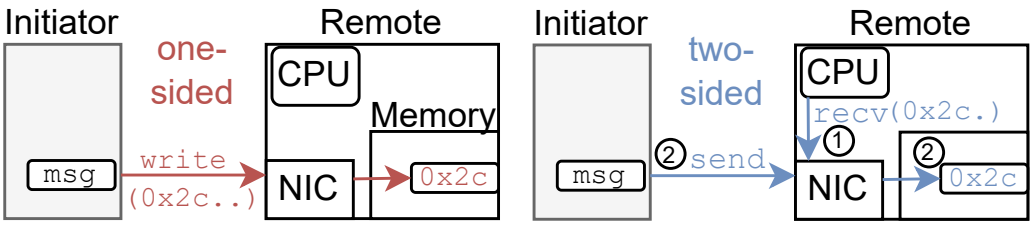
network bandwidth is undeniably impressive. Unfortunately, the bottleneck for most distributed OLTP databases is not the bandwidth but the network latency.

Therefore, many database vendors [29, 89, 128, 175] are increasingly adopting specialized interconnects such as InfiniBand, which offer low latency, often in the single-digit microsecond range [20]. These interconnects employ a communication stack known as *Remote Direct Memory Access (RDMA)*. Utilizing this stack requires software adaptations, especially in the context of existing legacy systems.

To enable the transition without modifying existing applications, IP over InfiniBand (IPoIB) implements a traditional TCP/IP stack as an overlay on InfiniBand. Although IPoIB allows for straightforward migration, it does not fully leverage the low latency. Figure 1.2 compares the latency of traditional TCP/IP over Ethernet, IPoIB, and RDMA for data transfers from 16 B to 4 kB the typical message sizes in data center applications [10, 174].

By utilizing optimization techniques such as kernel-bypass[2], which avoids the overhead of context switches and bypasses the operating system's network stack, RDMA primitives can outperform IPoIB by one order of magnitude. Compared to traditional socket-based networking (TCP/IP), this improvement is even greater. Consequently, a redesign of the communication stack is necessary to fully exploit the advantages of specialized interconnects like InfiniBand.

[2]See Chapter 2 for a detailed discussion on RDMA and its low-level details.



(a) one-sided RDMA (b) two-sided RDMA

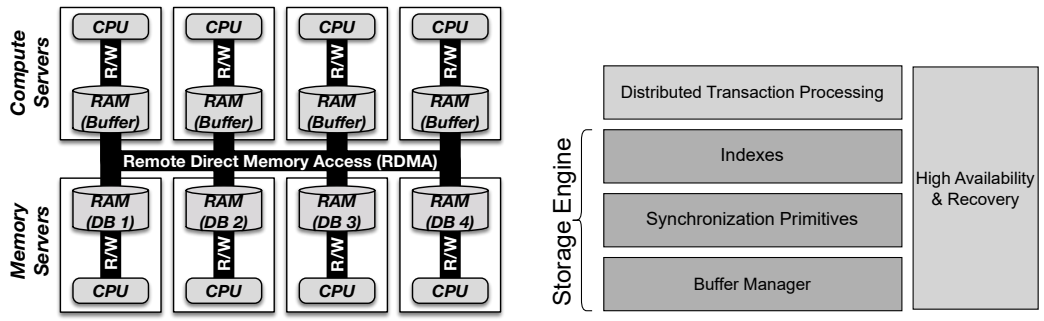
Figure 1.3: RDMA primitives: (a) Depicts a one-sided `write` that directly writes into the remote memory. (b) Shows a two-sided `send` and `receive` operation where the receive operation steers the incoming send to the memory location.

This redesign should incorporate RDMA primitives, which can be categorized into one-sided and two-sided verbs:

One-sided verbs. One-sided verbs (`read`, `write`, and `atomic`) provide remote memory access semantics. That means the sender (initiator) can directly read or manipulate remote memory as depicted in Figure 1.3a. When using one-sided verbs, the CPU of the remote node is not actively involved in the data transfer. Instead, the remote Network Interface Card (NIC) performs the entire memory access. For this to work, the initiator must provide the remote memory address to the remote NIC, as depicted in Figure 1.3a. This example shows a message transfer via one-sided `write` to the remote memory.

Two-sided verbs. Two-sided verbs, specifically `send` and `receive`, operate on channel semantics and involve the receiver's CPU in the communication process (see Figure 1.3b). The `receive` operation, which must be posted before the `send` operation, specifies the memory location for incoming data. This control mechanism allows the remote CPU to direct the incoming `send` to the appropriate memory location, thereby exercising granular control over data storage. Notably, the remote CPU's involvement is asynchronous and limited to providing the `receive` request, with no direct role in the NIC's memory access operation once the message arrives.

To summarize, RDMA verbs are fundamentally different compared to the traditional blocking socket-based POSIX API and allow for novel database designs. On the other hand, they provide unique challenges and adopting new primitives like RDMA into existing systems often fails to fully leverage specialized interconnects, given the different communication paradigms. In addition, it is not clear which RDMA primitive is best suited for databases. Hence, academia and industry have explored the numerous opportunities these novel communication primitives present as shown in the next section.



(a) Disaggregated Architecture (b) Database Stack

Figure 1.4: (a) Shows a disaggregated architecture as used by NAM-DB where the compute layer can access the storage via one-sided RDMA. (b) Depicts a typical database stack.

1.3 State of the Art

Database systems are highly sophisticated pieces of software composed of multiple components, as illustrated in Figure 1.4b. The conventional shared-nothing architecture typically isolates network communication within distributed transaction processing. Thus, the techniques in the lower layers, free from network communication, could be universally applied in both single-node and distributed databases.

The RDMA primitives, however, enable new designs where the network communication can be pushed down the stack to access remote data directly. For instance, some work proposes efficient techniques to ensure high availability, replication, and recovery [165, 166, 207, 222, 242]. In this line of work, Zamanian et al. argue that the bottleneck shifted from the network to the CPU. Thus, conventional replication schemes that re-execute the transaction logic on the replicas are becoming suboptimal. To address this challenge, the authors propose a high-availability solution that effectively utilizes one-sided RDMA to directly update the replicas' data, avoiding redundant computation.

Most existing research revolves around distributed transaction processing, a complex domain. For example, several papers optimize distributed lock management, an integral part of concurrency control, by using RDMA [33, 51, 158, 236]. In addition, other papers take a more comprehensive approach and propose radically new architectures. Unlike the shared-nothing architecture, which restricts access to local data, NAM-DB [20, 239] uses one-sided RDMA to enable network-wide data access. NAM-DB adopts a *disaggregated architecture* as shown in Figure 1.4a where compute and storage are decoupled. The benefit is that compute and storage can be added and removed independently. In NAM-

DB, all data is stored in the memory servers and can be accessed and manipulated by the compute nodes via one-sided RDMA.

Besides decoupled designs, more tightly coupled alternatives are still being explored. For example, FaRM [55, 56, 196] couples compute and storage tightly like the shared-nothing architecture so that local data can be accessed without network communication. However, for distributed transaction processing FaRM exposes a shared address space across all machines. Meaning every node can read remote data using one-sided RDMA. Another example of a tightly coupled design is FaSST [103] which presupposes static partitioning and optimizes the performance of the traditional shared-nothing design with RDMA. In contrast to NAM-DB and FaRM, the authors argue that one-sided RDMA is less scalable, and thus they exploit two-sided RDMA in FaSST.

The reviewed literature indicates a prevailing trend of using RDMA for efficient data access. In particular, FaRM and NAM-DB pushed the idea that data can be accessed from every node via one-sided RDMA. However, despite these advancements, some areas remain largely unexplored. Firstly, the emphasis has been predominantly on distributed transaction processing, leaving room for more research focusing on the lower components in the database stack, such as indexing, latching, and caching (cf. Figure 1.4b). Secondly, the balance between tightly coupled and decoupled computational and storage designs warrants further exploration. While NAM-DB is based on the disaggregated architecture, FaSST's arguments favoring static partitioning and two-sided RDMA indicate potential areas for further investigation. Below we explore these areas in more detail and highlight challenges that need to be addressed.

1.4 Storage Engine: The Missing Component

As we discussed earlier, the lower layers of the database stack, indexes, synchronization, and caching remained largely unexplored; those three layers combined are called the *storage-engine* (cf. Figure 1.4b) and ensure efficient data access:

Indexes are essential for any OLTP database, enabling efficient lookups by eliminating the need for full table scans and substantially reducing query execution time. Accompanying this, *synchronization primitives*[3] become indispensable for parallel index lookups to fully utilize modern multi-core CPUs, which have been the primary driver of performance improvement since the end of Moore's law. Lastly, the *buffer manager* is

[3] In this context, synchronization primitives refer to the low-level synchronization mechanisms, such as latching, rather than the higher-level concurrency control required in transaction execution. For most people outside the database community, latches are synonymous with locks, e.g., mutexes or spinlocks.

critical in optimizing data access efficiency. Its primary responsibility lies in maintaining a low-latency DRAM cache (the buffer pool), which effectively stores frequently accessed, or hot data, thereby avoiding access latency of slower storage devices. Less frequently accessed or cold data is evicted to the slower storage devices once the buffer pool capacity is reached.

Historically, hardware advancements have driven the continuous optimizations of these layers to improve data access. For example, the buffer manager in disk-based systems keeps frequently accessed data in memory to avoid very high disk latency. Fast-forward a few decades, in-memory systems have abandoned the buffer manager under the assumption that the data set fits into the DRAM, and memory prices continue to decline. With the increased speed of in-memory databases and the primary performance gains stemming from multi-core CPUs, the bottleneck shifted to synchronization and indexes, which were then heavily optimized.

In light of the recent advancements in RDMA-enabled databases, the landscape of data access has undergone a significant transformation. This requires a comprehensive redesign of the storage engine components to exploit the potential of RDMA. This need is underscored by the fact that efficient data access forms the cornerstone of the performance of all higher layers and the entire OLTP system. While state-of-the-art systems have improved the efficiency of distributed transactions, we have identified a shift in the performance bottleneck towards data access, as facilitated by the storage engine. As we will demonstrate in this dissertation, our prototype storage engine can outperform NAM-DB by up to 100×.

However, building such a network-optimized storage engine introduces several unresolved challenges. To address these challenges, we first revisit distributed database architectures before examining each storage engine layer in detail.

Challenge 1: Evaluating Distributed Architectures

The fundamental objective of a storage engine is to ensure efficient data access. However, there is an inherent dependency on the underlying distributed system architecture, i.e., *data access depends on where the data is stored* across multiple nodes and *how it can be accessed*.

A notable observation is that RDMA-enabled systems such as FaRM, NAM-DB, and FaSST implement divergent data access paths and design choices. For instance, FaSST [103] adopts a shared-nothing architecture with local access only, while NAM-DB [239] uses one-sided read and writes, allowing every node to access and manipulate

all data. FaRM positions itself in the middle, utilizing one-sided read operations to access all data directly. Nevertheless, it deviates from the NAM-DB model regarding write operations. Instead of implementing one-sided write operations, FaRM resorts to message-passing techniques, transmitting write requests to a partition's owner that applies the changes locally.

This multifaceted nature of available design decisions raises the question of which architecture to adopt and whether these architectures exhibit fundamental differences, e.g., in system scalability. Consequently, the initial challenge lies in comprehensively analyzing these architectures. In the second step, we can devise optimized storage engine components integrated optimally into the architecture, as discussed in the following.

Challenge 2: Designing RDMA-Optimized Indexes

Disaggregated RDMA-enabled DBMSs like NAM-DB demonstrate considerable flexibility, permitting independent scaling of compute and storage resources. Nevertheless, this decoupling introduces a unique challenge: the need for efficient data retrieval via the network from remote storage. While RDMA primitives provide part of the solution to retrieve data once the location is known, this is often not the case requiring indexes. B-Tree indexes are particularly prevalent in OLTP databases as they facilitate efficient point lookups, inserts, deletes, and range scans - crucial operations for any general-purpose OLTP database.

Yet, much of the current research focuses on alternative index structures, such as hashtables [55, 104, 204, 227, 233, 239], which, unfortunately, do not offer efficient range scans. Some research does look at tree-like indexes, but it is mostly within tightly-coupled architectures [103, 233], avoiding the need for RDMA-enabled index access. Cell [150] has been the only index that considers one-sided RDMA, but does focus on how to balance the CPU ressources in a client server setting instead of an evaluation of how to leverage RDMA for building efficient indexes.

As such, a principled study on building efficient RDMA-enabled tree-like indexes for disaggregated architectures has yet to be conducted. In particular, two critical design questions must be addressed: (1) Which RDMA verb to use for efficiently traversing the index, and (2) how should data be distributed across storage servers to optimize RDMA-based access and avoid hot spots? Addressing these unresolved questions forms the second challenge of this dissertation.

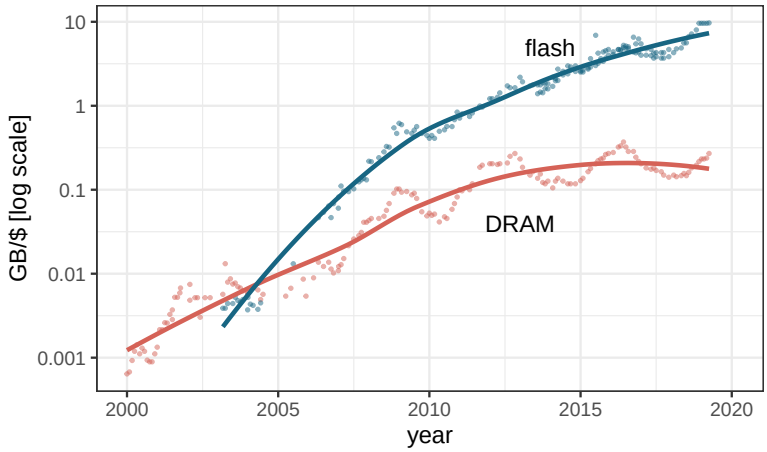


Figure 1.5: Price development for DRAM and flash (higher is cheaper). Source: https://jcmit.net/memoryprice.htm.

Challenge 3: Ensuring Correctness in One-Sided RDMA Synchronization

RDMA-enabled disaggregated DBMSs offer good scalability w.r.t. the number of compute nodes. However, this scalability inherently introduces increased concurrency, requiring careful and efficient synchronization. Given the ability of all compute nodes to access any data item via one-sided RDMA operations, traditional server-side synchronization techniques become obsolete. This is because server-side synchronization involves the remote CPU, which one-sided RDMA operations bypass. Thus, new synchronization techniques built with one-sided RDMA primitives are required.

While various one-sided synchronization techniques have been proposed in related work [55, 150, 223, 233, 256] it is hard to navigate the design space. In particular, since many designs rely on hardware behaviour that worked at that time, but our analysis reveals that some of these methods do not work anymore. Some suffer from scalabillity limitations and other published schemes fail to synchronize correctly on modern NICs, leading to potential data corruption. Our investigation suggests that these problems primarily stem from incorrect assumptions about the memory model of RDMA and the behaviour of modern NICs. This revelation emphasizes the need for comprehensive guidelines to assist engineers in creating scalable and reliable one-sided synchronization schemes.

Challenge 4: Designing a High-Performance Buffer Manager

The previously discussed systems were strictly in-memory; hence, they did not require a buffer manager. The design choice to abandon the buffer manager has been backed by two key factors in the past: (1) The high overhead associated with traditional buffer managers [80] and (2) the steady decline in memory costs. A decade ago, the prices of DRAM were consistently decreasing every year, making in-memory data processing seem economically viable. However, this trend has halted in the past decade and DRAM prices have plateaued, as demonstrated in Figure 1.5.

At the same time, there is a growing need to handle larger data sets, as witnessed by cloud vendors now regularly supporting databases up to 100 TB in size. This shift makes it economically impractical to maintain the entirety of the data (both relations and indexes) in-memory across a cluster of machines. Instead, a more cost-effective approach is required, where the storage engine, specifically the buffer manager component, can offload infrequently accessed data to economical storage options like NVMe flash storage while maintaining high performance.

However, integrating a buffer manager in a distributed system where all compute nodes can access all data presents its own challenges. In particular, how to integrate RDMA with secondary storage since fully one-sided RDMA designs intrinsically operate under the assumption that the necessary objects are in memory. However, this premise becomes invalid when data is evicted from the buffer pool. Moreover, data organization within these systems requires a different approach. Rather than managing in-memory objects of varied sizes, buffer-managed systems often adopt a fixed-size, page-based design where data is systematically organized in blocks to utilize SSDs as best as possible. Until now, no buffer manager has effectively combined RDMA with efficient NVMe storage, and designing such a system remains an open question. This constitutes the fourth challenge that this thesis aims to address.

1.5 Contributions

This thesis addresses the challenges enumerated above with the following contributions that can be summarized as follows:

Contribution 1: Analysis of Distributed Architectures

We conduct a comprehensive analysis of various distributed DBMS architectures and propose a novel taxonomy as a tool to categorize their scalability attributes. We posit that the disaggregated multi-writer architecture, exemplified by NAM-DB, exhibits considerable benefits regarding scalability and elasticity. However, to maximize this architecture's potential, it is essential to incorporate caching as a fundamental component. This thesis will explore this approach in further detail.

Contribution 2: RDMA-Optimized Indexes

We explore RDMA-optimized tree-like index designs tailored explicitly for disaggregated architectures. The two main aspects that we focus on are: (1) How the index itself should be distributed across several memory servers and (2) which RDMA primitives should be used by compute servers to access the distributed index structure in the most efficient manner. Our experimental evaluation shows the trade-offs for different distributed index design alternatives using a variety of workloads.

Contribution 3: One-Sided RDMA Synchronization Guidelines

Driven by the observation that achieving fast and scalable one-sided synchronization presents significant challenges, we conduct a comprehensive analysis of one-sided synchronization techniques We identify and address scalability and correctness issues in one-sided RDMA synchronization primitives. To resolve these issues, we develop a set of comprehensive guidelines for creating scalable and reliable one-sided RDMA synchronization primitives. Our research demonstrates that adherence to our guidelines not only guarantee correctness but also results in substantial performance enhancements.

Contribution 4: High-Performance Cache-Coherent Storage Engine

In the final contribution, we primarily focus on the last component — the buffer manager. Concurrently, we incorporate lessons learned from our preceding body of work to build a full-fledged storage engine (support for indexes, synchronization primitives, and buffer cache) in a holistic way. We present an innovative design for a distributed buffer manager that balances performance and cost-efficiency through NVMe flash storage. Our buffer manager employs a distinct cache-coherence protocol to address the perennial issue of buffer-cache consistency. Additionally, our buffer manager offers user-friendly abstractions with synchronization primitives for efficiently building distributed indexes. We show that

our storage engine is on par with the performance of distributed in-memory-only engines while effectively handling workloads that exceed available memory capacity.

1.6 Outline

The remainder of this dissertation is organized as follows: Chapter 2 introduces RDMA in detail. Chapter 3 revisits the question of how a scalable OLTP design should look like. We examine various designs based on their data access behavior and present a scalability comparison. Further, this chapter benchmarks the low-latency networks that are available in the cloud: RDMA and Elastic Fabric Adapter (EFA). Chapter 4 explores the design space of RDMA-enabled distributed indexes, and Chapter 5 develops guidelines for implementing efficient and correct one-sided RDMA synchronization primitives. Chapter 6 presents our buffer manager design that emerged based on the previous work. This chapter details how we combine cost-efficient NVMe storage with RDMA and evaluates the system against optimized distributed in-memory DBMSs. Finally, Chapter 7 concludes this thesis and outlines future work.

2 RDMA Primer

RDMA's low-level primitives are efficient yet notoriously tricky to use and pose unique design challenges to developers. Thus, this chapter describes RDMA in sufficient depth to understand the subsequent contributions that shed light on those challenges.

2.1 Capabilities

RDMA allows one machine to directly access remote memory over the network without interrupting the CPU on the remote system. Instead, specialized RDMA-enabled NIC (RNIC) perform the memory accesses on the remote side. Conversely, the same technique is used on the sender to *avoid unnecessary copys* from the user-space into the kernel-space. The sender essentially instructs the RNIC what memory needs to be sent, and the NIC's Direct Memory Access (DMA) engine copies the data and places it on the wire. This offloading is also called zero-copy transfer and decreases CPU usage and, thus, latency. Several network architectures offer RDMA – most notably InfiniBand and RDMA over Converged Ethernet (RoCE) [219]. This thesis focuses on the reliable RDMA connection type[1], *reliably connected queue pair (RC QP)*, as specified per the InfiniBand standard [91].

2.2 Primitives

Previously, we casually mentioned that the sender instructs the NIC. This is done with low-level primitives. In particular, the reliable connected RDMA implementations typically provide different primitives (called verbs) that can be categorized into the following two classes: (1) one-sided and (2) two-sided verbs.

[1] Besides reliable connection (RC), there is an unreliable connection (UC) and an unreliable datagram (UD)

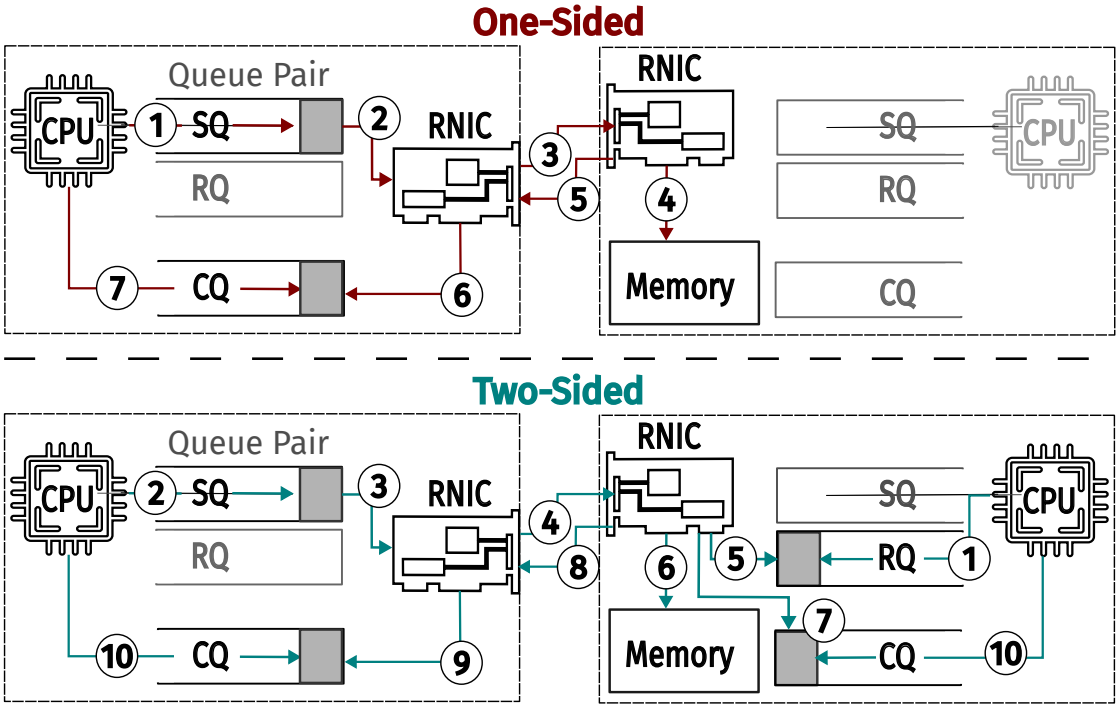


Figure 2.1: A detailed depiction of one-sided (upper) and two-sided RDMA (lower) operations. Depicting the components involved in the transfer from the sender (left) to the remote (right) and highlighting the differences in CPU involvement.

2.2.1 One-Sided Verbs

As mentioned before, one-sided verbs provide remote memory access semantics, where the initiator specifies the remote memory address that should be accessed. The one-sided operations are so named since they can be initiated by one party (the sender or initiator) without CPU involvement from the other party (the receiver or target). There are three operations supported:

- `read` operations allow the sender to read data directly from a remote memory.
- `write` operations allow the sender to write data directly into a remote memory.
- `atomics` include Compare-and-Swap (CAS) and Fetch-and-Add (FAA). These operations atomically modify remote memory and operate on 8 B target memory.

Figure 2.1 (upper) visually outlines the process of a one-sided operation. The sender is depicted on the left side of the figure, with the receiver on the right. Components not

engaged in the process are greyed out to emphasize the difference between one-sided and two-sided operations. In the *reliable connected queue pair* connection type, the *queue pair* is the mechanism to execute the primitives reliably. As shown in the figure, a queue pair consists of a *send queue* and a *receive queue*; in the following, we will see what each queue does.

The one-sided operation begins ① when the sender's CPU submits a *work request* to its local Send Queue (SQ), which is mapped to the memory. This work request specifies the remote target address, the target length in bytes, and the chosen operation (`read, write, or atomic`). The sender's RNIC then retrieves the work request from the SQ ② and forwards it to the receiver ③. The receiver's RNIC processes this incoming one-sided request by executing the specified operation directly on the target memory ④. It is noteworthy that throughout this entire process, the receiver's CPU is not involved. Upon the operation's completion, an acknowledgment is transferred from the receiver's RNIC to the sender ⑤. The sender's RNIC generates a *completion event* that is enqueued into the Completion Queue (CQ) ⑥. The completion event informs the sender's CPU that the operation has been completed. Note that the sender's CPU can asynchronously inspect the CQ for the completion event. In practical terms, this means that there can be numerous outstanding work requests in the SQ before the completion is checked and that the sender CPU accomplishes other work during this whole process.

2.2.2 Two-Sided Verbs

Two-sided verbs provide channel semantics. In contrast to one-sided verbs, there are only two operations that are executed pair-wise:

- `receive` operations are executed on the receiver and specify the remote memory region where the incoming message should be placed.
- `send` operations are executed on the sender and specify the payload, i.e., the message that should be transferred.

Contrasting with one-sided operations, both the sender's and receiver's CPUs participate in the two-sided operations, as depicted in Figure 2.1 (lower). The receiver's CPU ① initiates the two-sided process, where a *receive request* is posted to the Receive Queue (RQ). This step can be executed asynchronously, allowing for posting multiple receive requests in advance (as we will see, those are consumed by incoming requests and thus must be available). Note that after posting the receive request, the receiver CPU is not actively involved in the data transfer. Following this, the sender posts a *send request*

②, which is then processed ③ and relayed to the receiver's RNIC ④. Once an incoming send operation arrives, a receive request is fetched from the RQ ⑤. The receive request specifies the memory address that the RNIC utilizes for copying the message ⑥ – as such, the receiver has granular control over memory, unlike with one-sided where the sender has the control. The receiver's RNIC writes the incoming message to memory and generates a completion event ⑦. This event informs the receiver's CPU of the memory address to which the incoming message will be copied. The final steps are the same as for one-sided operations: an acknowledgment is dispatched to the sender ⑧, after which a completion event is produced ⑨. This event indicates the successful end of the two-sided operation.

This chapter has laid the foundation for understanding RDMA, setting the stage for subsequent discussions. As we delve deeper into the subject in later chapters, we will address more nuanced aspects of RDMA, such as maintaining consistency between local CPU processes and remote operations.

3 Scalable OLTP Architectures

The primary role of a storage engine revolves around ensuring efficient data access. As previously discussed, integrated components of the storage engine optimize data access, such as indexing, caching, and synchronization. Yet, data access is also inherently tied to the underpinnings of the distributed system architecture it resides on. In essence, efficient data access is based on *how well a design accommodates read and write operations* and the *technique employed for accessing the data*. Therefore, before diving into the storage engine, we first investigate those two areas (1) distributed architectures in Section 3.1 and (2) modern network technologies in Section 3.2.

3.1 Towards Scalable Online Transaction Processing

Publication. The work on scalable OLTP in the cloud is published in the peer-reviewed publication "Is Scalable OLTP in the Cloud a Solved Problem?" in the *13th Annual Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023, Online Proceedings* [251], cf. Chapter 8.

Contributions of the author. Tobias Ziegler is the leading author of the publication [251] mentioned above. He is responsible for mapping and analyzing the design space and deriving the taxonomy from the results. The co-authors Philip A. Bernstein, Viktor Leis, and Carsten Binnig have contributed invaluable feedback. All authors agree with the use of the publication for this dissertation.

The DBMS market has shifted significantly from on-premise to the cloud in the last few years. According to a recent market report[1], in 2021 DBMS revenue in the cloud was on par with the on-premise market. Consequently, classical DBMS vendors such as IBM and Oracle have been or are about to be overtaken by hyperscalers such as AWS, Microsoft, and Google, which have heavily invested in their cloud-native DBMSs.

[1] https://blogs.gartner.com/merv-adrian/2022/04/16/dbms-market-transformation-2021-the-big-picture/

Cloud-Native OLTP. Looking at cloud-native OLTP DBMSs, we see an interesting trend: Many commercially-available systems such as Amazon Aurora [217] and Microsoft Socrates [6], use a disaggregated design (shared-storage). In this design, all write transactions go to the primary node, which sends its write-ahead log to the shared storage so that secondary nodes can access it. The storage nodes use the log to reconstruct the data pages in the background. Secondary read-only nodes can read those reconstructed pages on demand and thus be spawned at any time with low overhead. While this design provides important features such as hot failover as well as elasticity and load-balancing for read-dominated workloads, it is limited by the capacity of the primary node – particularly for write-intensive workloads, which are common in OLTP [53].

Decades of OLTP Research. What makes this design choice interesting is decades of OLTP research proposing alternative designs for building scalable distributed systems. Early OLTP systems advocated the shared-nothing architecture for scaling OLTP beyond the limits of a single machine[105, 153]. Another option is the shared-storage architecture with multiple read-write nodes, which has been used successfully since the 1980s [32, 100, 135].

Research Question. Why have modern commercial cloud DBMSs not adopted these scalable designs? We argue that the cloud's arrival has disrupted our understanding of the requirements for distributed DBMS design, forcing us to reconsider established architectures and trade-offs. The need for disaggregation of storage and compute has become more prominent, and designs must incorporate the cloud applications' flexible and on-demand nature. Although the shared-nothing architecture was previously promoted as the preferred choice for building scalable distributed DBMSs, this is not so evident with the advent of cloud DBMSs utilizing shared storage.

Mapping the OLTP Landscape. Therefore, in this chapter, we explore the design space of distributed OLTP DBMSs to help system developers to understand their fundamental differences for the cloud. More specifically, we analyze the data access path of distributed OLTP systems to understand their scalability properties. We observe that the old taxonomy does not fully capture the scalability properties of modern cloud DBMSs.

3.1.1 Revisiting the Old Taxonomy

Decades of OLTP research have accumulated in a plethora of distributed OLTP DBMS designs. In 1985, Stonebraker provided a taxonomy for distributed DBMSs with the famous categorization into shared-nothing and shared-storage architectures found in

many DBMS textbooks today [202]. In a shared-nothing architecture, each node has private storage inaccessible to other nodes. In contrast, all nodes can access a single database copy in a shared-storage architecture. Note that there is no notion of how many nodes can update the data. It is implicitly assumed that all nodes can update and write. Stonebraker's paper not only coined the architectures but also provided a conceptual framework for assessing them based on several properties, including scalability. As such, it allows designers to understand potential performance characteristics based purely on the architecture.

Cloud blurs the lines. While the traditional categorization into shared-nothing and shared-storage architectures remains relevant, changes in the system landscape make it worth revisiting this taxonomy. The once-clear boundaries between these architectures have become blurred. Take, for instance, cloud-based shared-storage systems like Aurora single master [217] and Azure SQL Hyperscale [6]. To circumvent the issue of coordinating multiple writers for buffer cache coherence, these systems permit only one update node — a departure from the classic taxonomy, which presumes shared-storage systems accommodate multiple update nodes. Conversely, the traditional taxonomy posits that shared-nothing systems only allow one node to update a given record. Yet, many modern shared-nothing database systems facilitate updates from multiple nodes on node-private replicas of the same data item. This so-called 'multi-master' approach introduces significantly different scalability characteristics. FaRM [55], referenced briefly earlier, is another example of this categorization problem. FaRM offers a shared-storage abstraction where servers can read data from all other servers per the shared-storage design. However, its update mechanism aligns with the shared-nothing architecture, as writes are sent to a partition owner. Consequently, it remains unclear whether FaRM's scalability characteristics reflect shared-storage or shared-nothing properties. In light of these examples, the traditional taxonomy falls short in distinguishing the scalability properties of modern systems.

3.1.2 A New Taxonomy Based on the Data Access Path

The traditional taxonomy focused on data location and each design's resulting transaction synchronization challenges. For instance, the shared-nothing architecture only allows local data access on node-private storage, requiring a two-phase commit to ensure transaction atomicity in the event of cross-partition transactions. These factors remain undoubtedly crucial to any transaction-processing system.

Table 3.1: Data access archetypes for analyzing the asymptotic scalability of cloud OLTP DBMSs.

		Single-Writer	**Partitioned-Writer**	**Shared-Writer**	
				No Cache	*Coherent Cache*
Qualit. Features	System Complexity	Low	Medium	Medium	High
	Elasticity†	Only Reads	Limited	Yes	Yes
Asymptotic Scalability	Uniform Reads	Yes††	Yes	Yes	Yes
	Uniform Writes	No	Yes	Yes	Yes
	Skewed Reads	Yes††	Yes††	Yes††	Yes
	Skewed Writes	No	No	No	No

(†) Elasticity w.r.t. compute and storage separately (††) Only with the number of secondaries, i.e., replicas.

However, it is also essential how scalable data can be accessed. After all, if a system's data access cannot scale efficiently, the higher layers of the database stack (like transaction processing) are inherently limited. Consequently, we advocate for an updated taxonomy highlighting the data access path. Simply put, we focus on how a system can handle reads and writes and which servers can perform them.

Methodology and scope. To derive this taxonomy, we first perform a conceptual analysis to isolate the scalability properties of various architectures from implementation differences found in real-world systems. We survey existing distributed DBMSs and map the design space based on shared traits for data access. From there, we derive a taxonomy based on three *data access archetypes*. An archetype can be considered the "phenotype of a DBMS" that determines its observable asymptotic scalability independent of its implementation details. The *asymptotic scalability* describes how well a system scales w.r.t. the number of nodes for the following data access operations as the load in the system increases:

- uniform reads
- uniform reads
- skewed reads
- skewed writes

In addition, we discuss qualitative characteristics essential for the cloud. Table 3.1 compares each archetype on the collection of qualitative characteristics and workloads. In our discussions, we omit logging, recovery, and concurrency control [13, 143, 238] that are thoroughly investigated by Harding et al. [79], and Bernstein and Goodman [19]. We also omit deterministic databases [61, 139, 211], as they are not (yet) widely available.

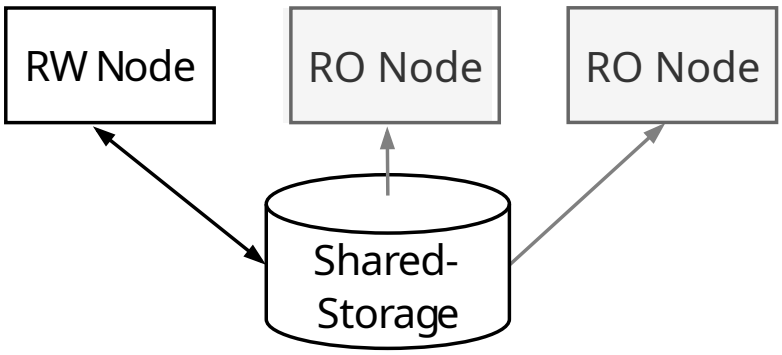


Figure 3.1: Archetype Single-Writer.

3.1.3 Data Access Archetypes

The following briefly describes the data access archetypes and discusses their scalability behavior as summarized in Table 3.1 (cf. Chapter 8 for the entire table).

Single-Writer. In the Single-Writer archetype, a single read-write node (RW-node) processes update transactions, and multiple read-only nodes (RO-nodes) can handle read-only transactions. We depict the basic architecture of systems implementing this archetype in Figure 3.1. Many modern cloud DBMSs, such as AWS Aurora (single-master) [218], Azure SQL Hyperscale [6], PolarDB [248], and AlloyDB [68] use shared-storage. They support multiple nodes that read from a single stored database copy, but only one dedicated RW-node can execute updates. We classify such systems as Single-Writer systems.

While Figure 3.1 shows a Single-Writer with shared-storage, many DBMSs with private storage also fall into this category. For example, a MySQL instance in which the primary node executes reads and writes and replicates writes to read-only replicas is also categorized as Single-Writer, even though the primary and replicas have node-private storage. The strictly separated *data access permission* is the distinguishing feature, not whether the storage is shared or private.

Asymptotic scalability. The asymptotic scalability of this archetype can be summarized as follows: Writes in this archetype are limited by the capacity of the single RW-node. I.e., as soon as the maximum number of requests that this node can handle is reached, the system's performance will stagnate or worsen. By contrast, read throughput can scale well by spawning more RO-nodes.

Qualitative features. This design provides important features for the cloud, such as fast failover and elasticity for read-dominated (analytical) workloads. Similarly, the separation of compute and storage allows cloud vendors to handle growing data sets. At

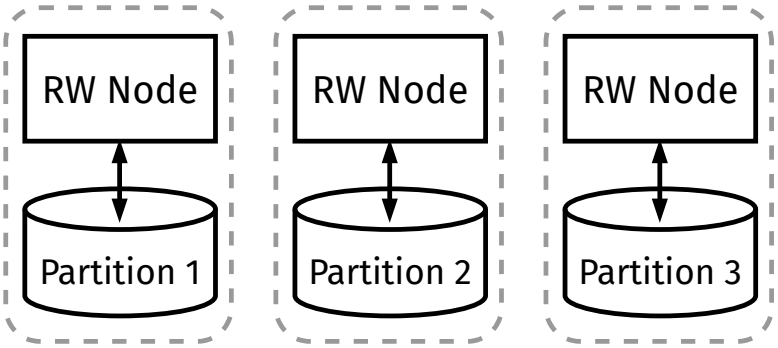


Figure 3.2: Archetype Partitioned-Writer.

the same time, the complexity of these systems is often lower since they are typically derived from single-node systems.

Archetype: Partitioned-Writer. In the Partitioned-Writer archetype, the database is split into partitions, each of which can be updated by only one RW-node. As shown in Figure 3.2, this archetype exploits data partitioning to spread the data across several nodes that are responsible for executing read-write operations on their local partition.

Like the Single-Writer design, the Partitioned-Writer archetype is popular for commercial cloud DBMSs. Examples include CockroachDB [115], AWS DynamoDB [188], Azure CosmosDB [145], and Spanner [45]. Recall that the data access path is the distinguishing feature, not whether the system uses instance local storage or (partitioned) disaggregated storage. The latter is commonly used in the cloud since a single partition can be dispersed to multiple storage nodes. In contrast, instance local storage typically offers lower latency albeit with a fixed storage capacity. Regardless of these implementation details, Partitioned-Writer systems provide the same asymptotic scalability.

Asymptotic scalability. Unlike the Single-Writer design, DBMSs that implement the Partitioned-Writer archetype can handle uniform reads and writes in a scalable manner since they can process each partition's workload independently. However, skewed workloads can be challenging since the same RW-node processes all requests for the hot keys.

One technique to handle read-skew is to couple this archetype with additional RO-nodes (e.g., per partition) similar to Single-Writer systems. However, this needs more resources for database copies and requires propagating updates to RO-nodes to keep them in sync. Since adding RO-nodes is an orthogonal mechanism that can be used for any archetype, in the rest of the paper, we discuss each archetype in its *pure* form. This way, we can identify archetypes that provide scalability under read-skew without additional resources.

Qualitative features. Classical partitioned systems offered limited elasticity since adding new nodes typically called for a full repartitioning of the database. Modern cloud DBMSs have added several optimizations to address this issue, e.g., consistent hashing or fine-grained range partitioning. For example, CockroachDB uses a fine-grained range partitioning of data chunks, which avoids complete repartitioning and improves elasticity. However, fine-grained partitioning can add complexity and overhead to locating data. To address this, CockroachDB stores data in a monolithic sorted map of key-value pairs and uses it to route queries to the responsible RW-node of a partition. Overall, the elasticity of a Partitioned-Writer system depends on the workload characteristics and the system's ability to repartition the data efficiently.

Although this archetype sounds very similar to the shared-nothing architecture, many shared-nothing DBMSs can not be classified as Partitioned-Writer, and thus, their asymptotic scalability is different. E.g., multi-master systems still use partitioned storage, but the data is replicated and can be modified by multiple RW-nodes leading to a different scalability behavior. The next archetype Shared-Writer captures this data access pattern.

Archetype: Shared-Writer. The Shared-Writer archetype allows multiple RW-nodes to modify data items – typically by utilizing shared-storage. Several recent research OLTP systems, such as NAM-DB [239], and a few established systems, such as Oracle RAC, can be classified as Shared-Writer systems. Because data must be kept consistent across writers, these systems must address the buffer-cache coherence problem [155, 177]. There are two ways of doing so: nodes always write and read from the shared storage to get the ground truth (*No Coherent Cache*), or the nodes participate in a cache coherence protocol (*Coherent Cache*).

Shared Writer with no coherent caches. First, we consider Shared-Writer systems without coherent caches. As shown in Figure 3.3, these systems support multiple RW-nodes that write to and read from the storage layer.

Asymptotic scalability. In contrast to the Partitioned-Writer archetype in which only one RW-node writes to a partition, the Shared-Writer design allows every compute node to be an RW-node and to access every data item. Nevertheless, the Shared-Writer archetype has the same asymptotic overall scalability. For example, under access skew, the storage layer might receive requests from multiple RW-nodes and become a bottleneck, limiting the scalability of both writes and reads. In the Partitioned-Writer design, compute and storage resources are usually scaled together. This is inflexible because the need to store more data or do more computation may change at different rates. Using a Shared-Writer design, we can scale compute and storage resources independently depending on the

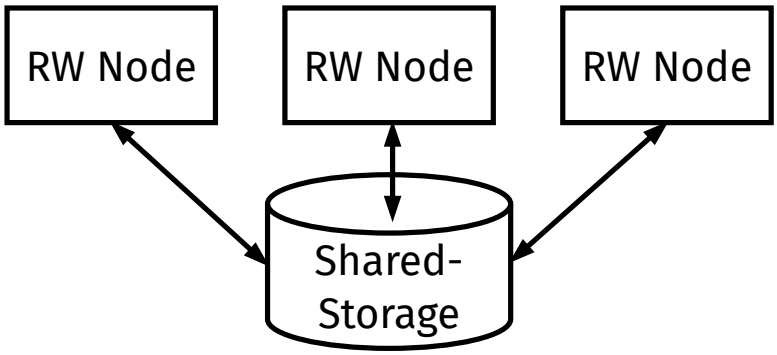


Figure 3.3: Archetype Shared-Writer without coherent cache.

demand. This enables better data scalability and elasticity since any compute node can access any data item, a desired property in the cloud.

However, one major challenge in this design is locating the data items in the storage layer. Like modern Partitioned-Writer DBMSs, modern disaggregated OLTP DBMSs use an index for this task. The index resides remotely in storage and is accessed by compute nodes to find relevant data items. The difficulty in index design increases system complexity, as shown in Table 3.1. We will discuss distributed index designs in Chapter 4.

Shared-Writer with coherent caches. The main downside of the previous sub-archetype (No Coherent Cache) is that every data access involves a network round trip to the storage layer, even for repeated access. Caching can considerably reduce that latency by keeping recently accessed data items in the compute nodes' memory. In this archetype, we specifically consider coherent caches. These caches support shared writers and allow caching read-only data locally while keeping the data coherent (i.e., data does not become stale). Shared-Writer with coherent caches is similar to Single-Writer and Partitioned-Writer with synchronous or eager asynchronous replication since the replicas are essentially caches for reads. The difference is that the caches are updatable.

Figure 3.4 shows that this archetype is, at first glance, similar to the previous Shared-Writer design. However, the coherent caches impact the asymptotic scalability, as discussed in the following.

Asymptotic scalability. This archetype has the best asymptotic scalability of all archetypes. The benefit of coherent caches is that they allow replicating frequently accessed (i.e., hot) data items on demand across multiple nodes. This workload-driven approach differs from the classical (orthogonal) replication used in the previous archetypes.

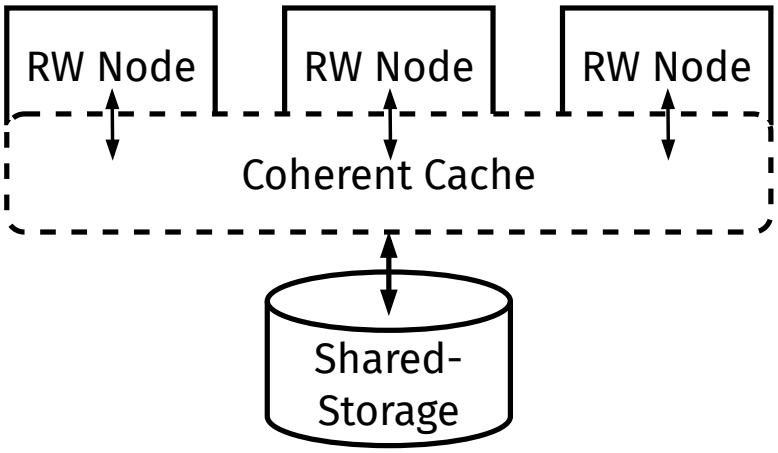


Figure 3.4: Archetype Shared-Writer with coherent cache.

Using the coherent cache, systems of this archetype can handle read skew efficiently, as shown in Table 3.1.

Unfortunately, processing skewed writes in a scalable manner is still challenging (similar to the other designs). However, the cache coherence protocol may lead to the write privilege bouncing between compute nodes since all compute nodes can modify all data items. Solving this requires complex synchronization, a challenging problem. In Chapter 6, we propose a protocol that alleviates the problem by queueing conflicts to ensure fairness and reduce latency.

Qualitative features. While coherence protocols are non-trivial to implement efficiently, they provide many benefits. Compared to operating directly on the shared-storage, shared-caching can reduce the latency of frequently accessed remote data since data items are cached on demand. For instance, when an index node is updated, e.g., a B-Tree leaf, the cache-coherence protocol invalidates other copies of the index node and delivers the newest data version when the leaf is re-accessed. Thus, all subsequent accesses are performed in the local cache, and the traversal is more efficient.

Furthermore, the caching-based design provides good elasticity. Admittedly, this is more complicated than the previous archetype without a cache since the coherence protocol must be designed accordingly. A new compute node can be added any time, incrementally filling its cache as data items are accessed. In addition, these systems can handle easy- and hard-to-partition workloads. They do not require user-defined partitioning but still profit from partitionable workloads. Moreover, they have the flexibility to react to changes in workload distribution.

3.1.4 Key Findings: A Road Towards Scalable OLTP

There are multiple lessons we can draw from our analysis. First, our proposed taxonomy offers a way to classify real-world systems that the old taxonomy failed to classify, such as FaRM. Interestingly, FaRM falls into a partitioned writer system despite its shared-memory abstraction. This is because, while any node can read data from any other node, write operations are exclusively performed by the read-write nodes owning the partition. Aurora multi-master presents another unique case [60, 123, 189], where all DB nodes have writing capabilities. This contrasts with Aurora's single-master system, which only permits one DBMS node to write. While Aurora's single-master conforms to our single-writer archetype and exhibits the characteristics we have discussed, Aurora's multi-master aligns more with a shared-writer system. This demonstrates that deployments influence scalability even within the same system.

Secondly, our analysis suggests that a Shared-Writer system with a coherent cache is well-positioned for the cloud. This system archetype provides the best asymptotic scalability, good data scalability, and elasticity. Moreover, the requirement for user-defined partitioning is relaxed in a Shared-Writer system where partitioning is regarded more as an optimization strategy than a strict requirement. This perspective allows the system to react to workload shifts. At the same time, it mitigates the burden placed on users to determine optimal partitioning, which can often be substantial and impractical in some cases. Therefore, the Shared-Writer system offers greater flexibility and adaptability than other archetypes. Consequently, in the foundational vision paper upon which this chapter is based, we advocated that research should focus on Shared-Writer designs besides the traditional emphasis on shared-nothing DBMSs.

We now shift our focus to a key element in achieving low query latency in the context of the Shared-Writer system, which is the use of low-latency networks. Until now, our discussion has largely disregarded network and implementation specifics. However, it is undeniable that these components significantly contribute to the performance of a system in practical applications. Accordingly, the following section will explore low-latency networks in the cloud.

3.2 Fast Networks in the Cloud

Publication. The work on fast networks in the cloud is published in the peer-reviewed publication "EFA: A Viable Alternative to RDMA over InfiniBand for DBMSs?" in the *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022* [254], cf. Chapter 9.

Contributions of the author. Tobias Ziegler is the leading author of the publication [254] mentioned above. He is responsible for the design and implementation of all the experiments and analyzing the experimental results constituting this principled study. The co-authors Dwarakanandan Bindiganavile Mohan, Viktor Leis, and Carsten Binnig have contributed invaluable feedback. All authors agree with the use of the publication for this dissertation.

Low-latency networks are the catalyst for a wide range of distributed applications such as distributed machine learning [98, 137, 187, 235], high-performance computing [133, 167, 179], and distributed database systems [20, 64, 102, 104, 129, 132, 149, 150, 184, 213, 226, 233, 239, 241, 243, 256]. In fact, distributed system designs previously considered inefficient had to be re-evaluated [64, 213, 239, 256]. For instance, distributed shared memory architectures are becoming increasingly common due to the low network latencies. One commonly employed low-latency network is RDMA over InfiniBand, which provides single-digit microsecond network latencies at an extremely high bandwidth. However, the requirement for specialized hardware to leverage RDMA's benefits creates a significant impediment in the cloud. On-premise hardware is not viable in the cloud, and the customers depend on the cloud providers' offerings.

Regrettably, among the top three cloud providers, only Microsoft Azure currently offers InfiniBand for a restricted set of instance families [147], namely, the H, HB, HC, and some of the N series. This limitation considerably reduces the flexibility of utilizing high-performance, low-latency networks in a cloud-based environment.

As an alternative to RDMA over InfiniBand, in 2018, the largest cloud provider, Amazon Web Services (AWS), introduced instances with *Elastic Fabric Adapter (EFA)*. EFA is a network interface for Amazon EC2 instances specifically designed to achieve low latencies in the cloud. Today, EFA is widely available in AWS, and Amazon presently offers 15 instance families with EFA support [191] to satisfy different application requirements, e.g., memory-optimized (r5dn, r5n) or storage-optimized instances (i3en).

3.2.1 A Detailed Comparison of EFA and RDMA

Surprisingly, even though EFA has become broadly available in AWS, there has not yet been a systematic evaluation of this technology from the data management community. The limited research that addresses EFA was driven by the HPC community and geared explicitly towards MPI in distributed HPC applications [31, 195, 234]. However, MPI is a higher-level library on top of the EFA networking stack and was built with different assumptions that do not necessarily match those of data-intensive systems [5, 131, 213].

To close this gap, we conduct a principled analysis of EFA from a data management perspective and compare it to the (academically) more widely used RDMA technology.

Methodology. We first focus on the qualitative features of EFA and RDMA and discuss both technologies, e.g., in terms of the provided primitives or ordering guarantees. We distill their commonalities and differences, which must be considered for a fair performance comparison. Second, we perform a thorough performance evaluation and compare both technologies regarding latency and bandwidth. Both are important metrics in the context of database workloads. We use the same benchmarking code for both network technologies to compare the experiments. That is, we investigate the fundamental properties of EFA and RDMA with the help of the well-known performance micro-benchmark library *perftest* (available at [76]). Perftest uses *ibverbs* directly and supports EFA (SRD and UD) and reliable connected RDMA over InfiniBand.

3.2.1.1 SRD: A Reliable and Unordered Protocol

Since we already discussed the fundamentals of RDMA previously (cf. Chapter 2), we describe the EFA stack as shown in Figure 3.5 in more detail in the following. Afterward, we compare both technologies based on their qualitative features.

We discuss the EFA stack from the bottom up, starting with the hardware and Amazon's Scalable Reliable Datagram (SRD) protocol. Subsequently, we move on to the software layers *ibverbs* and *libfabric*.

SRD is reliable. The foundation of EFA is SRD, a proprietary network protocol that provides reliable but unordered communication on top of commodity Ethernet switches [195]. Of course, reliability is not unique to SRD as many common protocols, including TCP/IP and reliable connected RDMA, guarantee reliable packet delivery. However, unlike Ethernet protocols such as TCP/IP, SRD is purpose-built for Amazon's data centers. Therefore, Amazon controls the hardware SRD operates on, enabling them to implement SRD's reliability in hardware efficiently (in the AWS Nitro network cards [190]) rather than in software.

SRD is out-of-order. Moreover, unlike other reliable protocols that often guarantee in-order delivery, SRD has out-of-order delivery. This is because in-order delivery may cause head-of-line blocking [195], and thus, SRD reduces tail latencies. Additionally, to avoid hot paths in the network and thus reduce the chance of packet drops, SRD packets are sent across multiple network paths. Thus, out-of-order delivery is a direct consequence of sending packets across multiple paths, and enforcing the order would require large intermediate buffers or dropping out-of-order messages.

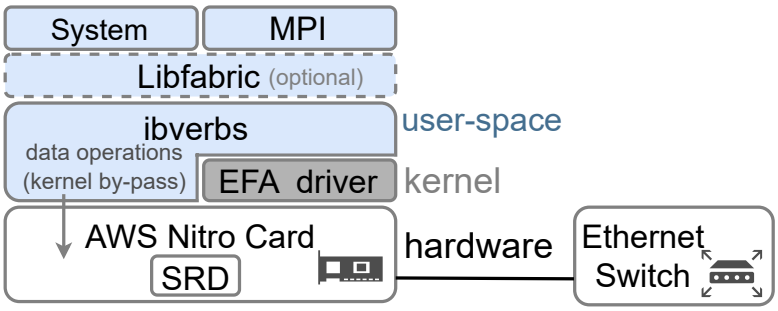


Figure 3.5: EFA stack: Software and hardware.

SRD is exposed to the application via the *ibverbs* library. *Ibverbs* is the same library that allows user-space processes to use RDMA primitives to perform high-throughput, low-latency network operations on InfiniBand. EFA and RDMA share the same low-level library, which is no coincidence because EFA closely resembles the InfiniBand verbs specification [195]. The library itself is split into a control path and a data path.

Control path. The control path is implemented through system calls to the kernel, which further calls the low-level EFA driver depicted gray in Figure 3.5. The control path creates, modifies, queries, and destroys resources needed for connection setup and communication. Because the control path may interact with the kernel, those operations are commonly avoided in the hot path.

Send/Receive queues. Since EFA and RDMA share the same low-level interface, EFA uses queue pairs as well (cf. Chapter 2).

No one-sided. The shared foundation between EFA and RDMA leads to an intersection of primitives as well. However, there are essential distinctions between EFA and reliable RDMA. Reliable connected RDMA provides two types of operations: One-sided *read/write/atomic* primitives and two-sided primitives *send/recv*. EFA does not support one-sided primitives [195] but is limited to two-sided primitives.

Connection scalability. On the other hand, SRD offers better scalability because a well-known limitation of reliable connected (RC) RDMA is lifted. That is, in RC RDMA, one connection endpoint can exclusively communicate only with one other endpoint. This means that many connections must be established to achieve all-to-all communication, impacting performance [103]. In contrast, in EFA one connection endpoint can communicate with all other endpoints without creating an endpoint for every connection. This reduces the number of connections in EFA drastically [195].

Table 3.2: SRD compared to reliable connected RDMA and traditional TCP/IP sockets.

SRD (EFA)	**RC RDMA (IB)**	**Sockets (TCP/IP)**
Ethernet	**InfiniBand**	Ethernet
reliable	reliable	reliable
Messages	Messages	**Stream**
unordered	ordered	ordered
user-space	user-space	**kernel-space**
asynchronous	asynchronous	synchronous
no one-sided	**one-sided**	no one-sided

3.2.1.2 Comparison to Reliable RDMA and Sockets

Table 3.2 summarizes the key differences between reliable EFA, reliable connected RDMA over InfiniBand, and traditional TCP/IP sockets. Due to their similar design objectives (low latency and high bandwidth), we can observe that EFA and RDMA share many common characteristics. Both are designed to use message-based semantics. These message-based semantics help SRD to handle out-of-order packets at the application level if needed. That would be infeasible with a byte streaming protocol as used by TCP/IP because message boundaries are opaque to the application. RDMA's unique feature is the native one-sided support. It is important to note that there are other combinations that we have not discussed or shown in the table. For instance, RDMA over Converged Ethernet (RoCE) uses Ethernet as its fabric instead of InfiniBand. However, those variations are outside of the scope of this thesis.

3.2.2 Benchmarking EFA and RDMA

In order to understand the impact of the qualitative differences of EFA and RDMA, we perform several micro-architectural benchmarks to study the behavior of both network technologies.

Experiment setup. Because EFA is only available on AWS, which does not offer RDMA, we use two different hardware platforms running Linux. We conduct the EFA experiments using two EC2 *c5n.18xlarge* instances with 192 GB main memory and 72 vCPUs connected via 100 Gigabit EFA. Both *c5n.18xlarge* instances are deployed in the recommended *cluster* placement group to achieve the best low-latency network performance [193]. We also replicate our experiments on *c5n.metal* and obtain similar results. The RDMA experiments are conducted on two bare-metal machines with 1 TB main memory and 56 CPUs connected with an InfiniBand network using Mellanox ConnectX-5 MT27800 NICs (InfiniBand EDR 4x, 100 Gbps).

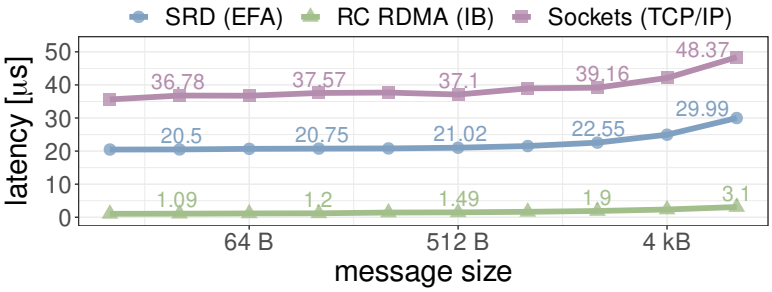


Figure 3.6: Latency with varying message sizes.

We briefly summarize the results from the paper in the following. The detailed evaluation can be found in Chapter 9.

Latency comparison.

Since latency is critical to OLTP, we begin our evaluation by comparing the latency of SRD (EFA) to RC RDMA (IB). To classify the latency improvement of SRD (EFA) over EC2's traditional 100 Gigabit network solution, we use sockets (TCP/IP) as a reference. Figure 3.6 compares the effect of different message sizes on the average latency, i.e., the half-round trip latency. Both EFA and RDMA offer lower latency than sockets. However, when comparing RDMA to SRD, we see that RDMA latency is about 20 *times* lower for messages under 512 bytes. For messages of 8 kB, RDMA's latency is still 10× lower than SRD's. So while the latency results for EFA are much better than for TCP/IP sockets, there is still a significant gap to RDMA (similar latencies for RDMA are achievable on Azure VMs [146]).

Bandwidth. The higher latency becomes the limiting factor in synchronous networking, i.e., when sending one message and waiting until the response. Subsequently, we investigate how asynchronous networking avoids this limitation. Therefore, we vary the number of outstanding messages (i.e., the transmission depth) and show how the bandwidth and message rate are affected in Figure 3.7.

First, let us focus on larger messages. With 4 kB messages and a transmission depth of about 64, RDMA achieves the maximum bandwidth (i.e., around 12GB/s). In contrast, EFA does not fully saturate the bandwidth and instead seems to be message bound at around 2 M messages, i.e., the same message rate as with smaller message sizes. When messages are sufficiently large, i.e., 8 kB, and larger, EFA and RDMA achieve the maximum bandwidth. However, Figure 3.7 shows that only RDMA can reach the full bandwidth when the transmission depth is small. The reason for that lies in RDMA's lower latency since it enables RDMA to process outstanding messages more quickly. For instance, a transmission depth of 8 means we always try to have 8 messages outstanding.

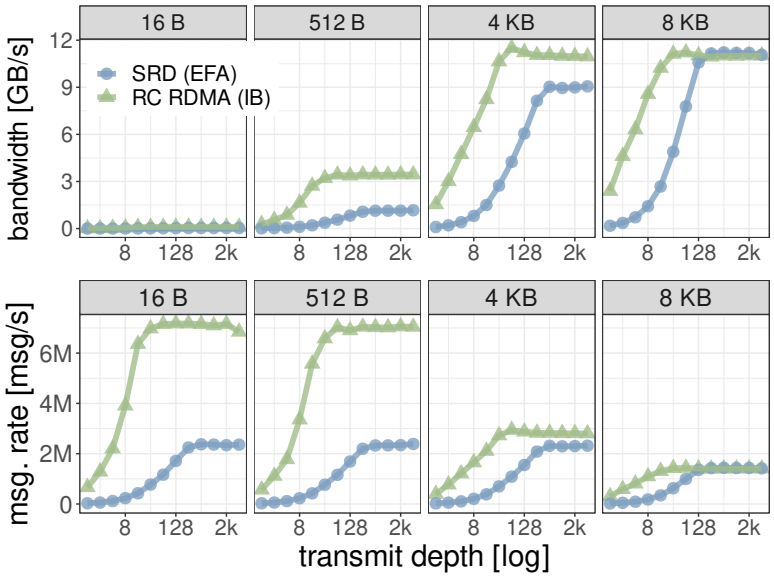


Figure 3.7: Effect of transmission depth on asynchronous bandwidth and message rate (single-threaded).

As RDMA's latency is lower, the completion for these 8 outstanding messages is generated faster, and new messages can be transmitted. Conversely, because EFA's latency is higher, the completion takes longer and thus requires a higher transmission depth to achieve the maximum bandwidth.

We now move on to smaller messages, i.e., 16 and 512 bytes. When comparing the respective message rates for 16 and 512 bytes, we can observe that the message size does not affect the maximum message rate. RDMA achieves around 7 M messages per second and EFA around 2 M messages per second for both message sizes. What is the limiting factor for EFA's message rate? We observed that the Nitro network cards are not as powerful as the RDMA NICs, and thus this workload is limited by the NIC.

3.2.3 Key Findings: RDMA Is Faster

Although both EFA and RDMA are advertised for a similar audience, their performance characteristics differ. The common theme in our evaluation is that RDMA has the edge over EFA in the most relevant metrics. In the following, we summarize our main findings and their implications on system design:

Latency. EFA's latency decreased twofold compared to traditional TCP/IP sockets. Nonetheless, the latencies of EFA are still an order of magnitude higher than those of RDMA.

Bandwidth. As shown in Figure 3.7, EFA's bandwidth strongly depends on the transmission depth and the message size. A large transmission depth (e.g., 256) and message sizes are important to saturate the bandwidth. When the message size is below 8 kB, NIC parallelism should be exploited using multiple connections for a single-threaded application or multiple threads. With even larger messages (e.g., 8 kB), those sophisticated optimizations are unnecessary to achieve the full bandwidth.

Qualitative Features. Besides performance characteristics, other factors, such as no one-sided support for EFA, play an important role. System designs based on one-sided primitives might need to be revisited. In addition, EFA can be exclusively used in AWS ec2 instances (vendor lock-in).

Given that RDMA has the advantage in low-latency communication, the rest of the thesis is limited to RDMA.

3.3 Summary

In this chapter, we laid the foundation for the rest of this thesis by analyzing various scalable designs. We show that a disaggregated architecture supporting multiple write nodes is desirable in the cloud for its scalability behavior and qualitative properties. In particular, disaggregation is a central element in the cloud era as it (1) allows better storage utilization and (2) enables independent scalability of compute and storage.

However, since disaggregation increases query latency due to the network accesses, we looked at low-latency networks in the second part of this chapter. Compared to Amazon's EFA technology, RDMA is particularly attractive, not only because of its very low latency but also because of its efficient primitives. In the following chapter, we will use these powerful primitives to build important building blocks for our fast network-optimized storage engine: indexes and synchronization primitives.

Throughout the remaining chapters of this dissertation, our primary focus will be on the Shared-Writer archetype. Initially, we consciously sidestep caching, directing our attention toward understanding how pure RDMA-based solutions perform and how the RDMA primitives can be exploited to efficiently manipulate remote memory. Therefore, we will focus on indexes in Chapter 4 and how to synchronize remote data accesses in Chapter 5. Subsequently, we investigate how to integrate caching into the storage layer, particularly in our buffer manager design in Chapter 6. This two-step approach allows us to explore and understand the individual elements before examining their interaction in a complete storage engine.

4 RDMA-Enabled Tree-Based Index Structures

Publication. The work on designing distributed RDMA-enabled tree-based index structures is published in the peer-reviewed publication "Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks" in the *2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019* [256], cf. Chapter 10.

Contributions of the author. The contributions to the above publication by Tobias Ziegler are as follows. Tobias Ziegler is the leading author and was thus responsible for the proposed approach for fine-grained and coarse-grained indexes, experimental evaluation, and the manuscript. The co-author Sumukha Tumkur Vani contributed to the first implementations of both designs and provided initial experiments. The remaining co-authors, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska contributed invaluable feedback.

In the preceding chapter, we observed that disaggregated systems are becoming more and more common in modern OLTP cloud offerings [6, 45, 218]. The strength of such disaggregated systems is that they allow to scale compute and storage resources independently, thereby achieving good resource utilization and cost-efficiency. However, this flexibility comes with a challenge: data must be found and accessed efficiently over the network. This chapter addresses this challenge by proposing designs for RDMA-enabled tree-based indexes for disaggregated DBMSs.

4.1 The Need for Tree-Based Indexes

One recently proposed architecture for building distributed in-memory database systems is the Network-Attached-Memory (NAM) architecture [20, 186, 239]. The NAM architecture was designed for high-performance RDMA-enabled networks and logically separates compute and memory servers as shown in Figure 4.1. The memory servers in the NAM architecture provide a shared and distributed memory pool for storing the tables and indexes. These can be accessed from compute servers during transaction processing via

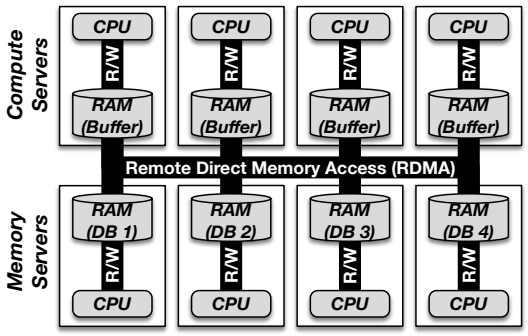


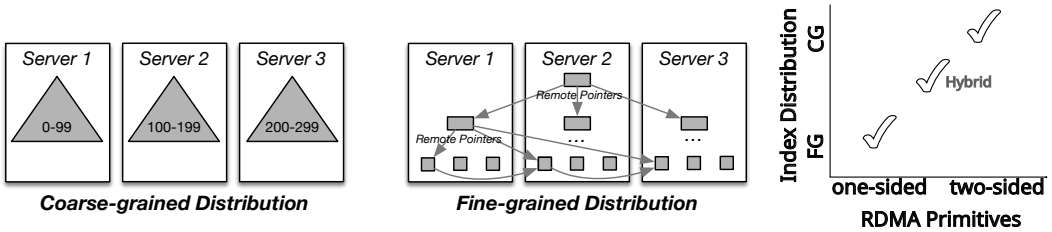
Figure 4.1: The NAM architecture.

one-sided RDMA. One-sided RDMA enables these systems to access data with a very low latency, typically measured in single-digit microseconds. However, it is important to note that this latency can only be achieved when the data's remote location (memory offset) is already known. In many cases, these remote locations are unknown, and auxiliary data structures (indexes) are used to locate the remote memory addresses. Thus, many papers have proposed RDMA-enabled hashtables as a solution to find tuples efficiently [55, 104, 204, 227, 233, 239].

While RDMA-enabled hashtables are an excellent fit for primary key lookups, they are not ideally suited to all database operations. For instance, operations such as scanning and filtering are pretty common in OLTP workloads and require other index structures, so-called *secondary indexes*. For these scenarios, B-Tree indexes are the better choice. Due to their capability to support point lookups, inserts, deletions, and range scans, B-Trees have become ubiquitous in general-purpose OLTP databases [42, 69]. In this contribution, we investigate if designing a scalable RDMA-optimized tree-based index structure for the NAM architecture is possible. In particular, we focus on two design questions (1) which RDMA verb to use for efficiently traversing the index, and (2) how data should be distributed across storage servers to optimize RDMA-based access. These two design questions are explored more thoroughly in the following section.

4.2 Mapping the Design Space

The advent of RDMA primitives allows for distributed index designs beyond traditional partitioning strategies. Traditionally, data is partitioned to assign each server a distinct key range. However, RDMA primitives can directly access the remote memory of any server with one-sided operations. This paradigm shift allows for a more nuanced



(a) Index distribution. (b) Design space.

Figure 4.2: Index distribution schemes and design space.

distribution of data. In the following, we will explore this design space that consists of two dimensions (1) *index distribution* and (2) *RDMA primitives*. Afterward, we derive the most promising design combinations.

Index Distribution

We first focus on the distribution of the index across multiple memory machines. When it comes to distributing the B-Tree index, two contrasting distribution strategies are possible:

Coarse-grained distribution (CG). This distribution scheme is primarily found in shared-nothing architectures. A *partitioning function*, e.g., range-based, is first applied to the indexed key space. This function maps a disjoint set of keys to a corresponding server as illustrated on the left-hand side of Figure 4.2a. Once the keys and their payloads have been allocated to their respective servers, a dedicated tree-based index is constructed on each memory server. In other words, each memory server is responsible for an index that manages a disjoint key range.

Fine-grained distribution (FG). The fine-grained distribution scheme represents the other end of the spectrum. In this case, the index is not partitioned. Instead, a global index is built over all keys, with index nodes (i.e., leaf and inner nodes) distributed individually over the memory of all machines. One possibility to assign the index nodes is in a round-robin fashing (see the right-hand side of Figure 4.2a).

RDMA Primitives

The second dimension we consider is which RDMA primitives can be used to implement indexes efficiently. As explained in Chapter 2, RDMA provides two different classes of operations to access remote memory, called one-sided and two-sided. In the following,

we discuss how one- and two-sided RDMA operations can be used to implement index access methods.

One-sided. Utilizing one-sided operations to traverse the index means that each node of a tree-based index is accessed independently, as RDMA `reads/writes` can only access one remote memory location at a time. Consequently, a key lookup requires one RDMA `read` operation for each index node along the path from the root to the leaf. For range queries that additionally traverse the linked leaf level, one more RDMA `read` operation is necessary for each scanned leaf. Implementing insert operations presents a more significant challenge since synchronization is required. Synchronization is a challenging topic in its own right. Thus we have dedicated Chapter 5 to address this topic in detail and will only briefly touch it in this chapter. In addition, insert operations may require two full index traversals from the root to the leaf and back. During the top-down pass, every index node from the root to the leaf is accessed using RDMA `reads`. During the bottom-up pass, at most, two pages need to be installed using RDMA `write` for each level (in the event of splits), and potentially an additional RDMA `write` might be necessary to install a new root node. This process is similar to a pure in-memory index; however, the latency until the operation is finished can be quite high when performing a split over the network.

Two-sided. Contrary to one-sided operations, two-sided operations do not directly enable us to read from a remote memory location. Instead, they require that the remote CPU is involved in the data access. Given that the remote CPU participates, it can execute the index operations through a Remote Procedure Call (RPC) protocol, similar to what has been described in [103]. An RPC call from one server to another can be implemented utilizing a pair of `send/receive` operations: one pair for transmitting the request from the host to the remote server and another pair for delivering the response, which might contain the result in the case of an index lookup.

Design Space Pruning

Referring to the dimensions previously discussed, we derived a design space as illustrated in Figure 4.2b. As the figure indicates, some combinations are more beneficial than others, thus allowing us to reduce the design space. In the following, we provide our rationale for considering certain design combinations less viable:

Coarse-grained and one-sided. This combination means a local index is built on each memory server (coarse-grained). At the same, this scheme cannot exploit any resulting data locality due to the numerous round trips required by one-sided RDMA operations.

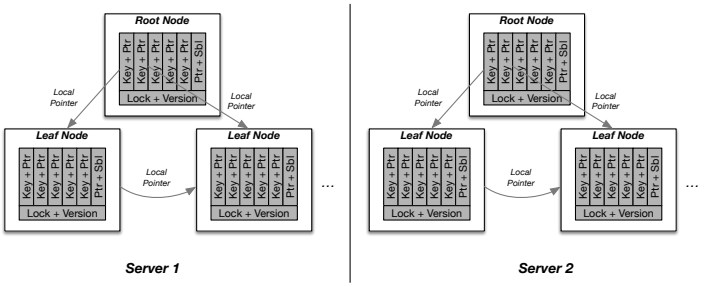


Figure 4.3: Design 1: Coarse-grained index.

Furthermore, the partitioning approach leaves it particularly vulnerable to hot-spot issues.

Fine-grained and two-sided. This design uses a global index spanning all storage servers. Given that the index nodes are distributed randomly or in a round-robin fashion, no meaningful locality can be capitalized. Therefore, this approach follows an access pattern similar to one-sided operations but uses two-sided operations, i.e., the RPC would request single index nodes. Since each index node is fetched from separate machines over the network, both RDMA operations yield comparable performance under a fine-grained scheme. We have decided to examine the fine-grained one-sided design as a representative design.

In the following, we will focus on the three promising designs.

4.3 Design 1: Coarse-Grained/Two-Sided

In this section, we discuss our first tree-based index structure design, which can be distributed over the memory of multiple servers and accessed by clients via RDMA (e.g., compute servers in the NAM architecture). First, we discuss the details of the distributed index structure itself. Afterward, we elaborate on how this index structure can be efficiently accessed using RDMA operations.

4.3.1 Index Structure

The first index structure leverages a coarse-grained distribution scheme as shown in Figure 4.2a. The basic idea of the coarse-grained index distribution scheme is to partition the key space by using a traditional range-based partitioning scheme between different memory servers. Afterward, each memory server builds a tree-based index for its assigned keys.

Figure 4.3 shows the internal index structure in each node and follows the basic concepts of a *B-link* tree [117]. However, unlike the original *B-link* tree, we use real memory pointers instead of page identifiers. More importantly, we introduce an 8-byte field per index node which stores a pair $(version, lock\text{-}bit)$ where the last bit represents a lock-bit. We use this field to implement a concurrency protocol based on optimistic-lock-coupling [120]. Our adaption of optimistic-lock-coupling for RDMA is explained in the next section.

4.3.2 RDMA-Based Accesses

In order to access the index structure, as shown in Figure 4.3, from a remote host, we use an RPC-based protocol that uses two-sided RDMA operations. This design thus follows a more traditional paradigm where operations are shipped to the data – similar to how database operations are executed in a shared-nothing architecture. Other index designs which use one-sided operations or a hybrid access protocol that mixes one-/two-sided operations are explained in Sections 4.4 and 4.5, respectively.

Our RPC implementation for this index design uses RDMA `send/receive` similar to the RPC implementation of [103]. Furthermore, to better scale with the number of clients, we use shared receive queues (SRQs) to handle the RDMA `receive` operations on the memory servers. SRQs allow all incoming clients to be mapped to a fixed number of receive queues instead of using one receive queue per client [203].

In the following, we mainly focus on how the remote procedures are executed on memory servers storing the part of the requested index.

Index Lookups. If an incoming RPC requests an index lookup (i.e., a point- or a range-query), a handler thread on the memory server traverses the index locally. The synchronization protocol is based on optimistic-lock-coupling [120], albeit modified to suit our tree-based index structure. There is a detailed discussion for optimistic lock-coupling in [119].

Index updates. We also support index inserts and deletes. In the following, we first discuss inserts and then delete operations. Similar to [120], the thread which handles the insertion RPC does not acquire any lock in the top-down pass from root to leaf nodes. Instead, it acquires the first lock on the leaf level using a local compare-and-swap (CAS). If a leaf needs to be split, the locks are propagated to the parent nodes. Delete operations are implemented by setting a deleted bit per index entry instead of removing the key. We use an epoch-based garbage collection scheme to remove deleted entries,

which runs on each memory server in a NAM architecture and is responsible for removing and re-balancing the index regularly.

When to choose CG and two-sided. If a workload is sensitive to latency, we argue that a coarse-grained design is the most suitable approach. This is because local memory access is still faster than accessing data over a network, so focusing on locality is beneficial. Coupling a coarse-grained distribution with two-sided operations allows the data traversals to be pushed to the storage side, taking full advantage of the lower DRAM latency. In particular, operations like point lookups that typically retrieve a single value or a small range scan where the range is clustered on the same server benefit from the locality to minimize latency.

4.4 Design 2: Fine-Grained/One-Sided

This section discusses our second design of a tree-based index structure.

4.4.1 Index Structure

The basic idea of the fine-grained index is that the index is distributed on a per-node basis to different memory servers in a round-robin fashion. An example index structure is shown in Figure 4.4.

As in the first design, in addition to the keys and pointers, each index node stores an 8-byte field with $(version, lock\text{-}bit)$ at the beginning of each node. However, different from the first design, pointers are implemented as so-called *remote pointers*. More precisely, a remote pointer is a 8-byte field which stores `(nullbit, node-ID, offset)`. The `nullbit` indicates whether a remote pointer is a NULL-pointer or not and the `node-ID` encodes the address of the remote memory server (using 7 Bit). The remaining 7 Byte encode an `offset` into the remote memory that can be accessed via RDMA. Furthermore, we introduce an optimization called *head nodes* on the leaf level that allows for efficient prefetching for larger scans (see Chapter 10 for a detailed description).

4.4.2 RDMA-Based Accesses

In order to access the index structure shown in Figure 4.4 from a remote host, we use an RDMA-based protocol based on one-sided operations. For the fine-grained distribution scheme, we need to access each index node separately (inner and leaf node); thus, one-sided operations are a good fit for the FG distribution scheme. Similar to the index

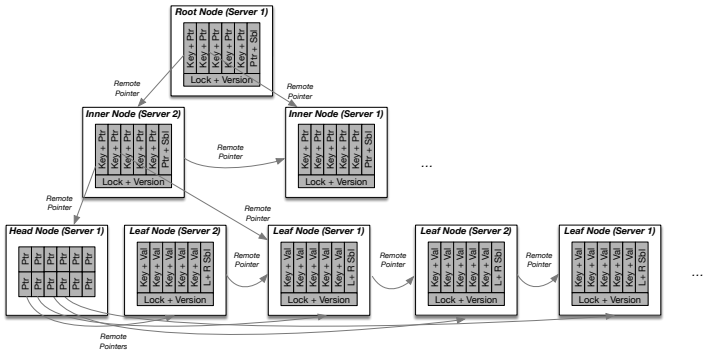


Figure 4.4: Design 2: Fine-grained index.

```
operation remoteLookup(key, remNodePtr) {
  node = remote_read(remNodePtr) // encapsulates RDMA read
  remote_readLockOrRestart(node, remNodePtr) // spinlock, restarts if needed

  if(isLeaf(node))
     value= getLeafValueFromNodeOrSiblings(node)
     return value
  else
    nextNodePtr = node.findChildInNodeOrSiblings(key)
    return remoteLookup(key, nextNodePtr)
}
```

Listing 4.1: Lookup in a fine-grained index.

design in Section 4.3, we use a protocol that is based on optimistic-lock-coupling. Yet, all operations are implemented using one-sided RDMA primitives.

Index lookups. Utilizing one-sided RDMA, a compute server can execute point queries to directly access the index node(s) stored on remote memory servers. The code for `remote_lookup` operation, which handles point-queries, is shown in Listing 4.1. Range queries work similarly but need to traverse the leaf level additionally. The lookup function is called recursively until the leaf level is found. The main difference to the local lookup protocol of Section 4.3 is that the `remote_lookup` first copies the accessed node (inner or leaf) with an RDMA `read` to the memory of the client. Afterward, the lookup operation checks on its local copy if the node is locked and fetches a new copy if the lock is set by implementing a remote spinlock. While implementing a remote one-sided spinlock may appear straightforward, its implementation is very challenging with potential pitfalls, a discussion we reserve for Chapter 5 (dedicated to one-sided synchronization).

Index updates. Similar to our previous design, we implement one-sided insert and delete operations. The high-level approach closely follows the lookup mechanism, with every operation executed using one-sided RDMA. Nevertheless, insert operations are

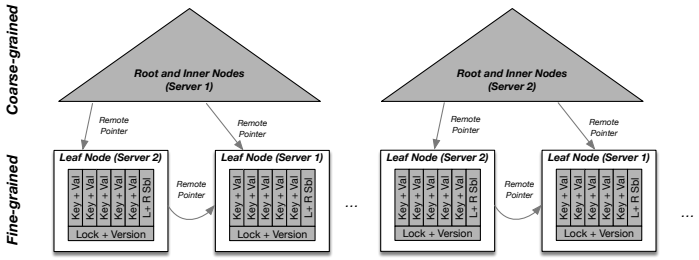


Figure 4.5: Design 3: Hybrid index.

more complex due to the potential for splits when a node is full. In a classical B-Tree, a split requires updating at least three nodes: the full node is split in two, and the parent. The parent update may trigger another split. This procedure requires holding latches to prevent inconsistent reads on all affected nodes. Due to the network latency, this is particularly expensive in our one-sided design.

To mitigate those long latch latency, we can exploit the property of the *B-link* tree [69, 117], which breaks the split procedure into two steps: The first step splits the nodes, and the second inserts a new separator key in the parent node. Since neighbors are linked in the *B-link* tree, a single key range in the parent node and its associated child pointer (the splitted child) may refer to two child nodes. Following this pointer, the search key is initially compared with the splitted child node's high fence key (the new separator) during a root-to-leaf search. The search continues by following the link to the right neighbor if the desired key is higher. This transitional state is only temporary and must be resolved timely by completing the second step. Only when the second step is completed the right neighbor is referenced in the parent node. A more comprehensive discussion on insert and delete operations, accompanied by pseudocode, can be found in the attached paper in Chapter 10.

When to choose FG and one-sided. A fine-grained distribution strategy proves advantageous when dealing with workloads constrained by bandwidth, such as larger-range scans. This is particularly evident in scenarios where the data accessed is not evenly distributed across all memory servers but exhibits skew, where certain ranges are accessed more frequently than others. Consider a case where a large-range scan must retrieve multiple leaf nodes. With a fine-grained design, requests for these nodes are randomly dispersed among multiple servers rather than being partitioned and possibly concentrated on a single one. This effectively spreads the load and results in better bandwidth utilization.

4.5 Design 3: Hybrid Scheme

We have presented the coarse-grained design that is better suited for latency-sensitive point lookups and the fine-grained design that tends to balance the bandwidth better across all storage servers. Unfortunately, database workloads typically do not consist of either point lookups or range scans; they are often mixed. To address this challenge, we derive a hybrid design to find a robust compromise that balances the trade-offs from both extremes.

4.5.1 Index Structure

Combining fine-grained and coarse-grained is possible since they can be applied to different levels of the tree: As shown in Figure 4.5 we use a coarse-grained scheme to partition the upper levels of the index (inner and root node) while we use a fine-grained scheme for the nodes on the leaf level. The intuition is that we combine the best of both designs, i.e., getting low latency by using an RPC-based index traversal and still leveraging the aggregated bandwidth of all memory servers by distributing leaves in a fine-grained manner. This way, leaf nodes are distributed uniformly to all memory servers, even if the index keys are skewed. This index design is a hybrid scheme combining the two schemes discussed in Section 4.3 and Section 4.4.

4.5.2 RDMA-Based Accesses

We also use a hybrid scheme of one-sided and two-sided RDMA operations to access the index.

Index lookups. The basic idea is that we (as discussed before) traverse the upper levels of the index using RPCs implemented using two-sided operations. However, instead of returning the actual data, the RPC only returns the remote pointer to the leaf node. Afterward, in case of a lookup (i.e., point- and range-queries), the compute server fetches the leaf nodes using one-sided RDMA `reads`.

Index updates. In case of an insertion, we again use an RPC that traverses the index and returns a remote pointer to a leaf page where the new key should be inserted. To install the key, the compute server uses the remote pointer and the one-sided protocol to install the new key to the leaf level. In case a new leaf node has to be inserted (due to a split operation), the compute node will perform the first step of the *B-link* tree insert protocol one-sided. For the second step, it issues an additional RPC over two-sided RDMA to the memory server, indicating that a new leaf node has been inserted (using

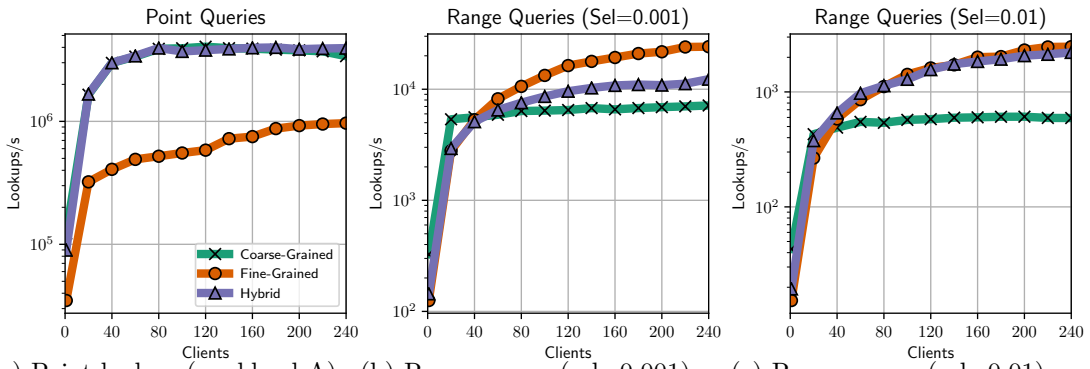
the start key and the new remote pointer as arguments). The memory server will then use the second part of the insertion protocol from Section 4.3 to install the new key into the upper index levels.

4.6 Experimental Evaluation

We evaluate the three proposed designs across various workloads to substantiate the previous discussion. For the synopsis, we focus on read-only point lookups and range scans to highlight the strengths and weaknesses of every design. Inserts, mixed workloads, and other experiments can be found in Chapter 10.

Experiment setup and workloads. We chose the Yahoo! Cloud Serving Benchmark (YCSB) [43] to mimic typical OLTP and OLAP index workloads. The original version of YCSB only supports queries for short ranges but does not explicitly support different selectivities (low/ high) for query ranges; therefore, we implemented a modified version. In this synopsis, we will focus on two workloads: *Workload A* models a read-only workload with 100% point queries. *Workload B* represents a read-only workload with range queries where the selectivity can be configured. In our experiment, we used $sel = 0.001$ (0.1%), $sel = 0.01$ (1%), and $sel = 0.1$ (10%) to model low and high selectivity. Moreover, the original YCSB only supports a skewed access pattern of queries by using a Zipfian distribution for the requested keys. However, for evaluating a tree-based index structure, we also want to evaluate attribute-value skew, which refers to the fact that the partitioning attribute is not unique. For instance, in a secondary index that maps last names to the customer id (primary key), the last names are typically not uniformly distributed, e.g., there may be multiple customers with the last name "Doe" resulting in skew.

For executing all the experiments, we used a cluster with 8 machines featuring a dual-port Mellanox Connect-IB card connected to a single InfiniBand FDR 4× switch. Each machine has two Intel Xeon E5-2660 v2 processors (each with 10 cores) and 256 GB RAM. We used $1 - 6$ compute servers (on $1 - 6$ physical machines), each running 40 compute threads (clients) to access the index. The remaining 2 machines host 4 memory servers (2 on each machine), exploiting the fact that our InfiniBand cards support two ports; i.e., each memory server uses a dedicated port on the networking card. The machines run Ubuntu 14.01 Server Edition (kernel 3.13.0-35-generic) as their operating system and use the Mellanox OFED 3.4.1 driver for the network. All three index designs are implemented using C++ 11 and compiled using GCC 4.8.5.



(a) Point lookup (workload A). (b) Range query (sel=0.001). (c) Range query (sel=0.01).

Figure 4.6: Throughput for workload A (uniform data, size 100M) and workloads B (skewed data, size 100M).

Scalability: Point queries (workload A). This experiment focuses on the scalability of point queries (workload A). We use $100M$ key/value pairs uniformly distributed with no skew and incrementally scale the number of clients to 240. Each client thread executes point queries in a closed loop (i.e., it waits for a lookup to finish before executing the next lookup). Figure 4.6a shows the scalability behavior of the three designs. The hybrid and coarse-grained designs dominate this workload due to the lower latency of every traversal. In this setup, the major limiting factor for both designs is that the compute resources on the memory servers are exhausted when reaching 80 clients. We will show techniques for more efficient message handling in Section 6.2.1. In contrast, the fine-grained design suffers from the higher latency that stems from needing multiple network round-trips per traversal.

Scalability: Range scans (workload B). For workload B, we look at different selectivities ($sel = 0.001$, $sel = 0.01$). Here, we model attribute-value skew by range partitioning 80% of the key/value pairs to the first memory server, 12% to the second, 5% to the third, and 3% to the last memory server. Consequently, 80% of the lookups must be sent to the first server since requests are spread uniformly across the key space. This is mainly relevant for coarse-grained since it is fully range partitioned and, to some extent, for the hybrid design. The hybrid design range partitions the inner nodes using a coarse-grained scheme and shuffles the leaf nodes in a round-robin manner using a fine-grained index distribution. Let us start with discussing the results of the lower selectivity range scan (0.001) as shown in Figure 4.6b. This low selectivity is interesting because it represents a workload between range-scans and point lookups. The skew is the limiting factor for the coarse-grained design in this workload. 80% of the requests end up in one storage server, quickly becoming the bottleneck. In contrast, one-sided is resilient

against this type of skew as it distributes the requests evenly across the servers. Hybrid is between both designs as it inherits the good properties but also some limitations from both schemes. On the one hand, the leaves are dispersed across the storage layer as in the fine-grained design, but on the other hand, the inner nodes are still partitioned. Consequently, 80% of the requests are routed to the first memory server (inner node traversal), which suffers from skew, and the actual data retrieval is dispersed (leaf node scan). We can see in Figure 4.6c that when the range scan gets longer, hybrid benefits from the evenly distributed leaf level for two reasons: 1) The bandwidth of the storage nodes can be better utilized compared to coarse-grained. 2) Since the clients perform the long-range scans (i.e., they are longer busy with scanning), the load on the skewed memory server is decreased. When comparing both selectivities, we can observe that the number of operations per second differs by almost an order of magnitude.

4.7 Key Findings: Non-Trivial Trade-Offs

The hybrid design involves partitioning the inner nodes across storage nodes while randomly distributing the leaf layer. This design choice aims to leverage the strengths of coarse-grained and fine-grained designs, providing a more adaptable and versatile solution. Our experimental evaluation confirms the trade-offs associated with each design. We observe that coarse-grained and fine-grained designs excel under specific workload patterns. On the other hand, the hybrid design demonstrates greater resilience and adaptability, dynamically adjusting its performance based on workload characteristics. It performs well in range scan scenarios, approaching the performance of fine-grained design while also delivering good results for point lookups. Fine-grained and hybrid designs demonstrate better bandwidth and storage resource utilization in cloud environments, where skew is a common challenge. However, it is important to note that incorporating one-sided RDMA introduces additional complexity, as we will explore in the upcoming chapter. Furthermore, the potential latency associated with one-sided approaches may not always be justifiable for OLTP workloads. Therefore, we explore alternative options in Chapter 6 by incorporating caching within our storage engine.

4.8 Summary

This chapter focused on indexes, an essential building block of any storage engine. We proposed three designs for distributed RDMA-enabled tree-based indexes and thoroughly

examined several design alternatives for index distribution and RDMA-based access protocols. The unique trade-offs of each presented design underscore the complexity of the design space. We found that fine-grained utilizes the storage bandwidth better, coarse-grained typically provides better latency, and hybrid is a robust compromise of both designs. Although our primary focus centered around the NAM architecture, which separates compute and memory servers, our insights are extendable to other distributed architectures.

While the designs provide a first insight, building high-performance indexes hinges on the implementation details, one of which is synchronization. In the upcoming chapter, we will dive deeper into one-sided synchronization primitives.

5 One-Sided RDMA Synchronization

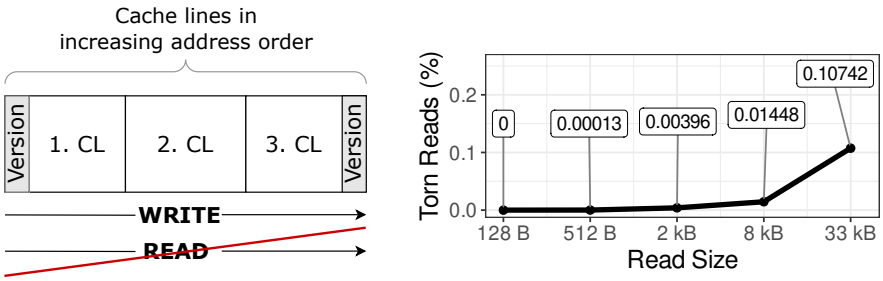
Publication. The work on design guidelines for correct, efficient, and scalable synchronization using one-sided RDMA is published in the peer-reviewed publication "Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA" in the *SIGMOD '23: International Conference on Management of Data, Seattle, WA, USA, June 18 - 23, 2023*

Contributions of the author. The contributions to the above publication by Tobias Ziegler are as follows. Tobias Ziegler is the leading author and was thus responsible for the proposed synchronization primitives and their optimizations, evaluation, and the manuscript. The remaining co-authors, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig contributed invaluable feedback.

Following the introduction of a one-sided index design in the previous chapter, this section explores the intricacies of one-sided RDMA synchronization, which was reserved for a more in-depth discussion here. Since the publication of our index paper, the use of one-sided verbs for effective remote data access in disaggregated DBMSs has attracted considerable interest [55, 57, 111, 134, 142, 210, 223, 227, 239, 241, 246, 256]. However, because one-sided operations bypass the remote CPU, traditional storage server-side synchronization techniques where the remote CPU is in charge do not work. Instead, various one-sided synchronization techniques have been proposed [55, 150, 233, 256]. Those techniques can be categorized into *pessimistic* and *optimistic* schemes. While pessimistic schemes *prevent* concurrent modifications, optimistic schemes *detect* (and handle) concurrent modifications. These approaches fundamentally differ in their scalability and performance characteristics.

5.1 The Need for Guidelines

Performance is key. Remote data structures may need to serve thousands of clients connecting from several compute servers. With such a high degree of concurrency, the performance depends on how well the implemented one-sided synchronization scheme performs. While individual papers have proposed various one-sided synchronization schemes [55, 149, 256], it is surprising that there has not yet been a systematic study of



(a) Faulty Optimistic Scheme (b) Percentage of Torn Reads

Figure 5.1: Incorrect optimistic synchronization

these schemes under comparable workloads and conditions. This contribution provides the first in-depth performance analysis. We show that small design choices when implementing a scheme can severely impact its performance and lead to performance bottlenecks. For example, contrary to expectations, data alignment hinders the scalability of pessimistic one-sided latches, even in uncontended workloads. In fact, if not carefully implemented, the performance for an uncontended workload can be as dismal as that for a highly contended one. To this end, we propose design principles to mitigate those pitfalls and present several optimizations that improve the performance of a well-known disaggregated RDMA-optimized DBMS [239] by 2×.

Correctness is hard. Achieving high performance in synchronization is unquestionably valuable, but ensuring correctness is mandatory. We have discovered that early techniques proposed in the literature fail to accurately synchronize concurrent operations on modern hardware, potentially resulting in hard-to-detect data inconsistencies. For example, consider an optimistic synchronization scheme as implemented in [150, 223] (shown in Figure 5.1a), which presumes that RDMA operations occur in ascending address order — a common assumption in many papers. This scheme implements an update by writing the head version first, then the data modification, and eventually updating the tail version. Under the assumption of operations being performed in increasing address order, a concurrent reader can detect concurrent modifications by comparing the head and tail versions.

Incorrect assumptions. Unfortunately, in contrast to the general assumption in many papers [150, 223], RDMA reads are not guaranteed to be performed in increasing address order. In fact, the RDMA specification does not state the ordering within a single RDMA read [9, 91]. As a result, in the previously mentioned synchronization scheme, an RDMA read may first read both versions (i.e., the first and third cache line) and then retrieves the data from the second cache line. At the same time, a writer concurrently modifies the data in address order. The concurrent data update may not be detected because the

versions were read first before the concurrent writer started. However, the reader and the writer overlapped at the second cache line leading to inconsistent data.

We validate the existence of this behavior with a simple experiment consisting of one storage node and two compute nodes. A single-threaded remote writer on one compute node repeatedly fills a block of its local memory, e.g., 512 bytes, with the same 8-byte version number and then writes it to a remote buffer (50 MB) on the storage node with a single RDMA write. The version number is incremented on each iteration, and the new block is written to the next slot in the buffer. Concurrently a reader on the other compute node reads a block in the remote buffer with a single RDMA read and then checks whether the header and footer version numbers are identical. If the header and footer version numbers match, then the intermediate values are examined to determine if an inconsistency exists. "Torn" reads – having an identical header and footer version but inconsistent intermediate values – are *undetectable* by a validation scheme that checks the block's leading and trailing version numbers.

Figure 5.1b shows the percentage of how many such torn reads are undetectable due to changes in the read order. Note that no torn read appears with 128 bytes as only the header and footer cache lines are read, i.e., if they are inconsistent, this can be detected. However, inconsistencies happen for more than 128 bytes, and while not frequently, often enough to corrupt the data. Surprisingly, this problem is not widely known, and techniques that assume ordering are still very popular [223]. We believe the main reason for this assumption is that a single RDMA request requires many protocols — not only RDMA but also PCIe, and cache coherence — to work in concert. Thus, it is challenging to understand which guarantees are provided by the respective specification.

Contribution. The primary goal of this work is to distill general principles for correct one-sided synchronization techniques. To our knowledge, this is the first principled analysis comparing the performance, scalability, and correctness of one-sided synchronization techniques. Our work demonstrates that understanding the specification and low-level hardware details is crucial for correct and efficient synchronization. Our underlying goal is to guide researchers and developers on how and when to use the different synchronization techniques. Finally, we open-sourced benchmarks[1] that help to transfer our findings to different hardware setups and future developments.

[1] https://github.com/DataManagementLab/RDMA_synchronization

5.2 Methodology

Before delving deeper into our analysis, we present the evaluation setup and framework used for our microbenchmarks scattered throughout the chapter.

Framework. We implement all techniques within a single code base to ensure a fair comparison of the different techniques. We use a perfectly sized remote hash table to highlight each scheme's overhead and prevent hash collisions. Additionally, we employ a remote B-Tree to observe the behavior of these schemes in a hierarchical structure where contention is inevitable due to concurrent root node traversals by multiple workers. While we utilize multi-threading in our experiments, we assign each thread to execute one operation until completion (avoiding batching or asynchronous execution) to facilitate the comparison of the different techniques.

Setup. We conduct all reported experiments on an 8-node cluster running Ubuntu 18.04.1 LTS, with a Linux 4.15.0 kernel. All nodes are connected to a SB7890 InfiniBand switch using one Mellanox ConnectX-5 MT27800 RNIC (InfiniBand EDR 4x, 100 Gbps) per node. Each node has two Intel Xeon Gold 5120 CPUs (14 cores) and 512 GB main-memory split between sockets.

Since the ConnectX-5 card is connected to one NUMA socket, we inevitably have NUMA effects when using more than 14 cores (i.e., 28 threads). To alleviate NUMA effects, we allocate the memory on the socket of the NIC, and assign threads round-robin (interleaved) to both sockets. Besides ConnectX-5, we also validate our results for different generations of RNICs, namely, ConnectX-3 and ConnectX-6. Our analysis focuses on Mellanox cards due to their wide-spread usage: From 30 recently analyzed papers, 28 used Mellanox cards. Further, besides on-premises, of the top three cloud providers, only Microsoft offers RDMA by using Mellanox cards in their instances. [107]. That being said, we believe that some findings are independent of the network card. The correctness discussion depends on the protocol specifications underpinning the RDMA communication. In particular, our testbed leverages the widely-used InfiniBand specification [91], which shares commonalities with alternative RDMA protocols and makes our results applicable to a broader range of deployments. The performance considerations shed light on possible pitfalls worth investigating when building an RDMA application. We open-source our benchmarks to help developers uncover performance and correctness bottlenecks in other RDMA system configurations.

For the sake of brevity, in the following analysis, we focus more of pessimistic approaches and only present a high-level view on optimistic approaches. A more detailed analysis of both approaches can be found in Chapter 11.

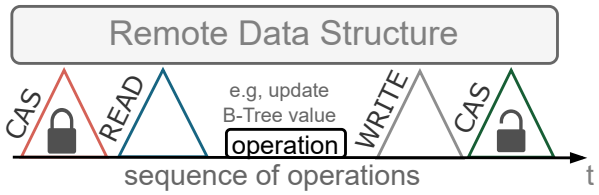


Figure 5.2: Example of an exclusive latch acquisition.

5.3 Pessimistic Synchronization

In order to prevent concurrent modifications of remote data structures such as a B-tree or a hash table, one-sided pessimistic synchronization techniques implement latches using one-sided RDMA operations. This section presents a basic implementation of such a latch that is a foundation for discussing possible optimizations. Since RDMA atomics are the fundamental building block of these one-sided latches, we will then drill down into the performance and scalability characteristics of these RDMA primitives. Afterward, we outline and evaluate possible optimizations for one-sided latches in NAM-DB, a modern disaggregated DBMS.

5.3.1 Basic Pessimistic Latch Implementation

Pessimistic schemes can be subdivided into two types of latches: While some latches such as RCC-NOWAIT [221] support only one latch mode (latched and unlatched), others distinguish between shared and exclusive modes, i.e., reader/writer latches. Both latch types can be implemented with atomic RDMA operations. We will use a reader/writer latch for the running example, but all optimizations generalize to both latch types.

We implement a typical reader/writer latch using an 8-byte value [141, 233]. A worker uses an RDMA CAS operation to set the latch bit (usually the trailing bit) for exclusive access. Readers increment the reader count, encoded in the remaining bits, with an FAA. In the basic variant, all operations are executed synchronously, i.e., the worker blocks after every operation until its result is returned.

Figure 5.2 gives an intuition on how this latch is used to access a remote data structure exclusively. First, the remote data structure is latched with an RDMA CAS operation on the 8-byte latch by setting the lock bit. Afterward, the desired data is read from the data structure, modified locally, and written back with an RDMA write operation. Finally, the remote data is unlatched with another RDMA CAS operation.

To access the remote data structure in shared mode, the clients use RDMA FAA to increment the reader count of the latch speculatively. The return value (i.e., the full

8-byte) of the operation allows the worker to check if the latch is in the exclusive mode, in which case the worker decrements the reader count and retries. Otherwise, the worker reads the data using an RDMA read and then decrements the counter to unlatch.

5.3.2 Performance of RDMA Atomics

Because every pessimistic approach relies on RDMA atomics (CAS and FAA), it is important to understand their isolated performance before discussing how our basic latch can be optimized.

Uncontended vs. contended RDMA atomics. In the first experiment, we examine the scalability behavior of contended and uncontended RDMA atomics. Both scenarios are equally vital, and while heavy contention is typically rare, it is unavoidable in some workloads, e.g., having hot tuples. To show the effect of contention, we perform an experiment in which all workers issue an RDMA CAS operation of the same 8-byte atomic counter. To understand how uncontended atomics scale, we assign a private 8-byte remote atomic counter to each worker. For reference, we compare uncontended atomics to the performance of an RDMA read.

Figure 5.3 shows the scalability behavior of both uncontended and contended RDMA atomic operations when increasing the number of workers. As we can observe, uncontended atomics scale like one-sided RDMA read operations, peaking at around 51.2 million operations per second with 128 workers. This suggests that the parallel uncontended atomic requests do not interfere, but we will demonstrate later that this does not hold for all scenarios. Note that the performance drop of uncontended atomics at 512 workers has nothing to do with the atomic operation but can be attributed to QP-thrashing [55] on the client machines. QP-thrashing means that the QP state cannot be cached on the RNIC and is swapped in and out to host memory. This occurs at around 128 utilized parallel QPs per client NIC in our hardware, i.e., 512 workers. As expected, when running the contended atomic workload, the peak is significantly lower at 2.32 million operations per second and atomic operations only scale until 8 workers.

In the literature, RDMA atomics seem to have a bad reputation for being fundamentally unscalable [102] – even for uncontended workloads. However, the above experiments demonstrate that uncontended atomics scale well w.r.t. the number of workers. In fact, they show comparable scalability to RDMA read operations. While this experiment offers valuable insights into the scalability of atomics, it is not the complete picture, as we will demonstrate.

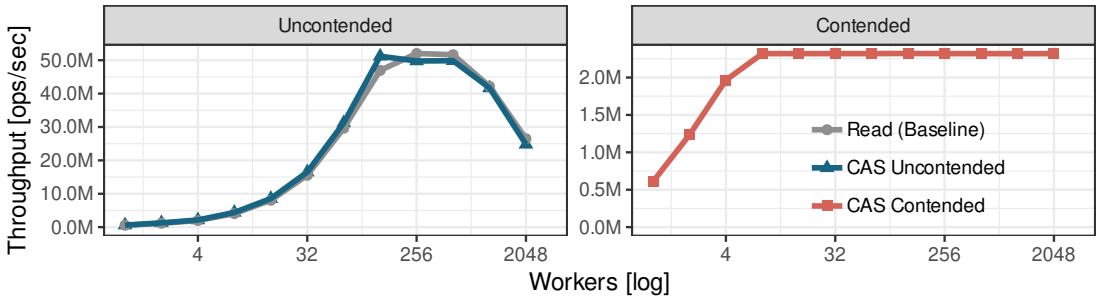


Figure 5.3: Scalability of contended and uncontended atomic RDMA operations when increasing the number of workers (4 compute nodes and 1 storage node)

Atomic stride size and alignment. Obviously, the scalability of RDMA atomics depends on the degree of parallelism. However, subtle details such as the data alignment can also interfere with the scalability. So far, our experiments placed values in a 64-byte stride, i.e., a cache line. We only used the first 8 bytes for the atomic counter and ignored the remaining 56 bytes. However, in practice, RDMA atomics are placed at larger strides as they protect data of various sizes, e.g., a 4 KB B-Tree node.

In the following experiment, we measure the effect of larger stride sizes by varying the distance between the atomic counters. As in the previous uncontended experiment, each worker has a private latch to avoid contention. Consequently, the expected outcome should be similar to the uncontended result in Figure 5.3. Surprisingly, Figure 5.4 shows that the stride size impairs the scalability tremendously. That is, with larger stride sizes, the inflection points w.r.t. throughput (highlighted in red) are reached earlier. With a 64-byte stride, the peak performance is $50M$ operations with 128 workers, i.e., the same upper-bound as in Figure 5.3. With a 256-byte stride, the peak performance is at $40M$ operations with 128 workers. With a 512-byte stride, the peak performance is halved and reached with fewer workers ($20M$ operations with 64 workers). Remember that there is no true latch contention, and we only vary the distance between the atomic counters and nothing else. The observed behavior must be based on a physical contention point in the RNIC.

NIC internals - physical contention. Through reverse engineering, we believe that the RNIC uses an internal locking table to serialize atomic operations. Since the locking table works similarly to a hash table, collisions can happen. Unrelated atomic operations can be assigned to the same slot, severely limiting the throughput of concurrent uncontended atomic operations. Based on our reverse engineering efforts, we determined that the lock slots are calculated from the last 12 bits of the atomic operation's target address. For instance, if we use a 4 KB-aligned address as illustrated in Figure 5.5b i), the last 12 bits

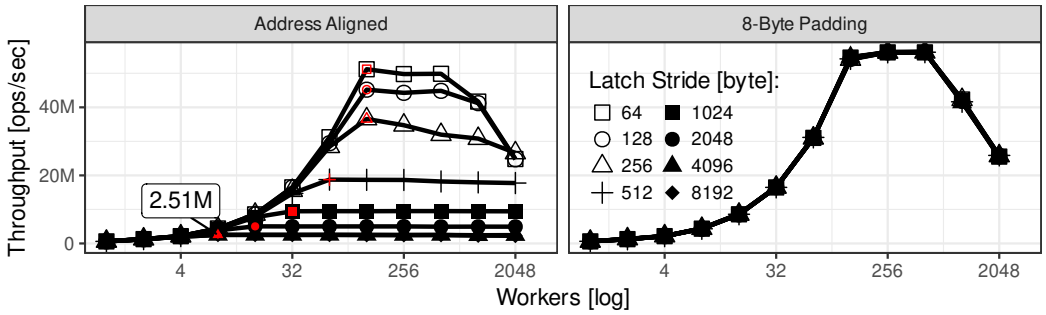
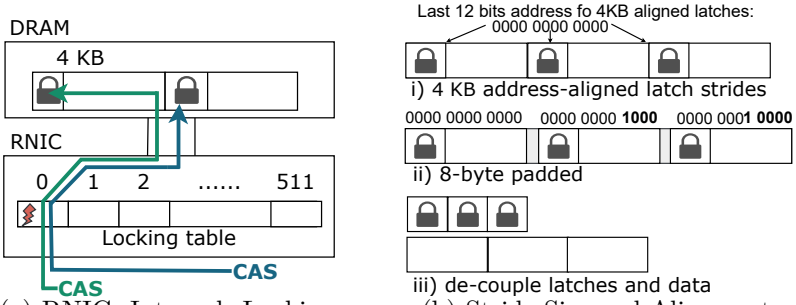


Figure 5.4: Scalability of uncontended atomics with varying strides between the latches (4 compute nodes and 1 storage node).

of the address are zeros, and they are assigned to the same lock slot. Even though the atomics do not contend on the same latch, the way the RNIC handles atomics, results in physical contention inside the locking table, as exemplified by the arrows in Figure 5.5a. The two CAS operations target different latches and are still serialized in the same lock slot, negatively impacting performance. Consequently, logically uncontended atomic operations do not scale very well when the data alignment is not carefully chosen, as shown in Figure 5.4. When we compare the results of our initial contended scalability experiment from Figure 5.3 with the performance of 4 KB-aligned stride sizes, we can see that the performance and scalability characteristics are very similar. Given the underlying hardware mechanism, this performance is now explainable: the operations are serialized in the same lock slot, whether through a collision of the address or the operations targeting the same latch. We validated the existence of a performance signature matching our hypothesized 12-bit lock table in three RNIC generations: ConnectX-3, ConnectX-5, and ConnectX-6 RNICs. Also, Kalia et al. [102] observed similar findings for an older network card (Connect-IB). However, it is hard to generalize our findings to all NICs since the implementation details are not made publicly available by the RNIC vendors. Therefore, like other papers, we can only infer implementation details [102, 223]. Instead, we emphasize that NIC hardware details are essential and demonstrate potential bottlenecks that should be carefully evaluated when building a high-performance system.

Improving the scalability of uncontented atomics. To improve the scalability of uncontended operations, we must avoid collisions in the locking table. The only way one can control this is through the data layout, i.e., the addresses of our latches. The goal is to place the latches so that the last 12 bits (used for the lock-slot calculation) are different. Consider the example in Figure 5.5b ii); instead of allocating the 4 KB blocks back-to-back, a padding of 8-byte is placed before the latches. Now, the last 12



(a) RNIC Internal Locking Table

(b) Stride Size and Alignment

Figure 5.5: Pessimistic synchronization performance can be impacted by RNIC architecture and by host memory layout.

bits of the latch addresses are not all zeros and, thus, will be assigned to different lock slots. The effect of this mitigation technique is illustrated in Figure 5.4. We can observe that all stride-sizes scale equally well (all measurements happen to be on the same line). Another possibility to better utilize the lock-slots is to decouple latches from the data as depicted in Figure 5.5b iii). In this technique, the latches are placed back-to-back. Since the latches are only 8 bytes, the last 12 bits of the latch address will vary and achieve the same scalability behavior as for the 8-byte padding. Note, in this experiment, we test the performance of the atomic operations, i.e., we do not read the data. This allows us to isolate the atomic contention effect, but in practice separating the latch from the data may have adverse effects due to locality. In particular, the translation from physical-to-virtual memory could suffer as every data access invokes two translations, i.e., one for the latch and one for the data. However, this depends on other parameters, such as the working set, tuple size, and NIC-cache size [102], and thus requires a holistic evaluation.

To conclude, despite contrasting beliefs, RDMA atomics can scale well w.r.t. the number of workers. However, the scalability depends on the number of concurrent accesses (contention) and the data alignment. While the first is workload-dependent, the latter can be carefully tuned to best utilize the internal RNIC hardware. While we demonstrate that padding is beneficial for RDMA atomics, it can also have consequences elsewhere, such as making page table lookup less efficient. Therefore, we argue that it is crucial to understand the internals of the RNIC and holistically optimize the DBMS system.

Pessimistic optimizations. With our findings on optimally utilizing atomics, we can now concentrate on designing optimized pessimistic latches. Remember that the basic latch variant executes all operations synchronously, as shown in Figure 5.2. After each

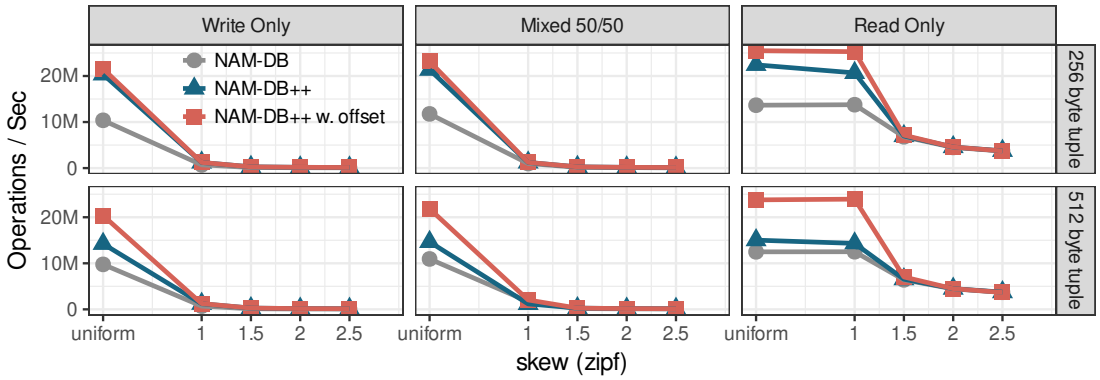


Figure 5.6: Effect of optimizations in NAM-DB (4 compute nodes and 4 storage nodes).

operation, the completion is awaited by polling the completion queue (cf. Chapter 2). While correct, this approach is inefficient and increases per-operation latency. The intuition of our optimizations is to save round-trip by exploiting the network order and guarantees of RDMA. However, we do not discuss these optimizations in detail due to space constraints in the synopsis (the optimizations can be found in Chapter 11).

5.3.3 Effect on a Disaggregated DBMS

As stated earlier, latch optimizations improve performance tremendously. Let us substantiate that claim by integrating them into an existing disaggregated DBMS. We use NAM-DB [239], which is a disaggregated RDMA-optimized DBMS. In our setup, we use 4 compute nodes with 28 workers each (112 total workers) and 4 storage nodes. Among those storage nodes, we distribute 20 million tuples. We measure the throughput in operations per second for representative tuple sizes of 256 and 512 byte. We implemented the highest optimization level and called this version NAM-DB++. In addition, we show the effect of manipulating the latches' data layout, as we discussed in Section 5.3.2. We compare the original NAM-DB with NAM-DB++ in a write-only, mixed, and read-only workload. To show the effect of contention, we vary the access skew.

Read-only performance.. Let us first focus on the 256 byte-sized tuples and the read-only workload (top right in Figure 5.6). The optimizations double the performance with uniform and slightly skewed (Zipf 1) access patterns. When the contention increases, the hardware limits the performance, and both systems converge. The dramatic performance degradation is nevertheless surprising in the read-only scenario. In theory, the read-only performance should not dramatically collapse since multiple readers can acquire a latch simultaneously. Unfortunately, the RNIC cannot handle the concurrent atomic RDMA

operations necessary to acquire the reader latch in the first place. As mentioned in the previous experiment, the RDMA atomics are serialized in the RNIC, which does not scale well when the lock slot is contended. Despite the hardware limitations, the read-only workload still performs much better than the write-only workload under high contention. For instance, with a Zipf factor of 2, the write-only performance is $170K$, whereas the read-only performance is $4.5M$.

Write performance. The contention exacerbates the performance issue in the write-only and mixed workload. The locks now logically contend in addition to the physical contention on the RNIC. In other words, the performance of the atomic operations decreases since those operations often target the same latch and are serialized in the same RNIC lock slot. Once the RNIC processes the atomic operation, the latch may have been already latched exclusively, which requires a restart and aggravates the problem.

Larger tuples. When looking at the 512 bytes-sized tuples in the read-only workload, we can see that the effect of applying the padding is more pronounced. As mentioned earlier, the larger latch strides lead to more contention inside the RNIC's lock-table, thus limiting the performance. We use 4 storage nodes, as opposed to one node in the experiment in Section 5.3.2, and still, the collisions inside the RNICs become the primary bottleneck. Therefore, with the padding optimization, the performance significantly improves by removing the bottleneck of physical contention, allowing the latch optimizations to achieve their potential.

5.4 Optimistic Synchronization

In the previous section, we observed that pessimistic synchronization scales well when latches are uncontended. However, some data structures inherently exhibit latch contention due to their design. For example, B-Tree operations require traversal through the root node. Although the root node is primarily latched in shared mode, this leads to (physical) contention on the RNIC with pessimistic synchronization, adversely affecting performance. Figure 5.7 illustrates this effect for a B-Tree with 4KB nodes stored on a single storage machine and accessed read-only from four client machines. Remarkably, the unsynchronized B-Tree saturates the bandwidth with only 16 workers, while the pessimistic synchronized version stagnates earlier and never reaches full bandwidth, even with the optimizations presented earlier.

This experiment motivates the exploration of optimistic synchronization schemes, where reads do not physically latch the data but detect concurrent modifications. Thus,

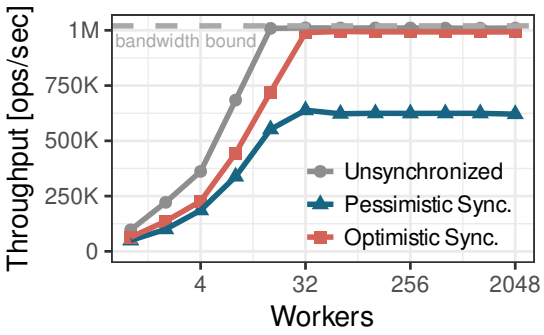


Figure 5.7: Lookups in one-sided B-Tree (4 compute nodes and 1 storage node).

optimistic schemes can avoid RDMA atomics while reading. Figure 5.7 demonstrates that optimistic synchronization can achieve full bandwidth. However, implementing optimistic synchronization is challenging, as ordering guarantees may lead to incorrect synchronization in some schemes.

Our work studies existing techniques to address these challenges [55, 149], and we quantify the overhead they incur and how they actually compare versus pessimistic schemes. However, due to the brevity of this synopsis, we will not present the specifics of our contributions here but refer to Chapter 11. The main goal of the previous discussion was to provide an intuition why optimistic synchronization can be an attractive and performant alternative to pessimistic synchronization schemes. This motivated our efforts to overcome its inherent difficulties as outlined in Chapter 11. In the following, we provide a summary of the main findings with regard to both optimistic and pessimistic one-sided synchronization.

5.5 Key Findings: One-Sided Synchronization Is Hard

We summarize two key findings (more are in the paper) of our analysis as lessons learned for correct one-sided synchronization.

Data alignment impacts the RDMA atomic scalability. Because of the lock table present in current RNICs, physical contention between independent latches can arise. We demonstrate that this is not only easily demonstrated by microbenchmarks that highlight the behavior but also have an important role in the scalability of a real system (i.e.,

NAM-DB). Hence, we advocate that system designers pay close attention to their data alignment when leveraging RDMA atomic operations to avoid this physical contention.
Pessimistic synchronization is more "future proof". As demonstrated, subtle hardware characteristics can have a profound impact on correctness. We highlight this using a widespread deployment but point out that disaggregated memory technologies are in active development. Changes to intermediate components of RDMA communication may alter the behavior of concurrent RDMA reads and writes, and therefore optimistic synchronization schemes, in unpredictable ways. In contrast, RDMA Atomic operations implement a well-established higher-level abstraction independent of hardware implementation. Hence, system designers should lean toward simple pessimistic synchronization when prioritizing production stability until the community has successfully converged on well-defined memory semantics for RDMA reads and writes.

5.6 Summary

This chapter explored one-sided synchronization primitives, revealing various pitfalls associated with their application. While pessimistic approaches typically ensure correctness, they can experience performance degradation due to internal hardware limitations, such as the lock table for RDMA atomics. On the other hand, optimistic approaches, while circumventing RDMA atomics, are hard to get correct. These methods often rely on implicit hardware assumptions that may have changed over the years, and the RDMA primitives (`read` and `write`) used for building these optimistic primitives are not designed with concurrency in mind. Therefore, some optimistic schemes fail to synchronize correctly. Our guidelines serve two primary purposes: 1) identifying potential pitfalls to avoid and 2) offering benchmarks to facilitate a more comprehensive understanding of the underlying hardware. Drawing on the insights obtained, developers can build one-sided synchronization primitives that are correct and scalable, tailoring them specifically to the system architecture.

In the next chapter, we will consolidate the lessons from this chapter for building a general-purpose storage engine.

6 Buffer-Managed Storage Engine

Publication. Most of this chapter is published in the peer-reviewed publication "ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA" in the *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022* [252], cf. Chapter 13

Contributions of the author. The contributions to the above publication by Tobias Ziegler are as follows. Tobias Ziegler is the leading author and was thus responsible for the proposed design of ScaleStore, implementation, evaluation, and the manuscript. The remaining co-authors, Carsten Binnig and Viktor Leis contributed invaluable feedback.

In response to the fact that in-memory DBMSs are becoming economically unattractive, primarily driven by the fact that DRMA prices stagnate, this chapter focuses on more cost-efficient alternatives such as flash Solid State Drivess (SSDs). We propose a high-performance, cost-effective distributed storage engine that builds on the insights from previous chapters. This chapter primarily focuses on the buffer manager - the fundamental component enabling the efficient use of NVMe Flash SSDs as secondary storage.

Like a traditional single-node buffer manager, our buffer manager organizes data into units termed *pages* and transparently loads and evicts pages to and from the SSD. All data, including tables and indexes, are stored on these fixed-size pages, allowing that these are managed transparently. However, unlike traditional buffer managers, ours leverages low-latency RDMA to enable a transparent memory abstraction that accesses the collective DRAM and NVMe storage across multiple nodes. The core of our buffer manager is a distributed caching strategy that dynamically decides which data to keep in memory (and which on SSDs) based on the workload.

Since our engine treats indexes identically to any other data, indexes can be transparently accessed across the entire cluster. This means indexes, synchronization primitives, and the buffer manager operate interdependently within a buffer-managed storage engine allowing us to reassess lessons learned from the previous chapters to avoid potential bottlenecks. For example, our findings indicate that one-sided synchronization primitives, like RDMA atomics, may potentially pose scalability issues, even under uncontended workloads. We have adopted a holistic design approach to the storage engine to avoid such pitfalls.

Therefore, we begin in Section 6.1 by drawing design considerations from previous work and outlining the design of our engine. This is followed by a detailed discussion on our RDMA-based message-handling framework in Section 6.2. Subsequently, we present our caching protocol in Section 6.3 and explain our approach to efficiently identify and evict infrequently accessed pages to SSD in Section 6.4. We further discuss the integration of synchronization primitives and indexes in Section 6.5. The chapter concludes by presenting a selection of our experiments in Section 6.2.2 and summarizing key insights derived from our work in Section 6.7.

6.1 ScaleStore: Design Overview

This section presents the overall design of our storage engine called ScaleStore. First, we reflect upon the insights gained from the previous sections to identify key design considerations that guide our approach. Building upon this foundation, we then present the primary building blocks of the system and illustrate a motivating example.

6.1.1 Design Considerations From Previous Work

This section distills the lessons learned from the previous chapters, identifying key considerations that influence the design of our storage engine.

Dynamic data placement. Earlier chapters, particularly Chapter 4 and Chapter 3, emphasized the limitations of static data partitioning, e.g., suffering from (read) hot spots or requiring repartitioning when the workload changes. In addition, asking users to decide the optimal partitioning can be a considerable and often impractical task. Therefore, ScaleStore adopts a more dynamic strategy favoring a workload-driven data placement over user-defined static partitioning.

Embracing flash storage. While our prior focus was on distributed in-memory DBMSs, the advantages of flash memory has become apparent. In-memory DBMSs, while fast, are increasingly uneconomical due to the current DRAM price development. In contrast, flash memory, being about 25 times more cost-effective than main memory [78], is economically attractive. However, a critical consideration is the substantially higher latency of NVMe, at least three orders of magnitude more than DRAM, which can significantly degrade performance when the working set exceeds memory capacity. Therefore, our system keeps the working set in the collective DRAM of the cluster that can be accessed with lower latency via RDMA and only sporadically loads and evicts pages from and to the cost-efficient SSDs.

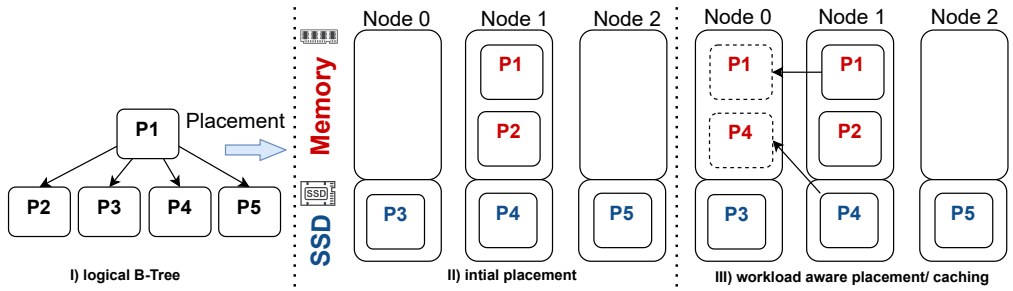


Figure 6.1: Overview of ScaleStore using a distributed B-Tree. II The B-tree pages can be spread across local/remote memory and SSDs. III The caching protocol optimizes how pages are laid out across machines

Use caching to exploit DRAM latency. Our initial designs deliberately avoided caching to explore the potential of pure RDMA-based systems. Nevertheless, the significantly lower latency offered by DRAM presents an opportunity for performance optimization. Especially when arranging data in flash storage-optimized pages, such as 4KB, the RDMA latency can be high, up to two orders of magnitude compared to traditional memory access. Accordingly, ScaleStore integrates caching as a fundamental component of its design.

Avoiding one-sided RDMA. We choose to eschew one-sided RDMA for accessing remote data directly for several reasons: Firstly, one-sided RDMA atomics may introduce scalability issues, even in uncontended workloads. Secondly, since caching is an integral part of ScaleStore, the lack of coherence between RDMA atomics and CPU atomics poses a challenge, requiring every local data access to use RDMA atomics for latching the page. Lastly, managing memory locations for one-sided primitives is complex, especially when data is evicted and loaded from secondary storage. Considering these challenges, we explore alternative approaches to data exchange with RDMA.

6.1.2 Main Building Blocks for ScaleStore

Based on the previously stated design considerations, we derive the main building blocks of our distributed storage engine, ScaleStore.

Distributed cache protocol. We base our design upon the shared-writer with cache archetype and develop, as a first core contribution, a *distributed caching protocol based on RDMA*. This protocol operates on fixed-size pages and is optimized for DBMS workloads. It provides transparent page access across machines and storage devices, allowing worker threads to access all pages as in a non-distributed system. Importantly, it dynamically handles shifts in the workload, such as changed access patterns, by repopulating the

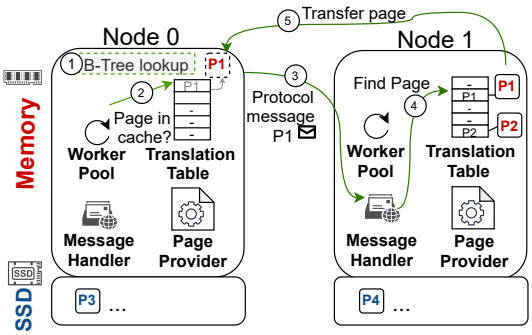


Figure 6.2: ScaleStore 's main components.

in-memory cache and migrating pages from one node to another. Furthermore, to optimize performance for workloads with high access locality, we dynamically cache frequently-accessed pages in DRAM on multiple nodes simultaneously. This caching protocol also coordinates page accesses across a cluster of nodes, ensuring a consistent view of the data despite concurrent modifications and multiple copies being cached by several nodes.

Efficient RDMA message handling. To efficiently implement the coherence protocol, we evaluate various techniques for RDMA message handling. While two-sided with `send/recv` seems like a natural fit for message handling, we find that one-sided message handling is actually preferable and yields better performance for a storage engine like ScaleStore. It is important to note that one-sided RDMA is not used to access data directly but to exchange protocol messages among the participants.

High performance eviction. Because ScaleStore caches pages and handles workload shifts, the local DRAM buffer may fill up at very high rates, and thus unused (cold) pages must be evicted efficiently. Existing strategies such as LRU or Second Chance are either too slow or not accurate enough. Moreover, to work with the distributed nature of ScaleStore, the eviction strategy must be closely integrated into our caching protocol. ScaleStore, therefore, employs a *distributed high performance replacement strategy* to identify cold pages and evict them efficiently.

These building blocks enable ScaleStore to handle shifts in workload efficiently, optimize performance for workloads with high access locality, and provide indexes and synchronization primitives. In addition, our eviction strategy generally applies to arbitrary data structures rather than being hard-coded to any particular data structure, making ScaleStore a general-purpose storage engine.

6.1.3 A Motivating Example

Next, we present a workflow example that highlights ScaleStore's core mechanics and the interaction of its key components.

In ScaleStore, page access is transparent: any node can access any page using its page identifier (PID), with the system automatically managing the page placement. Consider the B-tree illustrated in Figure 6.1 I, composed of a root page (P1) and four leaf pages (P2-P5). Figure 6.1 II depicts a possible initial distribution of pages across a three-node cluster: Pages P1 and P2 are cached by Node 1, while pages P3, P4, and P5 are not cached and only reside on the SSDs. Although P1 and P2 also have copies on SSDs, they are omitted for brevity. It is important to note that this placement represents a snapshot in time, with ScaleStore dynamically adjusting cached pages based on the workload during operation.

For instance, as shown in Figure 6.1 III, if Node 0 executes a lookup involving pages P1 and P4, it will replicate P1 from Node 1's remote main memory and P4 from Node 1's remote SSD. It is worth noting that Node 1 does not automatically cache P4. Consequently, P1 will be cached by both Node 0 and Node 1, providing substantial benefits for frequently-accessed pages such as the root of a B-tree.

When Node 2 accesses page P4, it could theoretically retrieve the page from either the SSD of Node 1 or the main memory cache of Node 0. In light of current hardware latencies, ScaleStore consistently opts for the latter choice. As a result, ScaleStore effectively combines the main memories of all nodes into a unified DRAM cache connected through a low-latency RDMA network. Moreover, since page placement is dynamic and workload-driven, ScaleStore can seamlessly adapt to various placement scenarios.

For example, suppose the read-only working set is small enough to fit into each node's cache. In that case, all data will eventually be fully replicated on each node, thereby enabling high performance by circumventing the network overhead associated with purely distributed systems. In another scenario, when the workload is partitioned, with each node predominantly working on a distinct data subset, each node will cache its specific subset over time, taking advantage of locality. Finally, it is crucial to note that our approach naturally adapts to workload shifts during runtime thanks to our invalidation-based page coherence protocol described in Section 6.3.

6.1.4 Main Components

As Figure 6.2 shows, each ScaleStore node consists of four main components: (1) *Worker Pool*, (2) *Translation Table*, (3) *Message Handler*, and (4) *Page Provider*. In the following, we briefly illustrate how these components interact at runtime.

Consider again the example from Figure 6.1 III where Node 0 wants to access the root node (page P1) from Node 1. Figure 6.2 illustrates how page access is implemented. The first step from the application perspective is to invoke a worker thread ① to execute an operation on a data structure (i.e., B-Tree lookup). The worker then consults the local translation table to check if page P1 is already in its cache ②. If the page is indeed in the local cache, the page ID of P1 is translated to the memory address of the cached page and returned to the application. If the page is not in the local cache, it has to be fetched from a remote node. For example, to request a page from Node 1, Node 0 invokes the distributed page coherence protocol. A page request ③ is sent to Node 1 to request page P1. When the message handler ④ on Node 1 finds that the page is already loaded into the remote memory using its translation table, the page is ⑤ directly transferred to Node 0. Otherwise, the page is loaded from SSD into the temporary memory of Node 1 before transferring it to Node 0.

6.2 Low-Latency RDMA Messaging

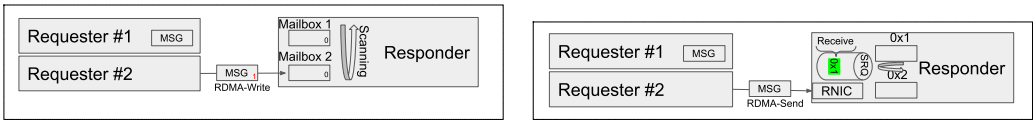
Publication. The work on RDMA communication patterns is published in the peer-reviewed publication "RDMA Communciation Patterns" in the [253], cf. Chapter 12

Contributions of the author. The contributions to the above publication by Tobias Ziegler are as follows. Tobias Ziegler is the leading author and was thus responsible for the proposed design space of efficient message handling in RDMA, benchmark implementation and evaluation, and the manuscript. The remaining co-authors, Viktor Leis and Carsten Binnig contributed invaluable feedback.

Before diving deeper into the caching-protocol, let us first focus on how we efficiently transfer protocol messages and pages with RDMA. As stated earlier, our design consciously avoids one-sided RDMA for directly accessing remote data. Instead, we explore alternative approaches that allow efficient message handling with RDMA.

6.2.1 Efficient Message Handling in RDMA

This evaluation focuses on a *synchronous communication* pattern between a requester (sender) and responder (message handler). For example, referring to Figure 6.2, a worker



(a) Write, *Mailboxes* (b) Send, *Shared Receive Queue*

Figure 6.3: Building Blocks — *Write with Mailboxes and Send with Shared Receive Queue*

on Node 0 requests page P1 from Node 1. This process includes a page request ③ being dispatched to Node 1. Upon receiving the request, the message handler (responder) at Node 1 processes it and sends a response back to Node 0. The response could be a page or another protocol message.

The question we want to answer in this work is how to efficiently realize message passing in RDMA. While RDMA two-sided operations are naturally suited for this type of communication, it might seem counterintuitive that RDMA one-sided operations can also be utilized effectively. The following sections will present these options and their respective implementation strategies.

Employing One-Sided Communication. When using RDMA `write` to transmit messages, it is essential to remember that it is a one-sided verb, i.e., directly writing to remote memory while bypassing the remote CPU (refer to Chapter 2). However, the responder must be informed when new messages arrive, which poses a challenge with CPU bypassing. To address this issue, a prevalent protocol [55, 64, 253] involves writing messages to designated memory regions, often referred to as *mailboxes*. Each requester knows the memory address of its dedicated mailbox, enabling the responder to continuously scan the mailboxes for incoming messages, as illustrated in Figure 6.3a. Here, scanning refers to reading the memory.

In order to safeguard against reading partially transmitted messages, the responder needs to verify that a message has been completely written into RDMA memory. To enable this, the responder can exploit that `write` operations are persisted in memory in increasing address order: the last bit of a message is written last. In practice, the sender marks the final bit in the message and commits it to the designated mailbox on the responder's side using one-sided RDMA `write`. Meanwhile, by checking the last bit, the responder continually scans the mailboxes to identify incoming messages, ensuring it reads only completely transmitted messages.

Utilizing Two-Sided Communication. In contrast to the `write` operation, `send` requires a `receive` to be posted to the receive queue in advance. As elaborated in Chapter 2, the `receive` contains information about the memory location where an

incoming `send` is copied. This is illustrated in Figure 6.3b: Two possible memory locations ($0x1$ and $0x2$) exist where the first location is registered at the RNIC.

When a `send` operation consumes a `receive`, the RNIC copies the incoming message to the memory location as per the `receive` and generates a completion event. Detecting incoming `send` messages requires continuous polling of the completion queues. This becomes a computationally expensive task, particularly for a responder handling multiple connections. To alleviate this overhead, RDMA provides a feature known as a *shared receive queue*. The shared receive queue manages multiple incoming connections, significantly reducing the polling overhead. It simplifies the process by allowing the responder to monitor a single queue to manage all incoming requests. Thus, we strongly recommend utilizing the shared receive queue on the responder side.

Design space exploration. Given our communication pattern, which differentiates the roles between the sender and responder, we recognized that each role could employ one-sided or two-sided communication. Consequently, there are four potential designs:

- Write/Write
- Send/Send
- Write/Send
- Send/Write

Each item combines two methods, separated by a '/'. The method listed before the slash refers to how the sender transmits the message to the responder. Conversely, the method after the slash indicates the receiver's response mechanism.

In the subsequent section, we will briefly present the key findings of our evaluation. We refer to our full paper for a more comprehensive exploration of message handling (see Chapter 12).

6.2.2 Evaluation

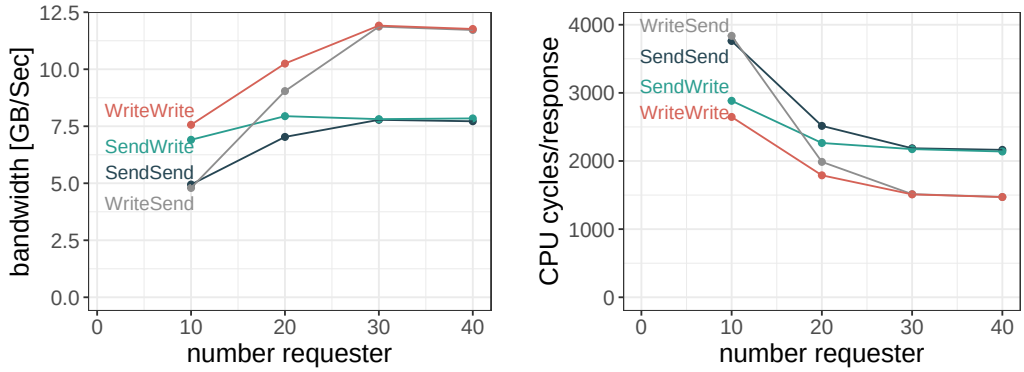
We use a 6-node cluster, each with two Intel Xeon Gold 5120 Skylake processors, 256GB RAM, and two Mellanox ConnectX5 cards connected via an InfiniBand EDR switch. The nodes run Ubuntu 18. 04 Server Edition and Mellanox OFED-5.0-1.0.0.0 network driver. The benchmark is implemented in C++17 and compiled with GCC 7.3.0.

N-to-1 evaluation: Bandwidth. This experiment explores an N-to-One scenario where the responder (i.e., message handler) operates on a single thread. We incrementally increase the number of requester servers from one to four, with ten threads running on

each server, resulting in a maximum of 40 requester threads. Figure 6.4a depicts the relationship between bandwidth and the number of requesters. This experiment uses a 64-byte message for requests and a 4 KB response size. Figure 6.4a demonstrates that the WriteWrite strategy outperforms the other strategies of up to 30 workers; afterward, it reaches the full bandwidth. Surprisingly, the fully two-sided SendSend and the SendWrite scheme achieve only approximately 7.5 GB/s. The reason is that the SendSend strategy is CPU-bound. Figure 6.4b illustrates the number of CPU cycles consumed per 4 KB response. It is apparent from this data that the SendSend and SendWrite methods do not show the same efficiency in amortizing CPU cycles as the WriteWrite does when the number of requesters increases. A deeper analysis of the shared receive queue reveals a significant expenditure of cycles in `pthread_spin_lock` and `mlx5_poll_cq`. This observation implies that polling on the shared receive queue incurs a consistent overhead that does not get amortized with more requesters.

Key findings. In our full paper attached in Chapter 12, we have evaluated a variety of topologies, including 1-to-1 and N-to-M, alongside diverse message size configurations and optimizations. A consistent pattern emerged throughout these experiments: WriteWrite is the most promising message-handling strategy in our targetted communication scenario.

Consequences for ScaleStore. Our findings suggest that a fully one-sided message-handling framework yields the best performance. Accordingly, ScaleStore utilizes this approach for all inter-node communication, encompassing protocol messages and page transmissions. To reduce the number of connections in ScaleStore (i.e. mailboxes), every worker is connected to a message handler on every remote node. The message handler provides a private mailbox for every worker, which is continuously monitored to detect incoming requests. When a new request arrives, the message handler processes this request and replies to the worker's thread-local mailboxes. In ScaleStore, all protocol messages are of cache-line size (64 byte), making this design very efficient in latency, i.e., sub-microseconds for such small messages. We carefully optimized our design to avoid the message handler becoming the bottleneck. Every data structure the message handler interacts with has been carefully designed to be scalable and non-blocking. As we will discuss in a subsequent section, we have offloaded most protocol work to the worker threads rather than burdening the message handler.



(a) N-to-1 Bandwidth Experiment

(b) CPU cycles per request

Figure 6.4: Design space evaluation.

6.3 Invalidation-Based Page Coherence Protocol

This section describes our distributed page coherence protocol that forms the foundation for our buffer-managed storage engine.

6.3.1 Protocol Overview

The basic idea of our protocol is inspired by cache coherence protocols like MESI that are used by multi-core CPUs to provide the illusion of a single unified main memory despite having multiple per-core caches and therefore duplicated copies of cache lines.

The MESI protocol. In MESI, each cache line has one of four eponymous states: (1) Modified (the cache line has been modified), (2) Exclusive (only a single copy exists), (3) Shared (there are multiple copies of this cache line), and (4) Invalidated (the cache line is out-dated). The protocol ensures coherence by using appropriate invalidation messages. For example, if a cache line is in the *Shared* state and a core wants to write to it, invalidation messages are sent to all cores holding it in the *Shared* state.

Our protocol. In contrast to MESI, to work in a distributed setting, our protocol has several important differences. Instead of cache lines (usually 64 bytes), our protocol uses pages (e.g., 4 KB) as the unit of coherency to amortize network overhead and enable SSD support. These pages can be fixed in the local cache, which prevents page invalidation until an ongoing operation is finished. In a distributed cache coherence system, this is required to avoid the same page being fetched multiple times from a remote node due to invalidations which could quickly lead to a significant increase in overall latency. We further ensure robustness and fairness with *anticipatory chaining*, a technique that

orders conflicts and thus ensures fairness and avoids starvation. Finally, our protocol implements sequential consistency, whereas multi-core CPUs typically implement lower memory consistency guarantees such as Total Store Order. Our protocol is separated into two distinct paths: (1) the local hot path and (2) the remote invocation. Figure 6.5 gives a high-level overview of the decisions in those paths. With this, ScaleStore follows the design principle "*make the common case fast*" and therefore, the local hot path is an important optimization to avoid unnecessary network messages when a page is already in the local page cache.

6.3.2 Local Hot Path

The local path in our protocol is a fast path that does not involve any remote messages – all decisions can be made locally. Especially for frequently-accessed pages, this obviously provides significant performance benefits. For instance, inner B-Tree pages that are rarely modified but frequently accessed are very likely to be cached on multiple nodes and can be accessed without any networking overhead. In the following, we explain the individual steps of the local hot path as shown in Figure 6.5.

Check ownership mode. The first step of every page access is to check the ownership mode. For this, the page translation table is used, which translates the page identifier (PID) to cache frames (similar to buffer frames in buffer managers). Besides the page data, we store the page latch, eviction information, and ownership metadata inside a cache frame. The ownership metadata describes the ownership mode for a page (i.e., what operations are allowed): (1) *node-exclusive* or (2) *node-shared*.

Before a worker thread accesses a page, it checks if the page is in the correct ownership mode. For example, the page has to be in the *node-exclusive* ownership mode for modifications (such as an update). Conversely, multiple nodes can access a page simultaneously only if the page is in the *node-shared* ownership mode. Note that node-exclusive and node-shared ownership regulate accesses on a *node-level basis*. That is, if a node owns a page in node-exclusive mode, then all worker threads can exclusively access this page.

Latch and return page. If the ownership mode is correct, the second step is that the page has to be latched for the concrete worker thread that wants to access the page. To efficiently synchronize worker threads within a node, we provide a hybrid latch [23] that combines a standard mutex with the option for optimistic access:

- *Exclusive:* Acquires cluster-wide exclusive access to a page for a worker thread. Note that this first requires a transition to the node-exclusive ownership mode for

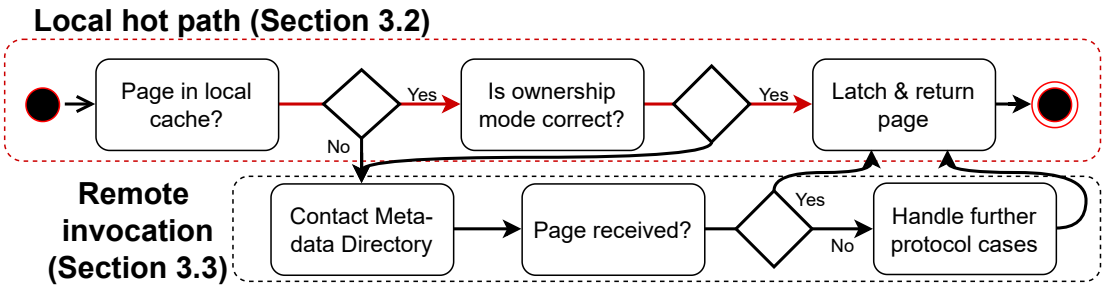


Figure 6.5: Protocol flow.

this page. Once the latch is acquired, the page is fixed, i.e., it cannot be evicted from the local cache until it is unlatched.

- *Shared:* Acquires shared access to a page where multiple threads (and nodes) can access it. The ownership mode for this latch can be either node-shared or node-exclusive. As before, the page is fixed in the local cache.
- *Optimistic:* Allows an optimistic page read without acquiring the latch. The ownership mode for this latch can be again either node-shared or node-exclusive. In optimistic mode, the page is not fixed and can be evicted from the local cache.

While shared and exclusive modes use a traditional OS-supported read/write mutex, the optimistic mode sidesteps the mutex. This avoids cache line invalidations for latch acquisition and is crucial for making reads scalable on multi-core CPUs. Optimistic mode relies on a version counter, which is incremented for every page modification, to detect concurrent page modifications or ownership changes. If the version has changed (or the page was evicted), the reader must restart its read operation. If restarts keep happening, one can easily fall back to the shared latching mode, which will ensure forward progress.

6.3.3 Remote Invocation

In cases where the page is not in the local cache, ScaleStore triggers the remote invocation path. The remote path consists of several steps, as shown in Figure 6.5 (lower part).

Directory. To query the current ownership mode and the location of a page, we first need to contact the *directory* node. To avoid a single directory node from becoming a performance bottleneck, every node is a directory for some of the pages. This also ensures that the SSD capacities are equally utilized because pages are only persisted at the directory nodes. In Figure 6.1, for example, the directory node of page P3 is Node 0. The directory has full knowledge about the state of the page, such as which other nodes

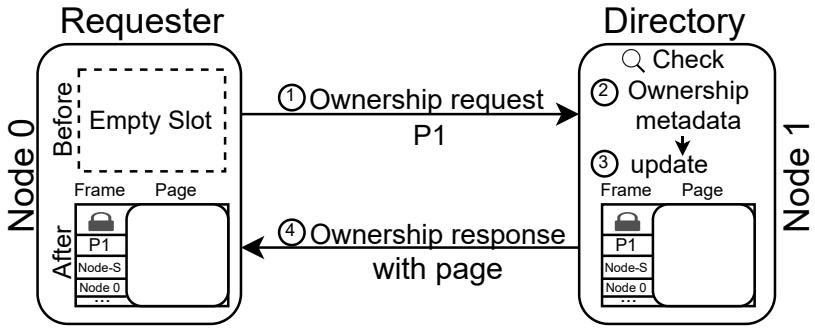


Figure 6.6: Base case of the remote invocation.

currently cache the page and in which ownership mode. As such their main responsibility lies in managing the metadata for the invalidation-based cache coherence protocol.

Base case. We can request the page as shown in Figure 6.6 (which illustrates the base case). Node 0 – the requester – sends an *ownership request* ① to the directory. The ownership request describes to the directory what page is needed and whether *node-exclusive* or *node-shared ownership* is required. In the example of our B-Tree lookup in Section 6.1.3, Node 0 needs node-shared ownership for page P1. The directory then checks the ownership metadata ②. This is necessary because conflicts could happen, but for now, let us assume there is no conflict. Subsequently, the ownership metadata in the cache frame is updated ③ on the directory node to reflect where the page is located. Finally, an *ownership response* ④ is sent and the page is copied to Node 0, which is then called a *node-shared owner*. Note, as discussed before, the node-shared owner stores the ownership mode in its local cache frame to support the local hot path.

Conflict cases. As outlined in step ② of Figure 6.6, two types of conflicts may arise during an ownership request: (1) exclusive and (2) shared conflicts. Conflicts in our protocol stem from incompatible ownership modes, i.e., when a requesting node (Requester) desires a specific ownership mode while another node already possesses the page in an incompatible mode, as depicted in the table below.

Requester wants:	Other Node has:	
	Node-exclusive	Node-shared
Node-exclusive	exclusive conflict	shared conflict
Node-shared	exclusive conflict	no conflict

In the interest of brevity, this synopsis will not dive into how conflict cases are handled; comprehensive information on this topic can be found in Chapter 13. Furthermore, one

of our main contributions in the protocol *Anticipatory Chaining*, a technique designed to enhance fairness and robustness, is best discussed in the full paper.

6.4 High Performance Page Eviction

ScaleStore achieves high transaction rates, which can quickly add new pages to the caches, exhausting their capacity. o avoid throttling the overall performance due to a shortage of free pages, ScaleStore must efficiently evict cold pages while retaining hot pages in the cache. This need is further underscored by the fact that every remote page access consumes a local free page when the requested page is copied. Given that this operation occurs frequently and with RDMA latency, it is crucial always to have free pages available. To achieve these goals, we separate our eviction process into two components: (1) a low-overhead strategy to track page accesses and (2) a dedicated background thread – the page provider – which utilizes the access information to evict pages. This separation of concerns allows worker threads to focus solely on query processing while the page provider handles the eviction in the background.

6.4.1 Epoch-Based LRU Approximation

Downsides of existing strategies. To distinguish between hot and cold pages, the access pattern has to be tracked in some way. A well-known eviction strategy is Least Recently Used (LRU), which orders pages based on their access recency by maintaining an LRU-list. Cold pages are stored at the end of the list, while hot pages are at the front, and thus one can reliably classify cold pages. While LRU would evict the right pages, maintaining an LRU list incurs high overhead for every page access and can easily become a scalability bottleneck in multi-core systems. An LRU-approximation such as *Second Chance* (or Clock) may be employed to avoid this overhead. Unfortunately, *Second Chance* only classifies hot pages well (since they are typically accessed very often), but it often evicts *warm* pages instead of cold pages. Evicting warm pages may lead to a significant slowdown in ScaleStore because required pages need to be read from remote memory or SSD, or even worse, pages may bounce between nodes. To identify cold pages more reliably, we need a more fine-grained distinction between the degree of hotness without incurring the overhead of LRU.

Epoch-based LRU approximation. The basic idea of our LRU approximation is to use a periodically-growing global epoch counter. This global epoch counter tracks the access time for each page. The current epoch is determined from the global epoch counter

and stored in the cache frame when accessing a page. This conceptually clusters similarly cold, warm, or hot pages, thus creating equivalency classes. In other words, some pages may share the same last accessed epoch and are treated equally by the eviction strategy. For instance, the B-tree root is likely accessed in the current epoch, whereas some leaf pages might have been accessed in an earlier epoch. This approach significantly reduces the cost of tracking access information because the global epoch is rarely incremented and checking if a page has been accessed in the current epoch is a mere `if`-statement:

```
if(gEpoch > frame.lastEpoch) // if avoids unnecessary
    frame.lastEpoch = gEpoch; // cache-line invalidations
```

Compared to LRU, our approximation results in much higher performance (and multi-core scalability) while still accumulating enough access history to identify cold pages.

6.4.2 Page Provider

The main goal of the page provider is to maintain a sufficient number of free cache frames per node, even under workload changes.

Sampling-based approach to find cold pages. With our epoch-based LRU approximation, the workers track access information efficiently, but what is left to discuss is how the page provider utilizes this information to find cold pages. To identify cold pages, we apply a sampling-based (randomized) approach that iterates over the translation table. We sample N pages, sort them and determine a configurable epoch eviction threshold, e.g., 10% smallest epochs from the sample. Based on this epoch eviction threshold, pages are evicted from the cache. The sampling phase is repeated when the epoch changes or if the rate at which free pages are needed cannot be satisfied.

Evicting pages. When the page provider evicts a page, it is important to know whether it is dirty (modified) or not. As we know from Section 6.3, only the directory has full knowledge about the state of the page and, inter alia, if it is dirty. Therefore, there are two cases when the page provider evicts a page: (1) the evicting node is the directory, (2) or not. When the current ScaleStore node is the directory, it knows if the page is dirty. Typically, dirty pages are persisted to SSD, but when the page is replicated on other nodes, the directory can evict the page immediately. When the current ScaleStore node is not the directory of the selected page, we need to inform the (remote) directory node: An eviction request is sent to the directory, which then latches the page exclusively and checks for ongoing concurrent page requests with the conflicting epoch. When the

directory discovers that the page is dirty, it copies the page to its cache and eventually evicts it to SSD.

6.5 Synchronization Primitives & Indexes

The key abstraction that ScaleStore 's interface offers for application developers are different latch guards.

Latch guards for page access. In ScaleStore, we introduce latch guards that wrap around page accesses so that distribution is fully transparent and sequential consistency is guaranteed. As such, page guards act as a proxy for translating PID to the memory address and acquiring the correct ownership and latch modes. For each latch mode, we provide a guard:

- `ExclusiveGuard(PID)`: Latches the desired page in exclusive mode and ensures that the page is in node-exclusive ownership.
- `SharedGuard(PID)`: Latches the desired page in shared mode and ensures that the page is in node-exclusive or node-shared ownership mode (as both are compatible with reads, see Section 6.3).
- `OptimisticGuard(PID)`: Latches the desired page in optimistic mode, i.e., saves the version, and ensures the same ownership modes as the `SharedGuard(PID)` guard. Due to its optimistic nature, a programmer needs to ensure that no concurrent changes occur. Therefore, this guard provides a `hasChanged` method, which indicates if a restart is necessary.

We further provide update and downgrade guards, e.g., an existing `SharedGuard` can be passed to a new `ExclusiveGuard` to upgrade from shared to exclusive. All guards have in common that they provide a `data` method to conveniently access the underlying page and the object encapsulated in that page, e.g., a B-Tree node. Moreover, with the destruction of the guards, the pages are unlatched.

6.5.1 Example: B-Tree Lookup

We now explain how the abstractions are applied in practice to develop the lookup operation in a distributed B-tree. Our B-tree code has only about 900 lines and thus is comparable to a single-node implementation. More importantly, the distributed implementation is as easy as for a non-distributed B-tree. We use this distributed B-Tree

implementation in our evaluation in Section 6.6. The abbreviated C++ lookup code looks as follows:

```
0 bool lookup(KeyType key, ValueType& returnValue) {
1    restart:
2    OptimisticGuard g_parent(catalogPID); // get catalog
3    // get rootPID from catalog ...
4    OptimisticGuard g_node(rootPID);
5    if (g_parent.hasChanged()) goto restart;
6    auto node = g_node.data<NodeBase>();
7    if (g_node.hasChanged()) goto restart;
8    while (node->type.isInner()) { // traverse inner nodes
9      auto& inner = reinterpret_cast<Inner&>(node);
10     if (g_parent.hasChanged()) goto restart;
11     PID nextPid = inner.children[inner.lowerBound(key)];
12     if (g_node.hasChanged()) goto restart;
13     g_parent = std::move(g_node); // node becomes parent
14     g_node = OptimisticGuard(nextPid);
15     node = g_node.data<NodeBase>(0); // get next node
16     if (g_node.hasChanged()) goto restart;
17   }
18   auto& leaf = reinterpret_cast<Leaf&>(node);
19   SharedGuard sg_node(std::move(g_node));
20   // leaf latched; search key and return it ...
21 }
```

Synchronization is done using optimistic lock coupling [119, 120], and consequently we traverse pages using two optimistic guards: For the parent page `g_parent` (Line 2) and `g_node` for the current page (Line 4). The root node is stored in a catalog page, which is buffer-managed like the B-tree. After every optimistic page access (Lines 5, 7, 10, 16), we validate whether that node was modified concurrently and restart if it was. Lines 8-17 traverse the inner nodes, replacing the parent with the current node at each level. Once we arrive at the leaf page (Line 18), we upgrade its optimistic guard to a shared guard.

Other data structures. As long as data is stored on fixed-size pages, the programming interface can implement arbitrary data structures, e.g., hash table, LSM tree, or columnar storage.

6.6 Experimental Evaluation

We conducted our experiments on a 5-node cluster running Ubuntu 18.04.1 LTS, with Linux 4.15.0 kernel. Each node is equipped with two Intel(R) Xeon(R) Gold 5120 CPUs (14 cores), 512 GB main-memory split between both sockets and four Samsung SSD 980 Pro M.2 1 TB SSDs connected via PCIe by one ASRock Hyper Quad M.2 PCIe card. We use the Linux md software RAID 0 implementation and use direct block device access. The nodes are connected with an InfiniBand network using one Mellanox ConnectX-5 MT27800 NIC (InfiniBand EDR 4x, 100 Gbps) per node.

When not noted otherwise, we configured ScaleStore as follows: Every node has a 150 GB in-memory cache, and we use 4 KB pages. We use 20 worker, 4 message handler, and 2 page provider threads (pinned to NUMA 0). We use an optimistically-latched B-Tree implemented on top of ScaleStore. In all experiments, the benchmark drivers are implemented in C++ and compiled into one binary with ScaleStore.

Workloads. We use YCSB, a widely-used OLTP-style benchmark [43]. For all experiments, the key-value pairs use 8 byte keys, and the values are randomly generated strings of 128 bytes. We use the following workloads: *100% Reads*, *95% Reads & 5% Writes*, *50% Reads & 50% Writes*, and *5% Reads & 95% Writes*. Reads are *point lookups*, and writes are *point updates*. While most experiments' default distribution is uniform, we also evaluate skewed workloads with a C++ Zipf generator [41]. Furthermore, the number of clients is defined by the number of worker threads, i.e., 20 clients per node. We execute one operation on a single client until completion, i.e., we do not batch operations nor execute them asynchronously. We used a 30 second warm-up phase to measure the steady-state performance, followed by three experiment phases of 30 seconds each. When not noted otherwise, we report the average system throughput, i.e., the accumulated average performance of every node. Finally, we use a distributed B-Tree, which is implemented on top of ScaleStore for serving all operations.

6.6.1 Scale-Out

We first conduct a scale-out experiment with the different YCSB workloads where we scale from 1 to 5 nodes and keep the data set size, i.e., hot set, constant at 280 GB. Hence, we show the different scenarios ScaleStore is designed for: if only one node is used, the data does not fit in the cache (150 GB). From 2 nodes on, however, the data can be fully cached. Furthermore, we use two access patterns: (1) a partitionable workload where nodes only cover distinct key ranges (and thus only parts of the data need to be cached per node) and (2) a random workload where workers access the full key range.

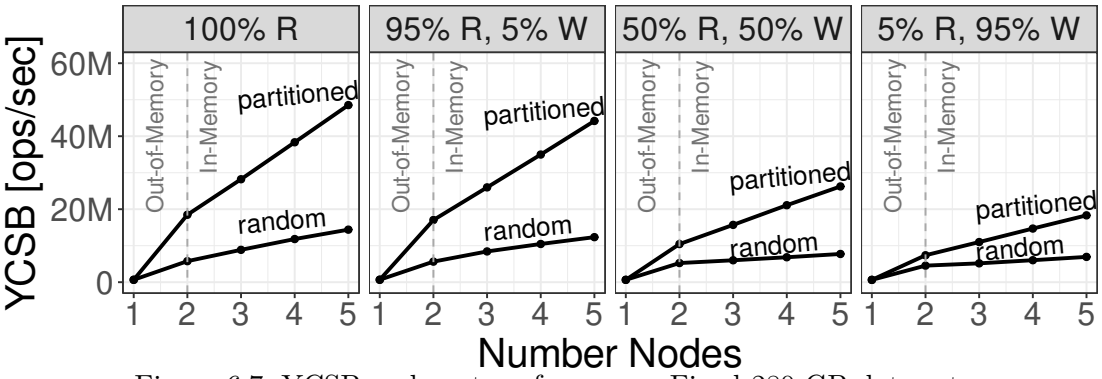


Figure 6.7: YCSB scale-out performance. Fixed 280 GB data set.

This random access pattern is certainly extreme since all nodes request all data. For that access pattern, the data set is too large that it can be fully cached at each node, and therefore, the performance is bound by the network latency.

Figure 6.7 shows the results of the experiment. When the workload is partitionable, ScaleStore achieves its peak performance with 5 nodes of up to $50M$ ops/sec in the read-only workload and $20M$ ops/sec in the update-heavy workload. When the accesses are randomly spread across the cluster ScaleStore still scales and achieves around $15M$ ops/sec for read-only and $8M$ ops/sec for the update heavy workload. An interesting finding is that the performance for the update heavy workload (95% writes) is strictly bound by network latency. This is already the case for the 50% writes workload, which is why both workloads perform similarly. However, the relative performance difference between read-only and more write-heavy workloads can be observed in both access patterns partitioned and random. For random accesses, it is slightly higher due to the average cache utilization, i.e., more cache misses. With 5 nodes, the average cache utilization in the update-heavy workload is just 35.1% compared to 98.7% in the read-only workload, which implies that pages are frequently invalidated.

When looking at the speedup between single-node performance of $800K$ ops/sec and 5 nodes, we can observe an increase between 10 and 60 times. This shows that when the hot set out-grows the memory, the performance can be considerably increased when scaling out to avoid the latency cliff of SSDs.

6.6.2 Data Scalability

Let us now focus on data scalability. We use all 5 nodes and analyze the read-only performance when increasing the YCSB data set size from 75 GB to 7.5 TB. Furthermore, we examine the effect of varying degrees of locality (skew). The results are shown in

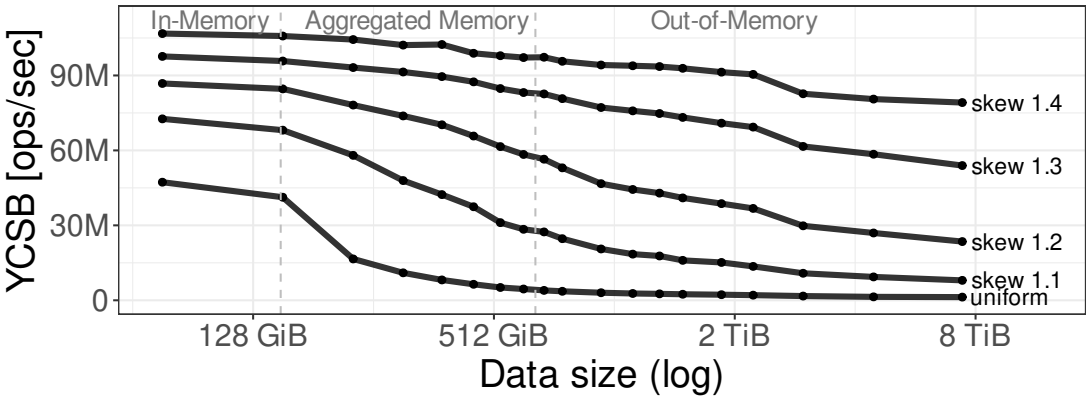


Figure 6.8: Data scalability with varying skew.

Figure 6.8. Overall, the results demonstrate that with ScaleStore we can gracefully bridge the latency cliff when data spills out from the local to aggregated cluster memory and then to SSDs.

We now look more deeply at the uniform (random) access pattern. We can see that with a data size of 150 GB and smaller, the performance is at its peak because the data set can be fully cached on all nodes. When increasing the data size, the performance degrades as expected. Finally, ScaleStore benefits tremendously from the locality in the access pattern as shown in Figure 6.8. This is because the hot path of our protocol can be exploited more frequently, and our epoch-based eviction can reliably find cold pages. We see that even with a low skew of 1.2, ScaleStore achieves around $22M$ ops/sec with a data set size of 7.5 TB. The more locality, the higher the performance resulting in around $85M$ ops/sec with 1.4 skew and 7.5 TB data size.

6.6.3 Distributed In-Memory DBMS Comparison

This section provides a comparative analysis of ScaleStore against other competitors, which are all purely distributed in-memory systems and cannot handle larger-than-memory workloads. One key feature of ScaleStore is that we do not require static partitioning but dynamically adapt on runtime. For this reason, we compare with *NAM-DB* [239]. NAM-DB does not rely on static partitioning either, as it reads remote data with one-sided reads and updates remote data with one-sided writes, respectively. We disabled transactions to compare the protocol overhead with ScaleStore.

Different from ScaleStore, NAM-DB uses shared and exclusive latches, which are (un)latched with RDMA atomics, i.e., one-sided. The second competitor is *FaRM* [55], which in contrast to NAM-DB, uses an optimistic synchronization scheme. Since FaRM

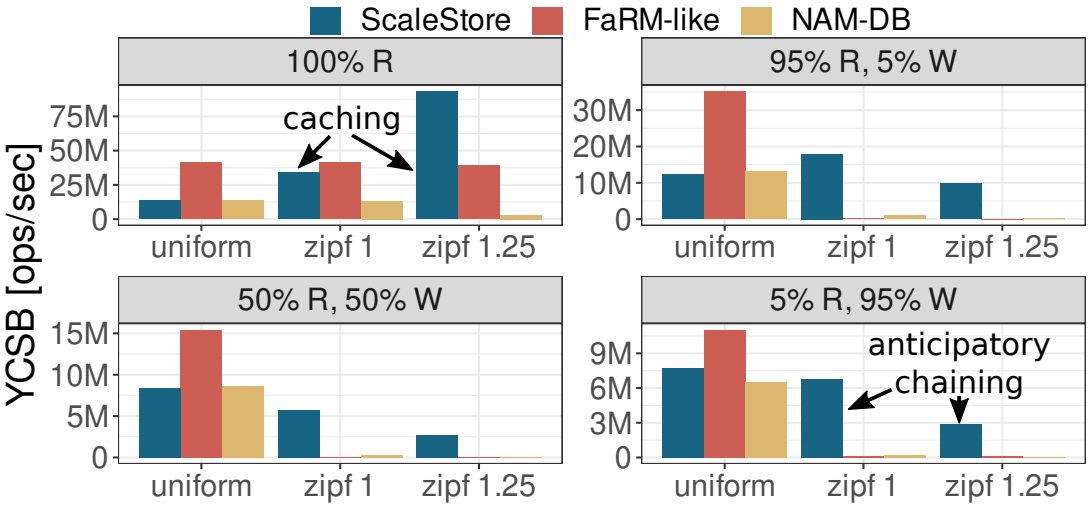


Figure 6.9: Distributed in-memory throughput comparison.

is not publicly available, we re-implemented a FaRM-like approach based on lock-free one-sided reads described in [55]. The lock-free one-sided read retrieves the remote value, and to check if the read was consistent, we validate the versions stored in every CPU cache line. For updates, the lock-free one-sided read is used to copy the record to a thread-local buffer; the record is then modified and sent back to the owner. The owner checks the version(s) of the modified record against the local version(s), if they are equal, the modified record is installed. Finally, for both competitors, we use a pre-allocated array (key-value pairs collocated), i.e., no collisions, which allows them to use a single one-sided read to fetch remote data. This differs from ScaleStore, which uses a distributed B-tree that supports range queries.

Throughput. We use $1B$ YCSB-records for this experiment, distributed across 5 nodes. Figure 6.9 shows the throughput for uniform and skewed accesses. First, we focus on the uniform access pattern. When looking at the read-only workload, one can see that FaRM outperforms NAM-DB and ScaleStore. This is because only a single very efficient one-sided lock-free read is needed to read a remote record. At the same time, NAM-DB requires 3 operations, i.e., two RDMA atomics and one one-sided read, and is thus bound by the latency of those operations. Similar to NAM-DB, ScaleStore is also latency bound. This is because only 50% of the data can be cached in this workload, and thus pages are read from remote nodes. Additionally, hot pages cannot be predicted with a uniform workload reliably.

With the skewed access patterns, the results look different. For the read-only workload, we can observe that with more locality (skew), the performance of ScaleStore increases.

With a light skew of 1, the performance is comparable to FaRM, and with 1.25 we outperform them. In contrast to FaRM, NAM-DB suffers from locality even in the read-only scenario. The reason is that RDMA atomics drastically limit the performance; the higher the skew, the more requests are routed to a subset of nodes that cannot sustain the load. Finally, when looking at the skewed write-heavy workloads, we can see that ScaleStore outperforms NAM-DB and FaRM and provides robust performance for higher skews (contention). In the write-heavy workload (95% writes), the performance of NAM-DB and FaRM drops significantly, whereas the performance of ScaleStore degrades gracefully. While FaRM and NAM-DB compete in a busy polling manner (with exponential back-off), we use anticipatory chaining. It is also important to highlight that the numbers we report for NAM-DB and FaRM represent their best-case scenarios, made possible using a perfectly sized hash table. These bypass any need to traverse index structures or catalog services to locate remote memory positions for one-sided operations. Contrastingly, ScaleStore employs a B-Tree, an inherently more complex structure that would be used under realistic conditions.

In summary, ScaleStore handles high contention scenarios relatively robust and is capable of competing with in-memory systems and outperforming them in the presence of locality in the workload. The full publication in Chapter 13 contains additional experiments demonstrating how ScaleStore responds to workload changes, performs compared to high-performance single-node engines, and more.

6.7 Summary

In this chapter, we introduced ScaleStore, a cache-coherent distributed storage engine optimized for DRAM, NVMe, and RDMA.

The design principles of ScaleStore are based on our previous work:

- the cache-coherent architecture discussed in Chapter 3,
- the page organization and index reflecting the fine-grained designs (c.f. Chapter 4)
- our investigation of one-sided synchronization primitives (c.f. Chapter 5) inspired the adoption of message-handling as an alternative means for accessing remote data.

Our evaluation reveals that by employing our distributed caching and eviction strategies, ScaleStore exhibits several beneficial properties for building modern distributed DBMSs:

- It efficiently scales with large data sets and effectively manages larger-than-memory workloads.
- ScaleStore adapts to workload changes seamlessly
- ScaleStore provides a user-friendly programming abstraction for implementing various distributed data structures managed by ScaleStore, such as hash tables, barriers, or columnar storage.

We think that ScaleStore can serve as the foundation for a distributed DBMS. We argue that our buffer manager employing a cache-coherence protocol provides attractive features. This approach allows for exploiting local DRAM latency by caching frequently accessed pages, using the collective DRAM, and making distributed programming easier. While we admit that the caching protocol inevitably complicates the buffer manager's design, it significantly simplifies the overall system architecture. With this approach, the higher-level components of the database system do not have to deal with additional complexities such as 2PC, which can be a considerable overhead in distributed transaction processing. In addition, distributed indexes a transparently managed and can be evicted at any time. Compared to our RDMA optimized one-sided, which cannot handle larger-than-memory workloads nor can it be cached, the buffer-managed B-Tree in ScaleStore is much simpler in terms of code complexity yet faster. In the subsequent chapter, we will conclude and sketch ideas for future work.

7 Concluding Remarks and Future Work

In this thesis, we have made multiple contributions to building a scalable database storage engine over fast networks for OLTP. In the following, we will reflect on these contributions and delineate potential avenues for future research.

7.1 Reflection

Before drilling down to the components of the storage engine, Chapter 3 studied how the architecture of a database influences the data access in a system. We analyzed existing cloud-based DBMS designs, devised a new taxonomy, and analyzed the architectures based on scalability. A disaggregated architecture with multiple write nodes is particularly suitable for scalable cloud DBMSs. This conclusion was underpinned by the architecture's scalability and qualitative features. The disaggregated architecture's storage utilization and the resulting elasticity from separating compute and storage resources make it a notable contender in the cloud. We called this *archetype* shared-writer. However, we also recognized the increased query latency due to network accesses intrinsic to disaggregation. Consequently, our exploration moved towards low-latency networks and their primitives, particularly in RDMA and EFA. The low latency and network primitives offered by those modern networks played a fundamental role in developing our novel, fast-network-optimized storage engine. The subsequent chapters of this thesis built upon the shared-writer architecture, focusing on the components for efficient data access.

Chapter 4 and Chapter 5 delved into the foundational elements of any storage engine: indexes and synchronization primitives. The exploration of distributed tree-based indexes tailored for RDMA uncovered a spectrum of designs, each with distinct trade-offs, emphasizing the complexity of the design space. By navigating this design space, we showed that the fine-grained index utilizes the bandwidth more efficiently while the coarse-grained offers lower latency. However, achieving high-performance in indexes hinges not only

on the chosen design but also on the implementation details, including synchronization. Remote data structures may need to accommodate thousands of clients connecting from multiple compute servers. The performance of such structures depends on the efficacy of the implemented synchronization scheme. While coarse-grained (two-sided) can utilize existing server-sided synchronization primitives, fine-grained (one-sided) bypasses the remote CPU, rendering existing primitives obsolete. Hence, Chapter 5 focused on one-sided synchronization schemes. We found that one-sided synchronization presented numerous pitfalls. While generally correct, pessimistic approaches can suffer from performance issues, causing an uncontended workload to perform as poorly as a heavily contended one. Conversely, optimistic approaches circumventing these performance issues by avoiding RDMA atomics pose significant implementation challenges. They use one-sided `read` and `write` that are not designed with concurrency in mind and often rely on specific assumptions about the underlying hardware. We showed that some of these assumptions do not hold, resulting in data corruption in some instances.

In Chapter 6, we presented ScaleStore, a cache-coherent distributed storage engine carefully optimized for DRAM, NVMe, and RDMA. The foundational principles underpinning ScaleStore are rooted in our previous findings, including the shared-writer cache-coherent design discussed in Chapter 3, the page organization and indexing inspired by the fine-grained design Chapter 4, and adoption of one-sided message-passing as an alternate approach for accessing remote data, instead of directly accessing remote data with one-sided (see Chapter 5). In addition, we proposed a cache-coherent buffer manager that allows data sharing between nodes.

Our evaluation showed that, by leveraging carefully designed distributed caching and eviction strategies, ScaleStore manifested several advantageous properties for distributed DBMSs:

- Data scalability: ScaleStore demonstrates efficient scalability with large datasets and effectively manages workloads exceeding memory capacity.
- Adaptability: Seamlessly adapting to workload fluctuations, ScaleStore exhibits resilience in dynamic data management scenarios.
- Performance: ScaleStore delivers performance on par with distributed in-memory engines without the limitation of requiring all data to be resident in memory.

To summarize, this thesis made another step toward a distributed OLTP database optimized for modern networks. With these properties, ScaleStore can serve as the

storage engine and, thus, the foundation for a novel distributed DBMS. In addition, it opens more research opportunities that we discuss in the next section.

7.2 Future Research Directions

Our initial focus is directed towards open challenges regarding ScaleStore. Subsequently, we expand our scope to consider the broader implications of current hardware trends.

7.2.1 Reconsidering Eviction & Admission

An important area of future research for buffer-managed systems like ScaleStore is the development of innovative eviction strategies. In particular, the high transaction rate within cloud data center networks may fill caches quickly in a shared-cache system. Thus the system must efficiently discard cold or unused pages and maintain hot pages in the cache. This quick turnover in shared-cache systems is required to prevent performance degradation, necessitating the development of innovative algorithms that deviate from traditional single-node system strategies. Traditional eviction strategies tend to be 'egoistic,' with nodes considering only their private cache and access history. However, we have identified scenarios where this approach does not provide optimal performance. Therefore, alternative strategies considering the broader distributed system may prove more effective. 'Altruistic' eviction strategies have been analyzed in the 1990s and shown promise in this regard [32, 116, 201]. The core concept behind this approach is that nodes cooperate in decision-making about which pages to evict and which to retain in the cluster memory. Unfortunately, the work was theoretical with no concrete implementation, and the hardware changed tremendously in the last decades. Therefore, it is unclear how to implement those approaches efficiently due to the need for global knowledge, like page access frequencies and the number of cached copies per page. Developing robust, efficient, adaptable heuristics or learned components may be viable to overcome this hurdle.

In addition, we intend to explore cache admission strategies. Instead of caching every page, selective admission effectively alleviates the page pressure on eviction mechanisms. Nevertheless, this introduces another challenge: distinguishing between pages worth caching and those that can be omitted. This calls for the design of heuristics or learned components capable of making these decisions.

7.2.2 Concurrency Control

There is a rich literature on locking-based concurrency control from decades ago that are suitable for ScaleStore [99, 154, 176]. For instance, DEC's Rdb/VMS [99] uses a distributed lock manager with techniques to reduce the number of network messages. It uses lock de-escalation, which means that processes will first acquire a lock on a large granule (e.g., a table) to permit operation on pages or records without additional lock requests. Mohan and Narang [154] describe LP-Locking, a technique that avoids this piggy-backed message altogether by granting local lock requests as long as the page is cached on the compute node. This approach is a good fit for ScaleStore's modus operandi as pages are cached as long as there is no conflict (i.e., invalidation). However, their design works at page-level granularity, not record-level, which may impair concurrency in highly parallel systems. We see two possible approaches to address this difficulty: Investigate new techniques such as contention split [3], which splits pages at contention points to achieve higher concurrency despite page-level locking; or use more involved schemes based on record-level locking [155], which require more network messages and may increase latency. An evaluation-based comparison of these approaches on modern hardware is an exciting avenue for future work.

While earlier approaches largely implemented two-phase locking and single-version storage, most contemporary systems have adopted Multiversion Concurrency Control (MVCC). Unlike single-version storage, MVCC updates tuples out-of-place and preserves old versions for visibility to concurrent transactions. Snapshot Isolation (SI) is commonly combined with MVCC, which allows reading transactions to advance without acquiring read locks (as in two-phase locking). This ensures that long-running read queries do not obstruct updates. In the context of ScaleStore, read queries can quickly become long-running due to the inherent latency when multiple pages need to be accessed remotely. Under pessimistic concurrency control strategies, such queries might be aborted or interfere with update queries. Consequently, the properties of SI combined with MVCC are highly desirable in this setting.

However, the application of MVCC in systems like ScaleStore has yet to be explored. Most existing methods have been designed for single-node setups[4, 24, 65, 162] and lean on in-memory data structures, particularly for functions such as garbage collection of older versions or commit logs. Additionally, version chains are often implemented with in-memory linked-lists, which are unsuitable for page-based systems that may need to evict pages, including older versions. In the context of ScaleStore, where all nodes have access to all pages, these data structures would also need to be shared, adding a

new layer of complexity. The overarching question becomes: Can such an approach be implemented scalably and efficiently in a distributed, page-based system like ScaleStore?

7.2.3 Durability & Recovery

Since ScaleStore organizes data in pages, the standard ARIES-style logging and recovery mechanism [152] of disk-based DBMSs is applicable. Typically, this involves a single sequential log for all concurrent transactions. Having all nodes write sequentially to a central log is not scalable. A scalable alternative is decentralized logging, where every node writes its private log file. Since every compute node can potentially touch any page, the changes for a single page can be dispersed across multiple local logs. Consequently, those log files must be merged in the storage layer during recovery or online at run-time in case of failure. Mohan and Narang [155] describe this approach and provide ARIES-style logging for shared-cache systems. Haubenschild et al. [81] show that single-node ARIES performance and scalability can be improved with techniques like Remote Flush Avoidance (RFA). Therefore, existing schemes [155] should be revisited further to improve performance and scalability in a modern shared-cache system.

7.2.4 Evolution of Hardware

The hardware landscape is constantly evolving, and modern database systems must adapt when they want to use the benefits offered by modern hardware. With the end of Moore's Law [50, 157], we have seen the end of an era where dramatic improvements in single-core performance could be anticipated. This marks a significant shift from when software vendors could rely on the steady improvements of processor speeds to enhance system performance naturally. To meet their performance requirements, applications have started employing specialized hardware such as Tensor Processing Unitss (TPUs) and Graphics Processing Unitss (GPUs) a prime example of machine learning. This development underscores a key challenge confronting modern systems: Hardware is becoming more heterogeneous. A key challenge of this trend is deciding which part of a query is executed on which accelerators. Moving forward, we will explore promising avenues that are likely to impact the design of databases.

Specialization in hardware. Devices such as GPUs, TPUs, and Field Programmable Gate Arrays (FPGAs) can potentially provide considerable performance benefits for data processing tasks, given their higher nominal computing power and memory bandwidth. However, such specialized devices demand unique programming models resulting in different algorithms and may require higher maintenance. Furthermore, data movement

presents another obstacle: transferring data over interconnects like PCIe to the specialized device has become a bottleneck in current systems. Consequently, recent research aims to optimize the use of the accelerators [95, 199]. For example, as data is transferred over the network to another node, it can be streamed directly through the GPU on the receiving end. That approach can be used in distributed databases when the data needs to be repartitioned, like in a distributed join [212].

Network infrastructure. The advent of smart NICs and programmable switches carry substantial implications for distributed databases. While these tools can offload network tasks and boost system performance, their use also introduces added complexity. For instance, programmable NICs have stringent real-time requirements, limiting the time to perform operations before data must be forwarded to the wire or the host. Therefore, substantial research is being conducted to explore the efficient utilization of programmable NICs and switches [21, 82, 85, 96] for DBMSs.

Envisioning RDMA's future. The current applicability of RDMA's low-level API to a wide variety of real-world applications remains a persistent challenge. Many distributed algorithms cannot be easily expressed using only RDMA `read` and `write` operations, thereby introducing complexity and error-prone implementations. Consequently, there has been a substantial effort in developing higher-level interfaces, as demonstrated by the solutions proposed in [5, 25, 213, 220]. It is important to note that despite these challenges, RDMA has seen success in large-scale deployments within data centers. A case in point are Microsoft's data centers, where RDMA extensively supports storage services, contributing to over 70% of data center traffic [12]. This underscores that while the challenges associated with RDMA's API are not trivial, they do not hinder wide adoption. However, as RDMA scales, other obstacles come into view, such as congestion control [77, 83, 250]. To counter these, industry experts anticipate an evolution in networking solutions that could help overcome the current constraints. According to a recent industry paper [84], modernized Ethernet-based high-performance solutions will emerge in the next decade. These solutions could potentially replace TCP and RoCE, suggesting a likely consolidation of networking paradigms that could significantly mitigate the complexity of diverse APIs.

Part II

Peer-Reviewed Publications

8 Is Scalable OLTP in the Cloud a Solved Problem?

Abstract

Many distributed cloud OLTP databases have settled on a shared-storage design coupled with a single-writer. This design choice is remarkable since conventional wisdom promotes using a shared-nothing architecture for building scalable systems. In this paper, we revisit the question of what a scalable OLTP design for the cloud should look like by analyzing the data access behavior of different systems. We find that a shared-storage design that supports multiple writers, combined with a coherent cache, has desirable properties for building scalable OLTP systems. For instance, this design provides scalability of writes and skewed reads by caching hot tuples on demand. At the same time, this design does not require user-defined partitioning and two-phase commit in contrast to a shared-nothing design. We present the lessons learned from building a multiple-writer cache-coherent cloud OLTP DBMS to show this architecture's merits and present open research challenges.

Bibliographic Information

The content of this chapter was previously published in the peer-reviewed work Tobias Ziegler, Philip A. Bernstein, Viktor Leis, and Carsten Binnig. "Is Scalable OLTP in the Cloud a Solved Problem?" In: *13th Annual Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023, Online Proceedings*. www.cidrdb.org, 2023. URL: https://www.cidrdb.org/cidr2023/papers/p50-ziegler.pdf. The contributions of the author of this dissertation are summarized in Section 3.1.

8.1 Introduction

The Cloud is Taking Over. The DBMS market has shifted significantly from on-premise to the cloud in the last few years. According to a recent market report[1], in 2021 DBMS revenue in the cloud was on par with the on-premise market. Given current growth rates, the cloud DBMS market will be substantially larger by CIDR 2023. Consequently, classical DBMS vendors such as IBM and Oracle have been or are about to be overtaken by hyperscalers such as AWS, Microsoft, and Google, which have heavily invested in their cloud-native DBMSs.

Cloud-Native OLTP. Looking at cloud-native OLTP DBMSs, we see an interesting trend: Many commercially-available systems such as Amazon Aurora [217] and Microsoft Socrates [6], use a disaggregated design (shared-storage) coupled with the primary-secondary paradigm. In this design, all write transactions go to the primary node, which sends its write-ahead log to the shared storage so that secondary nodes can access it. The storage nodes use the log to reconstruct the data pages in the background. Those reconstructed pages can be read by secondary nodes on demand and thus be spawned at any time with low overhead. While this design provides important features such as hot fail-over as well as elasticity and load-balancing for read-dominated workloads, it is limited by the capacity of the primary node – particularly for write-intensive workloads, which are common in OLTP [53].

Decades of OLTP Research. The design choice of using a shared-storage architecture with a single writer node is remarkable since decades of research proposed different designs for building scalable distributed systems [105, 112, 148, 168]. For example, many early OLTP systems promoted the use of the shared-nothing architecture for scaling OLTP beyond the capacity of a single machine [105, 153]. An alternative is a shared-storage architecture with multiple read-write nodes, which has been used successfully since the 1980s. It is implemented in on-premise systems such as IBM DB2 Data Sharing [100], DEC's Rdb/VMS for VAXcluster [135], and Oracle RAC [32].

Research Question. Why have modern commercial cloud DBMSs not adopted these scalable designs? We believe there are several reasons: First, despite the substantial body of research, the rise of the cloud has disrupted our understanding of the landscape and tradeoffs of different DBMS designs. For example, it was often thought that the shared-nothing architecture is preferable for building distributed DBMSs. However, this is no longer so clear with cloud DBMSs that use shared storage. Second, even though promising shared-storage designs have been proposed and used in on-premise DBMSs,

[1] https://blogs.gartner.com/merv-adrian/2022/04/16/dbms-market-transformation-2021-the-big-picture/

many of their core aspects crucial for a truly scalable cloud OLTP DBMS have yet to be fully explored. Third, some of the techniques for scalable OLTP are quite complex and may be risky to deploy at cloud scale. Simpler and more robust techniques are needed.

Table 8.1: Data Access Archetypes for Analyzing the Asymptotic Scalability of Cloud OLTP DBMSs.

		Single-Writer	**Partitioned-Writer**	**Shared-Writer**	
				No Coherent Cache	*Coherent Cache*
	Systems based on Archetypes	AWS Aurora [217], Azure SQL HyperScale [6], PolarDB [127], AlloyDB [68]	System R* [153], Spanner [45], YugabyteDB [90], H-Store [105] CockroachDB [206]	NAM-DB [239], Sherman [223]	ScaleStore [252], Oracle RAC [164], IBM DB2 Data Sharing [100], GAM [27], DEC VAXcluster [135]
Qualit. Features	Data Access Path	Single-writer, multiple readers	One writer per partition	Writers update (remote) shared storage	Writers update cache-local data governed by coherence protocol
	System Complexity	Low	Medium	Medium	High
	Elasticity†	Only Reads	Limited	Yes	Yes
Asymptotic Scalability	Uniform Reads	Yes††	Yes	Yes	Yes
	Uniform Writes	No	Yes	Yes	Yes
	Skewed Reads	Yes††	Yes††	Yes††	Yes
	Skewed Writes	No	No	No	No

(†) Elasticity w.r.t. compute and storage separately (††) Only with the number of secondaries, i.e., replicas.

Mapping the OLTP Landscape. In this paper, we explore the design space of distributed OLTP DBMSs to help system developers to understand their fundamental differences for the cloud. More specifically, we analyze the data access path of distributed OLTP systems to understand their scalability properties. We observe that the old taxonomy of shared-storage and shared-nothing does not fully capture the scalability properties of modern cloud DBMSs. For instance, it was usually assumed that a shared-storage system allows many machines to perform updates. However, many cloud DBMSs today only allow a single read-write node, which limits the system's scalability. Therefore, as the first contribution in this paper, we distill the data access behavior of distributed OLTP DBMS designs that have been proposed over several decades. We organize the design space into three data access archetypes with different features and scalability behaviors.

Towards a Scalable Cloud OLTP DBMS. As a main result, by analyzing the different OLTP archetypes, we show theoretically and experimentally that a shared-storage design supporting multiple writers, combined with a coherent cache, has desirable properties for building scalable OLTP systems. We advocate that future cloud DBMSs be based on this archetype. However, many challenges exist, and building blocks still need to be added to enable a full cloud DBMS based on a shared-caching design. As a second contribution, we discuss lessons learned from building such a cloud-native system and present research opportunities for the community.

8.2 OLTP Data Access Archetypes

Four decades of OLTP research have led to a plethora of distributed OLTP DBMS designs. In 1985, Stonebraker provided a taxonomy for distributed DBMSs with the famous categorization into shared-nothing and shared-storage architectures that can be found in many DBMS textbooks today [202]. In a shared-nothing architecture, each node has private storage that is not accessible to other nodes. In a shared-storage architecture, all nodes have access to a single stored copy of the database.

Revisiting the Old Taxonomy. Although these designs are still relevant, changes in the system landscape make it worth revisiting the categorization. The reason is that many of today's systems are hard to characterize with the traditional taxonomy. For instance, many cloud-based shared-storage systems, such as Aurora single master and Azure SQL Hyperscale, only permit one update node to avoid the problem of coordinating multiple writers for buffer cache coherence. By contrast, the classical taxonomy assumed shared-storage systems allow many update nodes.

Conversely, in the classical taxonomy, it is assumed that a shared-nothing system allows only one node to update a given data item. However, many shared-nothing database systems today allow multiple nodes to update node-private replicas of the same data item. This is commonly called *multi-master*, which leads to entirely different scalability properties. Another shared-nothing design allows multiple nodes to accept update requests on a given data item and forward them to the one-and-only node that is allowed to process updates on that data item. We call this *pseudo multi-master*. The conventional taxonomy does not highlight such differences, which are important for differentiating the scalability properties of systems.

A New Taxonomy for OLTP DBMSs. Therefore, we propose a taxonomy that focuses on the data access path, which is another useful basis for capturing the scalability properties of systems. Our taxonomy allows us to analyze *asymptotic scalability*, that is, how well a system scales w.r.t the number of nodes for basic data access operations when increasing the load in the system. As shown in Table 8.1, we look at fundamental operations for OLTP, namely key-based reads and writes for uniform and skewed key distributions.

The old taxonomy focused heavily on transaction synchronization problems that arise in each design, e.g., whether it requires two-phase commit for transaction atomicity or global lock management to synchronize buffer caches. In our taxonomy, we instead mainly focus on the storage layer, i.e., the data access path, since it is fundamental for any OLTP DBMS to execute reads/writes scalably. In our discussions, we omit logging, recovery,

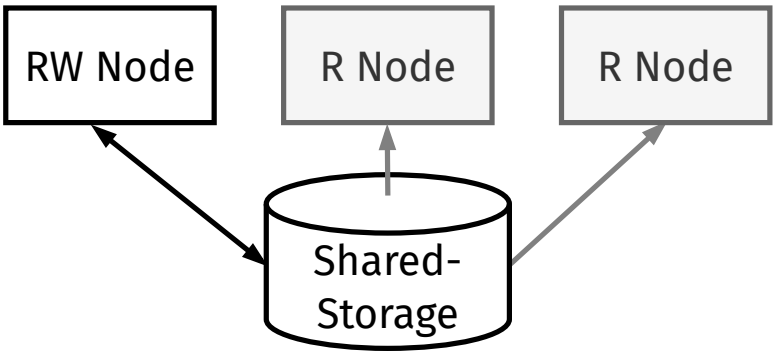


Figure 8.1: Archetype Single-Writer

and concurrency control [13, 143, 238] that are thoroughly investigated by Harding et al. [79], and Bernstein and Goodman [19]. We also omit deterministic databases [61, 139, 211]. Although they have some desirable properties, major cloud service providers do not offer them today.

We map the design space into the three data access archetypes shown in Table 8.1. An archetype can be considered the "phenotype of a DBMS" that determines its observable asymptotic scalability independent of its implementation details. Sections 8.2.1 - 8.2.3 describe the data access archetypes and their asymptotic scalability in more detail. Section 8.2.4 shows how our taxonomy is more descriptive for modern cloud DBMSs and highlights where the previous classification did not convey the data access properties. Section 8.2.5 discusses the effect of storage latency on the behavior of different archetypes.

8.2.1 Archetype: Single-Writer

In the Single-Writer archetype, a single read-write node (RW-node) processes update transactions, and multiple read-only nodes (RO-nodes) can handle read-only transactions. We depict the basic architecture of systems that implement this archetype in Figure 8.1. Many modern cloud DBMSs, such as AWS Aurora (single-master), Azure SQL Hyperscale, PolarDB, and AlloyDB use shared-storage. They support multiple nodes accessing a single stored database copy, but only one dedicated RW-node can execute updates. We classify such systems as Single-Writer systems.

While Figure 8.1 shows a Single-Writer with shared-storage, many DBMSs with private storage also fall into this category. For example, a MySQL instance in which the primary node executes reads and writes and replicates writes to read-only replicas is also categorized as Single-Writer even though the primary and replicas have node-private

storage. The strictly separated data access permission is the distinguishing feature, not whether the storage is shared or private.

Asymptotic Scalability. The asymptotic scalability of this archetype can be summarized as follows: Writes in this archetype are limited by the capacity of the single RW-node. By contrast, read throughput can scale well by spawning more RO-nodes.

Still, the concrete performance characteristics of systems can vary due to other design decisions. For instance, cloud DBMSs with a replicated shared-storage can spawn RO-nodes very fast and react to an increase in read-only transactions. On the other hand, a mirrored instance may take longer to deploy an additional RO-node, since it requires creating another copy of the database. Furthermore, some Single-Writer systems can attain high throughput on large multi-core processors. However, their asymptotic scalability remains bounded by their limitation of a single RW-node.

Qualitative Features. This design provides important features for the cloud, such as fast failover and elasticity for read-dominated workloads. Similarly, the separation of compute and storage allows cloud vendors to handle growing data sets. At the same time, the systems' complexity is often lower since they only need to support a single-writer.

8.2.2 Archetype: Partitioned-Writer

In the Partitioned-Writer archetype, the database is split into partitions, each of which can be updated by only one RW-node. As shown in Figure 8.2, this archetype exploits data partitioning to spread the data across several nodes that are responsible for executing read-write operations on their local partition.

Like the Single-Writer design, the Partitioned-Writer archetype is a popular one for commercial cloud DBMSs. Examples include CockroachDB [115], AWS DynamoDB [188], Azure CosmosDB [145], and Spanner [45]. Most of these systems implement a coordination protocol, such as two-phase commit, to ensure atomicity of cross-partition transactions.

In our taxonomy, the data access path is the distinguishing feature, not whether the system uses instance local storage or (partitioned) disaggregated storage. The latter is commonly used in the cloud since a single partition can be dispersed to multiple storage nodes. In contrast, instance local storage typically offers lower latency albeit with a fixed storage capacity (see Section 8.2.5). Regardless of these implementation details, Partitioned-Writer systems provide the same asymptotic scalability as discussed next.

Asymptotic Scalability. In contrast to the Single-Writer design, DBMSs that implement the Partitioned-Writer archetype can handle uniform reads and writes in a scalable manner since they can process each partition's workload independently. However, skewed

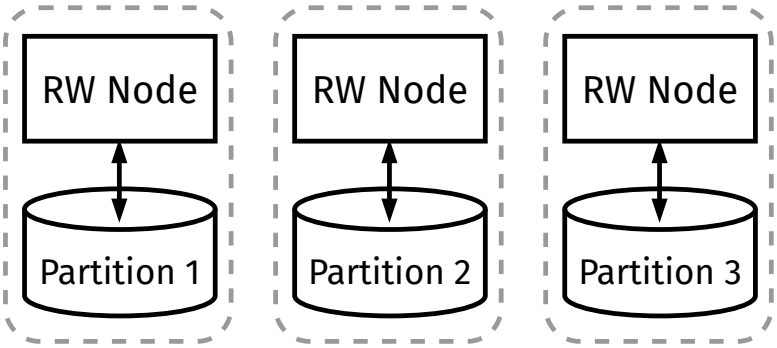


Figure 8.2: Archetype Partitioned-Writer

workloads can be challenging, since requests for hot keys are always processed by the same RW-node.

One technique to handle read-skew is to couple this archetype with additional RO-nodes (e.g., per partition) similar to Single-Writer systems. However, this needs more resources for database copies and requires propagating updates to RO-nodes to keep them in sync. Since adding RO-nodes is an orthogonal mechanism that can be used for any archetype, in the rest of the paper, we discuss each archetype in its *pure* form. This way, we can identify archetypes that provide scalability under read-skew without needing additional resources.

Qualitative Features. Classical partitioned systems offered limited elasticity since adding new nodes typically called for a full repartitioning of the database. Modern cloud DBMSs have added several optimizations to address this issue, e.g., consistent hashing or fine-grained range partitioning. For example, CockroachDB uses a fine-grained range partitioning of data chunks, which avoids a complete repartitioning and improves elasticity. However, fine-grained partitioning can add complexity and overhead to locating data. To address this, CockroachDB stores data in a monolithic sorted map of key-value pairs and uses it to route queries to the responsible RW-node of a partition. Overall, the elasticity of a Partitioned-Writer system depends on the workload characteristics and the system's ability to repartition the data efficiently.

Although this archetype sounds very similar to the shared-nothing architecture, many shared-nothing DBMSs can not be classified as Partitioned-Writer, and thus, their asymptotic scalability is different. For instance, some shared-nothing systems are multi-master. While they still use partitioned storage the data is replicated and can be modified by multiple RW-nodes leading to a different scalability behavior. The next archetype Shared-Writer captures this data access pattern.

8.2.3 Archetype: Shared-Writer

The Shared-Writer archetype allows multiple RW-nodes to modify data items – typically by utilizing shared-storage. Several recent research OLTP systems, such as NAM-DB [239], and a few established systems, such as Oracle RAC, can be classified as Shared-Writer systems. Because data must be kept consistent across writers, these systems must address the buffer-cache coherence problem [155, 177]. There are two ways of doing this: nodes always write and read from the shared storage to get the ground truth (*No Coherent Cache*), or the nodes participate in a cache coherence protocol (*Coherent Cache*). In the following, we will drill into these two cases.

8.2.3.1 Shared Writer With No Coherent Caches

First, we consider Shared-Writer systems without coherent caches. As shown in Figure 8.3, these systems support multiple RW-nodes that write to and read from the storage layer.
Asymptotic Scalability. In contrast to the Partitioned-Writer archetype in which only one RW-node writes to a partition, the Shared-Writer design allows every compute node to be an RW-node and to access every data item. Nevertheless, the Shared-Writer archetype has the same asymptotic overall scalability. For example, under access skew, the storage layer might receive requests from multiple RW-nodes and become a bottleneck, limiting the scalability of both writes and reads.
Qualitative Features. In the Partitioned-Writer design, compute and storage resources are usually scaled together. This is inflexible because the need to store more data or do more computation may change at different rates. Using a Shared-Writer design, we can scale compute and storage resources independently depending on the demand. This enables better data scalability and elasticity since any compute node can access any data item which is a desired property in the cloud.

However, one major challenge in this design is locating the data items in the storage layer. Like modern Partitioned-Writer DBMSs, modern disaggregated OLTP DBMSs use an index to locate items. The index resides remotely in storage and is accessed by compute nodes to find relevant data items. This additional requirement for indexes increases system complexity as shown in Table 8.1.

8.2.3.2 Shared-Writer With Coherent Caches

The main downside of the previous sub-archetype (No Coherent Cache) is that every data access involves a network round trip to the storage layer, even for repeated access. Caching can considerably reduce that latency by keeping recently accessed data items

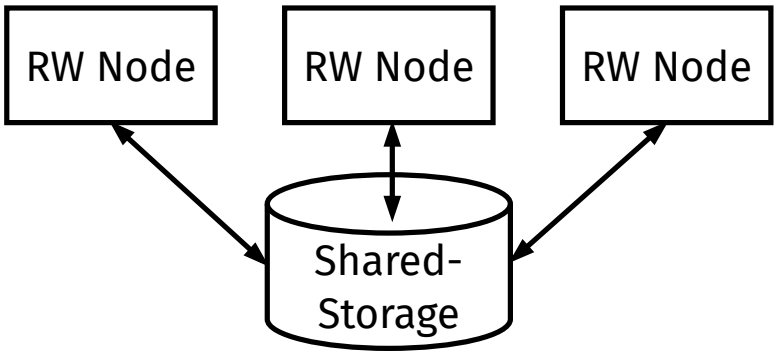


Figure 8.3: Archetype Shared-Writer without Coherent Cache

in the compute nodes' memory. In this archetype, we specifically consider coherent caches. These caches support shared writers and allow caching read-only data locally while keeping the data coherent (i.e., data does not become stale). Shared-Writer with coherent caches is similar to Single-Writer and Partitioned-Writer with synchronous or eager asynchronous replication since the replicas are essentially caches for reads. The difference is that the caches in this archetype are updatable.

Figure 8.4 shows that this archetype is, at first glance, similar to the previous Shared-Writer design. However, the coherent caches impact the asymptotic scalability as discussed in the following.

Asymptotic Scalability. This archetype has the best asymptotic scalability of all archetypes. The benefit of coherent caches is that they allow replicating frequently accessed (i.e., hot) data items on demand across multiple nodes. This workload-driven approach differs from the classical replication used in the previous archetypes. Using the coherent cache, systems of this archetype can handle read skew efficiently, as shown in Table 8.1.

Unfortunately, skewed writes can also not be processed in a scalable manner. This is no different than the other archetypes since each update must be exclusively handled by a single node to keep the data consistent. However, the cache coherence protocol may lead to the write privilege bouncing between compute nodes since all compute nodes can modify all data items. Solving this requires complex synchronization, a challenging problem.

Qualitative Features. While coherence protocols are non-trivial to implement efficiently, they provide many benefits. Compared to operating directly on the shared-storage, shared-caching makes accessing and maintaining remote indexes easier since index nodes are cached on demand. For instance, when an index node is updated, e.g., a B-Tree leaf,

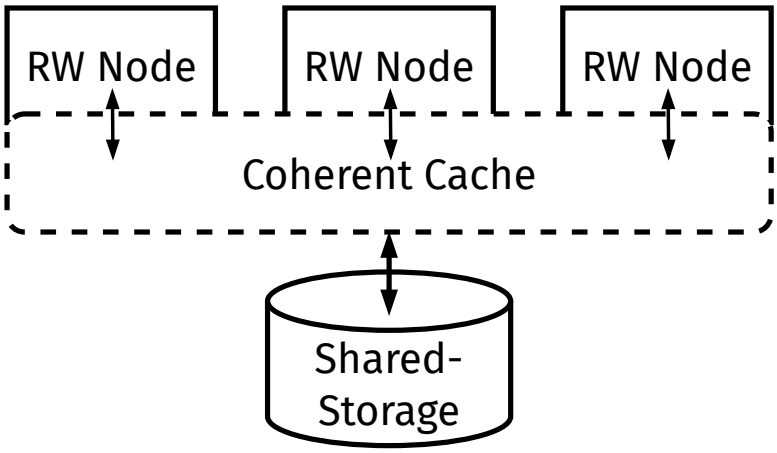


Figure 8.4: Archetype Shared-Writer with Coherent Cache

the cache-coherence protocol invalidates other copies of the index node and delivers the newest data version when the leaf is re-accessed.

Furthermore, the caching-based design provides good elasticity. Admittedly, this is more complicated than in the previous archetype without a cache since the coherence protocol must be designed accordingly (see Section 8.3.3). A new compute node can be added at any time, incrementally filling its cache as data items are accessed. In addition, these systems can handle easy- and hard-to-partition workloads. They do not require user-defined partitioning but still profit from partitionable workloads. Moreover, they have the flexibility to react to changes in workload distribution.

The lesson from our analysis is that a caching-based design is very promising for cloud DBMSs. Compared to the other archetypes, it has the best asymptotic scalability and provides data scalability and elasticity.

8.2.4 Categorizing Systems With Archetypes

In the following, we discuss how our new taxonomy can be used for categorizing the scalability properties of existing systems.

New Taxonomy Focuses on Data Access. As mentioned before, we see the archetypes introduced in our taxonomy as themes that better help developers understand the scalability properties of DBMSs. This is in contrast to the old taxonomy, which often falls short in classifying systems. A good example where the old taxonomy fails to express the system's scalability is FaRM [55] from Microsoft.

With the old taxonomy, this system cannot be easily classified as a shared-nothing or a shared-storage design rendering the discussion of the scalability properties difficult.

FaRM provides a shared-storage abstraction in which workers can read data from every other node. Thus, for reads FaRM behaves as a shared-storage system. However, updates are shipped to the partition owner as in a shared-nothing. So it is unclear if the scalability characteristics are inherited from shared-storage or shared-nothing.

Instead, we can categorize FaRM as a Partitioned Writer using our taxonomy. That is, while every node can read from every other node, the writes are performed by the RW-nodes, which own the partition. Therefore, the system's asymptotic scalability is inherited from the Partitioned Writer design.

Multiple Archetypes for One System. Another interesting DBMS is Aurora multi-master [60, 123, 189], in which all DB nodes can be writers. This contrasts with Aurora's single-master, in which only one DB node can act as a writer. While Aurora single-master fits our Archetype of a single-writer and has the above-discussed characteristics, Aurora multi-master is a Shared-Writer system since it allows multiple compute nodes as writers.

Aurora multi-master allows each RW-node to read and write the whole database. It uses optimistic concurrency control (OCC) to detect conflicts with operations from other writers. The underlying storage system is very similar to Aurora single-master [217], except that it performs the OCC test. A writer sends each update to all storage replicas with a copy of the data it wrote. A storage replica accepts the first write it receives for the latest page version and rejects ones that arrive later. The write succeeds if a quorum of storage servers accepts it. Otherwise, the write fails, and the transaction that issued the write aborts.

Writers send their committed updates to other writers, which the other writers use to update their cache. This is how Aurora maintains a weak form of cache coherence. It is weak in the sense that stale versions of an updated page are not immediately invalidated in all caches. It is similar to update propagation on replicas in a single-writer system, but for a different purpose. Here, update propagation increases the chance that future writes by a node operate on fresh data and, hence, will be accepted by the OCC check.

Caching in Single-Writer Systems. A third interesting system is PolarDB Serverless [29]. While traditional PolarDB is very similar to AWS Aurora due to using log servers and log replay on the RO-nodes, PolarDB Serverless uses a cache-coherence protocol to update the RO-nodes. Yet, the fact that they use a cache coherence protocol does not turn PolarDB Severless into a shared-writer system since the data access path only allows a single RW-node.

Optimizations for Partitioned-Writer. A further relevant research direction is shared-nothing databases that dynamically repartition data. Several papers repartition the shards online when (severe) workload imbalances are detected [59, 108, 130, 170,

205, 232]. This process balances the load at runtime while minimizing the impact on running transactions. This allows Partitioned-Writer databases to behave similarly to the shared-writer archetype with a coherent cache in that they can scale better than Single-Writer under skew. However, the fundamental difference is that only one RW-node can process operations on a given partition; thus, adapting to changing workloads requires expensive repartitioning. As such, very skewed workloads can still overload a partition and limit scalability.

8.2.5 The Importance of Latency

So far, our discussion of scalability has focused exclusively on throughput. However, latency is also an important factor for OLTP performance since it influences lock holding times and other factors that can affect the throughput of an OLTP DBMS. In the following, we first discuss factors that influence latency and ways it affects throughput. We then discuss considerations on how to reduce it.

The Importance of Latency to Throughput. Latency in an OLTP DBMS depends very much on workload characteristics. Transactions that need to navigate the relationships between tuples of different tables can require many storage accesses. Tuples have complex and sometimes subtle relationship graphs in the form of joins, foreign keys, being modified by the same transaction, being part of the same group by clause, matching the same filters, etc. Evaluating those relationships and maintaining their integrity typically requires many separate sequential memory accesses, which drives up response and lock holding time, which reduces throughput.

Transaction latency could be seen as independent of throughput scalability. But in practice higher transaction latency is quite problematic at scale. For example, many distributed relational DBMSs adopt the protocol of an existing relational DBMS, such as PostgreSQL, to support existing libraries and tools. These protocols are typically synchronous and interactive, which effectively bounds achievable throughput to the number of sessions divided by average response time. Increasing the number of sessions to achieve higher throughput is not always practical and generally has adverse effects such as increasing memory pressure and resource contention. Higher response time also leads to longer lock holding time, which further limits achievable concurrency and throughput. Finally, tools and libraries built for a single-node relational DBMS rarely anticipate high response times. For instance, web development frameworks routinely do a dozen sequential queries for a single page load because they expect queries to return in less than a millisecond. In the cloud, that can take much longer. Custom libraries and

protocols could help achieve scalability in the presence of high response times, but that limits applicability. Thus, even if throughput is a more important user requirement than latency, optimizing latency is essential to reach the throughput goal.

Cutting Latency by Push-down. Storage latency contributes to transaction latency and hence can limit throughput scalability. Storage latency in the cloud is relatively high, since there is rarely enough capacity to have all servers in close proximity, e.g., in the same rack. Moreover, close proximity might be undesirable since it degrades availability due to correlated failures. One way to reduce storage latency is by using node-local storage instead of disaggregated storage. However, this is typically not desirable in the cloud since this prevents not only independent scalability of the storage but also hinders other properties, such as elasticity. A technique that helps to reduce latency in systems that use a disaggregated (shared) storage design is to push down the access to storage servers. While this is harder to do in a shared-storage DBMS that uses a simple file system, it is conceivable with a shared-storage DBMS with a custom-built storage system, such as Aurora and SQL Server Hyperscale. However, computation push-down risks consuming too much of the storage system's limited compute capacity, thereby increasing the latency of simple storage accesses. Exploring query and update push-down to the storage layer of OLTP systems is thus potentially an interesting research opportunity.

Cutting Latency by Caching. Another direction in a shared-storage system that can mitigate the impact of storage latency is keeping the working set in a node-local cache which is possible for all archetypes as discussed before. This cache can avoid distributed processing of transactions and thus remote data accesses. Moreover, caching can be used by both RW-nodes and RO-nodes. However, caching designs come with their own challenges. For example, to ensure that read-only replicas remain fresh, they need to apply the log of committed updates to their cached data. Hence, cutting down the latency of log replay is one interesting challenge. Another research opportunity in disaggregated (shared-storage) designs is to find efficient ways to mitigate storage latency when the working set does not fit in the cache.

Latency and Throughput are Important. In summary, our analysis of the throughput scalability of archetypes is only part of the story. When latency is taken into account, the tradeoffs can be even more complex. This is also reflected in the discussions of the following section.

8.3 Towards a Scalable Cloud OLTP DBMS

Section 8.2 showed that a design with shared-writers and coherent caches, in the following called shared-cache systems, provide the best asymptotic scalability and are a solid foundation for modern cloud OLTP DBMSs. This section delves deeper into that design.

There was a large body of research on shared-cache DBMSs in the early 1990's [99, 116, 154, 155, 201]. Several commercial OLTP DBMSs of that era used that design, such as IBM DB2 Data Sharing [100], DEC's Rdb/VMS for VAXcluster [135], and Oracle RAC [32]. However, these approaches were not designed for the cloud. Some systems relied on proprietary hardware, such as the coupling facility of IBM, that limit multi-cloud support since the custom interconnection hardware is unlikely to be supported in other vendors' clouds.

Meanwhile hardware has been evolving. Today, we rely on data center networks and large multicore servers. Customizable processors, such as FPGAs and ASICs, are becoming commonplace. To minimize cost, multitenancy is usually needed. And although Oracle is actively working on optimizing Oracle RAC for the cloud[2], the research community has not been focusing on shared-cache DBMSs for some years. Given these trends, we believe it is time to revisit, adapt, and improve upon existing research to design a cloud-native DBMS. We are not the only researchers with this sentiment. Mohan pointed out in his keynote that much of the work done in this area can now be reused in cloud DBMSs [151].

In the past year, some of us have started to build a new cloud shared-caching OLTP DBMS called ScaleStore[3][252]. In the following, we first discuss ScaleStore's design and then present research opportunities.

8.3.1 A Blueprint for a Shared-Caching DBMS

A key feature of shared-caching systems is that they do not rely on statically partitioned data. Instead, compute nodes cache data dynamically based on the access patterns of a workload. For instance, consider the index shown in Figure 8.5(i), which consists of pages P1-P5. We can see that the three compute nodes can cache different index pages depending on their workload.

Invalidation-based Coherence Protocol. Because an index page can be cached on multiple nodes (e.g., the root of the index P1), a coherence protocol is necessary. Thus,

[2] https://www.oracle.com/technetwork/database/options/clustering/overview/new-generation-oracle-rac-5975370.pdf

[3] https://github.com/DataManagementLab/ScaleStore

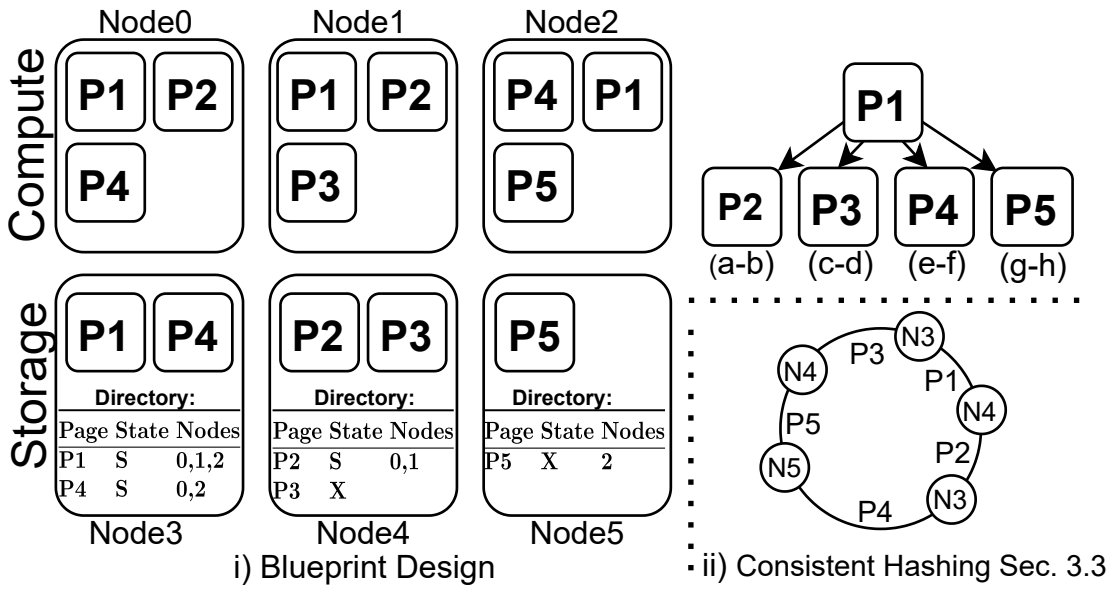


Figure 8.5: Shared-Cache Design of ScaleStore

modifying a page in one node makes all copies of that page in other nodes obsolete. The coherence protocol has to ensure that these changes are detected so that nodes can access the most recent page. ScaleStore uses a *directory-based invalidation-based protocol* which provides sequential consistency at page-granularity. Thus, whenever a node plans to modify a page, other nodes must invalidate the page. However, instead of broadcasting the modification intent to all nodes, we use directories to track nodes that currently cache the page. Consequently, we send the invalidations only to the required nodes. Note that recently proposed memory coherence protocols [237, 245] are attractive alternatives to a directory coherence protocol and need to be evaluated in a distributed DBMS.

Directory per Storage Node. Every storage node is a directory for some of the pages as shown in Figure 8.5(i). Storage nodes keep track of those pages that are stored on them. For instance, Node 3 is the directory of P1 and P4. The directories are mainly responsible for managing the metadata for the invalidation-based cache coherence protocol. In Figure 8.5(i) this metadata is depicted in the tables on the storage nodes and reflects the state of the compute caches. For instance, P5 is held in exclusive-mode by Node 2 whereas P1 is cached on all compute nodes in shared-mode. Since tracking on a per-tuple granularity would be prohibitively expensive, we organize multiple tuples in fixed-size pages to amortize the bookkeeping overhead. These pages (e.g., 4 KB) are kept coherent across all nodes.

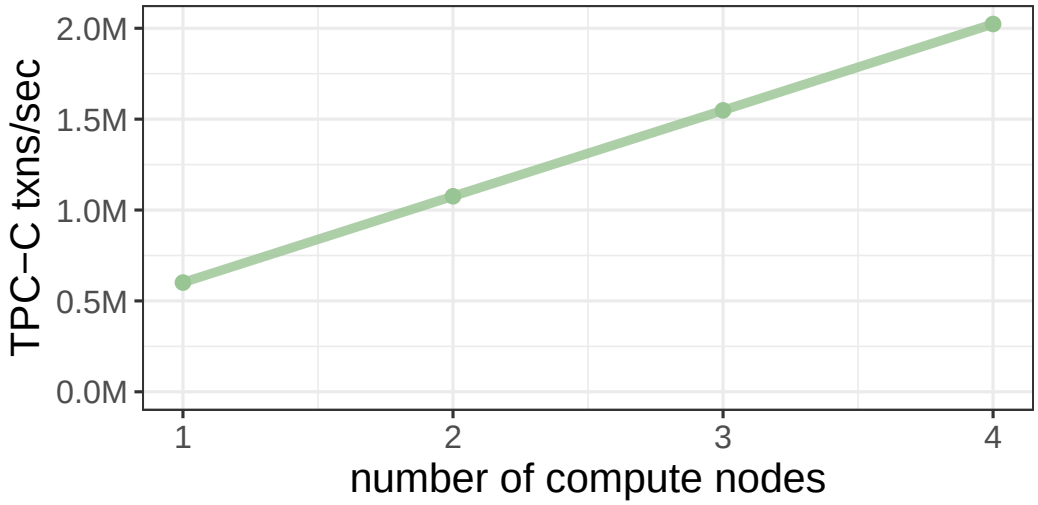


Figure 8.6: TPC-C Scale-Out Performance of ScaleStore with 4 Compute and Storage Nodes all Connected with RDMA. We use 20 Warehouses per Compute Node. Every Compute Node uses 20 Threads.

Supporting Arbitrary OLTP Workloads. For cloud DBMSs one important requirement is that they can support arbitrary (and potentially changing) workloads in a scalable manner. The page-based organization of data is a perfect fit for supporting primary and secondary indexes to access data efficiently. Those indexes allow the user to perform lookups, updates, and short-range scans very efficiently. Furthermore, the beauty of a page-based index organization is that every node of a distributed B-Tree is automatically backed by the coherence protocol. Thus, all participating compute nodes that currently cache a page observe changes to this page. In addition, the coherence protocol allows compute resources to dynamically cache the hot parts of the index (such as the inner B-Tree nodes). As a result, if the workload changes, the cache will adapt automatically.

Proof of Concept. With the current design ScaleStore can already execute arbitrary OLTP workloads. We implemented a full-fledged TPC-C benchmark using our B-Trees as primary and secondary indexes. All experiments are run in read uncommitted mode, since ScaleStore does not yet implement full transaction semantics. The invalidation-based coherence protocol as described above only provides sequential consistency on a page-level. Figure 8.6 shows the results for a setup with 4 compute and 4 storage nodes connected with RDMA. We use 20 warehouses and threads per compute node. As we can observe, the throughput of ScaleStore scales with the number of nodes for the write-heavy TPC-C benchmark. This underlines the asymptotic scalability of the storage engine as discussed in Section 8.2.

8.3.2 Caching and Eviction Go Hand in Hand

Modern cloud data center networks are fast – which means that new pages may be added to the caches of compute nodes at very high rates. Thus, shared-cache systems need to evict cold, i.e., unused, pages quickly while making sure that hot pages remain in the cache. Otherwise, the performance may be impaired significantly. Although many good page replacement algorithms are known for single-node systems, optimal performance in a distributed setting of a shared caching system requires different algorithms.

Why is this hard?. Consider an example in which a working set of 150GB is processed on 4 nodes with a capacity of 50GB each (total aggregated 200GB). For the sake of simplicity, assume that we perform uniform point lookups. Clearly, the working set should fit in the aggregated cache of the compute nodes. Consequently, no eviction to the secondary storage (i.e., storage nodes) should be triggered – in theory.

However, when every node only uses a local eviction strategy, latencies will increase, which affects the performance as illustrated in Figure 8.7. The main reason for this non-obvious behavior lies in the egoistic caching strategy of each node. For instance, if page A is cached on all 4 nodes, four slots are used instead of just one. In the extreme, if every page was cached 4 times, we could only effectively store 50 GB. This means that the nodes cannot cache all the data and thus pages may be evicted to secondary storage instead of simply dropping the cached copies. Note that the nodes do not know which copies are cached on other nodes as they only have their local view. The next time a node reads such a page from secondary storage the latency is much higher than expected. After all, it would be much faster to read from the remote caches than from secondary storage.

Research Opportunity: Altruistic Eviction. A theoretical solution was proposed 30 years ago [116, 201]. However, we are not aware of any system that implements these ideas. Even commercial systems such as Oracle RAC use a sub-optimal local strategy [32]. The main roadblock is that the proposed altruistic approaches rely on global knowledge, such as page access frequencies and the number of cached copies of each page. However, global knowledge is too expensive to maintain in high-performance distributed systems. Hence, designing a robust and adaptive heuristic could be an interesting research opportunity.

8.3.3 Elasticity

Pay-as-you-go pricing is the de-facto standard in the cloud. Therefore, support for compute and storage elasticity is a major competitive and operational advantage. Scaling the compute layer comes naturally in a shared-caching system since new compute nodes

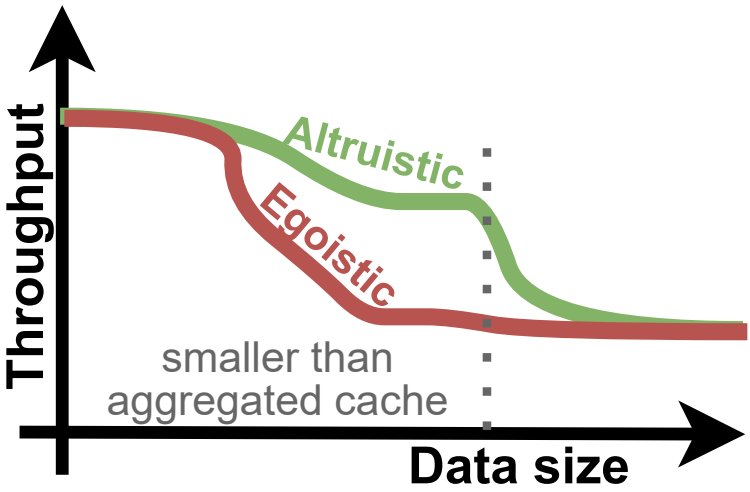


Figure 8.7: Effect of Egoistic vs. Altruistic Eviction when the Working Set is Smaller than the Aggregated Memory of Compute Nodes.

can join any time. However, achieving elasticity of the storage layer needs a more careful design. In the following, we sketch ideas on how to achieve elasticity in order to scale the storage layer up and down at runtime.

Research Opportunity: Elastic Storage & Caching. To achieve elasticity on the storage layer of ScaleStore, some of the state must be moved whenever a new storage node joins or old one leaves. To relocate pages in storage, consistent hashing [106] can be used. In consistent hashing, all nodes are organized in a hash ring which represents hash-ranges as illustrated in Figure 8.5(ii). Every node in the ring is assigned to a specific range, based on *the hash of the page identifiers*. The key idea is to leverage the storage layer in a way such that every physical storage node is placed randomly multiple times on this ring and thereby responsible for multiple of those ranges. In our example from Figure 8.5 we can see that Node 3 is placed twice on the ring and manages P1 and P4. Once a storage node leaves the cluster, all its pages are moved to the neighbors in the ring. For instance, when Node 3 leaves, P1 needs to be moved to Node 4 and P4 to Node 5. Conversely, whenever a new node joins it gets some of the pages from its neighbors. Clearly, using a consistent-hashing scheme provides elasticity and has been used in the past. However, while the data pages can be moved asynchronously in the background, moving the directory (and its tracking data) requires more attention. For example, an inconsistency can arise if two directories manage access to the same page and they grant exclusive access to different nodes. An open question is thus how to update the directory state at run-time.

8.3.4 ACID Guarantees

Currently, ScaleStore is merely a distributed storage engine that provides page-level sequential consistency but without transaction support. Therefore, let us next discuss how to support ACID transactions on top of an elastic caching-based DBMS.

Built-In Atomicity and Consistency. A key challenge of distributed DBMSs is to ensure all nodes agree on whether a particular transaction commits. In shared-nothing systems, this is achieved using a two-phase commit protocol (2PC) [71]. In a shared-caching system, every transaction becomes a local transaction since all of the transaction's data can be cached at one node. Thus, 2PC is not required [108]. Consistency is straightforward too since every compute node can validate declarative constraints locally by simply loading and accessing the data from the cache/remote nodes.

Research Opportunity: Isolation. There is a rich literature on locking-based concurrency control in shared-cache systems such as [99, 154, 176]. For instance, DEC's Rdb/VMS [99] uses a distributed lock manager with techniques to reduce the number of network messages. It uses lock de-escalation which means that processes will first acquire a lock on a large granule (e.g., a table) so as to permit operation on pages or records without additional lock requests.

Along the same lines, Rahm's Primary Copy Locking scheme [176] reduces network messages by integrating the concurrency control with an *on-request invalidation coherence scheme*. In contrast to our scheme, which invalidates as soon as they are outdated, on-request invalidation leaves outdated pages in the cache. Thus every page access must validate that the page is still up-to-date. This is achieved by comparing a cached page's version number with its version number on the storage node, and incrementing the version number on updates. Naturally, a lock request can be piggy-backed with the required validation message. E.g., when a node wants to read P1 in the cache, it sends a message with the page version and the shared-lock request to the storage node.

Mohan and Narang [154] describe LP-Locking, a technique that avoids this piggybacked message altogether by granting local lock requests as long as the page is cached on the compute node. This approach is a good fit for ScaleStore's modus operandi as pages are cached as long as there is no conflict (i.e., invalidation). However, their design works at page-granularity, not record-level, which may impair concurrency in highly parallel systems. We see two possible approaches to address this difficulty: investigate new techniques such as contention split [3], which splits pages at contention points to achieve higher concurrency despite page-level locking; or use more involved schemes based on record-level locking [155] which, however, require more network messages and

may increase latency. We think an evaluation-based comparison of these approaches on modern hardware is an interesting avenue for future work.

The previous schemes rely on pessimistic CC. Another direction is to evaluate modern optimistic CC schemes [215]. Especially for ScaleStore, such schemes are interesting as they avoid all shared-memory writes for read operations. That is, no pages must be invalidated and thus no network messages must be sent at all. AWS Aurora uses an optimistic scheme where conflicts are detected during log replay on storage servers. This solution is simpler than pessimistic lock protocols. However, the delay in detecting conflicts significantly limits throughput of transactions on write-hot data, since it leads to many aborts.

Research Opportunity: Durability & Recovery. Since ScaleStore organizes data in pages, the standard ARIES-style logging and recovery mechanism [152] of disk-based DBMSs is applicable. Typically, this involves a single sequential log for all concurrent transactions. Having all nodes write sequentially to a central log is clearly not scalable. A scalable alternative is decentralized logging, where every node writes its private log file. Since every compute node can potentially touch any page, the changes for a single-page can be dispersed across multiple local logs. Consequently, in case of a failure, those log files must be merged in the storage layer, either during recovery or online at run-time. Mohan and Narang [155] describe this approach and provide ARIES-style logging for shared-cache systems. As shown by Haubenschild et al. [81], single-node ARIES performance and scalability can be improved with techniques like Remote Flush Avoidance (RFA). Therefore, we believe that existing schemes [155] should be revisited to further improve performance and scalability in a modern shared-cache system.

8.3.5 Cloud Infrastructure and Services

There are also many challenges and opportunities when building an OLTP DBMS that arise from the fact that come with modern cloud infrastructures and services.

Cloud Services. So far, we have assumed that the compute and storage layers are under the control of the DBMS. However, there are now many opportunities to utilize off-the-shelf cloud services instead. Those services may be more cost-effective than hand-rolled solutions and often come with high availability guarantees. For instance, AWS's object store S3 offers built-in replication. A shared-caching DBMS could use such a storage service to replace the storage layer. Of course, S3 does not offer the directory functionality, which then must be moved to the compute node or even decoupled completely into its own layer. Unfortunately, using such a proprietary service comes

with its own technical challenges. For instance, S3's latency is in the tens of milliseconds which is unacceptable in latency-critical workloads such as OLTP. One could circumvent this by equipping the compute nodes with SSDs having microsecond latency to store warm data and only use S3 to read cold data or save checkpoints/log asynchronously.

Multi-Cloud Support. A further complication is that customers want multi-cloud support. Thus, when using a storage service, a DBMS must support the storage services of multiple vendors such as AWS, Microsoft, and Google. This adds code complexity not only due to different APIs but also due to different performance, availability, and cost characteristics. Another challenge is that data center network latency varies across cloud vendors. RDMA offers single-digit microseconds latency but it is only available in Microsoft Azure and only for certain virtual machine types. EFA is AWS's low-latency network offering, which has 10× higher latency than RDMA [254]. Therefore, a latency-adaptive coherence and eviction protocol is required. That is, buffer-to-buffer communication may not be faster than reading from a VM's local SSD, which means that the coherence protocol should load and cache pages to local SSD. Conversely, the eviction should adapt for the use of the egoistic variant (cf. Section 8.3.2), which is inefficient if buffer-to-buffer communication is fast, but preferable if remote accesses are costly.

Hyperscaler Opportunities. Hyperscalers have many possibilities to co-design services and infrastructure. For instance, Microsoft rolled their own internal low-latency logging service called XLOG [6]. Another trend is that data center networks are equipped with programmable switches and network cards. This opens the possibility to embed the coherence protocol and conflict resolution inside the network. For instance, the directory can be placed inside the switches and thus be completely transparent to the DBMS.

Multitenancy and Workload Placement. Another exciting avenue of work is multitenancy and workload placement. A scalable system allows multiple tenants to execute their workload in the same instance, necessitating fairness in resource allocation and isolation between tenants. Of course, this is a challenging task that requires policies and strategies but enables better resource utilization. Especially coupled with workload placement which could allow pairing customer workloads that are more CPU intensive with other customer workloads which require more I/O. The flexibility of shared-caching systems allows to provision additional compute instances to avoid SLA violations. Therefore, such a system is predestined to run autonomously in the cloud, e.g., as a backend for Database-as-a-Service.

8.4 Summary

The title of this paper asked if scalable OLTP in the cloud is a solved problem. To approach this question, we analyzed the data access path of different distributed OLTP systems and devised a taxonomy that helps to characterize their asymptotic scalability. We found that most modern cloud systems are either based on the single-writer or the partitioned-writer paradigm. While these architectures certainly have their merits, we believe that the elasticity and scalability properties of a shared-cache design (shared-writer with coherent caches) is a better foundation for building cloud-native OLTP DBMSs. In this paper, we presented our findings and lessons learned from building such a DBMS. However, as we outlined in this paper, many interesting research opportunities must be solved to build a full-fledged shared-cache DBMS for the cloud.

8.5 Acknowledgments

This work was partially funded by the German Research Foundation priority program 2037 (DFG) under the grants BI2011/1 & BI2011/2, the DFG Collaborative Research Center 1053 (MAKI), the BMBF and the state of Hesse as part of the NHR Program. We also thank hessian.AI and DFKI for the support. We thank Marco Slot for insights about the importance of low latency access, and the anonymous CIDR reviewers for many excellent suggestions on improving our original submission.

9 EFA: A Viable Alternative to RDMA Over InfiniBand for DBMSs?

Abstract

RDMA over InfiniBand offers high bandwidth and low latency which provides many benefits for distributed DBMSs. However, in the cloud RDMA is still not widely available. Instead, cloud providers often invest in their own high-speed networking technology and start to expose their own native networking interfaces. For example, the largest cloud provider, Amazon Web Services (AWS), introduced instances with Elastic Fabric Adapter (EFA) in 2018. In this paper, we aim to analyze EFA as an alternative to RDMA in the cloud by performing an in-depth and systematic evaluation.

Bibliographic Information

The content of this chapter was previously published in the peer-reviewed work Tobias Ziegler, Dwarakanandan Bindiganavile Mohan, Viktor Leis, and Carsten Binnig. "EFA: A Viable Alternative to RDMA over InfiniBand for DBMSs?" In: *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022*. Ed. by Spyros Blanas and Norman May. ACM, 2022, 10:1–10:5. DOI: 10.1145/3533737.3538506. URL: https://doi.org/10.1145/3533737.3538506. The contributions of the author of this dissertation are summarized in Section 3.2.

9.1 Introduction

InfiniBand offers low network latency. Remote Direct Memory Access (RDMA) over InfiniBand offers single-digit microsecond network latencies at extremely high bandwidth. As such RDMA over InfiniBand had a tremendous impact on distributed machine learning [98, 137, 187, 235], high performance computing [133, 167, 179], and distributed database systems [20, 64, 102, 104, 129, 132, 149, 150, 184, 213, 226, 233, 239, 241, 243, 256]. In fact, distributed system designs that were previously thought to be inefficient had to be re-evaluated [64, 213, 239, 256]. For instance, distributed shared memory architectures are becoming increasingly common due to the low network latencies. This is evidenced by several distributed database systems that expose their aggregated memory via RDMA such as FaRM [55, 56, 196], NAM-DB [239], or Tell [134]. Unfortunately, to get the benefits of RDMA, specialized and expensive hardware is required.

InfiniBand is not widely available in the cloud. At the same time, many companies transition from on-premise to the cloud and are dependent on currently-available cloud offerings. Unfortunately, even though cloud providers offer hundreds of heterogeneous instance types, InfiniBand is *not widely available* in public clouds. In fact, of the three major cloud providers, only Microsoft Azure offers InfiniBand for a very limited set of instance families [147] (i.e., instance types of H, HB, HC, and some of the N series). This limited availability takes away one of the main benefits of the cloud; a broad hardware landscape to satisfy every application requirement. And although several vendors offer networking that achieves the same bandwidth as RDMA over InfiniBand (100 Gbit), the low network latencies of InfiniBand are still unmatched.

AWS's low latency network EFA. As an alternative to RDMA over InfiniBand, in 2018 the largest cloud provider, Amazon Web Services (AWS), introduced instances with *Elastic Fabric Adapter (EFA)*. EFA is a network interface for Amazon EC2 instances that is specifically designed to achieve low latencies in the cloud. Today, EFA is widely available in AWS, and Amazon presently offers 15 instance families with EFA support [191] to satisfy different application requirements, e.g., memory optimized (r5dn, r5n) or storage optimized instances (i3en).

No systematic evaluation of EFA. Surprisingly, even though EFA has become broadly available in AWS, there has not yet been a systematic evaluation of this technology from the data management community. The limited research that addresses EFA was driven by the HPC community and specifically geared towards MPI in distributed HPC applications [31, 195, 234]. However, MPI is a higher-level library on top of the EFA networking stack and was built with different assumptions that do not necessarily match

those of data-intensive systems [5, 131, 213]. Therefore, the following question remains: *Can EFA be a viable alternative to RDMA over InfiniBand for data-intensive systems?*

Contributions. In this paper, we answer this question by performing an in-depth systematic evaluation. First, we describe the EFA stack and highlight qualitative differences to RDMA over InfiniBand which we think is an already highly valuable contribution for system design since today the information about EFA is scattered across many resources, Second, we evaluate the performance of EFA and compare it with RDMA over InfiniBand in a set of reproducible microbenchmarks. Third, based on our evaluation we derived lessons learned that can be used by system designers to utilize EFA in data-intensive systems.

9.2 Elastic Fabric Adapter

In this section, we describe the EFA stack as shown in Figure 9.1. We do this from the bottom up, starting with the hardware and Amazon's *Scalable Reliable Datagram* (SRD) protocol. Afterwards, we move on to the software layers *ibverbs* and *libfabric*.

9.2.1 SRD: A Reliable and Unordered Protocol

SRD is reliable. The foundation of EFA is SRD, a proprietary network protocol that provides reliable but unordered communication on top of commodity Ethernet switches [195]. Of course, reliability is not unique to SRD as many common protocols, including TCP/IP and reliable connected RDMA, guarantee reliable packet delivery. However, unlike existing Ethernet protocols such as TCP/IP, SRD is purpose-built for Amazon's data centers. Therefore, Amazon controls the hardware SRD operates on, which in turn enables them to implement SRD's reliability efficiently in hardware (in the AWS Nitro network cards [190]) rather than in software.

SRD is out-of-order. Moreover, unlike other reliable protocols that often guarantee in-order delivery, SRD has out-of-order delivery. This is because in-order delivery may cause head-of-line blocking [195] and thus SRD reduces tail latencies. Additionally, to avoid hot paths in the network and thus reduce the chance of packet drops, SRD packets are sent across multiple network paths. Thus, out-of-order delivery is a direct consequence of sending packets across multiple paths, and enforcing the order would require large intermediate buffers or dropping out-of-order messages.

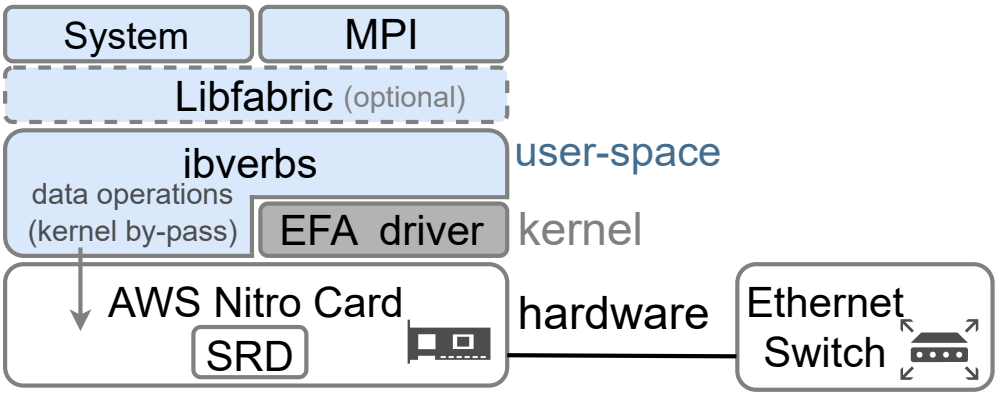


Figure 9.1: EFA Stack: Software and Hardware.

9.2.2 Ibverbs: Low Level Interface

SRD is exposed to the application via the *ibverbs* library. *Ibverbs* is the same library that allows user-space processes to use RDMA primitives to perform high-throughput, low-latency network operations on InfiniBand. The fact that EFA and RDMA share the same low-level library is no coincidence because EFA closely resembles the InfiniBand verbs specification [195]. The library itself is split into a control path and a data path.

Control path. The control path is implemented through system calls to the kernel, which further calls the low-level EFA driver depicted gray in Figure 9.1. The control path creates, modifies, queries, and destroys resources that are needed for connection setup and communication. Because the control path may interact with the kernel, those operations are commonly avoided in the hot path.

Data path. The data path avoids the kernel and network stack completely and instead interacts directly with the hardware. The reason for kernel and network stack bypass is that the traditional kernel network stack was shown to be prohibitively expensive and thus ill-suited for high-performance networks [20, 66]. This is evident by the rise of available frameworks such as DPDK [173], mTCP [97], or eRPC [101]. Furthermore, *ibverbs* is designed to provide zero-copy and asynchronous networking.

Send/receive queues. Modern network interfaces get the data buffer from the application and notify the application when data is actually sent, via traditional synchronization mechanisms such as completion queues. EFA follows this approach and offers send/ receive queues with the respective completion queues. (1) The *send queue* is used by the sender to issue a send operation that returns immediately to achieve asynchronous networking. (2) The *receive queue* is used by the receiving side to issue receive requests. A receive request instructs the network interface card (NIC) in which memory buffer to

copy the incoming data via direct memory access (DMA). Therefore, to handle incoming messages the application needs to post such receive requests beforehand.

No one-sided. As mentioned earlier, because EFA and RDMA share the same low-level interface they have a common intersection of primitives. However, there are important distinctions between EFA and reliable RDMA. Reliable connected RDMA provides two types of operations: One-sided *read/write/atomic* primitives and two-sided primitives *send/recv*. EFA does not support one-sided primitives [195], but is limited to two-sided primitives.

Connection scalability. On the other hand, SRD offers better scalability because a well-known limitation of reliable connected (RC) RDMA is lifted. That is, in RC RDMA, one connection endpoint can exclusively communicate only with one other end-point This means to achieve an all-to-all communication many connections need to be established which impacts performance [103]. In contrast, in EFA one connection end-point can communicate with all other end-points without creating an end-point for every connection. This reduces the number of connections in EFA drastically [195].

9.2.3 Libfabric: EFA's Programming Interface

Even though *ibverbs* provides all available primitives, the recommended interface for application developers is *libfabric* [192], which provides a higher level interface on top of *ibverbs* as shown in Figure 9.1. *Libfabric* is a framework that provides a unified abstraction for high-performance network devices [74]. In contrast to *ibverbs*, *libfabric* offers higher level functionality and is somewhat easier to use. It provides a standard set of APIs that are agnostic to the underlying network protocol and hardware device.

Provider. In *libfabric*, hardware devices are termed as *provider* which hook into the framework and implement optimized device-specific functionality. EFA is essentially one such provider under *libfabric* [75] (called *fi_efa*). This EFA provider is layered above *ibverbs* and thus also provides control and data transfer operations.

Endpoint types. Internally, endpoints use the primitives provided by *ibverbs* and may expose some higher- level functionality to the application developer. For instance, the reliable datagram (RDM) endpoint that exposes the proprietary SRD protocol provides higher-level features such as message segmentation that are implemented by *libfabric* in software.

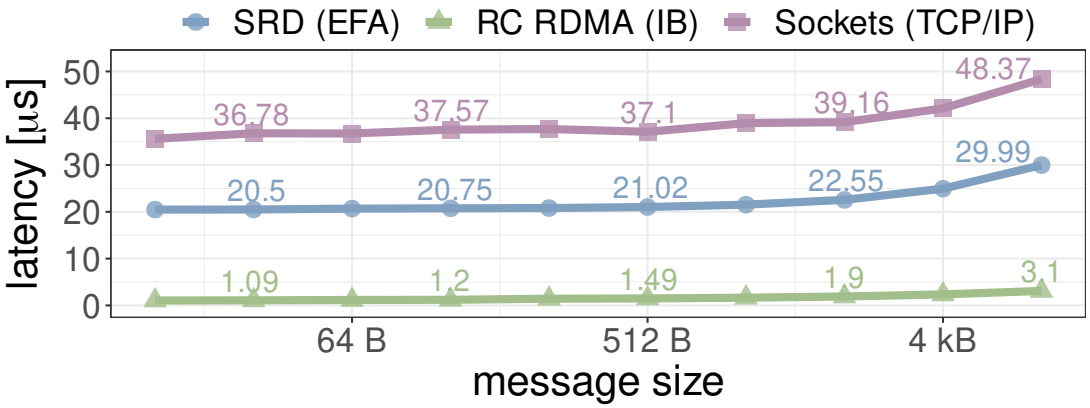


Figure 9.2: Latency with varying message sizes.

9.3 Experimental Evaluation

In the following, we discuss the results of our extensive evaluation which compares the performance of EFA with RDMA over InfiniBand in a set of reproducible microbenchmarks.

Methodology. To isolate the fundamental properties of EFA and RDMA we used the well-known performance micro-benchmark library *perftest* (available at [76]). Perftest uses *ibverbs* directly and supports EFA (SRD and UD) as well as reliable connected RDMA over InfiniBand. This makes the experiments directly comparable as both implementations use the same benchmarking code.

Setup. Because EFA is only available on AWS which does not offer RDMA we used two different hardware platforms both running Linux. We conducted the EFA experiments using two EC2 *c5n.18xlarge* instances with 192 GB main memory and 72 vCPUs connected via 100 Gigabit EFA. Both *c5n.18xlarge* instances were deployed in the recommended *cluster* placement group to achieve the best low-latency network performance [193]. We also replicated our experiments on *c5n.metal* and obtained similar results. The RDMA experiments were conducted on two bare-metal machines with 1 TB main memory and 56 CPUs connected with an InfiniBand network using Mellanox ConnectX-5 MT27800 NICs (InfiniBand EDR 4x, 100 Gbps).

9.3.1 Latency Comparison

As latency is critical for many applications, we start our evaluation by comparing the latency of SRD (EFA) with RC RDMA (IB). To classify the latency improvement of SRD (EFA) over EC2's traditional 100 Gigabit networking solution we included sockets

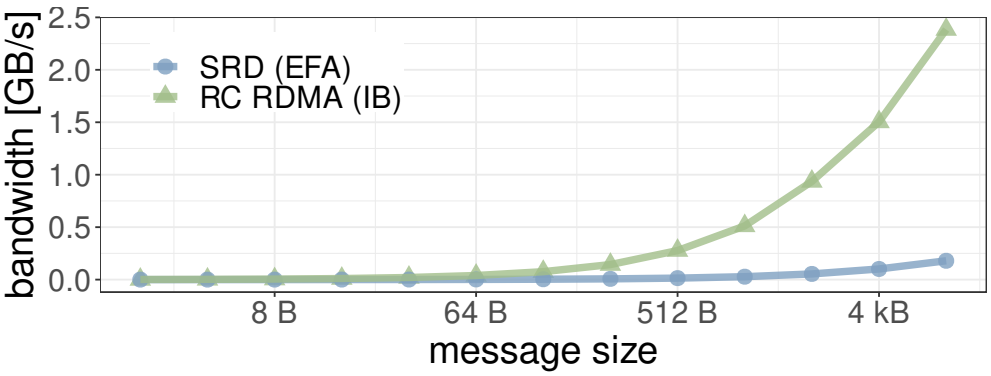


Figure 9.3: Synchronous bandwidth (single-threaded).

(TCP/IP) as a reference. As mentioned, we use perftest [76] for EFA as well as RDMA and sockperf [144] for sockets.

Figure 9.2 compares the effect of varying message sizes on the average latency, i.e., half-round-trip latency. Both EFA and RDMA provide lower latencies than sockets. However, when comparing RDMA with SRD, we can observe that RDMA's latency is around 20× lower for messages below 512 bytes. For 8 kB messages RDMA's latency is still 10× lower than SRD's latency. Thus, although the latency results for EFA are considerably better than for the TCP/IP sockets, there remains a substantial gap to RDMA (similar latencies for RDMA are achievable on Azure VMs [146]).

9.3.2 Synchronous Bandwidth

We now turn our attention from latency to bandwidth. In Figure 9.3, we examine the synchronous bandwidth of SRD (EFA) and RC RDMA (IB) with varying message sizes using a single thread. In this context, *synchronous* means that we wait for the completion of the previous message before sending the next message. SRD and RC RDMA both adhere to a delivery-complete semantics which means that the completion is generated when the remote network card receives the message. For this setup, we can see in Figure 9.3 that RDMA achieves a much higher bandwidth. The reason is that RDMA's latency is much smaller and thus when cross-referencing the bandwidth results with the latency results from Figure 9.2 the achieved bandwidth is not surprising. Therefore, the achievable bandwidth of both is limited by their respective latencies. For instance, with 8 kB RDMA achieves roughly 2.5 GB/s which is 10 times more than SRD (similar to the latency gap).

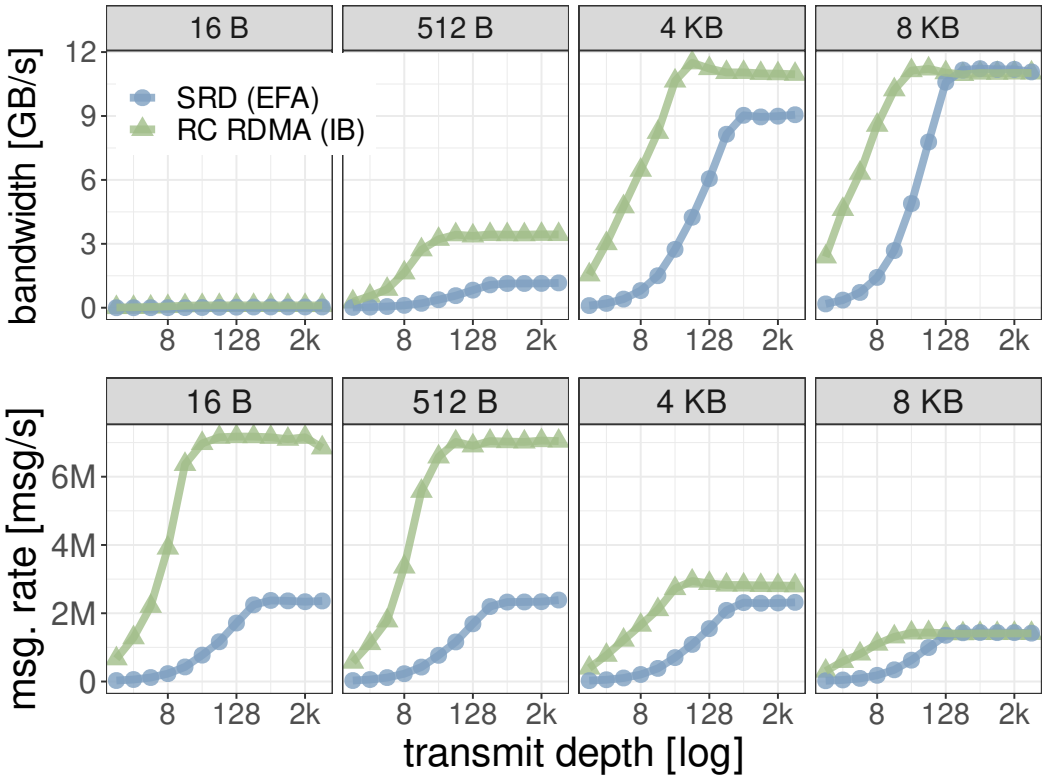


Figure 9.4: Effect of transmission depth on asynchronous bandwidth and message rate (single-threaded).

9.3.3 Asynchronous Networking

As shown above, latency becomes the limiting factor in synchronous networking. Subsequently, we investigate how asynchronous networking avoids this limitation. Therefore, we vary the number of outstanding messages (i.e, the transmission depth) and show in Figure 9.4 how the bandwidth and message rate are affected.

First, let us focus on larger messages. With 4 kB messages and a transmission depth of about 64, RDMA achieves the maximum bandwidth (i.e., around 12GB/s). In contrast, EFA does not fully saturate the bandwidth and instead seems to be message bound at around 2 M messages, i.e., the same message rate as with smaller message sizes. When messages are sufficiently large, i.e., 8 kB and larger, both EFA and RDMA achieve the maximum bandwidth. However, Figure 9.4 shows that only RDMA is able to reach the full bandwidth when the transmission depth is small. The reason for that lies in RDMA's lower latency since it enables RDMA to process outstanding messages more quickly. For instance, a transmission depth of 8 means that we always try to have 8 messages

outstanding. As RDMA's latency is lower, the completion for these 8 outstanding messages is generated faster and new messages can be transmitted. Conversely, because EFA's latency is higher the completion takes longer and thus requires a higher transmission depth to achieve the maximum bandwidth.

We now move on to smaller messages, i.e., 16 and 512 byte. When comparing the respective message rates for both 16 and 512 byte, we can observe that the message size does not affect the maximum message rate. RDMA achieves around 7 M messages per second and EFA around 2 M messages per second for both message sizes. One may now wonder what the limiting factor for EFA's message rate is. What we can already rule out is being latency-bound because we send multiple messages asynchronously. Additionally, we are not bandwidth bound either for 16 and 512 byte messages as we can clearly observe from Figure 9.4. Hence, we argue that the message rate is limited by the network card, as discussed in the next experiment.

9.3.4 NIC Parallelism

In this experiment, we examine if the processing unit (PU) of the network card became the limiting factor in the previous experiment. This can happen if a powerful CPU core overwhelms a less powerful PU on the NIC. Often a single connection is handled by a single NIC PU [102]. Consequently, increasing the number of connections may lead to the utilization of multiple NIC PUs, which ultimately improves the message rates.

To evaluate if NIC parallelism can be exploited we use a single thread and increase the number of connections. Figure 9.5 shows that a single CPU core utilizes multiple connections with SRD and thus achieves a peak message rate of around 4 M with 4 connections (16 and 512 byte messages). With 4 kB message size, EFA eventually reaches the full bandwidth with 2 connections (same message rate as RDMA which achieves full bandwidth). In contrast, RDMA does not profit from having multiple connections, instead, performance degrades slightly for smaller messages. We can conclude that RDMA is CPU-bound with small messages. We confirmed this statement by using a single connection and posting a linked list of multiple messages to the NIC, i.e., in perftest this is done by setting the *post list* parameter to 16. This allows the NIC to process the requests at full speed without being dependent on the CPU. For RDMA the message rate with 64 byte messages peaked at around 17 M messages per second whereas EFA's messages per second remained at 2 M due to the processing limit of a single NIC PU.

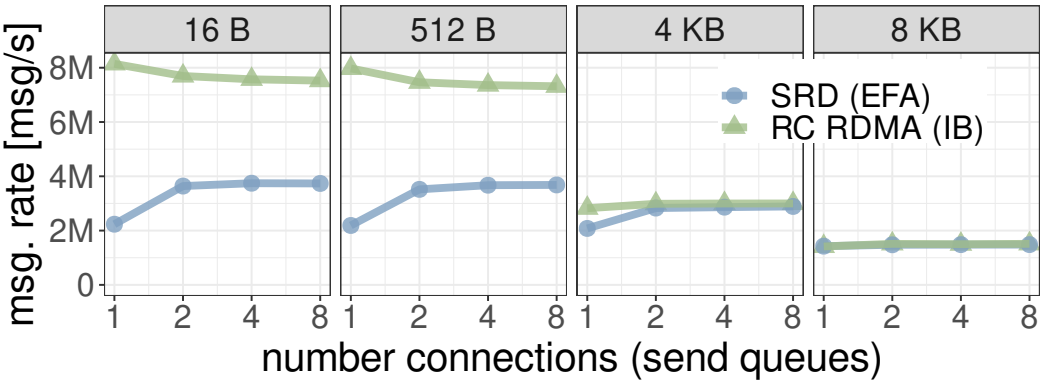


Figure 9.5: NIC parallelism (single-threaded, tx depth 256).

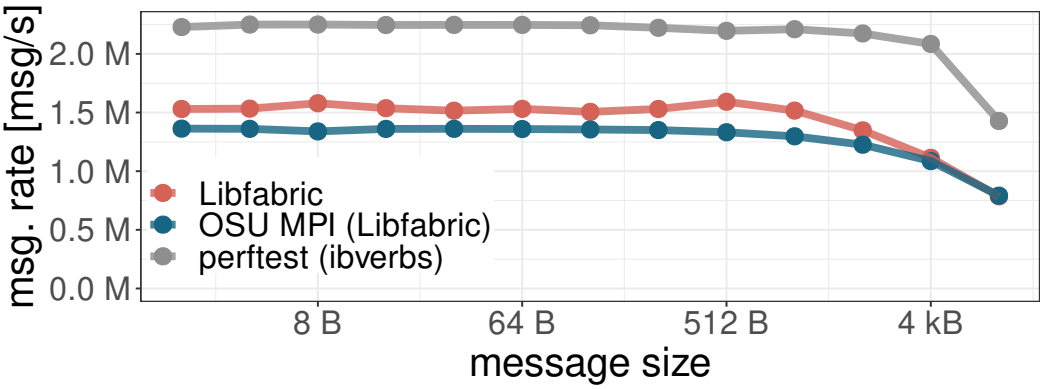


Figure 9.6: EFA interfaces (single-threaded, tx depth 256).

Based on these results, we can reasonably speculate that a single Nitro Card PU can achieve around 2 M messages per second. To achieve the maximum of 4 M messages per second multiple Nitro Card PUs need to be exploited.

9.3.5 EFA Interface Evaluation

We now compare the single-threaded performance of *libfabric* with the lower-level *ibverbs* interface. We examine the bandwidth as well as the message rate in Figure 9.6. To verify that our *libfabric* implementation performs as expected, we compared it to the OSU MPI performance test, which uses *libfabric* as well. Figure 9.6 shows that using the lower-level library *ibverbs* yields the best performance (50% more msg/second). Additionally, libfabric has many performance knobs and was not easier to use than *ibverbs*.

9.4 Related Work

We think that the broad availability of EFA has the potential to trigger a redesign of distributed databases. Unfortunately, there has only been very limited work on EFA in general.

Most research was driven by the HPC community and specifically geared towards MPI as it is the de-facto standard in distributed HPC applications [31, 195, 234]. Most notably is the paper from Shalev et al. (Amazon) [195] which discusses some of the design decisions behind the proprietary SRD protocol. The other two papers [31, 195, 234] specifically focus on MPI primitives. The only database paper which mentions EFA is from Barthels et al. [17] in which they show in one experiment that their findings are transferable to the off-the-shelf network infrastructure of AWS.

9.5 Lessons Learned and Summary

Although both EFA and RDMA are advertised for a similar audience, their performance characteristics are quite different. This has major implications on how distributed data processing systems for the cloud should be designed. In the following, we summarize our main findings and their implications on system design:

Latency. As we have seen in Figure 9.2, EFA's latency decreased twofold compared to traditional TCP/IP sockets. Nonetheless, the latencies of EFA are still an order of magnitude higher than those of RDMA. Due to EFA's higher latency it is not as beneficial to send small messages as in RDMA.

Bandwidth. As shown in Figure 9.4, the bandwidth of EFA is strongly dependent on the transmission depth and the message size. To saturate the bandwidth, a large transmission depth (e.g., 256) and message sizes are important. When the message size is below 8 kB, NIC-parallelism should be exploited by either using multiple connections for a single-threaded application or multiple threads. With even larger messages (e.g., 8 kB) those sophisticated optimizations are not necessary to achieve the full bandwidth.

Message Rate. Figure 9.4 shows that the achievable message rate for small messages in EFA is considerably smaller than in RDMA. NIC parallelism is crucial to utilize the maximum message rate, preferable with multiple threads to achieve 8 M messages/second.

No one-sided operations. Besides performance characteristics, other factors such as that there is no one-sided support for EFA plays an important role. System designs based on one-sided primitives might need to be revisited.

Out-of-order-delivery. In contrast to reliable connected RDMA, EFA does not guarantee in-order delivery. Systems which rely on ordering need to handle re-ordering in the software stack which may induce some overhead.

EFA is proprietary. Unlike InfiniBand, it is not yet possible to equip on-premise machines with EFA. Therefore, EFA can exclusively be used in AWS ec2 instances (vendor lock-in).

9.6 Acknowledgments

This work was partially funded by the German Research Foundation priority program 2037 (DFG) under the grants BI2011/1 & BI2011/2, the DFG Collaborative Research Center 1053 (MAKI), the BMBF and the state of Hesse as part of the NHR and 3AI Program. We also thank hessian.AI for the support.

10 Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks

Abstract

Over the past decade, in-memory database systems have become prevalent in academia and industry. However, large data sets often need to be stored distributed across the memory of several nodes in a cluster, since they often do not fit into the memory of a single machine. A database architecture that has recently been proposed for building distributed in-memory databases for fast RDMA-capable networks is the Network-Attached-Memory (NAM) architecture. The NAM architecture logically separates compute and memory servers and thus provides independent scalability of both resources. One important key challenge in the NAM architecture, is to provide efficient remote access methods for compute nodes to access data residing in memory nodes.

In this paper, we therefore discuss design alternatives for distributed tree-based index structures in the NAM architecture. The two main aspects that we focus on in our paper are: (1) how the index itself should be distributed across several memory servers and (2) which RDMA primitives should be used by compute servers to access the distributed index structure in the most efficient manner. Our experimental evaluation shows the trade-offs for different distributed index design alternatives using a variety of workloads. While the focus of this paper is on the NAM architecture, we believe that the findings can also help to understand the design space on how to build distributed tree-based indexes for other RDMA-based distributed database architectures in general.

Bibliographic Information

The content of this chapter was previously published in the peer-reviewed work Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. "Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks." In: *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. The contributions of the author of this dissertation are summarized in Chapter 4.

10.1 Introduction

Motivation. In the last years, in-memory database systems have become dominant in academia and industry. This is not only demonstrated by the multitude of academic projects including MonetDB, Peloton and HyPer but also by the variety of available commercial in-memory database systems such as SAP HANA, Oracle Exalytics, IBM DB2 BLU, and Microsoft Hekaton. A major challenge of in-memory systems, however, is that large data sets often do not fit into the memory of a single machine anymore. To that end, in-memory databases often need to be stored distributed across the memory of a cluster of machines. For example, Walmart — the world's largest company by revenue — uses a cluster of multiple servers that in total provide 64 terabytes of main memory to process their business data.

An architecture that has recently been proposed for building distributed in-memory database systems is the Network-Attached-Memory (NAM) architecture [20, 186, 239]. The NAM architecture was specifically designed for high-performance RDMA-enabled networks and logically separates compute and memory servers as shown in Figure 10.1. The memory servers in the NAM architecture provide a shared and distributed memory pool for storing the tables and indexes, which can be accessed from compute servers that execute queries and transactions.

A major advantage of the NAM architecture over the classical shared-nothing architecture is that compute and memory servers can be scaled independently and thus the NAM architecture can efficiently support the resource requirements for various different data-intensive workloads (OLTP, OLAP, and ML [20]). Moreover, as shown in [239], the NAM architecture is less sensitive towards data-locality and can thus support workloads where the database is not trivially partitionable. As a result, a recent paper [239] has shown that the NAM architecture can scale out nearly linearly for transactional workloads (OLTP) up to clusters with more than 50 nodes while the classical shared-nothing architecture stops scaling after only a few nodes.

However, what enables the scalability of the NAM architecture is the advent of affordable high-performance networks such as InfiniBand, RoCE, or OmniPath. These networks not only provide high-bandwidth and low-latency, but also allow to bypass the CPU for many of the required data transfer operations using Remote-Direct-Memory-Access (RDMA), minimizing the CPU overhead for every data transfer. Unfortunately, taking full advantage of RDMA especially for smaller data transfers is not easy and as [239] points out requires a careful design for all in-memory data structures.

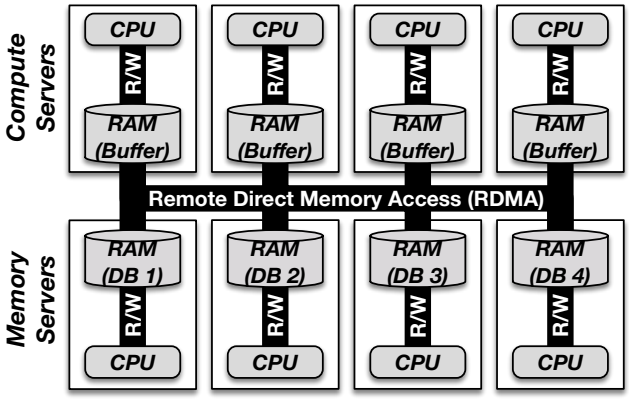


Figure 10.1: The NAM Architecture

Previous work [55, 56, 103, 239] therefore made several proposals on how to design data structures mainly to support concurrent updates. However, all these systems assume that secondary indexes are not distributed and do not span more than one server. While it is a reasonable assumption for some workloads, it cannot only severely limit the scalability of the entire system, but also create hot-spots if the index is commonly read/updated, destroying one of the key advantages of the NAM architecture.

In this paper we therefore investigate if it is possible to design a scalable tree-based index structure for RDMA. Our focus is on tree structures in order to handle range queries efficiently, and on the NAM architecture because of its scalability, as well as its capability to separately scale compute and in-memory storage. However, designing a scalable tree-based index structure is not trivial and many design choices exists. For instance, accessing indexes via RDMA leaves the option whether to use one-sided RDMA operations, which do not involve the remote CPU, or two-sided RDMA operations, which are essentially RPC calls. One-sided operations are more scalable as they have less overhead, but unfortunately, they are more complicated to use [20, 54, 104, 239], and might require more than one round-trip. Furthermore, many approaches exists on how the index (inner and leaf nodes) should be distributed across the storage servers. Ideally, the distribution scheme not only leverages the memory resources of all available servers in a fair manner (e.g., the memory requirements are distributed uniformly across all machines), but is also robust towards different access patterns (i.e., uniform vs. skewed, different selectivities, different read/write ratios etc.)

Contributions: In summary we make the following contributions: (1) We discuss the design options of distributed tree-based index structures for the NAM architecture that

can be efficiently accessed via RDMA. (2) We present three different possible index implementations that vary in the data distribution scheme as well as the underlying RDMA primitives used to access and update the index. (3) We analyze the performance of the proposed index designs using various workloads ranging from read-only workloads with various access patterns and selectivities to mixed workloads with different write intensities. Furthermore, the workloads used in our evaluation cover uniform and skewed distributions to show the robustness of the suggested index designs. As we will show in our experiments, both design questions, which RDMA primitives to use and how to distribute the index nodes, play a significant role on the resulting scalability and robustness of the index structure. Finally, we believe that the findings of this paper are not only applicable for the NAM architecture, but also represent a more general guideline to build distributed indexes for other architectures (e.g., the shared-nothing architecture) and applications (e.g., ordered key-value stores) over RDMA-capable networks.

Outline: The remainder of this paper is organized as follows: In Section 10.2 we first give an overview of the capabilities of RDMA-enabled networks and then discuss the design space for tree-based indexes in the NAM architecture. Afterwards, based on the design space we derive three possible tree-based indexing schemes, that we then discuss in detail in Sections 10.3 to 10.5. The evaluation, in Section 10.6, examines these index alternatives with various workloads. As mentioned before, we believe that the findings of this paper generalize to other distributed architectures. Some initial ideas in this direction are discussed in Section 10.7. Finally, we conclude with an overview of the related work in Section 10.8 and a summary of the findings and possible avenues of future work in Section 10.9.

10.2 Overview

This section provides an overview of the background of RDMA-capable networks relevant for this paper, discusses the design space of distributed indexes for RDMA and analyzes the scalability of the different alternatives. Readers familiar with RDMA can skip Section 10.2.1 and continue with Section 10.2.2.

10.2.1 RDMA Basics

Remote Direct Memory Access (RDMA) is a networking protocol that provides high bandwidth and low latency accesses to a remote node's main memory. This is achieved by using zero-copy transfer from the application space to bypass the OS kernel. There

are several RDMA implementations available — most notably InfiniBand and RDMA over Converged Ethernet (RoCE) [219].

RDMA implementations typically provide different operations (called verbs) that can be categorized into the following two classes: (1) one-sided and (2) two-sided verbs.

One-sided verbs: One-sided verbs (READ/WRITE) provide remote memory access semantics, where the host specifies the memory address of the remote node that should be accessed. When using one-sided verbs, the CPU of the remote node is not actively involved in the data transfer.

Two-sided verbs: Two-sided verbs (SEND/RECEIVE) provide channel semantics. In order to transfer data between the host and the remote node, the remote node first needs to publish a RECEIVE request before the host can transfer the data with a SEND operation. Different from one-sided verbs, the host does not specify the target remote memory address. Instead, the remote host defines the target address in its RECEIVE operation. Another difference is that the remote CPU is actively involved in the data transfer.

A further important category of verbs is *Atomic verbs*. These verbs fall into the category of one-sided verbs and enable multiple host nodes to access the same remote memory address concurrently, while preventing data races at the same time. In RDMA, there are two atomic operations available: remote compare-and-swap (CAS) and remote fetch-and-add (FA). An important difference to READ/WRITE operations is that both atomic operations (CAS and FA) can only modify exactly 8 Bytes on the remote side.

Whether to use one-sided or two-sided verbs strongly depends on the application. While one-sided operations are appealing since they do not involve the remote CPU for being executed, they typically require more complex communication protocols using multiple round-trips between the host and the remote node. On the other hand, two-sided verbs are capable of implementing an RPC-based protocol which requires only two round-trips but involves the remote CPU (potentially heavily) in the execution of the RPC and thus limits the scalability of the application. In this paper, we study these trade-offs for the design of distributed tree-based index structures. A more general analysis of whether to use one-sided or two-sided operations can be found in [54].

10.2.2 Design Space for RDMA-Based Indexes

In this section, we discuss the design space of distributed tree-based index structures for RDMA-based networks. The focus of this paper is on the NAM architecture and thus on the question of how to distribute the index over memory resources of multiple

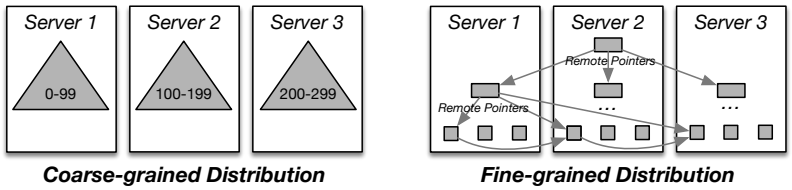


Figure 10.2: Index Distribution Schemes

memory servers, as well as how to access the distributed index from compute servers (see Figure 10.1). However, we believe that the design discussion in this section can also help to understand the design space on how to build distributed tree-based indexes for RDMA in general and thus the findings can be applied to other architectures (e.g., in a shared-nothing architecture). We will discuss some of these other scenarios in more detail in Section 10.7.

In this section, we assume that the tree-based index has a structure similar to a B-$^{+}$-tree (or more precisely a *B-link* tree [117]); i.e., inner nodes only store separators and leaf nodes store the actual keys of the index. We will describe the details of how we adapted the *B-link* tree in Sections 10.3 to 10.5 for our implementation. The index designs discussed in this paper are applicable to primary/secondary as well as clustered/non-clustered tree-based indexes with and without unique keys.

Since tree-based indexes and in particular *B-link* trees are often used as secondary indexes, the following discussions assume a secondary (i.e., non-clustered) index with non-unique keys, where duplicate keys can be stored in the leaf/inner nodes of the index while leaves map secondary keys (called keys further on) to primary keys (called payload further on). The generalization to the other cases (primary index, clustered index) is straightforward.

Index Distribution. In order to distribute the index (inner and leaf nodes) across the memory of multiple machines, two different extreme forms of distribution schemes can be applied — namely a coarse-grained and a fine-grained distribution scheme (see Figure 10.2):

(1) Coarse-grained Distribution (CG): This scheme applies a classical approach known from shared-nothing architectures. In order to distribute the index over multiple servers, we first apply a partitioning function (either range-based, round-robin, or a hash-based) to the keys that are being indexed and thus decide on which server a key and its payload should be stored. Once all keys and the payload are assigned to servers, we can build a

separate tree-based index on each of the memory servers (i.e., we are co-locating inner and leaf nodes on the same server).

(2) Fine-grained Distribution (FG): In the fine-grained distribution scheme, we implement the other extreme case, where we do not partition the index at all but instead build a global index over all keys and distribute the index nodes (i.e., leaf and inner nodes) on a per node-basis over the memory of all machines in a round-robin manner level by level. In order to connect the index nodes, we use remote memory pointers that encode not only the memory address but also the storage location (i.e., the memory server) which holds the remote node. Details about the implementation of remote memory pointers will be discussed later in Section 10.4.

While the coarse-grained scheme is the dominant solution to distribute indexes (and data) in slow networks when the network bandwidth is a major limiting factor, the fine-grained scheme is an interesting alternative for fast RDMA-capable networks since it can farm out index requests across all servers and thus lead to better load balancing. This is particularly interesting if local and remote memory bandwidth converges which is already true for the most recent InfiniBand standard HDR 4× that can provide approx. $50GB/s$ for remote memory accesses per dual-port RDMA network card.

At the end of this section, we provide a formal analysis of the scalability of both distribution schemes.

RDMA-Based Accesses. As explained in Section 10.2.1, RDMA provides two different classes of operations to access remote memory, called one-sided and two-sided. In the following, we discuss how one- and two-sided RDMA operations can be used to implement index access methods. A more general discussion about the trade-offs of one- and two-sided can be found in [54].

If we use one-sided operations to access an index from a remote host, each node of a tree-based index typically needs to be accessed independently since RDMA READ/WRITEs can only access one remote memory location at a time. Thus, a lookup of a key in a leaf needs one RDMA READ operation for each index node from the index root down to the leaf level following the remote pointers. Range queries additionally need to traverse the (linked) leaf level and need one additional RDMA READ operation for each scanned leaf.

Implementing insert operations is even more complex since we also need to take care of concurrency control (e.g., using RDMA atomics). In addition, to concurrency control operations, inserts need at least two full index passes from the root to the leaves and back. In the top-down pass, we access every index node from the root to the leaf using RDMA READs and in the bottom-up pass we then need to install at most two pages using

Description	Symbol	Example
# of Memory Servers	S	4
Bandwidth per Memory Server (GB/s)	BW	50GB/s
Page Size of Index Nodes (in Bytes)	P	1024B
Data Size (# of tuples)	D	100M
Key Size (in Bytes) - same as Value/Pointer Size -	K	8B
Fanout (per index node)	M=P/(3·K)	42
Leaves (# of nodes)	L=D/M	approx. 2.3M
Max. index height (FG, Unif./Skew)	H_{FG}=$\log_M$(L)	4
Max. index height (CG, Unif.)	H^U_{CG}=$\log_M$(L/S)	4
Max. index height (CG, Skew)	H^S_{CG}=$\log_M$(L)	4

Table 10.1: Overview of Symbols

	Fine-grained (1-sided)	Coarse-grained (2-sided)	
		Range	Hash
Step (1): Avail. BW:			
Total BW Uniform	S·BW	S·BW	S·BW
Total BW Skew	S·BW	1·BW	1·BW
Step (2): BW per Q			
Point (Unif., sel=1/L)	H_{FG}·P	H^U_{CG}·P	H^U_{CG}·P
Point (Skew, sel=z/L)	H_{FG}·P+z·P	H^S_{CG}·P+z·P	H^S_{CG}·P+z·P
Range (Unif., sel=s)	H_{FG}·P+s·L·P	H^U_{CG}·P+s·L·P	H^U_{CG}·P·S+s·L·P
Range (Skew, sel= s_z)	H_{FG}·P+s_z·L·P	H^S_{CG}·P+s_z·L·P	H^S_{CG}·P·S+s_z·L·P
Step (3): Max. Q/sec			
Max. Throughput	Avail. BW / BW requirement per Q		

Table 10.2: Scalability Analysis (Theoretical)

RDMA WRITEs for every level (in case splits happen) plus potentially one additional RDMA WRITE for installing a new root node.

In case we use two-sided RDMA operations for index accesses, we can leverage the fact that we can implement an RPC protocol using an approach similar to [103] where the remote CPU is involved to apply the index operations. An RPC call from one server to another can be implemented using a pair of SEND/RECEIVE operations: one pair for sending the request from host to remote server, and one pair for the response (e.g., which contains the result in case of an index lookup). Thus, two-sided operations seem to be more efficient for implementing remote index accesses.

However, again when assuming that the local memory bandwidth and network bandwidth are equal, one-sided operations are not worse than two-sided operations w.r.t. the bandwidth and thus throughput. Nevertheless, latency might increase. On the other hand, using one-sided operations do not involve the remote CPU at all. This is especially beneficial for skewed workloads in high-load scenarios leading to higher throughput and lower latency of index accesses, as we will show in our experiments in Section 10.6.

10.2.3 Scalability Analysis

In this section, we now formally analyze the theoretical maximal throughput for the different index design variants introduced before. The basic idea of the scalability analysis is to compute the theoretical maximal throughput (i.e., number of index accesses per second) based on the number of available servers that hold index data (i.e., memory servers in the NAM architecture). The theoretical maximal throughput will be computed by the total aggregated (remote) memory bandwidth available that can be provided by all memory servers divided by the bandwidth requirements for each index access (i.e., for each query Q).

In the analysis we did not consider latency which is definitively higher for FG because multiple network round-trips are needed for each query. In our evaluation in Section 10.6, we discuss the latency of the different index designs and show that under high-load the FG scheme outperforms the CG scheme in terms of latency while being slightly higher for normal load.

Assumptions. For analyzing maximal throughput for S memory servers, we assume that memory bandwidth and network bandwidth are equal and thus we do not differentiate between both. This assumption is realistic as shown in [20]. Since [20] was published, a new generation of InfiniBand hardware (HDR 4×) appeared. With this hardware, the remote memory bandwidth when using two network cards (or one with a dual-port interface) will give us around $50GB/s$, which is close to what we can expect today from the local memory bus of one CPU socket with 4 memory channels.

In our scalability analysis, we further consider different combinations of how the index is distributed across memory servers (fine-grained/FG vs. coarse-grained/CG), as well as different workload characteristics of how index accesses are distributed (uniform vs. skewed). Since, we cannot analyze all possible alternatives we make the following restrictions: (a) We do not differentiate between one-sided and two-sided RDMA operations since we assume memory and network bandwidth to be equal as discussed before. Moreover, we also do not include the CPU load in our analysis. To that end, searching an index page from a compute server using a RPC (based on two-sided operations) or a read using one-sided operations is not different anymore w.r.t. required (remote) memory bandwidth and thus the resulting throughput. (b) We only consider read-only index accesses for the theoretical analysis since read accesses dominate OLTP workloads [113] (and OLAP workloads as well). In our experimental evaluation in Section 10.6, we also include workloads with write accesses which show similar effects as the theoretical analysis for read-only workloads in terms of throughput. (c) For analyzing skewed workloads, we

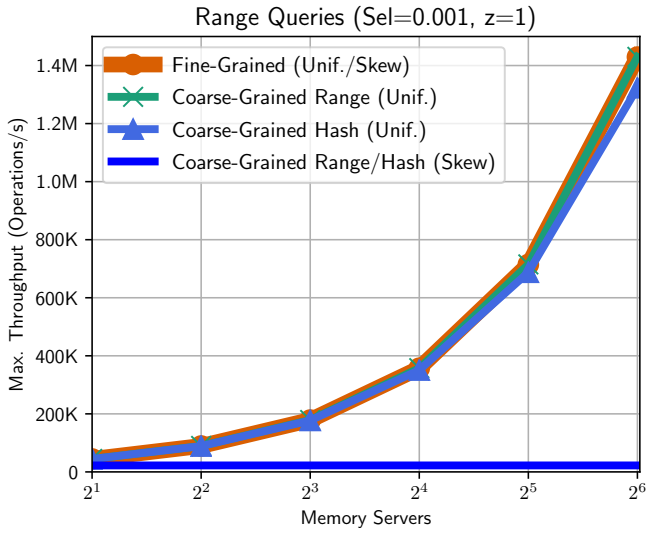


Figure 10.3: Maximal Throughput (Theoretical)

assume attribute-value-skew on the indexed attribute; i.e., one distinct key dominates the distribution of the secondary index. In case of CG-distribution (hash and range) this means that the majority of index entries (i.e., inner and leaf nodes) will end up being stored on a single memory server.

Analysis and Findings. We now discuss the scalability of throughput for different index designs. For the scalability analysis, we use the symbols that we introduce in Table 10.1. The findings of the analysis are summarized in Table 10.2. In the following, we explain the idea behind the analysis in a stepwise manner as indicated in the table.

Step (1) in Table 10.2: First, for different workload distributions we model the effectively available aggregated bandwidth of all memory servers that hold the index. We assume that we have a cluster with S memory servers, each contributing a bandwidth BW to the aggregated bandwidth. The total available aggregated bandwidth for FG-distribution is always $S \cdot BW$. The reason is that even if workload is skewed (i.e., one secondary key dominates the distribution), index accesses will always be farmed out to all memory servers due to the round-robin distribution of index nodes to servers. This is different for CG-distribution (hash and range). In the skewed case, the very same memory server stores most of the index data, thus effectively limiting the bandwidth to only $1 \cdot BW$ in the worst case.

Step (2), line 2 in Table 10.2: In a second step, we now consider the (remote) memory bandwidth requirements of an individual index access (i.e., one query) and start with point queries and then continue with range queries.

For uniform distribution, we assume that only one leaf page needs to be read by the point query (i.e., the selectivity is $sel = 1/L$). In both cases (FG and CG), we thus only need to traverse the index height. To that end the memory bandwidth requirement is $H_{FG} \cdot P$ (where P is the size of an index node in bytes). For FG distribution, the index is built over all leaf nodes L and thus the height is $log_M(L)$ where M is the fanout of an index node. The index height for CG (uniform) is only $log_M(L/S)$ since data is first partitioned on S servers. In the CG (skew) case, we assume that the maximal index height under the CG scheme is the same as for the FG scheme. The reason is that most leaf nodes will be stored in one memory server (which also increases the index height). Moreover, we assume a read-amplification of z for skewed workloads; i.e., z leaf pages need to be retrieved instead of 1 only (i.e., the selectivity is $sel = z/L$) resulting in an additional memory bandwidth requirement of $z \cdot P$ for a point query.

For range queries, we assume that under uniform workload a fraction of s leaf pages needs to be retrieved (i.e., the selectivity is $sel = s$). For skewed workloads this will again be amplified by a factor of z (i.e., the selectivity is $sel = z \cdot s = s_z$). Moreover, in the skewed case, we again assume that CG has the same maximal height as FG since most data will be stored in one memory server. Additionally, for the hash-based scheme queries must be send to all S memory servers resulting in S index traversals from root to leaves.

Step (3), line 3 in Table 10.2: Based on the results in step (1) and (2), we can now derive the theoretical maximal throughput for S memory servers by dividing the available aggregated memory bandwidth (step 1) by the bandwidth requirements for the different queries (step 2).

Example Results. Figure 10.3 plots the results of our analysis for range queries (for a uniform and a skewed workload). Point queries show a similar trend; thus we do not show their plot. For Figure 10.3 plot, we use the example values provided in Table 10.1 (rightmost column), however, additionally varying the available memory servers between $S = 2...64$. We choose a selectivity of $s = 0.001$ and skew amplification of $z = 10$.

We can see that all index designs scale well for uniform workloads. The reason why the CG scheme (uniform) scales slightly worse for hash-partitioning than for range-partitioning is that queries for hash-partitioning need to be sent to all memory servers since all servers might hold relevant index entries for range queries resulting in the fact that the indexes on all S machines need to be traversed. This is different for skewed

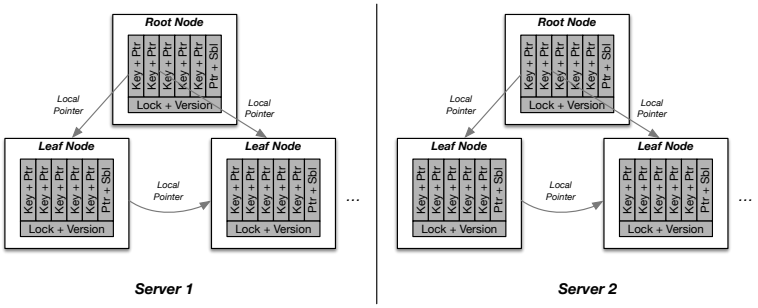


Figure 10.4: Design 1 - Coarse-Grained Index

workloads. Here the FG scheme still shows the same scalability as for a uniform workload while the CG scheme stagnates with increasing memory servers. The main reason is that the total available bandwidth is limited to only BW under skew and is independent from the number of memory servers available. To that end, we can see that in terms of throughput the FG scheme is the only scheme which achieves a throughput which scales with the available memory servers independent of the workload.
Next, we present the details of our index implementations using a coarse-grained (CG), a fine-grained (FG), and a hybrid distribution scheme that mixes CG and FG. For each of the index implementations we also discuss which RDMA operations are being used for accessing the index from a remote machine (i.e., from a compute server).

10.3 Design 1: Coarse-Grained/Two-Sided

In this section, we discuss our first tree-based index structure design which can be distributed over the memory of multiple servers and accessed by clients via RDMA (e.g., compute servers in the NAM architecture). First, we discuss the details of the distributed index structure itself. Afterwards, we elaborate on how this index structure can be efficiently accessed using RDMA operations.

10.3.1 Index Structure

The first index structure leverages a coarse-grained distribution scheme as shown in Figure 10.2. The basic idea of the coarse-grained index distribution scheme is to partition the key space by using a traditional partitioning scheme (either hash- or range-based), between different memory servers. Afterwards, each memory server individually builds a tree-based index for its assigned keys.

The internal index structure in each node is shown in Figure 10.4 and follows the basic concepts of a *B-link* tree [117]. However, different from the original *B-link* tree, we use real memory pointers instead of page identifiers. More importantly, we additionally introduce an 8-byte field per index node which stores a pair $(version, lock\text{-}bit)$ where the last bit represents a lock-bit. We use this field to implement a concurrency protocol based on optimistic-lock-coupling [120]. Our adaption of optimistic-lock-coupling for RDMA is explained in the next section.

```
operation lookup(key, node, parent, parentVersion) {
  version = readLockOrRestart(node);
  if(parent!= null)
    readUnlockOrRestart(parent, versionParent)

  if(isLeaf(node))
     value= getLeafValue(node)
     checkOrRestart(node, version)
     return value
  else
    nextNode = node.findChildInNodeOrSingling(key)
    checkOrRestart(node, version)
    return lookup(key, nextNode, node, version)
}

operation insert(key, value, node, parent, parentVersion){
  version = readLockOrRestart(node);
  if(parent!= null)
    readUnlockOrRestart(parent, versionParent)

  if(isLeaf(node))
    upgradeToWriteLockOrRestart(node, version)
    splitKey = node.insert(key, value)
    writeUnlock(node)
    return splitKey
  else
    nextNode = node.findChild(key)
    splitKey = insert(key, value, nextNode, node,version)
    if(splitKey!=NULL)
      upgradeToWriteLockOrRestart(node, version)
      parentSplitKey = node.insert(key, value)
      writeUnlock(node)
      return parentSplitKey
    return NULL
```

```
}
```

Listing 10.1: Operations of a Coarse-Grained Index

10.3.2 RDMA-Based Accesses

In order to access the index structure, as shown in Figure 10.4, from a remote host, we use an RPC-based protocol that uses two-sided RDMA operations. This design thus follows a more traditional paradigm where operations are shipped to the data – similar to how database operations are executed in a shared-nothing architecture. Other index designs which use one-sided operations or a hybrid access protocol that mixes one-/two-sided operations are explained in Sections 10.4 and 10.5 respectively.

Our RPC implementation for this index design is using RDMA SEND/RECEIVE similar to the RPC implementation of [103]. In contrast to [103], we are not using unreliable datagrams (UD) to implement the RPC but reliable connections (RC). Typically the overall throughput of index operations is limited by either the CPU or the memory bandwidth as we will show in our experiments; rather than by the number of RDMA operations that the network card can execute (which was the main motivation of using UD in [103]). Furthermore, to better scale-out with the number of clients, we are using shared receive queues (SRQs) to handle the RDMA RECEIVE operations on the memory servers. SRQs allow all incoming clients to be mapped to a fixed number of receive queues, instead of using one receive queue per client [203].

In the following, we mainly focus on how the remote procedures are executed on memory servers which store the part of the index being requested.

Index Lookups. If an incoming RPC represents an index lookup (i.e., a point- or a range-query), a thread which handles the RPC in a memory server traverses the index using a concurrency protocol based on optimistic-lock-coupling [120] but adapted for our tree-based index structure.

Index Updates. Furthermore, we also support index inserts and deletes as on RPCs. In the following, we first discuss inserts and then delete operations. Similar to [120], the thread which handles the insertion RPC, does not acquire any lock in the top-down pass from root to leaf nodes. Instead it acquires the first lock on the leaf level using a local compare-and-swap (CAS). If a leaf needs to be split, due to an insert, the locks are propagated to the parent nodes. Delete operations are implemented by setting a delete bit per index entry instead of removing the key. For removing deleted entries we use an epoch-based garbage collection scheme which runs on each memory server in a NAM

architecture and is responsible for removing and re-balancing the index in regular intervals.

The code for the two operations `lookup` (point-query) and `insert` that are executed locally on a memory server is shown in Listing 10.1. The concurrency scheme of the coarse-grained index relies on the same methods as [120]. Range-queries work similar and only need to traverse the leaf level additionally. The helper methods used in the code of Listing 10.1 is shown in Appendix 10.10.1. In order to implement the lookup operation of Listing 10.1, `readLockOrRestart` is used which implements a spinlock on the lock-bit to enter a node. Furthermore, after scanning the content of a node, the lookup operation calls `checkOrRestart` which uses the full version information (including the lock-bit) to check whether a concurrent modification has happened while searching the node. The insert operation of Listing 10.1 additionally uses a compare-and-swap operation in its `upgradeToWriteLockOrRestart` operation to set the lock-bit before modifying a node and insert a new key. For releasing the lock in `writeUnlock` a fetch-and-add operation is used to atomically reset the lock-bit and carry the bit over to increase the version counter.

10.4 Design 2: Fine-Grained/One-Sided

In this section, we discuss our second design of a tree-based index structure.

10.4.1 Index Structure

The basic idea of the fine-grained index is that the index is distributed on a per-node basis to different memory servers in a round robin fashion as discussed in Section 10.2. An example index structure is shown in Figure 10.5.

As in the first design, in addition to the keys and pointers each index node stores an 8-byte field with $(version, lock\text{-}bit)$ at the beginning of each node. However, different from the first design, pointers are implemented as so called *remote pointers*. More precisely, a remote pointer is a 8-byte field which stores `(nullbit, node-ID, offset)`. The `nullbit` indicates whether a remote pointer is a NULL-pointer or not and the `node-ID` encodes the address of the remote memory server (using 7 Bit). The remaining 7 Byte encode an `offset` into the remote memory that can be accessed via RDMA.

Furthermore, we introduce an optimization called head nodes on the leaf level. The optimization will be discussed at the end of this section.

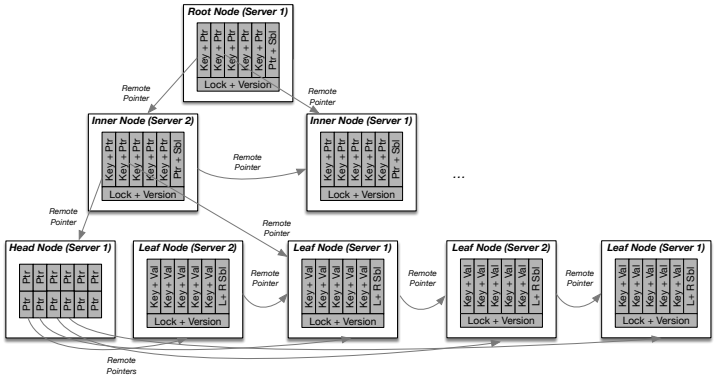


Figure 10.5: Design 2 - Fine-Grained Index

10.4.2 RDMA-Based Accesses

In order to access the index structure shown in Figure 10.5 from a remote host, we use an RDMA-based protocol that is based on one-sided operations. For the fine-grained distribution scheme we anyway need to access each index node separately (inner and leaf node) and thus one-sided operations are a good fit for the FG distribution scheme. Similar to the index design in Section 10.3, we use a protocol that is based on optimistic-lock-coupling. Yet, all operations are implemented using one-sided RDMA primitives.

Index Lookups. The intention of the lookup-operation for point-queries is that it can be executed by a compute server to access a remote memory server(s) which store the index node(s). The code for the `remote_lookup` operation which implements a point-query is shown in Listing 10.2. Range-queries work similar and only need to traverse the leaf level additionally. The code of the helper methods is shown in Listing 10.4 in Appendix 10.10.1.

The main difference to the lookup protocol of Section 10.3 is that the `remote_lookup` first copies the accessed node (inner or leaf) with an `RDMA_READ` to the memory of the client. Afterwards, the lookup operation checks on its local copy if the lock is not set (i.e., it is set to 0) and fetches a new copy if the lock is set by implementing a remote spinlock.

An interesting observation is that different from the original protocol in [120], we do not need the version of the node after searching the node. Version checking in the coarse-grained scheme (as shown in line 8 and 12 in Listing 10.1 for the coarse-grained scheme) is used since a reader may see an intermediate inconsistent state of an index node. In the fine-grained scheme, however, the client holds a local consistent copy which cannot be modified by other clients, hence version checking is not needed. Furthermore, we also do not check the version of the parent again once we traversed down to the next

level (as shown in line 4 in Listing 10.1 for the coarse-grained scheme). Therefore, a concurrent split on the current level might not be detected. However, after a split the first node is written in-place (as discussed below). Thus, we can use the fact that we implement a *B-link* tree and continue the search with the sibling if the search key is not found in the current node.

Finally, a last modification, when using a one-sided protocol, is that compute servers need to know the remote pointer for the root node. This can be implemented as part of a catalog service that is anyway used during query compilation and optimization to access the metadata of the database.

Index Updates. As before, we also support inserts and deletes, which will be discussed in the following.

Inserts are more complex than lookups, since they modify the index structure. Similar to the `remote_lookup` operation the compute server first fetches a local copy and checks if no lock was set. To that end, version checking after traversing from one leaf level to the next is not needed anymore since clients hold a copy of an index node. However, since clients only hold a local copy of the index node, the `remote_upgradeToWriteLockOrRestart` operation uses an `RDMA_CAS` operation for setting the lock-bit on the remote memory server. Moreover, `remote_writeUnlock` resets the lock-bit remotely with `RDMA_FETCH_AND_ADD`. This method additionally installs the modified version of the node on the remote side using an `RDMA_WRITE` as shown in Listing 10.4 in Appendix 10.10.1. In case a node has to be split (i.e., the *splitkey* is not $NULL$), the `remote_writeUnlock` method additionally writes the second node resulting from the split.

Finally, delete operations (no shown in Listing 10.2) are again implemented by setting a delete bit using a similar protocol as inserts, which modifies a local copy of the page and then writes the node back to the memory server. Moreover, we use an epoch based garbage collection scheme similar to Section 10.3, although the garbage collection thread is run by a compute server globally for the complete index. The reason is that deletions also need to lock the index nodes (same as writes). In order to implemented these locks, we have to use the same one-sided protocol as for potential concurrent writes which relies on RDMA-based atomics. The reason why we cannot run garbage collection as a local thread on a memory server is that atomicity cannot be guaranteed if remote and local atomic operations would both be used concurrently on the same memory addresses [33].

```
operation remoteLookup(key, remNodePtr) {
  node = remote_read(remNodePtr)
  remote_readLockOrRestart(node, remNodePtr)
```

```
  if(isLeaf(node))
     value= getLeafValueFromNodeOrSiblings(node)
     return value
  else
    nextNodePtr = node.findChildInNodeOrSiblings(key)
    return remoteLookup(key, nextNodePtr)
}

operation remoteInsert(key, value, remNodePtr){
  node = remote_read(remNodePtr)
  version = remote_readLockOrRestart(node, remNodePtr)

  if(isLeaf(node))
    remote_upgradeToWriteLockOrRestart(node, remNodePtr,version)
    splitKey = node.insert(key, value)
    remote_writeUnlock(node, remNodePtr)
    return splitKey
  else
    nextNodePtr = node.findChildInNodeOrSiblings(key)
    splitKey = remoteInsert(key, value, nextNodePtr)
    if(splitKey!=NULL)
      remote_upgradeToWriteLockOrRestart(node, remNodePtr,version)
      parentSplitKey = node.insert(key, value)
      remote_writeUnlock(node, remNodePtr)
      return parentSplitKey
    return NULL
}
```

Listing 10.2: Operations of a Fine-Grained Index

10.4.3 Optimization of Index Structure

In addition to the basic index design, we introduce so-called head nodes in the leaf level. Head nodes are additional leaf nodes (with no actual index data) that are installed after every n-th real leaf node. The idea of the head nodes is that they redundantly store remote pointers to all $n-1$ following leaf nodes (i.e., the pointers between leaf nodes are still kept). That way, a compute node which reads a head node during a leaf level scan (which is necessary for a range query) can use the remote pointers to prefetch leaves. This technique is based on selectively signaled RDMA READs as already presented in [186]. Prefetching reduces the network latency by masking network transfer with computation.

One difficulty that arises when using head nodes is that they need to be updated after a leaf node splits. Since head nodes are only an optimization and we keep the actual

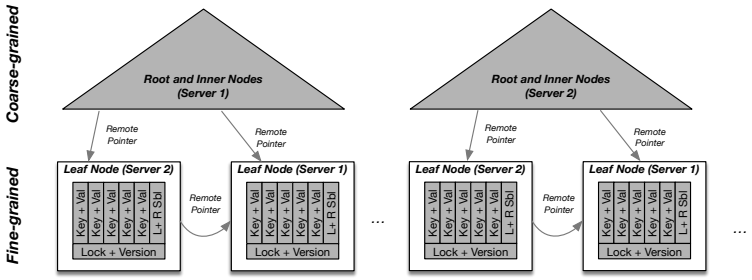


Figure 10.6: Design 3 - Hybrid Index

sibling pointers additionally in each leaf node, we do this similar to garbage collection in an epoch-based manner using an additional thread that scans through the leaf nodes of all memory servers and installs new head nodes (resp. removes the old head nodes). A compute server can detect outdated head nodes during traversing the leaf level; i.e., if a sibling pointer in a leaf node is pointing to a leaf node whose remote pointer was not in a head node (i.e., it was not prefetched), then the compute server which traverses the leaf level simply needs to execute an additional remote read for this pointer. This will cause some increase in latency but the scan of the leaf level will still be correct.

10.5 Design 3: Hybrid Scheme

This index design is a hybrid scheme combining the two schemes discussed in Section 10.3 and Section 10.4.

10.5.1 Index Structure

For distributing the index, as shown in Figure 10.6 we use a coarse-grained scheme to partition the upper levels of the index (inner and root node) while we use a fine-grained scheme for the nodes on the leaf level. The intuition is that we combine the best of both designs; i.e., getting low latency by using an RPC-based index traversal and still being able to leverage the aggregated bandwidth of all memory servers by distributing leaves in a fine-grained manner.

That way, even if attribute-value-skew on index key occurs, leaf nodes are still distributed uniformly to all memory servers. Additionally, the leaf level in this index structure can leverage head nodes, similar to the design in Section 10.4, to enable prefetching for range queries.

Workload	Point Queries	Range Queries (sel=s)	Inserts
A	100%		
B		100%	
C	95%		5%
D	50%		50%

Table 10.3: Workloads of our Evaluation

10.5.2 RDMA-Based Accesses

For accessing the index, we also use a hybrid scheme of one-sided and two-sided RDMA operations.

Index Lookups. The basic idea is that we (as already discussed before) traverse the upper levels of the index using RPCs that are implemented using two-sided operations. However, instead of returning the actual data, the RPC only returns the remote pointer to the leaf node. Afterwards, in case of a lookup (i.e., point- and range-queries), the compute server fetches the leaf nodes using one-sided RDMA READs.

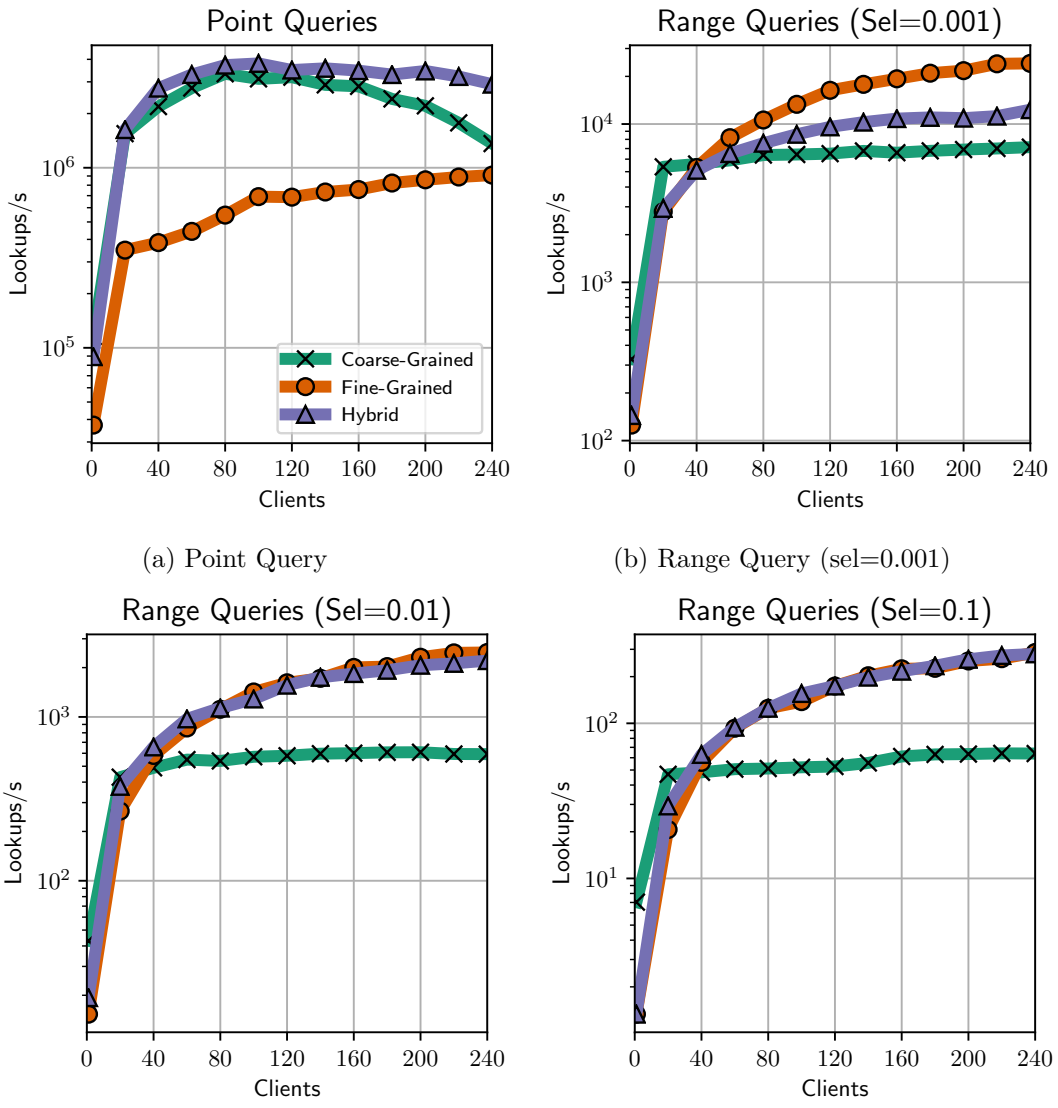
Index Updates. In case of an insertion, we again use an RPC that traverses the index and returns a remote pointer to a leaf page where the new key should be inserted to. For actually installing the key, the compute server uses the remote pointer and the one-sided protocol from Section 10.4 to install the new key to the leaf level.

In case a new leaf node has to be inserted (due to a split operation), the compute node will issue an additional RPC over two-sided RDMA to the memory server indicating that a new leaf node has been inserted (using the start key and the new remote pointer as arguments). The memory server will then use the second part of insertion protocol from Section 10.3 to install the new key into the upper levels of the index.

Finally, deletes are handled again by an epoch-based garbage collection. In this scheme, a global garbage collection thread is again executed on a compute server handles deletes for all the leaf nodes while local garbage collection threads on memory servers handle the upper levels. There is no need to synchronize the local garbage collection threads on memory servers with the global garbage collector for leaves since the delete operation already takes care of setting the delete bit in a consistent manner.

10.6 Experimental Evaluation

The goal of our experiments is to analyze the different index designs (CG/2-Sided, FG/1-sided, Hybrid) presented in Sections 10.3 to 10.5.



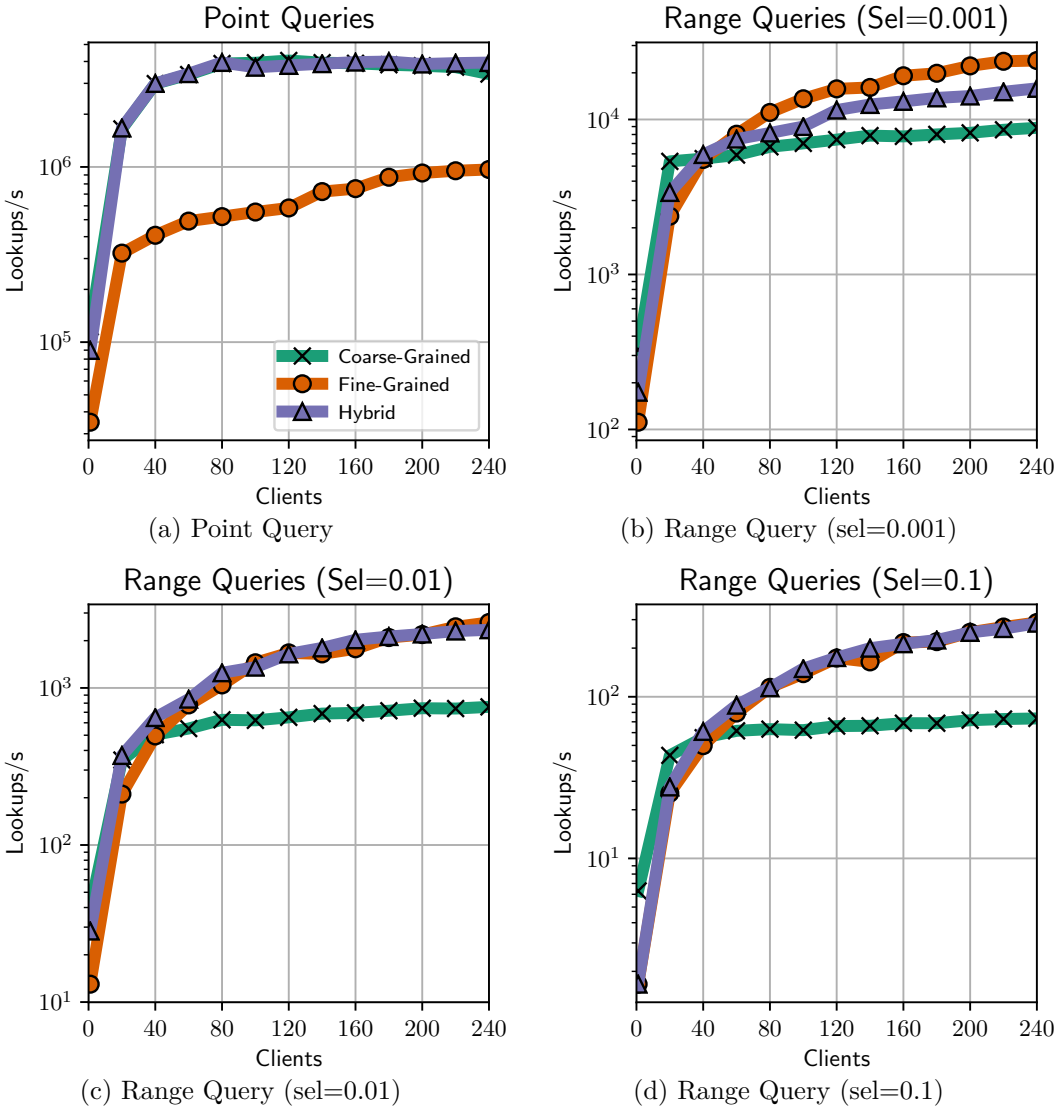
(a) Point Query

(b) Range Query (sel=0.001)

(c) Range Query (sel=0.01)

(d) Range Query (sel=0.1)

Figure 10.7: Throughput for Workloads A and B (Skewed Data, Size 100M)



(a) Point Query

(b) Range Query (sel=0.001)

(c) Range Query (sel=0.01)

(d) Range Query (sel=0.1)

Figure 10.8: Throughput for Workloads A and B (Uniform Data, Size 100M)

Workloads. We chose the Yahoo! Cloud Serving Benchmark (YCSB) to mimic typical OLTP and OLAP index workloads. Since the original version of the YCSB workload [43] does not cover all relevant cases for tree-based index structures we implemented a modified version. For example, the original version of YCSB only supports queries for short ranges but does not explicitly support different selectivities (low/ high) for query ranges. Table 10.3 summarizes the workloads of our modified version of the YCSB benchmark: *Workload A* models a read-only workload with 100% point queries while *Workload B* represents read-only workload with range queries where the selectivity can be configured to different values. In our experiment we used $sel = 0.001$ (0.1%), $sel = 0.01$ (1%) and $sel = 0.1$ (10%) to model different cases from low selectivity to high selectivity.

Moreover, the original YCSB only supports a skewed access pattern of queries by using a Zipfian distribution for the requested keys. However, for evaluating a tree-based index structure we also want to modify the skewness of the data itself (to introduce attribute-value skew), as it has a significant impact on the index performance as already discussed in Section 10.2. Therefore, we generated data sets with monotonically increasing integer keys and values (each 4-Byte) with different sizes: $10M$, $100M$, and $1B$. In order to simulate non-unique data with attribute-value skew, we use the same data with unique keys/values and assign the data based on key ranges to servers to enforce a skewed distribution; e.g., if we use two servers, we could assign 80% to one server and 20% of the data to the other server.

Setup. For executing all the experiments, we used a cluster with 8 machines featuring a dual-port Mellanox Connect-IB card connected to a single InfiniBand FDR 4× switch. Each machine has two Intel Xeon E5-2660 v2 processors (each with 10 cores) and 256GB RAM. The machines run Ubuntu 14.01 Server Edition (kernel 3.13.0-35-generic) as their operating system and use the Mellanox OFED 3.4.1 driver for the network. All three index designs where implemented using C++ 11 and compiled using GCC 4.8.5.

10.6.1 Exp.1: Throughput

In our first experiment, we analyze the throughput of the different indexing variants presented in Section 10.3 to Section 10.5. The main goal of this experiment is to show the behaviour of the different index designs under a varying load for workloads with and without skew. As discussed in Section 10.2 a major difference of the individual index designs is how data is distributed and accessed via RDMA. This determines how efficiently an indexing variant can leverage the total aggregated bandwidth of all memory servers in a NAM architecture.

In order to analyze the efficiency of the different indexing strategies and model the throughput behavior, we deployed a NAM cluster with 4 memory servers (on 2 physical machines) and used $1-6$ compute servers (on $1-6$ physical machines) each running 40 compute threads to access the index. We deployed 2 memory servers on each physical machine to exploit the fact that our InfiniBand cards support two ports; i.e., each memory server was thus using its own dedicated port on the networking cards.

In this experiment, we first focus on the read-only workloads A (point queries) and B (range queries). Workloads C and D, which include insertions, will be used in Section 10.6.3. For workload B, we ran different variants each having a different selectivity ($sel = 0.001$, $sel = 0.01$, and $sel = 0.1$). We use $100M$ key/value pairs throughout this experiment.

For running the workloads under low- and high-load scenarios, we ran each of these workloads with a different number of compute servers starting with one server that hosts 20 compute threads (called clients in this experiment). Each of the client threads executes index lookups (point and range queries) in a closed loop (i.e., it waits for a lookup to finish before executing the next lookup) and spreads lookups uniformly at random over the complete key space.

In order to model attribute-value skew in this experiment, we use range partitioning for the coarse-grained index to assign 80% of the key/value pairs to the first memory server, 12% to the second , 5% to the third, and 3% to the last memory server. Consequently, 80% of the lookups need to be sent to the first server since requests are spread uniformly across the key space. For the hybrid index, we use a similar scheme and only shuffle the leaf nodes in a round robin manner using fine-grained index distribution.

In the following, we first show the throughput results for uniform and skewed data and then discuss the network utilization of the different schemes (coarse-grained, fine-grained, and hybrid).

Discussion of Throughput:. The throughput for all schemes with skewed and uniform data are shown in Figure 10.7 and 10.8. The x-axis shows the number of clients used and the y-axis the resulting aggregated throughput.

A first interesting observation is that the hybrid approach, combining ideas of coarse- and fine-grained, performs in the most robust manner not only for point and range queries but also for different data distributions (uniform and skewed) as shown in Figure 10.7 and Figure 10.8. The reason is that it combines the best of both other schemes: getting low latency by using a RPC-based index traversal as in the coarse-grained scheme, and still being able to leverage the aggregated bandwidth of all memory servers by distributing leaves as in the fine-grained scheme. As a result the hybrid scheme has

the highest throughput in (almost) all cases. Only for lower loads (i.e., $\leq$ 20 clients), the coarse-grained outperforms the hybrid scheme minimally. The reason is that the coarse-grained scheme is slightly more communication efficient since it gets the qualifying data of an index lookup directly in the RPC response. In contrast, the hybrid scheme only gets a remote pointer and then additionally need to read the leaf data using RDMA READ operations.

For scenarios with moderate and higher loads (i.e., $>$ 20 clients), the hybrid scheme clearly outperforms the coarse-grained scheme. The reason for the stagnation of coarse-grained is that the memory servers becomes CPU bound with more than 20 clients, which in the case of coarse-grained becomes the bottleneck. One could argue that four memory servers (executed on two physical machines each having 20 cores) should allow coarse-grained to scale to 40 clients. However, the RDMA network card is attached to one socket only (while each machine has two sockets). To that end, the second memory server on each machine needs to cross the QPI link for every index lookup leading to less throughput for this server (i.e., the experiments show the point when the first memory server on each machine becomes saturated). Furthermore, we can see that the throughput of the coarse-grained scheme for skewed data (Figure 10.7) is approx. 20% below uniform data (Figure 10.8) and even declines under a high load for point queries (Figure 10.7a).

Finally, another interesting observation is that the fine-grained approach performs almost as good as the hybrid scheme except for point queries in which the fine-grained scheme achieves a much lower throughput. After all, the fine-grained scheme has a much lower network efficiency (as discussed next) for point queries; i.e., each index lookup needs to transfer multiple index pages over the network to traverse the index. This problem is mitigated in the hybrid scheme since it uses a two-sided RDMA implementation to traverse the index using an RPC to the memory server.

Discussion of Network Utilization:. In the following, we discuss the effects that we see in the network utilization as shown in Figure 10.9. One effect that becomes visible, is that fine-grained scheme, which relies purely on one-sided RDMA operations, is less network efficient for point queries as the hybrid and the coarse-grained scheme. For point queries, the coarse-grained scheme, which uses two-sided RDMA, only needs to transfer one key for the request and value for the response between compute and memory servers to implement the RPC. A similar observation holds for the hybrid scheme, which only needs one additional RDMA operation (i.e., a READ) to fetch the result. In contrast to the coarse-grained and hybrid, the fine-grained scheme needs multiple round-trips to traverse the index (i.e., read n index pages). This results in a higher network load and also translates into lower throughput.

For range queries, instead, the network communication between compute and memory servers is dominated by the data from the leaf level that needs to be transferred. For example, in the case of $s = 0.001$, fine- and coarse-grained need to transfer approx. 1600 pages in our experiments for a data size of $100M$ from the leaf level between compute and memory servers. The index pages that need to be transferred for the fine-grained scheme to traverse the index, thus do not add a noticeable overhead. For example, in our experiments with data sizes of up to $100M$, the index height is only 4; i.e., only 4 pages in addition to the 1600 pages need to be transferred in the fine-grained scheme compared to coarse-grained and hybrid scheme.

Important to note is that for fine-grained as well as for the hybrid scheme, the leaf level is distributed across all memory servers on a per page basis. Therefore, both schemes utilize the remote memory bandwidth of all memory servers when executing range queries (as shown in Figure 10.9). This allows the fine-grained as well as the hybrid scheme to achieve the same throughput for range queries under skew and uniform data, while the coarse-grained scheme is limited by the bandwidth of one memory server.

10.6.2 Exp.2: Scalability

***.**Exp.2a: Varying Data Size

The goal of this experiment is to demonstrate how the operations per second change with different data sizes while maintaining the same number of memory servers. We again analyze all three different index designs using the same setup as before with 4 memory servers on 2 physical machines. Moreover, we used 6 compute servers with a total of 240 clients to show the effect of high load and a uniform data distribution which better leverages all available resources in the different designs.

The results of the experiment can be seen in Figure 10.10 for point queries and range queries with high selectivities of 10% to show the extreme cases. For point queries, we see that all indexing approaches behave similarly for the different data sizes; i.e., with increasing data size the throughput only drops minimally in all cases. However, for range queries we can see a significant drop for fine-grained and hybrid indexes when data size increases. The reason is that both approaches become network-bound for range queries with a selectivity of $sel = 0.1$.

As we show in our next experiment, adding more memory servers helps to further increase throughput. This underlines the advantage of the NAM architecture of being able to scale the different resources (compute and memory servers) individually if one becomes a bottleneck.

***.**Exp.2b: Varying # of Memory Servers

In this experiment, we analyze the throughput of the coarse-grained and fine-grained indexing scheme when using a different number of memory servers. We do not show the hybrid scheme since the results are, as in the previous experiments, very similar to coarse-grained for point queries and to fine-grained indexes for range queries.

For the setup, we use only 3 machines for compute clients with a total number of 120 clients (i.e. 40 per compute server). Moreover, we used $1-8$ memory servers to distribute the index; where two memory servers always shared the same physical machine as before. Furthermore, as workloads we again use all the different queries (point and range queries with $sel = 0.01$) with different data distribution for a data size of $100M$ index entries.

Figure 10.11 shows the results. The x-axis shows the number of memory servers used for each run and the y-axis the resulting throughput aggregated over all clients. An interesting result of this experiment is that the fine-grained approach can make use of all memory servers for all workloads while the coarse-grained index only can benefit from an increased number of memory servers if data distribution is not skewed. Furthermore, the hybrid scheme is also sensitive to skew in point queries since the index access is dominated by the two-sided RPC-based access (which becomes the bottleneck under skew). For range queries with a high selectivity it behaves similar to the one-sided scheme again and efficiently can make use of an increased number of memory servers.

10.6.3 Exp.3: Workloads With Inserts

In the last experiment, we analyze the throughput of the different indexing variants using time workloads C and D which also include insert operations. Workload C is a workload with a low insertion rate (only 5%) whereas workload D has a relatively high insertion rate (50%). The other accesses which are not insertions are comprised only of point queries in those workloads.

In this experiment we use the same experimental setup as in Exp. 1 (Section 10.6.1) with 4 memory servers and an increasing number of clients. As data set we use the size of $100M$ index entries distributed uniformly across all 4 memory servers. The results of this experiment can be seen in Figure 10.12. The x-axis shows the number of clients used for each run and the y-axis the resulting throughput of all operations (inserts and lookups) aggregated over all clients.

Again, the hybrid index is the most robust one and clearly outperforms coarse-grained. Furthermore, the hybrid index also dominates the fine-grained index for scenarios with a load with less than 140 clients. For higher loads the fine-grained scheme has a higher

throughput while the coarse-gained and hybrid scheme degrade. The main reason is that for the coarse-gained and hybrid scheme, a higher load increases the wait time in the memory servers for spin locks. In consequence, the threads that traverse the index are busy waiting and cannot accept lookups/inserts from other clients. In case of the fine-grained scheme, the clients use remote spin locks, which allow threads in the compute server to progress if they access other nodes of the index.

10.7 Other Architectures

In this section, we discuss how the tree-based index alternatives of this paper could be adapted to other architectures than the NAM architecture.

Shared-Nothing Architecture. A classical architecture for distributed in-memory databases is the shared-nothing architecture. In this architecture, data is partitioned across the memory of all nodes, and each node has direct access only to its local memory. Furthermore, indexes are also created locally per partition. The results of this paper can be applied to the shared-nothing architecture in different ways. In the following, we discuss two potential ideas.

First, we could directly use the coarse-grained index design to make indexes that are built locally per partition (i.e., per node) accessible via RDMA also from other nodes. That way indexes could be accessed remotely using RDMA by distributed transactions that not only need to access data on a single node but also need to access data on other nodes. Moreover, transactions that run on the same node where the index resides can leverage locality (i.e., use local memory accesses) and avoid remote memory accesses completely. An additional experiment, which shows the benefits that result from locality in a shared-nothing architecture is shown in an additional experiment in Appendix 10.10.3.

Second, another problem is that indexes often do not fit into the memory of a single node. A recent study [244] shows that the indexes created for typical OLTP workloads can consume up to 55% of the total memory available in a single node in-memory DBMS. This overhead not only limits the amount of space available to store new data but also reduces space for intermediates that can be helpful when processing existing data. Consequently, another idea is to use the hybrid or fine-grained scheme in a shared nothing architecture to leverage the available memory from other nodes (e.g., in a cloud setup). A similar idea has been discussed in [128] where the buffer pool was extended to other machines that have memory available using RDMA.

Shared-Storage Architectures. Shared-storage architectures separating persistent storage from data processing is a preferred architecture for cloud databases since it can provide elasticity and high-availability [11, 26, 47]. Many of these shared-storage based systems aim to push filter operations into the storage layer to reduce data movements. Recent results show that combining Non-volatile memory (NVM) and RDMA facilitate high-performance distributed storage designs [140]. In these designs, our indexing schemes developed for the NAM architecture could also be applied to push filter operations into the RDMA-enabled storage layer.

Many-Core Architectures (Single-Node). Multi-socket, many-core architectures have replaced single-core architectures in the last decade. So far, a typical design has been that the coherency between CPU caches is managed by the hardware. However, these designs have shown to not scale to a large number of cores distributed across multiple sockets. This problem motivated non-cache-coherent (nCC) multi-core architectures where the machines can be partitioned into independent *hardware islands* [35, 73]. One direction to provide software-managed cache coherence is to use RDMA operations to transfer data between the hardware islands [35]. Thus, we believe that our index designs also become relevant when deploying single-node database on future non-cache-coherent architectures.

10.8 Related Work

Distributed Databases and RDMA. An important line of work related to this paper are new database designs based on RDMA [14, 55, 56, 103, 134, 239]. Most related to this paper is the work on the NAM architecture [20, 186, 239]. While [239] identified distributed indexes as a challenge, they only discussed them as an afterthought. Furthermore, there exists other database systems that separate storage from compute nodes [28, 47, 125], all of them treated RDMA as an afterthought and none of them discussed index design for RDMA.

Another recent work [134] is similar to the NAM architecture as it also separates storage from compute nodes. The authors discuss indexes for retrieving data from remote storage. However, their assumption is that the index is small enough to be cached completely by a compute node — an assumption which limits the applicability of their proposal. [27] discusses caching of remote memory using RDMA in general. The main insight of the paper is that finding an ideal caching strategy heavily depends on the

workload. This is an observation that we also made for tree-based indexes. Our initial results about caching can be found Appendix 10.10.4.

Other systems that focus on RDMA for building distributed database systems are FaRM [55, 56] and FaSST [103]. FaRM exposes the memory of all machines in the cluster as a shared address space. Threads in FaRM can use transactions as an abstraction to allocate, read, write, and free objects in the address space using strict serializability without worrying about the location of objects. In contrast to FaRM, FaSST discusses how remote procedure calls (RPCs) over two-sided RDMA operations can be implemented efficiently. In this paper, we built on these results: Similar to FaRM, we use an abstraction (called remote pointers) to access remote (and local) data in our fine-grained indexing scheme without worrying about the data location. Similar to FaSST, we also leverage RPC calls based on two-sided RDMA operations for implementing our coarse-grained and the hybrid index scheme. However, different from the ideas discussed in FaRM and FaSST, we implemented optimizations targeting tree-based indexes (e.g., using head pages for pre-fetching) as well as different design decisions such as using shared-receive queues to better support scale-out of compute servers connected to a fixed set of memory servers. Furthermore, both — FaRM and FaSST — discuss indexes (typically hash-tables) only as potential applications of their programming model and do not focus on distributed (tree-based) indexes as we do in this paper.

There has also been some work on RDMA-based lock managers [33, 51, 158, 236] which is relevant to this paper to implement concurrency control protocols for distributed databases. However, lock-managers which implement general purpose solutions for coarse-grained concurrency control. In our indexing schemes instead, we developed a concurrency control protocol for fine-grained latching that is based on so called optimistic-lock coupling [120].

Furthermore, many other projects in academia have also targeted RDMA for OLAP workloads, such as distributed join processing [15, 16, 183] or RDMA-based shuffle operations [132]. As opposed to our work these papers discuss RDMA in a traditional shared-nothing architecture only and they also do not consider the redesign of indexes.

Finally, industrial-strength DBMSs have also adopted RDMA. For example, Oracle RAC [175] has RDMA support, including the use of RDMA atomic primitives. Furthermore, SQLServer [128] uses RDMA to extend the buffer pool of a single node instance but does not discuss the effect on distributed databases at all. After all, none of these systems has discussed distributed indexes for RDMA-based architectures.

***.**Distributed Indexes The design of distributed indexes has not only been discussed in databases [136] but also in the context of information retrieval [49] and web databases [63].

However, none of these directions has particularly focused on the design of distributed indexes for RDMA.

A distributed RDMA-enabled key/value store such as the ones in [104, 114, 126, 138, 149, 166] can also be seen as a distributed index that can be accessed via RDMA. Different from our work, these papers typically focus on put/get for RDMA-based distributed hash tables.

citeKVDirect additionally leverages programmable NICs to extend the one-sided RDMA primitives with operations that allow clients to add / retrieve new entries from hash-tables in one round-trip instead of multiple ones. However, different from tree-based indexes, distributed hash tables do not support range queries, which are an important class of queries in OLAP and OLTP workloads. To that end, this line of work complements our work and the results can be used as another form of distributed index for point queries. In fact, in [239] the authors used results from this work to build primary clustered indexes.

10.9 Conclusions

In this paper we presented distributed tree-based indexes for RDMA. We have discussed different design alternatives regarding the index distribution and the RDMA-based access protocols. While the focus of this paper was on the NAM architecture which separates compute and memory servers, we believe that the discussions and findings can also help to understand the design space also for other distributed architectures in general. Furthermore, there are other important dimensions such as caching to improve the index performance. As mentioned before, we discuss our initial results for caching in Appendix 10.10.4. However, studying caching in detail is beyond the scope of this paper and represents an interesting avenue of future work.

Appendix

```
uint64_t readLockOrRestart(Node node){
  uint64_t version = awaitNodeUnlocked(node)
  return version
}

void readUnlockOrRestart(Node node, uint64_t version){
  if(version != node.version)
    restart()
```

```
}

void upgradeToWriteLockOrRestart(Node node, uint64_t version){
  if(!CAS(node.version, setLockBit(version)){
    restart()
}

void writeUnlock(Node node){
  fetch_add(node.version, 1)
}

uint64_t awaitNodeUnlocked(Node node){
  uint64_t version = node.version
  while (version & 1) == 1 // spinlock
    pause()

  return node.version;
}
```

Listing 10.3: Helper Methods for Coarse-Grained Index

```
Node remote_read(NodePtr remotePtr){
  return RDMA_READ(remotePtr)
}

uint64_t remote_readLockOrRestart(Node node, NodePtr remotePtr){
  uint64_t version = remote_awaitNodeUnlocked(node, remotePtr)
  return version
}

void remote_upgradeToWriteLockOrRestart(Node node, NodePtr remotePtr,
    uint64_t version){
  if(!RDMA_CAS(remotePtr, node.version, setLockBit(version)){
    restart()
}

void remote_writeUnlock(NodePtr remotePtr, Node node){
  if(node.right_node != NULL){ //node was split
    remNodePtr2 = RDMA_ALLOC(size(node.right_node))
    RDMA_WRITE(remNodePtr2, node.right_node)
  }
  RDMA_WRITE(remNodePtr, node);
  RDMA_FETCH_AND_ADD(remotePtr, 1)
}
```

```
uint64_t remote_awaitNodeUnlocked(Node node, NodePtr remotePtr){
  uint64_t version = node.version
  while (version & 1) == 1 // spinlock
    pause()
    node = RDMA_READ(remotePtr)

  return node.version;
}
```

Listing 10.4: Helper Methods for Fine-Grained Index

10.10 Appendix

10.10.1 Additional Index Operations

In the following, we show operations used to implement the indexing variants in Sections 10.3 to Sections 10.5. The operations in Listing 10.3 are used by the coarse-grained index (Section 10.3). These operations are called from compute servers by RPC and are then executed by memory servers. The operations in Listing 10.4 are used by the fine-grained index (Section 10.4) where compute servers use one-sided RDMA operations to access the index in the memory servers.

10.10.2 Latency of Index Designs

In this experiment, we analyze the latency of the different workloads using the same experimental setup and data as in experiment 1 (Section 10.6.1). Figure 10.13 and Figure 10.14 show the results for skewed and uniform data distribution. The x-axis again shows the number of clients used for each run and the y-axis the resulting latency for executing one instance of a query in a client.

A general pattern that we can see in all workloads is that the latency of the coarse-grained index is best for a low load ($<$ 20 clients). The reason is that this indexing scheme uses RPCs and as long as the memory servers are not becoming CPU bound, the latency in this indexing scheme benefits from using less round-trips between compute and memory servers for index lookups. However, for high load scenarios the fine-grained or the hybrid scheme show smaller latencies than the coarse-grained indexing scheme.

10.10.3 Effect of Co-location

In this experiment, we analyze the throughput when co-locating compute and memory servers in a NAM architecture. Co-location in the NAM architecture was also discussed in [20, 186, 239] and can be used to mimic a shared-nothing (like) architecture where data and compute is also co-located. The difference is that the memory of all nodes in a co-located NAM architecture is directly accessible not only via RPC but also via one-sided RDMA. The goal of this experiment was to show how the coarse-grained scheme behaves compared to the fine-grained scheme under co-location using typical workload characteristics (i.e., medium load and no skew).

In this experiment, we were running the same workloads as in Exp. 1 (i.e., the read-only workloads A and B) with data of size 100M. For running the workload, we used two NAM variants: one variant with and one without co-location of compute and memory servers. For the variant with co-location, we used 4 physical machines and each of the physical machines hosted a compute and a memory server each running 20 threads pinned to one of the sockets. For the variant without co-location, we used 4 additional dedicated physical machines for compute server, however only one socket of those machine was used for executing the compute server threads. Moreover, the memory servers were deployed on the other 4 machines. To simulate the same resources in the variant without co-location were we used in total 8 machines instead of 4, the compute and memory server used only one socket of each machine (instead of both). Moreover, in both variants (with and without co-location), each memory server used only one port of the RDMA NIC.

For running the workload, every client (i.e., each compute thread) selected the requested key (range) uniformly at random. The throughput results are summarized in Figure 10.15 for coarse- and fine-grained. We do not show the results for the hybrid scheme, which again behaved very similar to the coarse-grained for point queries and similar to fine-grained for range queries. As one effect, we can see that all workloads (point and range queries) have a similar relative gain independent when running in the co-located variant. The reason is that in both cases (coarse- and fine-grained), 25% of the memory accesses required by index lookups can be executed locally on the same physical machine where the compute server is running. For coarse-grained, for example, the complete index traversal can be executed locally, if the data requested by a compute server resides in a memory server on the same physical machine. For fine-grained, a compute server cannot execute a complete traversal locally but it can also use local memory accesses for those index pages that reside on the same physical machine with the compute server

which requests the data. This also results on average in 25% local accesses since we use 4 dedicated machines for the co-located variant.

This observation is similar to the observations made for co-location in [20, 239] where the authors also report that co-location enables a constant factor of higher throughput depending on the ratio of local/distributed transactions. Furthermore, when looking at the absolute throughput, as expected, under co-location point queries can achieve the highest throughput using the coarse-grained scheme. For range queries, the fine-grained scheme has still the highest throughput similar to what we saw in Exp. 1 (see Section 10.6.1).

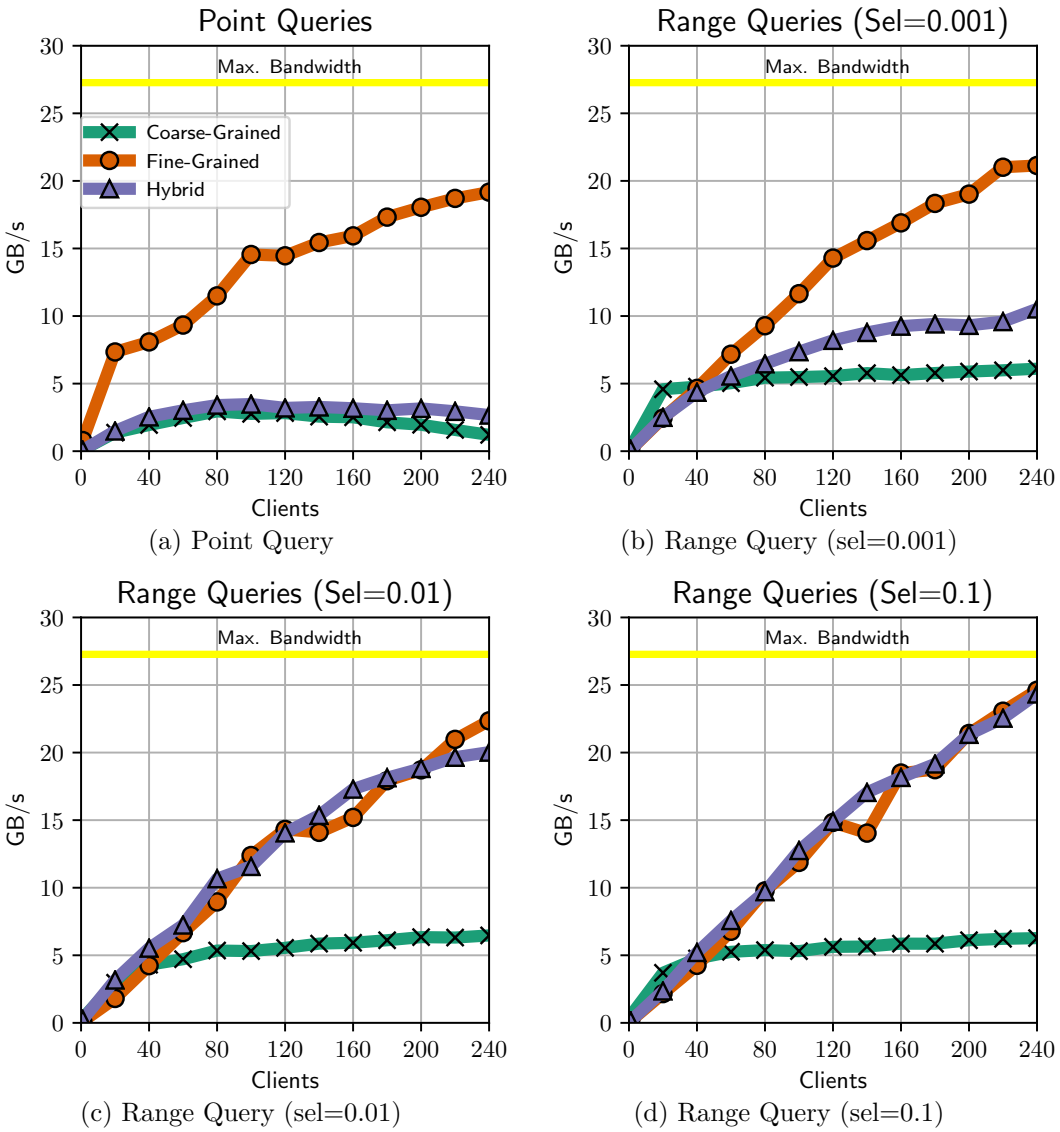
10.10.4 Opportunities and Challenges of Caching

An interesting dimension in the NAM architecture is caching of hot index nodes in compute servers that are frequently accessed. Caching allows compute servers to avoid remote memory transfers from memory servers. This is similar to the co-location of compute and memory servers as discussed in Appendix 10.10.3, which also allows compute servers to make use of locality. However, different from co-location, for caching, index nodes are replicated from memory servers to compute serves and thus it requires cache invalidation if the index in the memory servers is updated.

For read-only workloads, caching can thus help to avoid expensive remote memory accesses and significantly improve the lookup performance of tree-based indexes in a NAM architecture since no invalidation of cached data is required. We believe that especially the fine-grained scheme benefits from caching since it requires multiple round-trips between compute and memory servers to traverse the index. Furthermore, the other indexing schemes (coarse-grained and hybrid) can also benefit, especially for range-queries, since (potentially large) results do not need to be transferred fully from memory to compute servers anymore. However, for workloads which include writes (i.e., inserts and deletes of index entries), caching becomes a non-trivial problem since cached index nodes on compute servers need to be invalidated if the index on the memory servers changes.

The problem of cache invalidation has also been discussed in [27], which presents general caching strategies for remote memory accesses using RDMA. For tree-based indexes, where inserts and deletes might propagate up to the index from the leaf level to the root node, we observed that cache invalidation is even a more severe issue since one insert/delete operation might need to invalidate multiple cached index nodes. To that end, we believe that there is a need for developing new caching strategies that take the particularities of tree-based indexes into account to decide whether or not to cache an

index node. However, providing an in-depth discussion and analysis of different caching strategies for tree-based indexes that can adapt themselves to the workload is beyond the scope of this paper and presents an interesting avenue of future work.



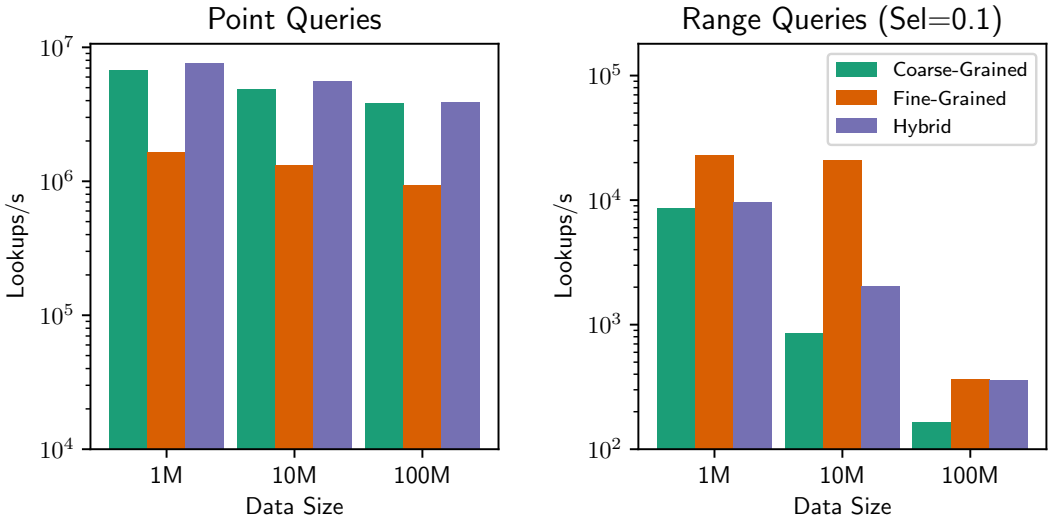
(a) Point Query

(b) Range Query (sel=0.001)

(c) Range Query (sel=0.01)

(d) Range Query (sel=0.1)

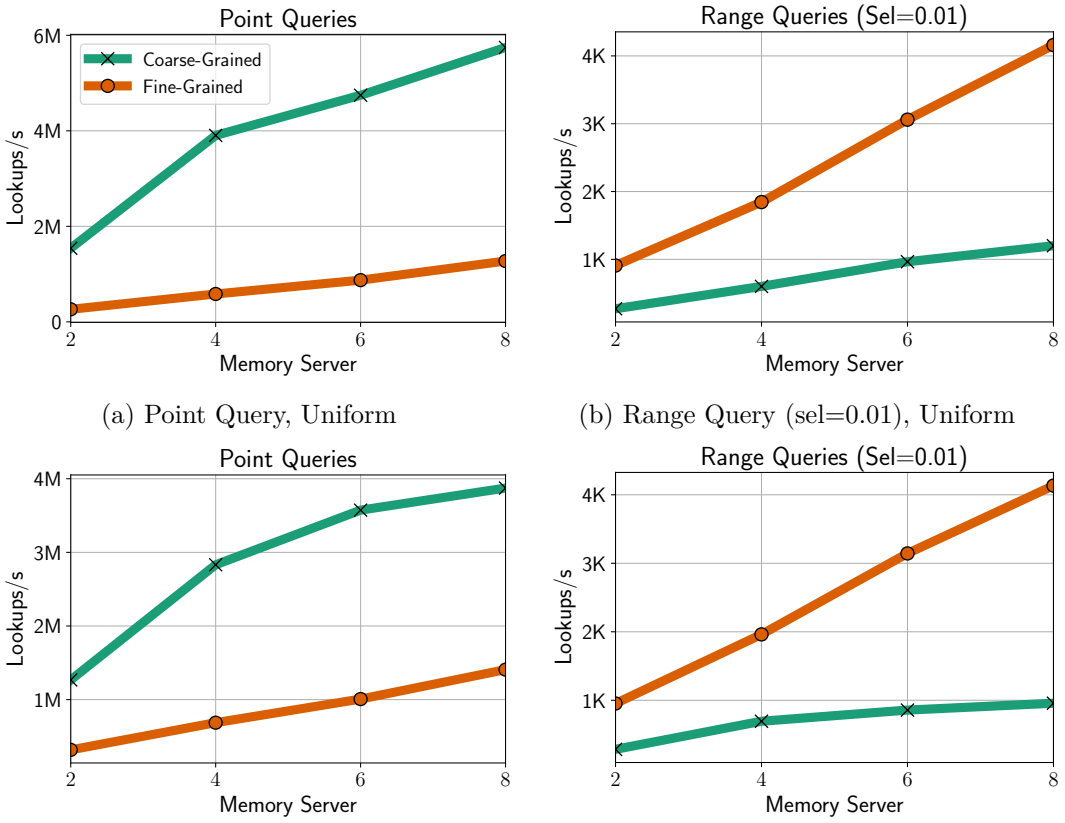
Figure 10.9: Network Utilization for Workloads A and B (Skewed Data, Size 100M)



(a) Point Query

(b) Range Query (sel=0.1)

Figure 10.10: Varying Data Size for Workloads A and B (Uniform Data, 240 Clients)



(a) Point Query, Uniform

(b) Range Query (sel=0.01), Uniform

(c) Point Query, Skew

(d) Range Query (sel=0.01), Skew

Figure 10.11: Varying # of Memory Servers for Workloads A and B (Size 100M, 120 Clients)

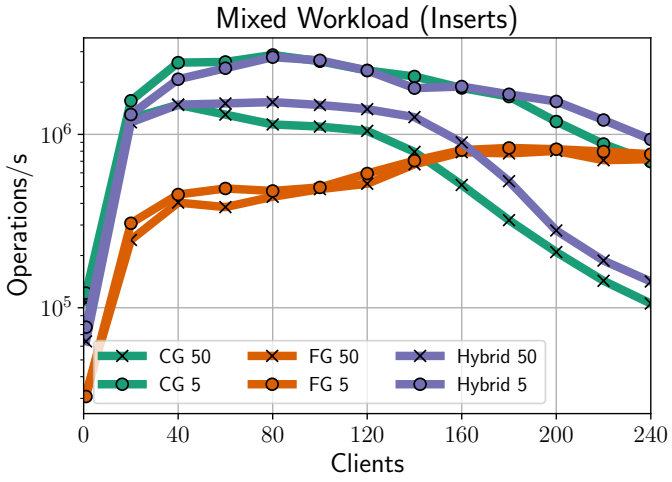
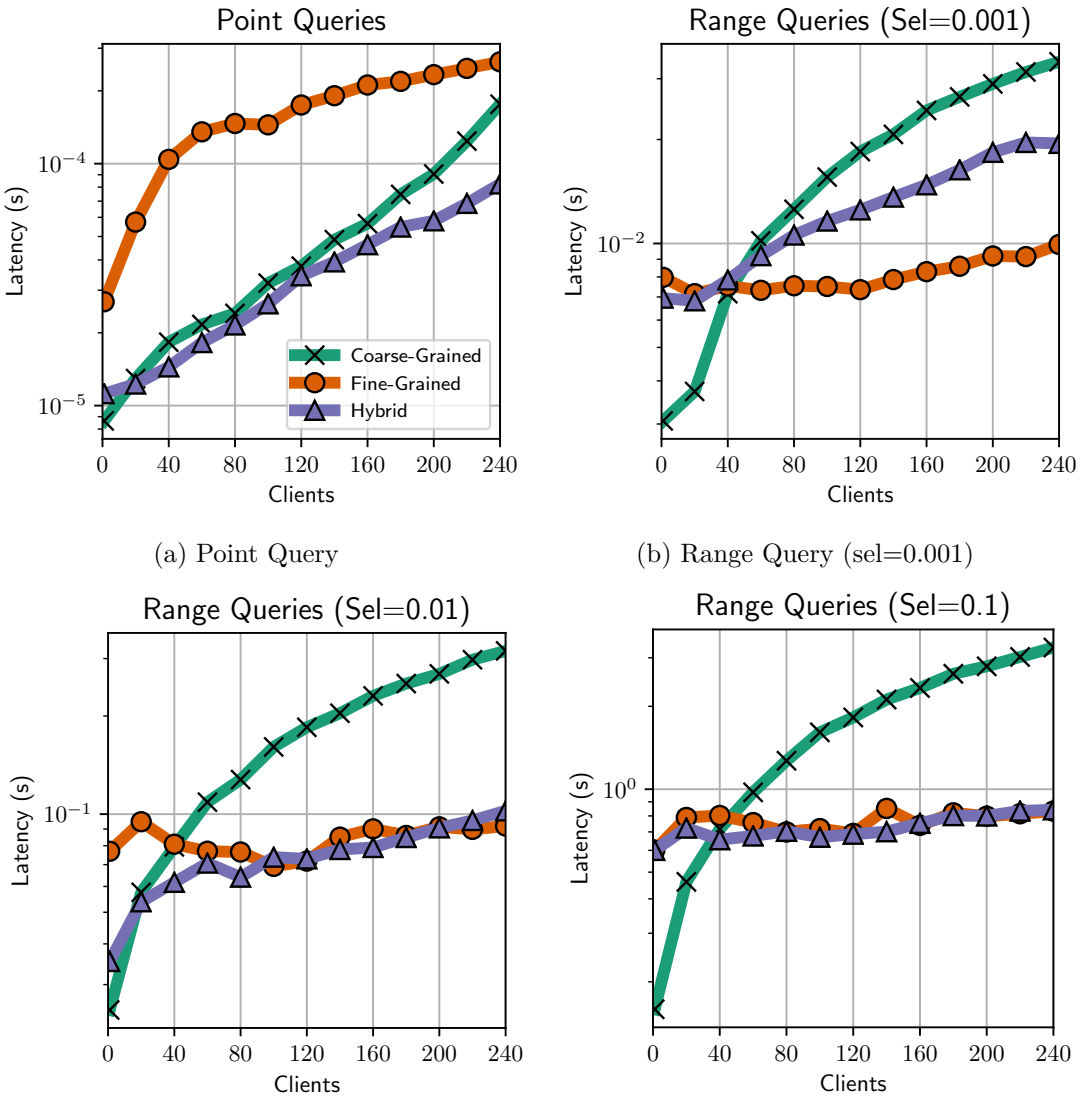


Figure 10.12: Throughput for Workloads C & D with Inserts (Uniform Data, Size 100M)



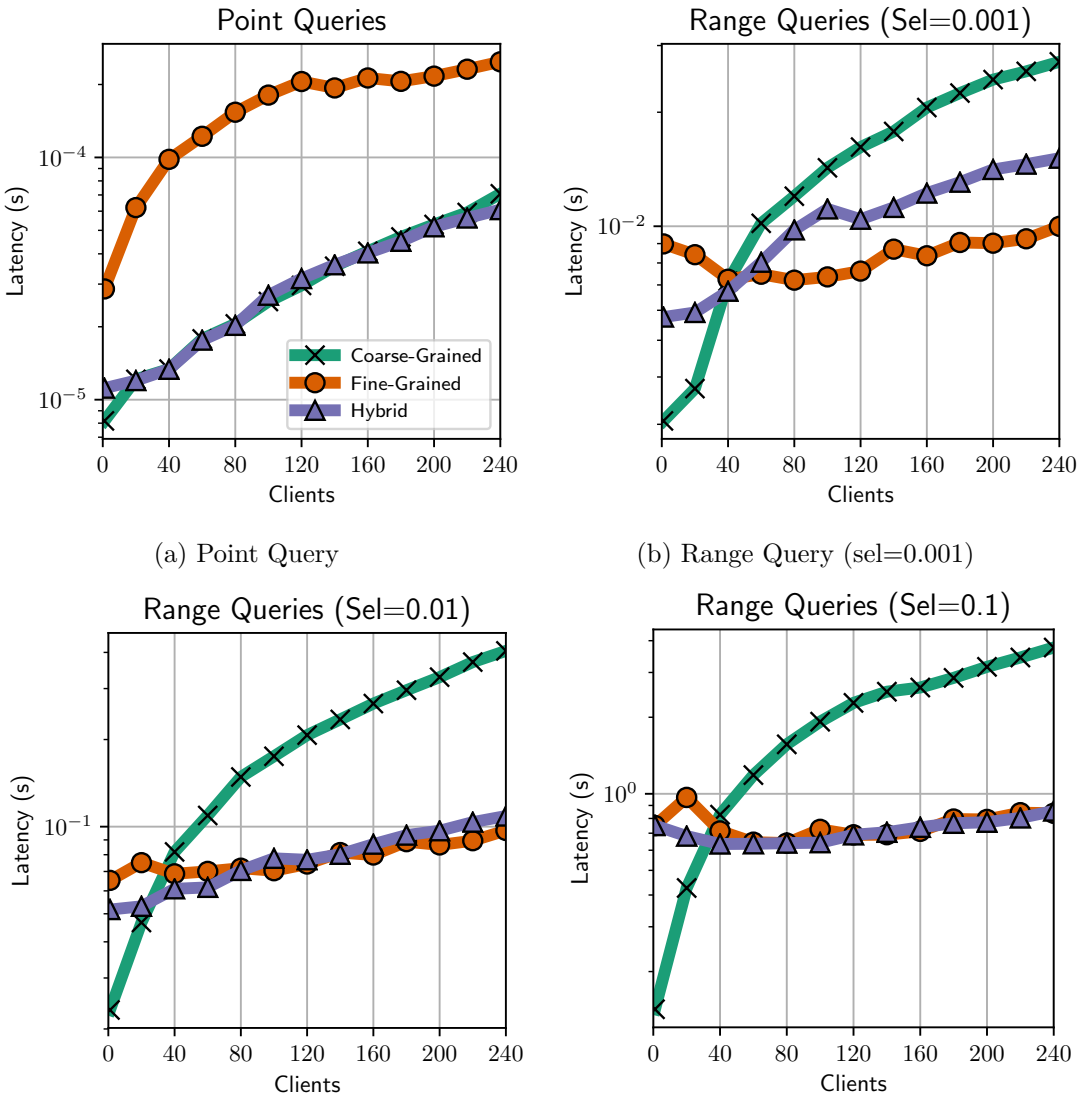
(a) Point Query

(b) Range Query (sel=0.001)

(c) Range Query (sel=0.01)

(d) Range Query (sel=0.1)

Figure 10.13: Latency for Workloads A and B (Skewed Data, Size 100M)



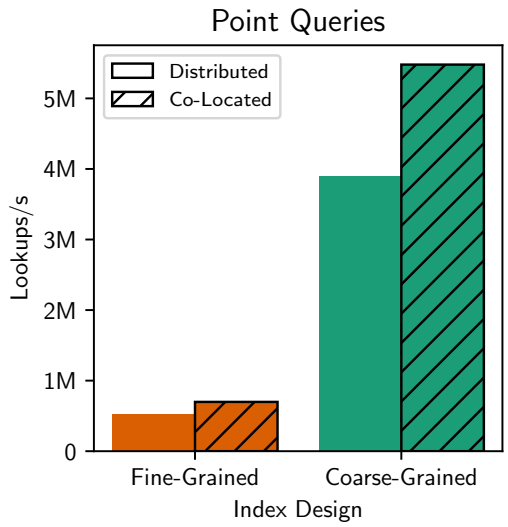
(a) Point Query

(b) Range Query (sel=0.001)

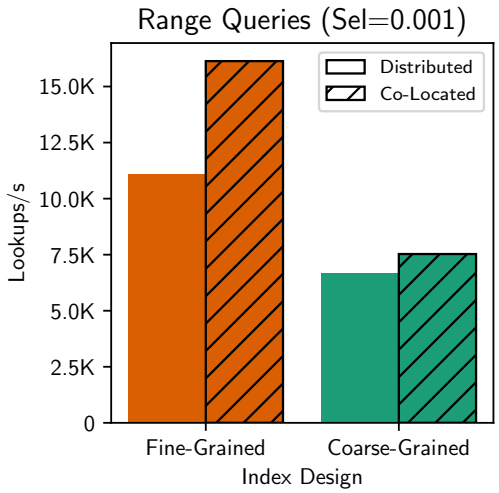
(c) Range Query (sel=0.01)

(d) Range Query (sel=0.1)

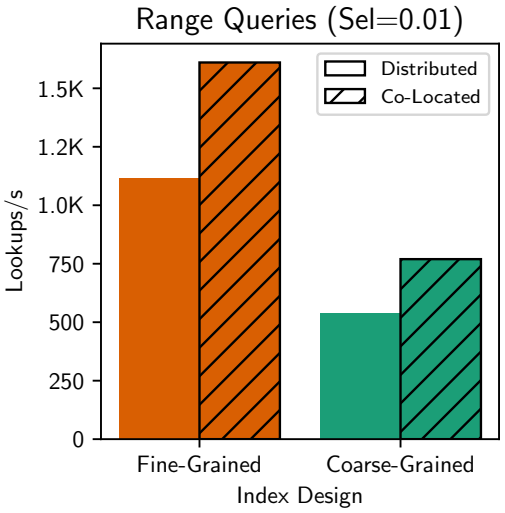
Figure 10.14: Latency for Workloads A and B (Uniform Data, Size 100M)



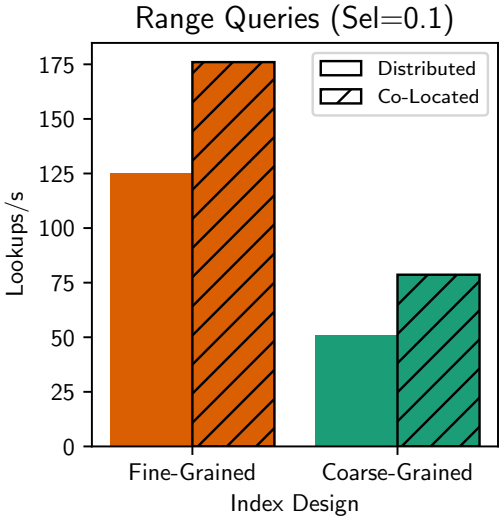
(a) Point Query



(b) Range Query (sel=0.001)



(c) Range Query (sel=0.01)



(d) Range Query (sel=0.1)

Figure 10.15: Effects of Co-location on Throughput (Uniform Data, Size 100M, 80 Clients)

11 Design Guidelines for Correct, Efficient, and Scalable Synchronization Using One-Sided RDMA

Abstract

Remote data structures built with one-sided Remote Direct Memory Access (RDMA) are at the heart of many disaggregated database management systems today. Concurrent access to these data structures by thousands of remote workers necessitates a highly efficient synchronization scheme. Remarkably, our investigation reveals that existing synchronization schemes display substantial variations in performance and scalability. Even worse, some schemes do not correctly synchronize, resulting in rare and hard-to-detect data corruption. Motivated by these observations, we conduct the first comprehensive analysis of one-sided synchronization techniques and provide general principles for correct synchronization using one-sided RDMA. Our research demonstrates that adherence to these principles not only guarantees correctness but also results in substantial performance enhancements.

Bibliographic Information

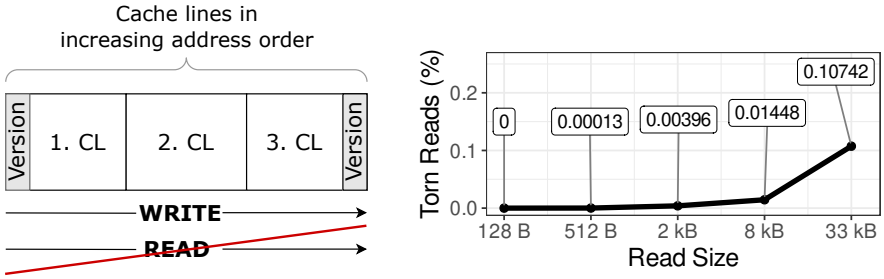
The content of this chapter was previously published in the peer-reviewed work Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. "Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA." in: *SIGMOD '23: International Conference on Management of Data, Seattle, WA, USA, June 18 - 23, 2023*. ACM, 2023. The contributions of the author of this dissertation are summarized in Chapter 5.

11.1 Introduction

RDMA & disaggregated databases. Remote Direct Memory Access (RDMA) has quickly become one of the indispensable tools for building disaggregated database systems. Not only does RDMA provide single-digit microsecond network latencies, but it also provides efficient primitives for remote memory access. In particular, RDMA's *one-sided verbs* allow a compute server to read or write directly to a remote memory server while bypassing the remote CPU. Since memory servers frequently possess near-zero computational capacity [224] and most computational power is concentrated within the compute layer, one-sided RDMA proves to be well-suited for disaggregated DBMSs. Consequently, recent literature has explored how disaggregated DBMSs can leverage one-sided verbs [55, 57, 111, 134, 210, 227, 239, 241, 246], B-Trees [223, 256], or SkipLists [142] which enable efficient access to remote data. But because one-sided operations bypass the remote CPU, traditional storage server-side synchronization techniques where the remote CPU is in charge[1] do not work. Instead, various one-sided synchronization techniques have been proposed [55, 150, 233, 256]. Those techniques can be categorized into *pessimistic* and *optimistic* schemes. While pessimistic schemes *prevent* concurrent modifications, optimistic schemes *detect* (and handle) concurrent modifications. These approaches fundamentally differ in their scalability and performance characteristics.

Performance is key. Remote data structures may need to serve thousands of clients connecting from several compute servers. With such a high degree of concurrency, the performance depends on how well the implemented one-sided synchronization scheme performs. While individual papers have proposed various one-sided synchronization schemes [55, 149, 256], it is surprising that there has not yet been a systematic study of these schemes under comparable workloads and conditions. This paper provides the first in-depth performance analysis. We show that small design choices when implementing a scheme can severely impact its performance and lead to performance bottlenecks. For example, contrary to expectations, data alignment hinders the scalability of pessimistic one-sided latches, even in uncontended workloads. In fact, if not carefully implemented, the performance for an uncontended workload can be as dismal as that for a highly contended one. To this end, this paper proposes design principles to mitigate those pitfalls and presents several optimizations that improve the performance of a well-known disaggregated RDMA-optimized DBMS [239] by 2×.

[1] In this paper, *synchronization techniques* refer to the low-level synchronization mechanism, i.e., latching, not higher-level concurrency control.



(a) Faulty Optimistic Scheme (b) Percentage of Torn Reads

Figure 11.1: Incorrect optimistic synchronization

Correctness is hard. Achieving high performance in synchronization is unquestionably valuable, but ensuring correctness is mandatory. We have discovered that early techniques proposed in the literature fail to accurately synchronize concurrent operations, potentially resulting in hard-to-detect data inconsistencies. For example, consider an optimistic synchronization scheme as implemented in [150] (shown in Figure 11.1a), which presumes that RDMA operations occur in ascending address order — a common assumption in many papers. This scheme implements an update by writing the head version first, then the data modification, and eventually updating the tail version. Under the assumption of operations being performed in increasing address order, a concurrent reader can detect concurrent modifications by comparing the head and tail versions.

Incorrect assumptions. Unfortunately, in contrast to the general assumption in many papers [150, 223], RDMA reads are not guaranteed to be performed in increasing address order. In fact, the RDMA specification does not state the ordering within a single RDMA read. As a result, in the previously mentioned synchronization scheme, an RDMA read may first read both versions (i.e., the first and third cache line) and then retrieves the data from the second cache line. At the same time, a writer concurrently modifies the data in address order. The concurrent data update may not be detected because the versions were read first before the concurrent writer started. However, the reader and the writer overlapped at the second cache line leading to inconsistent data.

We validate the existence of this behavior with a simple experiment consisting of one storage node and two compute nodes. A single-threaded remote writer on one compute node repeatedly fills a block of its local memory, e.g., 512 bytes, with the same 8-byte version number and then writes it to a remote buffer (50 MB) on the storage node with a single RDMA write. The version number is incremented on each iteration, and the new block is written to the next slot in the buffer. Concurrently a reader on the other compute node reads a block in the remote buffer with a single RDMA read and then checks whether the header and footer version numbers are identical. If the header and

footer version numbers match, then the intermediate values are examined to determine if an inconsistency exists. "Torn" reads – having an identical header and footer version but inconsistent intermediate values – are *undetectable* by a validation scheme that checks the block's leading and trailing version numbers.

Figure 11.1b shows the percentage of how many such torn reads are undetectable due to changes in the read order. Note that no torn read appears with 128 bytes as only the header and footer cache lines are read, i.e., if they are inconsistent, this can be detected. However, inconsistencies happen for more than 128 bytes, and while not frequently, often enough to corrupt the data. Surprisingly, this problem is not widely known, and techniques that assume ordering are still very popular [223]. We believe the main reason for this assumption is that a single RDMA request requires many protocols — not only RDMA but also PCIe, and cache coherence — to work in concert. Thus, it is challenging to understand which guarantees are provided by the respective specification.

Contribution. The primary goal of this paper is to distill general principles for correct one-sided synchronization techniques. To our knowledge, this paper is the first principled analysis comparing the performance, scalability, and correctness of one-sided synchronization techniques. Our work demonstrates that understanding the specification and low-level hardware details is crucial for correct and efficient synchronization. Our underlying goal is to provide researchers and developers with guidelines on how and when to use the different synchronization techniques. Finally, we open-sourced benchmarks[2] that help to transfer our findings to different hardware setups and future developments.

11.2 Background and Methodology

While many systems [55, 149, 150, 256] implement various synchronization schemes, they do not isolate the impact of their synchronization on the overall performance. However, as we will show, synchronization techniques significantly affect scalability and system performance. This section describes the necessary background on RDMA and the typical RDMA hardware stack, existing synchronization techniques, and our experiment methodology.

11.2.1 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is a networking protocol that provides high bandwidth and low latency access to a remote node's main memory [180], using zero-copy

[2] https://github.com/DataManagementLab/RDMA_synchronization

transfers from the application space. Several RDMA implementations are available – most notably InfiniBand [91], RDMA over Converged Ethernet [3] (RoCE) [92], and iWARP [180]. RDMA offers four transport types: reliable or unreliable, which can be connected or unconnected. This paper focuses on the *reliable connected* transport as it is the only configuration that fully supports one-sided primitives.

RDMA one-sided verbs. RDMA implementations provide two communication paradigms (called verbs) (1) *two-sided* and (2) *one-sided* verbs. Two-sided verbs are similar to traditional socket-based programming in that both sides (sender and receiver) are involved. In contrast, one-sided verbs (read, write, and atomics) provide remote memory access semantics, in which a process specifies the memory address of the remote node that should be accessed. The CPU of the remote node is not actively involved in the data transfer, i.e., only one side is involved. In this paper, we solely focus on one-sided primitives. RDMA read and write enable applications to read and write remote memory directly without remote CPU involvement. To support highly concurrent applications, RDMA specifications provide atomic operations [91, 194]. One-sided compare-and-swap (CAS) and fetch-and-add (FAA) operations atomically read, modify, and write memory at a remote destination. Those operations work similarly to local CPU CAS and FAA instructions. CAS atomically swaps the current value with a new one if it equals the expected value. FAA increments the current value with some user-defined value and then returns the original value to the caller. RDMA atomics are limited to 64-bit, 8-byte aligned values.

RDMA two-sided verbs. Two-sided verbs are widely used in high-performance RDMA systems [103, 104, 231] to send messages between distributed processes. The sender issues an RDMA send request, which consumes a waiting RDMA receive request at the destination. The receiving process issues the receive request to dictate the destination address for the sent payload. After the payload is written, the receiving process is notified of its arrival. Since the receiving process is explicitly involved in the communication, synchronization of remote processes is managed using request handlers and traditional multiprocessing approaches. In contrast, disaggregated memory systems leveraging one-sided verbs must synchronize processes directly through remote memory access, which requires careful consideration due to the behaviors we highlight in this paper. In addition, two-sided RDMA requires explicit processing on the storage side, which is typically not ideal with limited computational resources on the storage servers. Therefore, two-sided

[3] RoCE is an attempt to combine RDMA with Ethernet. Refer to [84] for the shortcomings of RoCE in modern datacenters.

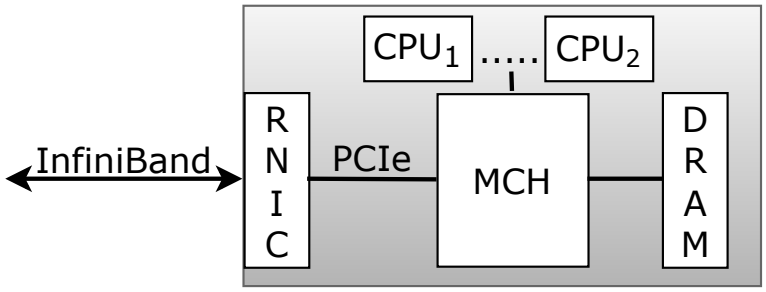


Figure 11.2: Hardware components involved in RDMA

RDMA is not in the scope of this paper; however, several works do explore the relative merits of one-sided vs. two-sided RDMA [102–104, 231, 256]

Interface. Modern network interfaces typically provide asynchronous networking. This means that a network operation is dispatched to the NIC, which notifies the application once the operation is completed. RDMA's interface uses so-called send/receive queues to post operations: (1) While a *send queue* is used by the requester to issue operations such as read, write, send as well as atomics (2) the *receive queue* is used by the responder to issue receive requests. With RDMA, a connection between a requester and a responder bundles these two queues and is therefore called queue pair (QP). More precisely, the application must create queue pairs on both ends and connect them to initiate a connection between a requester and a responder. To issue RDMA operations, a host creates a *work queue element* (WQE). The WQE specifies parameters such as the verb and other metadata (e.g., the remote target address). The requester then adds the WQE to its send queue and informs the local RDMA NIC (RNIC) via Programmed IO (PIO) to process the WQE. For a signaled WQE, the local RNIC pushes a completion event into the *completion queue* (CQ) via a DMA WRITE once the remote side has processed the WQE. To enforce synchronous communication, the application can also block until the network card generates the completion event. Typically, RDMA is used asynchronously, meaning multiple WQE are simultaneously registered with the RNIC, and later the application checks for completions in the CQ. This technique is often referred to as *doorbell batching*.

11.2.2 RDMA Hardware Stack

As mentioned in Section 11.1, synchronization techniques rely on certain hardware guarantees; thus, we briefly present the required hardware in Figure 11.2. An RNIC connects via the PCIe bus to the memory controller hub (MCH) of a multi-core CPU [171]. The MCH is responsible for handling memory access by both the CPU and peripheral devices. Among others, the MCH contains the memory controller, the coherency engine,

	Variant	Ops.	Systems	Correct
Pess.	Reader/Writer	2	[29], [233], [239]	Yes
	Exclusive	2	[221], [36],[158]	Yes
Opt.	CRC	2	[200],[257],[149],[214]	Prob.
	Versioning	2	[256], [150], [223]	No†
	Cache line ver.	2	[55], [230]	Yes

† can be fixed with an additional RDMA read (see Section 11.4.3)

Table 11.1: Pessimistic and optimistic techniques for one-sided RDMA synchronization.

and acts as the root complex for the PCIe bus. Because modern server architectures implement cache coherent I/O, e.g., Intel DDIO [93] and ARM CCI [7], knowing that an RDMA access to system memory will snoop (lookup) CPU cache for conflicting addresses is essential. If a conflicting address is found, it can be served from the CPU cache; otherwise, it is fetched from main memory. Cache coherent I/O is a core enabler for many of the techniques described in Section 11.4.

11.2.3 Existing Synchronization Techniques

The one-sided synchronization techniques of modern distributed systems can be categorized into pessimistic and optimistic approaches, as shown in Table 11.1. Pessimistic approaches mirror traditional latching with the distinction of having only a single latch mode or two for reader-writer support; meanwhile, optimistic approaches can be further subdivided. Each optimistic technique offers different characteristics, like memory and computation overhead, but all embed metadata in the object, which is updated by writers and validated by readers. We now briefly introduce existing optimistic techniques from Table 11.1: *CRC*, *versioning*, and *cache line versioning* before we discuss them in more detail in Section 11.4.

Checksums. A common detection technique used to identify potential inconsistencies [149, 200, 257]. After updating the object, writers write back a new checksum (typically a 64-bit CRC). Readers then recompute a checksum locally and compare it to the observed checksum. While effective in practice, there is still a non-zero probability of a collision causing validation to succeed on corrupted data.

Versioning. Versioning is a strategy that augments the data with a single version. During execution, writers increment the version before and after updating the data. Readers read the version before and after their operation to validate whether an update occurred concurrently. Optimistic lock coupling is a special case of versioning in which sequence locks are used for both write-write synchronization and validating reads (e.g.,

[119, 256]). As we will explain in Section 11.4.3, there are pitfalls with this approach that impact its correctness.

Cache line versioning. Finally, cache line versioning maintains the object's version in each constituent cache line [55]. Writers increment all versions before updating the data, then increment them upon completion. Unlike versioning, readers detect inconsistencies with a single remote read by validating that all versions match.

To understand the differences between the sundry approaches, we perform a principled analysis and evaluation of one-sided synchronization techniques. To the best of our knowledge, this paper is the first to do so.

11.2.4 Evaluation Methodology

Framework. We implemented all techniques in the same code base to isolate the fundamental properties of the synchronization schemes from incidental differences. We highlight each synchronization scheme's overhead using a perfectly-sized remote hash table to avoid hash collisions. In addition to the hash table, we use a remote B-Tree to show how the synchronization schemes behave when contention is inherent to the data structure due to its hierarchical nature (all workers start their traversal at the root node). We use multi-threading and execute one operation on a single thread (worker) until completion, i.e., no batching or asynchronous execution. This allows a fair comparison of the approaches.

Setup. We conducted all reported experiments on an 8-node cluster running Ubuntu 18.04.1 LTS, with a Linux 4.15.0 kernel. All nodes are connected to a SB7890 InfiniBand switch using one Mellanox ConnectX-5 MT27800 RNICs (InfiniBand EDR 4x, 100 Gbps) per node. Each node has two Intel Xeon Gold 5120 CPUs (14 cores) and 512 GB main-memory split between sockets.

Since the ConnectX-5 is connected to one NUMA socket, we inevitably have NUMA effects when using more than 14 cores (i.e., 28 threads). We allocate the memory on the socket of the NIC, and to alleviate the NUMA effects, we assign threads round-robin (interleaved) to both sockets. More details of NUMA effects on RDMA can be found in [160, 181]. Besides ConnectX-5, we also validate our results for different generations of RNIC, namely, ConnectX-3 and ConnectX-6. Mellanox is widely used; from 30 papers, we recently analyzed 28 used Mellanox cards. Besides on-premises, only Microsoft offers RDMA of the top three cloud providers and they use Mellanox cards in their instances. [107]. That being said, we believe that most findings are independent of the network card. The correctness discussion mainly depends on the protocol specifications

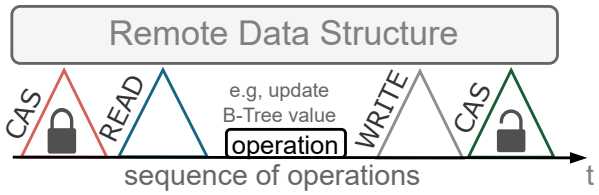


Figure 11.3: Example of an exclusive latch acquisition.

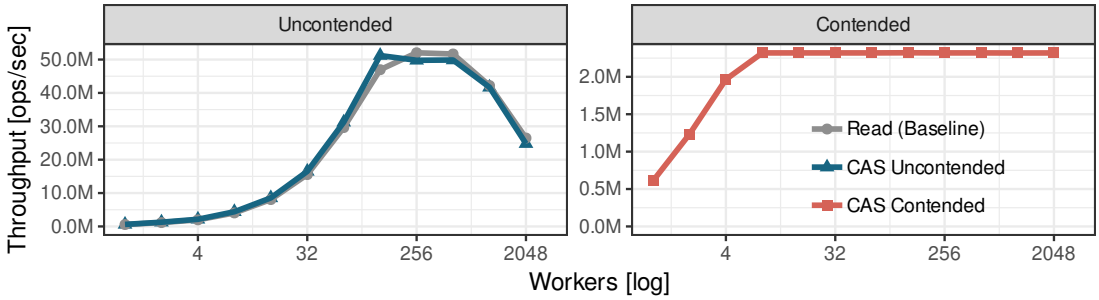


Figure 11.4: Scalability of contended and uncontended atomic RDMA operations when increasing the number of workers (4 compute nodes and 1 storage node)

underpinning the RDMA communication. In particular, our testbed leverages the widely-used InfiniBand specification [91] which shares commonalities with alternative RDMA protocols and make our results applicable to a broader range of deployments. The performance considerations shed light on possible pitfalls worth investigating when building an RDMA application. We open-source our benchmarks to help developers uncover performance and correctness bottlenecks in other RDMA system configurations.

11.3 Pessimistic Synchronization

In order to prevent concurrent modifications of remote data structures such as a B-tree or a hash table, one-sided pessimistic synchronization techniques implement latches using one-sided RDMA operations. In this section, we present the basic implementation of such a latch and use it as a running example to discuss possible optimizations. Since RDMA atomics are the fundamental building block of these one-sided latches, we will then drill down into the performance and scalability characteristics of these RDMA primitives. Afterward, we outline and evaluate possible optimizations for one-sided latches.

11.3.1 Running Example of a Pessimistic Latch

Table 11.1 shows that existing pessimistic schemes are subdivided into two types of latches: While some latches such as RCC-NOWAIT [221] support only one latch mode (latched and unlatched), others distinguish between shared and exclusive modes, i.e., reader/writer latches. Both latch types can be implemented with atomic RDMA operations. We will use a reader/writer latch for the running example, but all optimizations generalize to both latch types.

We implement a typical reader/writer latch using an 8-byte value [141, 233]. A worker uses an RDMA compare-and-swap (CAS) operation to set the latch bit (usually the trailing bit) for exclusive access. Readers increment the reader count, encoded in the remaining bits, with a fetch-and-add (FAA). In the basic variant, all operations are executed synchronously, i.e., the worker blocks after every operation until its result is returned.

Figure 11.3 gives an intuition on how this latch is used to access a remote data structure exlusively. First, the remote data structure is latched with an RDMA CAS operation on the 8-byte latch by setting the lock bit. Afterward, the desired data is read from the data structure, modified locally, and written back with an RDMA write operation. Finally, the remote data is unlatched with another RDMA CAS operation on the 8-byte latch.

To access the remote data structure in shared mode, the clients use RDMA FAA to increment the reader count of the latch speculatively. The return value (i.e., the full 8-byte) of the operation allows the worker to check if the latch is in the exclusive mode, in which case the worker decrements the reader count and retries. Otherwise, the worker reads the data using an RDMA read and then decrements the counter to unlatch.

11.3.2 Performance of RDMA Atomics

Because every pessimistic approach relies on RDMA atomics (CAS and FAA), it is important to understand their isolated performance before discussing how our basic latch can be optimized.

Uncontended vs. contended RDMA atomics. In the first experiment, we examine the scalability behavior of contended and uncontended RDMA atomics. Both scenarios are equally vital, and while heavy contention is typically rare, it is unavoidable in some workloads, e.g., having hot tuples. To show the effect of contention, we perform an experiment in which all workers issue an RDMA CAS operation of the same 8-byte atomic counter. To understand how uncontended atomics scale, we assign a private

8-byte remote atomic counter to each worker. For reference, we compare uncontended atomics to the performance of an RDMA read.

Figure 11.4 shows the scalability behavior of both uncontended and contended RDMA atomic operations when increasing the number of workers. As we can observe, uncontended atomics scale like one-sided RDMA read operations, peaking at around 51.2 million operations per second with 128 workers. This suggests that the parallel uncontended atomic requests do not interfere, but we will demonstrate later that this does not hold for all scenarios. Note that the performance drop of uncontended atomics at 512 workers has nothing to do with the atomic operation but can be attributed to QP-thrashing [55] on the client machines. QP-thrashing means that the QP state cannot be cached on the RNIC and is swapped in and out to host memory. This occurs at around 128 utilized parallel QPs per client NIC in our hardware, i.e., 512 workers.

As expected, when running the contended atomic workload, the peak is significantly lower at 2.32 million operations per second and atomic operations only scale until 8 workers.

In the literature, RDMA atomics seem to have a bad reputation for being fundamentally unscalable [102] – even for uncontended workloads. However, the above experiments demonstrate that uncontended atomics scale well w.r.t. the number of workers. In fact, they show comparable scalability to RDMA read operations. While this experiment offers valuable insights into the scalability of atomics, it is not the complete picture, as we will demonstrate.

Atomic stride size and alignment. Obviously, the scalability of RDMA atomics depends on the degree of parallelism. However, subtle details such as the data alignment can also interfere with the scalability. So far, in our experiments, values are placed in a 64-byte stride, i.e., a cache line. We only used the first 8 bytes for the atomic counter and ignored the remaining 56 bytes. However, in practice, RDMA atomics are placed at larger strides as they protect data of various sizes, e.g., a 4 KB B-Tree node.

In the following experiment, we measure the effect of larger stride sizes by varying the distance between the atomic counters. As in the previous uncontended experiment, each worker has a private latch to avoid contention. Consequently, the expected outcome should be similar to the uncontended result in Figure 11.4. Surprisingly, Figure 11.5 shows that the stride size impairs the scalability tremendously. That is, with larger stride sizes, the inflection points w.r.t. throughput (highlighted in red) are reached earlier. With a 64-byte stride, the peak performance is $50M$ operations with 128 workers, i.e., the same upper-bound as in Figure 11.4. With a 256-byte stride, the peak performance is at $40M$ operations with 128 workers. With a 512-byte stride, the peak performance is

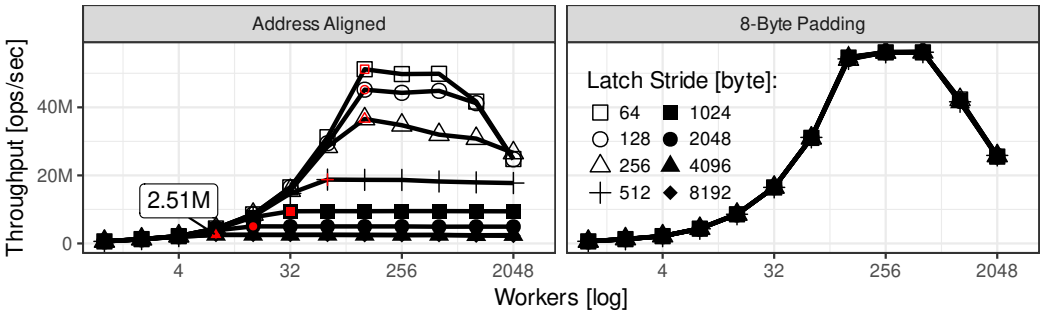
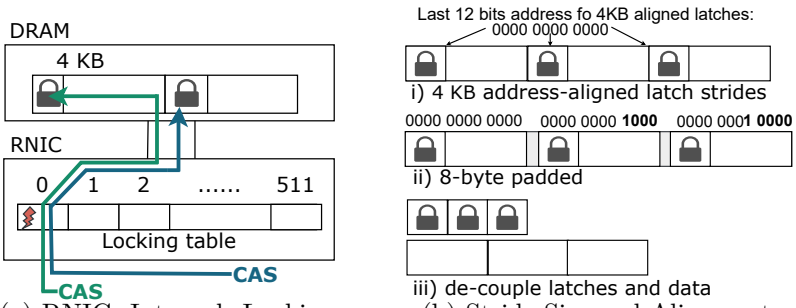


Figure 11.5: Scalability of uncontended atomics with varying strides between the latches (4 compute nodes and 1 storage node).

halved and reached with fewer workers ($20M$ operations with 64 workers). Remember that there is no true latch contention, and we only vary the distance between the atomic counters and nothing else. The observed behavior must be based on a physical contention point in the RNIC.

NIC internals - physical contention. Through reverse engineering, we believe that the RNIC uses an internal locking table to serialize atomic operations. Since the locking table works similarly to a hash table, collisions can happen. Unrelated atomic operations can be assigned to the same slot, severely limiting the throughput of concurrent uncontended atomic operations. The lock slots are determined based on the last 12 bits of the atomic operation's target address. For instance, if we use a 4 KB-aligned address as illustrated in Figure 11.6b i), the last 12 bits of the address are zeros, and they are assigned to the same lock slot. Even though the atomics do not contend on the same latch, the way the RNIC handles atomics results in physical contention inside the locking table, as exemplified by the arrows in Figure 11.6a. The two CAS operations target different latches and are still serialized in the same lock slot, negatively impacting performance. Consequently, logically uncontended atomic operations do not scale very well when the data alignment is not carefully chosen, as shown in Figure 11.5. When we compare the results of our initial contended scalability experiment from Figure 11.4 with the performance of 4 KB-aligned stride sizes, we can see that the performance and scalability characteristics are very similar. Given the underlying hardware mechanism, this performance is now explainable: the operations are serialized in the same lock slot, whether through a collision of the address or the operations targeting the same latch. We validated the existence of a performance signature matching our hypothesized 12-bit lock table in three RNIC generations: ConnectX-3, ConnectX-5, and ConnectX-6 RNICs. Also, Kalia et al. [102] observed similar findings for an older network card (Connect-IB).



(a) RNIC Internal Locking Table

(b) Stride Size and Alignment

Figure 11.6: Pessimistic synchronization performance can be impacted by RNIC architecture and by host memory layout.

However, it is hard to generalize our findings to all NICs since the implementation details are not made publicly available by the RNIC vendors. Therefore, like other papers, we can only infer implementation details [102, 223]. Instead, we emphasize that NIC hardware details are essential and demonstrate potential bottlenecks that should be carefully evaluated when building a high-performance system.

Improving scalability of uncontented atomics. To improve the scalability of uncontended operations, we must avoid collisions in the locking table. The only way one can control this is through the data layout, i.e., the addresses of our latches. The goal is to place the latches so that the last 12 bits (used for the lock-slot calculation) are different. Consider the example in Figure 11.6b ii); instead of allocating the 4 KB blocks back-to-back, a padding of 8-byte is placed before the latches. Now, the last 12 bits of the latch addresses are not all zeros and, thus, will be assigned to different lock slots. The effect of this mitigation technique is illustrated in Figure 11.5. We can observe that all stride-sizes scale equally well (all measurements happen to be on the same line). Another possibility to better utilize the lock-slots is to decouple latches from the data as depicted in Figure 11.6b iii). In this technique, the latches are placed back-to-back. Since the latches are only 8 bytes, the last 12 bits of the latch address will vary and achieve the same scalability behavior as for the 8-byte padding. Note, in this experiment, we test the performance of the atomic operations, i.e., we do not read the data. This allows us to isolate the atomic contention effect, but in practice separating the latch from the data may have adverse effects due to locality. In particular, the translation from physical-to-virtual memory could suffer as every data access invokes two translations, i.e., one for the latch and one for the data. However, this depends on other parameters, such as the working set, tuple size, and NIC-cache size [102], and thus requires a holistic evaluation.

To conclude, despite contrasting beliefs, RDMA atomics can scale well w.r.t. the number of workers. However, the scalability depends on the number of concurrent accesses (contention) and the data alignment. While the first is workload-dependent, the latter can be carefully tuned to best utilize the internal RNIC hardware. While we demonstrate that padding is beneficial for RDMA atomics, it can also have consequences elsewhere, such as making page table lookup less efficient. Therefore, we argue that it is crucial to understand the internals of the RNIC and holistically optimize the DBMS system.

11.3.3 Optimized Pessimistic Latches

Equipped with our findings on how to utilize atomics optimally, we can now focus on how to design optimized pessimistic latches. Recall that the basic latch variant executes all operations synchronously, as illustrated in Figure 11.3. After every operation, the completion is awaited by polling the completion queue (cf. Section 11.2). While this is certainly correct, it is inefficient and increases the per-operation latency.

Latch optimizations. To reduce the operation latency, many systems use optimizations. Unfortunately, those optimizations are often only briefly discussed by the authors. We compiled a list of existing optimizations following numerous small hints in related work and a careful study of published source code. This is the first paper that describes these optimizations in detail and discusses why those optimizations guarantee correctness. In the following, we present those optimizations and highlight possible pitfalls.

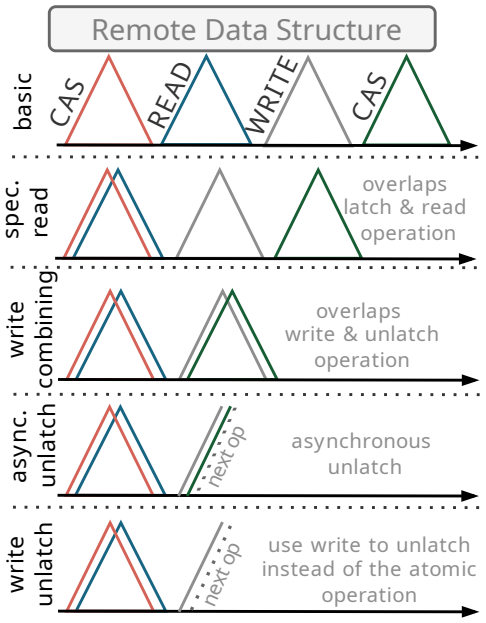


Figure 11.7: Evolution of exclusive latch optimizations.

The *speculative read* optimization overlaps the latch with the read operation (cf. Figure 11.7) to hide the latency of the read. If the latch request is successful, we can proceed, and if it is unsuccessful, the latch request is restarted. However, the correctness of this optimization relies on the order of the operations. That is, it must be

guaranteed that the latch operation takes place before the read happens. Fortunately, RDMA atomics are executed prior to subsequent RDMA operations on the same QP as per the InfiniBand specification [91].

Similarly, *write combining* overlaps the write and the unlatch operation (CAS) as illustrated in Figure 11.7. The intuition is that the unlatch operation can mask the write latency. This optimization is only applicable for exclusive access since read-only operations do not update the data and thus do not involve an RDMA write.

The *asynchronous unlatch* optimization is an optimization that goes even further and does not wait for the last completion event synchronously and thus immediately processes the next operation. In contrast to the synchronous variants, however, the memory buffer containing the write's payload cannot be reused immediately for the next operation. The issue is that when immediately re-using the buffer, the data from the previous operation could be overwritten as the previous RDMA write may still be outstanding; as such, we need to ensure that the operations are finished before re-using the buffers. The typical solution to avoid overwriting in-flight buffers is to use multiple buffers per QP. For instance, if the worker wants to modify two tuples, then the first tuple is written to the first (local) buffer. Subsequently, the remote content of the first tuple is updated and asynchronously unlatched. Since the remote operations are executed asynchronously, the RDMA operations (CAS and write) may still be outstanding, and the first buffer should not be re-used for the second tuple. Therefore, the second tuple is is written to a second (local) buffer. Using separate buffers gives the outstanding operation from the first buffer time to complete without any risks of overwriting the content. The first buffer can be safely re-used when the second operation on the same QP generates a completion event, i.e., after the payload is read.

Write unlatch is an optimization that relies on the fact that RDMA writes are performed in increasing address order. This optimization often works in practice but is similar to RDMA reads, not specified by the RDMA standard. However, because many essential applications such as MPI [133] and many other systems rely on last-bit polling [55, 64, 142, 252], RNIC vendors typically implement RDMA writes in increasing address order [55] for compatibility reasons. Note that the latch must be located at the highest address (typically as a footer) to protect the data until the full write has been completed. For example, the last 8 bytes of the value of an item residing in an RDMA-enabled key-value store could encode the lock protecting it. The payload of an RDMA write to update the value would be suffixed by an unlocked value to also release the lock. Since this optimization uses the write to unlatch the data, it saves a complete atomic operation. Unfortunately, there is a catch; the optimization only works in *certain cases*.

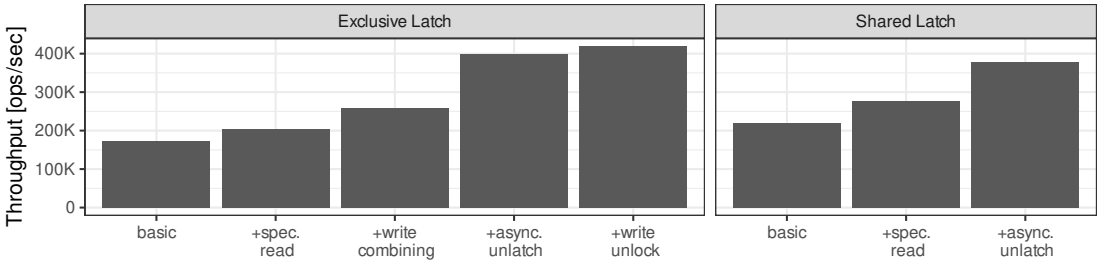


Figure 11.8: Single-threaded ablation study of latch optimizations (1 compute node and 1 storage node).

The InfiniBand specification makes no guarantees that non-atomic and atomic RDMA operations are ordered and visible to each other when issued from different QPs [91]. That means that the RDMA write that unlatches the data item may not be visible to subsequent atomic operations from other workers. This can lead to behavior that might seem surprising because sequential consistency is not guaranteed.

We believe this is because the RNIC can buffer atomic operations for fast completion in a special on-chip buffer. When an atomic operation arrives at the RNIC, the cache line in which the value is stored, is read into this buffer from the memory. But because there is no guarantee w.r.t. interference of non-atomic operations, it may happen that once the atomic operation reads the current value in the buffer, an RDMA write changes the value on the cache line, i.e., unlatch. To make this more concrete, assume the data item is exclusively latched. Now, an atomic FAA operation wants to increment the reader count. The value is read to the RNIC buffer, but in the meantime, the latch-holder unlatches the data item with an RDMA write. Unfortunately, there are now two incoherent states: (1) in the RNIC buffer, the latch is still latched (2) in the cache line, the latch is unlatched. The RNIC increments the old (latched) value and overwrites the unlatch state on the cache line. Therefore, the original unlatch operation is lost, and the latch will remain latched, leading to a deadlock. Hence, the write unlatch optimization cannot be used with FAA operations and only works in combination with CAS operations, as used in RCC-NOWAIT [221]. The reason is that CAS operations conditionally overwrite the state. The operation detects that the old state is still latched and does not modify the latch. Thus, a concurrent worker may detect the unlatch operation delayed for the incoherent states, but eventually, the cache coherence protocol ensures that the RNIC sees the newest version. Consequently, creating a reader/writer latch in combination with the write unlatch optimization is impratical and only one latch mode can be supported (no distinction between reader and writer) (cf. Table 11.1 RCC-NOWAIT).

11.3.4 Ablation Study of Latch Optimizations

To better understand how the latch optimizations perform, we first enable them step-by-step in a single-threaded ablation study. We show the throughput for exclusive and shared latch acquisitions in Figure 11.8. The numbers include all necessary operations: *CAS, read, write, and CAS* for a modification, and *FAA, read, FAA* for a read operation. As mentioned, some optimizations are only applicable for the exclusive latch acquisition. For the analysis, we repeatedly latch and perform the operations on a 256 byte-sized tuple placed on a storage node.

We can see that all exclusive latch optimizations in Figure 11.8 (left) increase the single-threaded performance. One of the most effective optimizations is asynchronous unlatch since we can immediately start the next operation. In contrast, write unlatch does not significantly increase the performance further. Combined with the fact that it is impractical to implement a reader/writer latch, we will not further consider this optimization in this section. However, we will leverage this optimization again in Section 11.4.

When focusing on the shared latch performance in Figure 11.8 (right), we can observe that the basic performance is higher since the read consists of only three sequential operations. The advantage of fewer operations vanishes with the higher optimization levels as they aggressively overlap messages (cf. Figure 11.7). Eventually, the performance of exclusive and shared latch acquisitions converges with higher optimizations and becomes latency bound.

Scalability of latches. So far, we have focused on the single-threaded performance for pessimistic latch acquisitions – however, most disaggregated database systems run with multiple workers dispersed across several compute nodes. Thus in the following, we discuss how the optimizations perform with an increasing number of workers equally distributed across 4 compute nodes. We use a data set with 20 thousand 256 byte-sized tuples stored on one storage node. We also measured with 20 million tuples, but the results behaved similarly. The accesses are randomly distributed across the data set.

We only show the performance of the write-only workload, i.e., the exclusive latch optimizations, since the read-only workload behaves very similarly in this experiment. As the upper bound, we include an unsynchronized version (an RDMA read and RDMA write). Figure 11.9 shows the results for 8, 32, and 128 workers. The first observation is that the performance is on par with the unsynchronized version when all optimizations are enabled, i.e., *asynchronous unlatch*, for 8 and 32 workers. The optimizations effectively hide the latch operations with the data movement operations. Furthermore, the

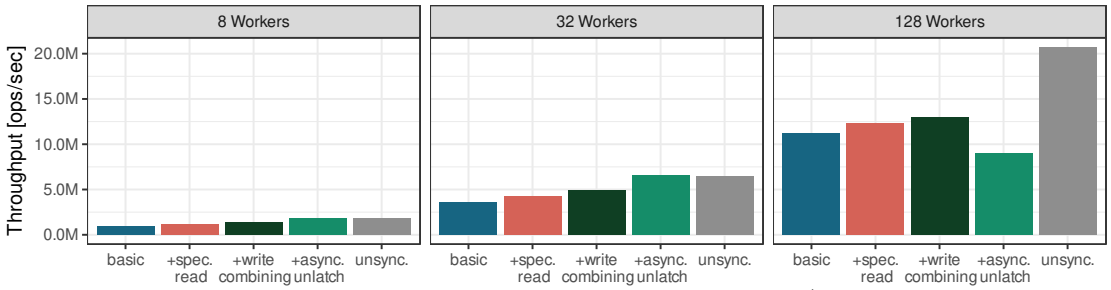


Figure 11.9: Multi-threaded performance of optimized latches (4 compute nodes and 1 storage node).

performance scales near-linear when increasing the worker count from 8 to 32. However, with 128 workers, the highest optimization level seems to be counterproductive. We attribute this behavior to the fact that the next operation may start sooner. Acquiring the next lock earlier introduces additional lock contention in the presence of more workers. For instance, one worker already acquires the next lock even though the old lock is still latched due to the asynchronous nature, i.e., a worker can hold two locks for a limited period. It can also lead to increased RNIC contention on the storage node, with more in-flight operations. In practice, the *asynchronous unlatch* optimization is still worthwhile since there are typically more storage nodes, as we will see in the following experiment.

11.3.5 Effect on a Disaggregated DBMS

As stated earlier, latch optimizations improve performance tremendously. However, the previous results were only obtained in micro-benchmarks. Let us substantiate that claim by integrating them into an existing disaggregated DBMS. We use NAM-DB [239], which is a disaggregated RDMA-optimized DBMS. In our setup, we use 4 compute nodes with 28 workers each (112 total workers) and 4 storage nodes. Among those storage nodes, we distribute 20 million tuples. We measure the throughput in operations per second for representative tuple sizes of 256 and 512 byte. We implemented the highest optimization level., i.e., *asynchronous unlatch*, and call this version NAM-DB++. In addition, we show the effect of manipulating the latches' data layout, as we discussed in Section 11.3.2. We compare the original NAM-DB with NAM-DB++ in a write-only, mixed, and read-only workload. To show the effect of contention, we vary the access skew.

Read-only performance.. Let us first focus on the 256 byte-sized tuples and the read-only workload (top right in Figure 11.10). The optimizations double the performance with uniform and slightly skewed (Zipf 1) access patterns. When the contention increases, the hardware limits the performance, and both systems converge. The dramatic performance

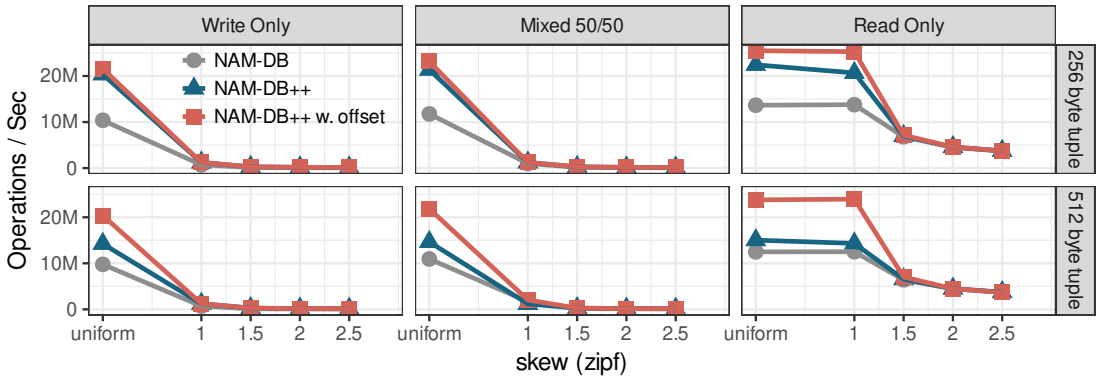


Figure 11.10: Effect of optimizations in NAM-DB (4 compute nodes and 4 storage nodes).

degradation is nevertheless surprising in the read-only scenario. In theory, the read-only performance should not dramatically collapse since multiple readers can acquire a latch simultaneously. Unfortunately, the RNIC cannot handle the concurrent atomic RDMA operations necessary to acquire the reader latch in the first place. As mentioned in the previous experiment, the RDMA atomics are serialized in the RNIC, which does not scale well when the lock slot is contended. Despite the hardware limitations, the read-only workload still performs much better than the write-only workload under high contention. For instance, with a Zipf factor of 2 the write-only performance is $170K$, whereas the read-only performance is $4.5M$.

Write performance. The contention exacerbates the performance issue in the write-only and mixed workload. The locks now logically contend in addition to the physical contention on the RNIC. In other words, the performance of the atomic operations decreases since those operations often target the same latch and are serialized in the same RNIC lock slot. Once the RNIC processes the atomic operation, the latch may have been already latched exclusively, which requires a restart and aggravates the problem.

Larger tuples. When looking at the 512 bytes-sized tuples in the read-only workload, we can see that the effect of applying the padding is more pronounced. As mentioned earlier, the larger latch strides lead to more contention inside the RNIC's lock-table, thus limiting the performance. We use 4 storage nodes, as opposed to the experiment in Section 11.3.2, and still, the collisions inside the RNICs become the primary bottleneck. Therefore, with the padding optimization, the performance significantly improves by removing the bottleneck of physical contention, allowing the latch optimizations to achieve their potential.

11.4 Optimistic Synchronization

In the previous section, we have seen that pessimistic synchronization scales when the latches are uncontended. However, in some data structures, latch contention is unavoidable and, in fact, inherent to the data structure design. For instance, B-Tree operations have to traverse the B-Tree root node. Although the root node is mainly latched in shared mode, this creates (physical) contention on the RNIC when using pessimistic synchronization, negatively impacting the performance. Figure 11.11 shows this effect for a B-Tree with 4 KB nodes stored on a single storage machine and accessed read-only from 4 client machines. As we can observe, the unsynchronized B-Tree can saturate the bandwidth with only 16 workers, whereas the pessimistic synchronized version stagnates much earlier and never achieves the full bandwidth. Note that the pessimistically synchronized version includes all the optimizations presented above, and even then, the performance is disappointing in this experiment.

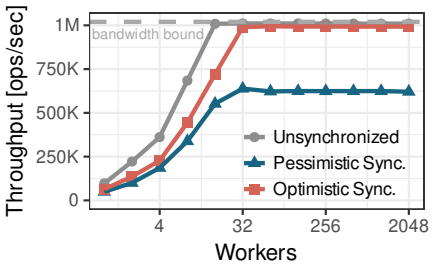


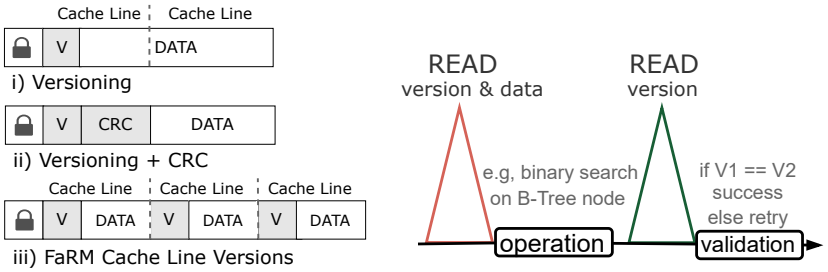
Figure 11.11: Lookups in one-sided B-Tree (4 compute nodes and 1 storage node).

This is why many papers eschew RDMA atomics and propose an optimistic synchronization scheme in which reads do not physically latch the data but detect concurrent modifications. In contrast to pessimistic synchronization, Figure 11.11 demonstrates that optimistic synchronization can achieve the full bandwidth. We start this section by providing an intuition on how optimistic synchronization works. After that, we discuss PCIe's ordering guarantees and its devastating effects on some optimistic schemes leading to an incorrect synchronization. We then present correct optimistic synchronization schemes andgt evaluate their performance.

11.4.1 Intuition for Optimistic Synchronization

Optimistic reads. The intuition for optimistic synchronization is that readers proceed optimistically and then validate, while writers physically acquire an underlying pessimistic lock to avoid write-write conflicts. To achieve the same guarantees as shared pessimistic latches[4] readers must check that the data item did not change during their operation. This is typically realized by augmenting the data item with a version counter and incrementing

[4]We will discuss relaxed guarantees in Section 11.5



(a) Optimistic Layouts (b) Naïve Optimistic Technique

Figure 11.12: Implementation of optimistic synchronization

it upon each modification, which allows readers to detect concurrent writes. The layout of such an augmented optimistic lock is shown in Figure 11.12a. It consists of a pessimistic latch used for exclusive access and the version counter. Using this version counter, readers validate that the version did not change during their operation. If the validation fails, the operation is restarted.

Naïve implementation. One way of implementing a one-sided optimistic lock is depicted in Figure 11.12b, which we call the *naïve implementation*. This approach uses a single RDMA read to copy the latch, version, and data to the worker. The worker can then check if the item is exclusively latched and possibly restart. Otherwise, if the check on the latch succeeds, the operation, e.g., a binary search in a B-Tree node, is performed optimistically. Once the operation is finished, the version is read once more (via RDMA) and compared to the initial value.

This after-the-fact validation is crucial to detect concurrent modifications and to get the same guarantees as a pessimistic latch. Thus, the validation is effectively equivalent to an unlatch operation. These semantics are critical for higher-level synchronization techniques such as optimistic lock-coupling [120, 256] on a B-Tree since we need to ensure that the B-Tree node did not split and the child node is still valid.

11.4.2 PCIe's Ordering Guarantees

Intermediate protocols. Unfortunately, the naïve implementation in Figure 11.12b is not correct. As pointed out by Taranov et al. [209] there are three factors that can affect data deliver order of transmitted messages: (1) Message ordering, (2) packet ordering within a single message, and (3) DMA ordering. The first two factors are generally ensured by InfiniBand and RoCE[5]. But even though message ordering is guaranteed, it is not guaranteed that DMA operations are performed in-order. The InfiniBand RDMA

[5]There are RNICs that deviate from this and offer out-of-order delivery [44]

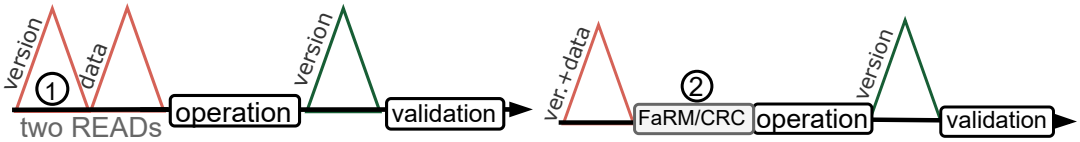
specification [91, 92] does not itself provide any ordering guarantees concerning the order of bytes read by an RDMA read. In other words, RDMA operations are not designed with concurrency in mind (except for RDMA atomics). However, in practice, many academic and industry-grade systems use RDMA concurrently on the same memory regions. Since RDMA does not specify the behavior of those concurrent accesses, the intermediate protocols are important. For example, the lack of order for reads permits intermediate protocols involved in the operation, e.g., PCIe, to retrieve host memory as they see fit. Therefore, to fully understand why an RDMA read may not execute in increasing address order, it is critical to investigate the impact of PCIe [171] and cache coherence protocol. Understanding the underlying protocols subsequently allows us to extract correct synchronization techniques.

As shown in Section 11.2, an RDMA read request is sent over the network to the remote node. The RNIC then dispatches the request to the PCIe controller, which fetches the requested data region from the host memory. The data is transferred via PCIe to the RNIC and then sent back to the requester in one or more RDMA packets. An important aspect is that PCIe requests serviced by the host are cache coherent on modern server architectures. Modern architectures provide *direct cache access* [88, 121, 208] to allow high-performance I/O such as RDMA to access processor caches directly. For example, the x86 machines in our testbed are equipped with Intel's DDIO [93], and a similar mechanism is available for ARM [8].

PCIe reordering. Since the actual data transfer from the remote memory to the remote RNIC is initiated and carried out via PCIe, we must look to the PCIe specification. RDMA requests are translated to PCIe transactions consisting of reads and writes that are processed by the so-called PCIe root complex. Hence, the guarantees provided at this layer of the hardware stack play a pivotal role. The root complex services PCIe read requests, which are coherent at a cache line granularity. Once a request is initiated, the PCIe protocol transfers that data to the endpoint via so-called *completions*.

Multiple completions are used for reads larger than a given size, e.g., 64 bytes. Herein lies the problem. The PCIe specification states, "Memory Read Request serviced with multiple completions, the completions will be returned in address order." [171]. This is hard to interpret, but it only determines the order of the completions and not in which order the data is retrieved from memory. In fact, an implementation note permits that a "single Memory Read that requests multiple cachelines from host memory is permitted to fetch multiple cachelines concurrently" [171].

Implications on correctness. Due to the concurrent cache line retrieval, we cannot reliably detect concurrent modifications with our naïve implementation from Figure 11.12b.



(a) Two RDMA reads (b) FaRM and CRC technique

Figure 11.13: Correct optimistic techniques

For example, assume a concurrent writer and a reader access a data item as depicted in Figure 11.12a i). With the lack of order, the reader could first retrieve the second cache line in which the writer currently modifies data. In the meantime, the writer increments the version and unlatches the data item again. Only then does the reader retrieve the first cache line containing the latch and the version counter. Consequently, despite the validation step at the end in which it will retrieve the version again, the reader will falsely assume that the version did not change. In our introduction, we have already shown that this is a real risk for modern RNICs (cf. Figure 11.1b.) This may be surprising for two reasons: (1) The PCIe standard guarantees that RDMA writes are performed in increasing address order while reads can be read out of order, as discussed before. (2) many papers rely on this unspecified behavior. In fact, an earlier paper of ours [256] suffered from this wrong ordering assumption. More details on the memory semantics of RDMA can be found in [48].

Referring back to Figure 11.1b, we can see that data is rarely corrupted, but it happens, and if data corruption occurs, it is almost undetectable. We tested this behavior on three generations of modern RNICs, including the RNICs available in the cloud, and observed that no RNIC provides additional ordering guarantees beyond the PCIe specification. Fortunately, there exist techniques that prevent this issue, but, as we will see in the following section, they are not for free and come with different performance characteristics and trade-offs.

11.4.3 Correct Optimistic Synchronization

Optimistic synchronization relies on the initial RDMA read to observe a consistent view of the version and data. Due to the lack of ordering at the PCIe bus, we must rely on an additional mechanism to provide this capability. Only then the after-the-fact validation will correctly detect concurrent modifications. We will discuss existing mechanisms[6] in the following.

[6]Note of caution: Some optimistic techniques rely on hardware details that may not hold. E.g., when data is not aligned or larger than the MTU.

Versioning (using two RDMA reads). This technique is not very different from the naïve version; however, it requires two dedicated RDMA reads in the beginning, as shown in Figure 11.13 a). The first RDMA read targets only the latch and the version to ensure correct serialization, and only the second reads the data. Because the version is always read before the data, every concurrent update will be detected. When looking at Figure 11.13 one may wonder if the two reads can be overlapped similarly to the operations in Figure 11.7. Unfortunately, this is not possible with two RDMA reads since they could be re-ordered at the PCIe level or even in the network. This is due to another unexpected pitfall: Although RDMA operations are ordered in a connected queue pair, the ordering only holds for the RDMA completion events. There are some exceptions in which the order is ensured, e.g., for atomic operations. On the other hand, the two reads must be issued sequentially.

Besides the correct order of the two reads, the latch and version must be stored in one cache line to exploit the cache coherence protocol and read both consistently. Only this enables a reader to detect concurrent modifications reliably. Unfortunately, this technique needs two RDMA reads, which may be expensive. The following two approaches only require a single RDMA read by embedding additional metadata in the object as shown in Figure 11.12a

Checksum by CRC. This scheme detects inconsistencies by using a checksum in the data, e.g., CRC64, that allows the worker to verify the data with high probability. The CRC will not match the corrupt data if there is a concurrent writer. Therefore, in the best case, only one RDMA read is required (cf. Figure 11.13 b)). The downside, however, is that (1) the CRC generation is computationally expensive and (2) it is only probabilistic. Admittedly, the probability of a collision is low for CRC64. However, if a collision occurs, the data corruption is hard-to-detect.

FaRM cache line versions. As with CRC, FaRM [55, 56] proposes a technique that requires one RDMA read in the best case. However, instead of computing a checksum, FaRM relies on coherent cache DMA (specified by x86) to detect if a single RDMA is consistent. FaRM stores a version at the beginning of *every* cache line as illustrated in Figure 11.12a. Therefore, we can detect if a concurrent modification happens by comparing all cache line versions. To enable this, writers first latch the data item, read it, modify the data, increment the version locally, and then write the data back via RDMA in address order. Although not as computationally expensive as CRC, every cache line must be accessed to validate the versions introducing additional cache misses, i.e., stalled CPU cycles. In addition, this technique imposes additional storage overhead to accommodate a version in every cache line.

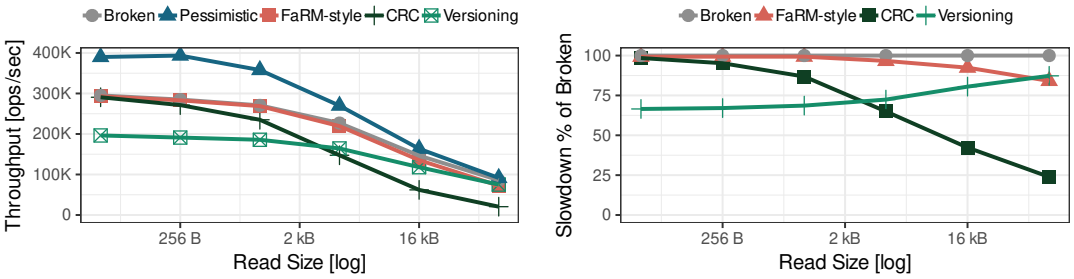
11.4.4 Single-Threaded Performance

We initially focus on single-threaded performance to understand the different trade-offs of the correct schemes. Figure 11.14a compares the single-threaded read-only performance with varying tuple sizes. We also include the "broken" optimistic scheme and the optimized (and correct) pessimistic scheme from Section 11.3.

Optimistic vs. pessimistic. Maybe unexpectedly, the pessimistic synchronization scheme performs better than all optimistic schemes, including the incorrect one. Although the broken optimistic scheme only requires 2 messages as opposed to the pessimistic scheme that requires 3 messages, the pessimistic scheme can exploit the asynchronous unlatch optimization. This optimization unlatches asynchronously, and the subsequent operation begins immediately (cf. Figure 11.7). Analogous behavior is not possible for optimistic schemes as the validation, i.e., the unlatch operation, determines if the operation failed or succeeded. Consequently, the validation must be synchronous (see Figure 11.13).

Correct schemes vs. broken. Let us now quantify the induced overhead of the correct schemes compared to the broken scheme, which only consists of a single RDMA request and no consistency checks to acquire the data and version information (i.e., the first round trip in Figure 11.12b). From Figure 11.14a, we can observe that the broken scheme performs better than the correct schemes. The closest competitor is the FaRM scheme, which performs nearly as well as the broken scheme and only drops with larger message sizes. CRC can only compete with small tuple sizes, contrary to the versioning scheme, which catches up with large tuple sizes.

To highlight those effects, Figure 11.14b shows the slowdown in percentage compared to the broken scheme. We can see that both CRC and FaRM suffer from larger tuple sizes. The computational overhead for both schemes is $O(n)$ since CRC calculates the checksum for every byte, and FaRM checks the versions in every cache line. Consequently, both schemes get linearly more expensive with increasing tuple sizes, more precisely, the consistency check (phase ②) as shown in Figure 11.13. However, because the CRC calculation is considerably more expensive than checking every cache line sequentially, the performance degrades more severely. For FaRM the sequential HW prefetcher and the out-of-order execution hide the cache misses. In contrast, the versioning approach incurs a substantial *constant* overhead by requiring an additional read for the version. Therefore, the versioning approach amortizes the initial cost with larger tuples sizes and performs better than FaRM.



(a) Throughput comparison (b) Relative slowdown of correct schemes

Figure 11.14: Single-threaded performance of optimistic reads (1 compute node and 1 storage node).

To summarize, looking at the results of the single-threaded experiments, we found (1) the pessimistic schemes out-performs the optimistic schemes in the single-threaded setup, (2) FaRM performs better with smaller tuple sizes ($\leq$ 64 KB), (3) while versioning is beneficial with larger tuples.

11.4.5 Scalability of Optimistic Techniques

While the optimistically latched single-threaded performance was worse than the pessimistic scheme, it is not indicative of its scalability behavior. After all, the main benefit of optimistic schemes is that they avoid RDMA atomic operations during a read and, thus, avoid physical contention on the RNIC generated by multiple workers.

Read-only scalability. Figure 11.15 shows the scalability w.r.t. the number of workers for small, i.e., 256-byte sized, tuples. Again with fewer workers (8), the pessimistic approach performs better than the optimistic schemes. In line with the previous results, CRC and FaRM are close to the broken scheme, whereas the versioning scheme is far behind. However, under the highest contention when using 128 workers, all the optimistic schemes perform better than the pessimistic scheme. In particular, FaRM and CRC perform almost twice as well as pessimistic synchronization. This confirms our intuition that by avoiding RDMA atomics, the physical contention effect does not limit the performance in optimistic schemes.

We also compared the scalability of larger, 16 KB sized tuples and found that the pessimistic scheme performs better in this scenario. The reason is that with larger tuples, the workload becomes network bound before the RNIC contention limits the performance. In other words, only 8 workers are required to reach the full bandwidth (12 GB/s) with the pessimistic synchronization, and the RNIC can easily sustain RDMA atomic operations at such a rate.

Figure 11.15: Read-only and write-only optimistic scalability (4 compute nodes and 1 storage node)

Write-only scalability. Remember that only reads are performed optimistically, while modifications are still latched pessimistically. However, since the optimistic readers do not use RDMA atomics, we can now leverage the write-unlatch optimization, which uses a write instead of an atomic operation to unlatch (see Figure 11.7). To remind ourselves, we have not considered them before because the write-unlatch optimization does not work with concurrent FAA operations required to implement reader-writer latches. But because optimistic schemes only require one latch mode, i.e., exclusively latched or unlatched, write-unlatch can be used to further reduce the number of RDMA atomic operations.

Figure 11.15 (right side) shows the effect of this optimization when increasing the number of workers up to 128. We can observe that the write-unlatch optimization enables better scalability by avoiding one RDMA atomic operation. When comparing the performance of the write-unlatch optimization with 128 worker and the results from Figure 11.9 we can see that it approaches the performance of the unsynchronized variant. Important to note is that the layout as shown in Figure 11.12a must be reversed so that the version and latch are placed at the end of the data to exploit address-ordered RDMA writes. That is, the last byte written must unlatch the record. The reversed layout does not affect the performance of the optimistic approaches since it merely changes the location of the version.

Qualitative discussion. So far, we have focused on performance numbers. However, the correct techniques come with trade-offs, such as the number of required messages. In the best case, the versioning approach requires 3 messages and can be a sub-optimal strategy

when the workload is message-bound. In contrast, CRC requires only 2 messages but is computationally more expensive, which may be detrimental if the threads can do other valuable work. Moreover, CRC is probabilistic, and although the collision probability is low, the question remains: why risk data corruption when reliable schemes exist? Lastly, although FaRM is certainly efficient, it adds $2-8$ bytes storage overhead [55] per cache line, which means it is not only $O(n)$ in terms of computing but also storage overhead. In addition, the higher-level logic must handle the interleaved cache line versions. For instance, any string operation (comparison, regexp etc.), e.g., on a value in a B-Tree, must explicitly deal with strings chunked across cache lines. Because only a few existing libraries support chunking, one would have to copy larger strings into a contiguous memory before being able to use them and potentially offset the gains by the efficient scheme. Consequently, choosing the correct scheme has trade-offs that must be carefully weighed against each other and co-designed with the system.

Summary. To conclude, we have found that the optimistic techniques scale and perform better than pessimistic techniques for workloads in which the RNIC's capability of handling RDMA atomic is the limiting factor. For instance, this happens in a read-only workload with small tuples and 128 or more workers. When using larger tuples, the workload tends to become network bound, reducing the number of possible parallel operations and shifting the bottleneck for pessimistic schemes from the RNIC contention to the network. Thus, a pessimistic scheme often performs better in such scenarios. Another interesting finding is that in combination with optimistic reads, we can improve the scalability of writes by using the write-unlatch optimization. In contrast to pessimistic schemes, optimistic schemes come with different trade-offs regarding computational and storage requirements.

11.4.6 Effect on a Disaggregated DBMS

Finally, we compare the optimistic schemes by implementing them in NAM-DB and call this system OPT-DB. Similar to the experiment in Section 11.3.5, we vary the contention and show write-only, mixed, and read-only workloads. The configuration is unchanged with 4 compute and storage nodes storing $20M$ tuples. Unlike the experiment in Section 11.3.5, we stick to 512 byte but vary the number of workers from 112 to 224.

Contended shared latches. In the read-only workload in Figure 11.16, we can observe that all optimistic techniques are pretty robust against contention, contrary to the pessimistic synchronization in NAM-DB++, which already degrades with light skew (1.5). Furthermore, we can back up the previous findings that the pessimistic scheme performs

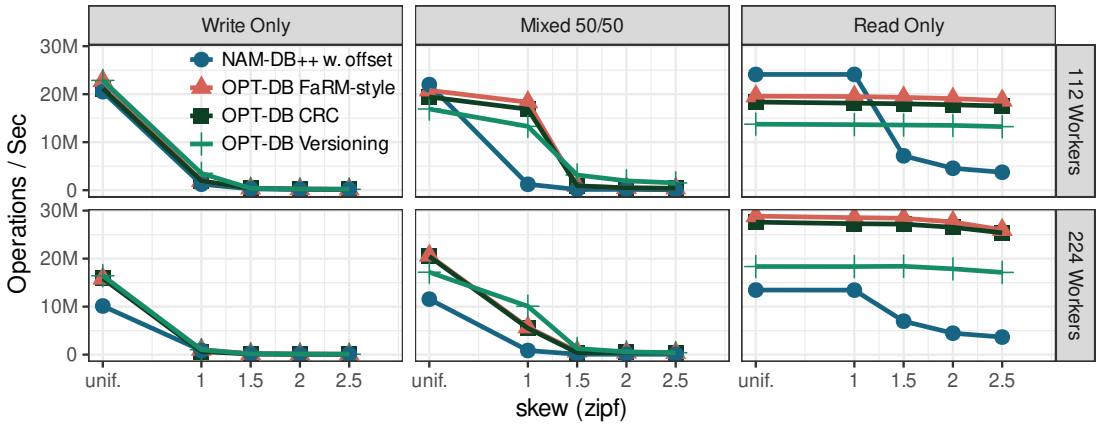


Figure 11.16: Optimistic techniques in a disaggregated DBMS (4 compute nodes and 4 storage nodes).

better with fewer workers than the optimistic schemes. The optimistic approaches perform better when increasing the number of workers to 224. Overall, the OPT-DB implemented with FaRM performs the best, followed by CRC and Versioning (for 512 byte tuples).

Reader/writer contention and aborts. There is a well-known trade-off for optimistic approaches in reader-writer contention: optimistic approaches typically lead to more restarts, resulting in wasted work. Nevertheless, the mixed workload shows that the optimistic approaches out-perform NAM-DB++ despite the restarts.

The reason lies again in the fewer RDMA atomic operations. Especially under contention (skew), the atomics fall in the same lock slot, which creates physical contention in the RNIC. Moreover, in this experiment, the logical contention exacerbates this physical effect. Once the RDMA atomic operation is executed, it does not necessarily mean that the lock operation was successful, e.g., it may be already latched exclusively by another worker. Thus, for a pessimistic scheme, the operations may restart, which incurs additional atomic operations. To this end, for the mixed workload, the optimistic approaches, which get away with a single atomic operation for exclusive latches and none for optimistic (read) acquisitions, perform much better.

The same holds for the write-only benchmark; since the optimistic approaches harmonize with the write-unlatch optimization, they perform on par or sometimes even better (e.g., with low contention and 224 workers as shown in Figure 11.16). Moreover, in the write-only workload with 112 workers, the versioning approach is slightly faster since the writers do not need to perform any additional computation, such as CRC checks or incrementing the cache line versions. When contention increases, all the approaches converge at merely $100K$ operations per second.

11.5 Discussion of Other Approaches

This section discusses other synchronization techniques with relaxed guarantees.

Consistent optimistic read. The synchronization schemes that we discussed before captured latch semantics. That is, they include an additional validation step after retrieving a data value and operating on it to make sure that the data did not change in the meantime. However, these strong semantics are not required for all use cases [149, 150, 257]. A classic example is a key-value store that is only concerned about returning consistent data to clients and does not consider the clients' operations on the data. Our presented optimistic schemes can be easily adapted to support such relaxed semantics by removing the additional validation step at the end.

Other incorrect schemes. Another common technique is called bookend versioning which we briefly showed in Figure 11.1a. While incorrect, the intuition of this approach is similar to versioning, where the second version (embedded in the data) validates that the data was not concurrently updated during the RDMA read. This approach suffers from the ordering problem, and its use in recent literature (e.g., [223]) is a testament to the subtleties involved in one-sided synchronization.

Out-of-place updates. An alternative design to the data consistency techniques described in Section 11.4.3 is to leverage out-of-place updates [30, 185, 228], eliminating concurrent data modifications using a copy-on-write strategy. This approach has the same communication overhead as the (broken) single RDMA read versioning technique but comes at the obvious cost of additional storage overhead. Out-of-place updates also enable the use of a technique we call *marking*. Marking is a simple detection technique that can alert readers by relying on a logical flag to indicate that a concurrent write is taking place. This approach manifests as logical insertion or deletion and a busy or pending update flag. However, when combined with in-place updates, marking suffers from the same re-ordering problem as versioning and can fail to detect an inconsistency because of the PCIe bus. Unfortunately, this incorrect use of marking has found its way into existing literature [86, 246].

11.6 Lessons Learned and Conclusion

As the first analytical study of RDMA synchronization primitives, we aim to leave the reader with a distilled perspective on the lessons learned from our work. These lessons are cast as *anti-patterns*, which reflect common mistakes that lead to incorrect designs, and

important design *considerations*, which do not impact correctness but can be detrimental to performance and system complexity.

Anti-pattern #1.
Reliance on cache line order within a single RDMA read.

The first anti-pattern stems directly from the lack of ordering in the PCIe bus, which was first introduced in Section 11.1 and discussed in more detail in Section 11.4.2. Designs fall prey to this anti-pattern and thus incorrectly assume that an RDMA read observes the same order in which writes to different cache lines are made. Examples that relied on this anti-pattern include bookend versioning [150, 223] and marking with in-place updates [86, 246]. For instance, in [246], a writer first marks data as invisible to readers, then updates the value. As we have already established (cf. Section 11.4.2), a reader may observe these two steps out-of-order and, therefore, could accidentally assume a corrupted read is consistent. Worse, the version retrieved during this initial read could later be (incorrectly) validated. This behavior is identical to the broken versioning scheme discussed in Section 11.4.2.

Anti-pattern #2.
Reliance on the ordering of overlapping reads.

One solution to address the first anti-pattern is to manually enforce an order by issuing multiple reads. However, there are also pitfalls to this approach because distinct RDMA read operations also have subtle ordering guarantees. The second anti-pattern captures the lack of ordering between overlapping reads from the same connection. While RDMA operations can be fenced to enforce ordering in specific cases, this does not pertain to RDMA reads. In other words, there is currently no mechanism to enforce the order in which a remote machine processes RDMA reads without waiting for completion. Optimistic techniques like versioning hence require two sequential reads to ensure that the version is read before the data, as explained in Section 11.4.3. One-shot optimistic approaches, such as FaRM, do not suffer from this because they detect inconsistency at the granularity of cache lines.

Anti-pattern #3.
Reliance on the atomicity of RDMA writes and RDMA atomics.

The third anti-pattern results from the lack of atomicity guarantees by RDMA write and RDMA atomic operations in the RDMA specification. As Section 11.3.3 describes, various optimizations to improve pessimistic synchronization techniques exist. Recall that the write-unlatch optimization is only permissible when the locking primitive exclusively

utilizes CAS operations. Again, this is because there is a store buffer in the RNIC. CAS operations are a special case because the write-back to memory is conditional. While it is possible to correctly utilize the write-unlatch optimization [221], as shown in Section 11.4.5, it requires a careful understanding of the underlying hardware. Therefore, we generally caution against designs that combine RDMA writes with RDMA atomic operations but acknowledge that in some instances where the lack of atomicity does not break semantics, it can yield better performance.

Consideration #1.
Data alignment impacts the RDMA atomic scalability.

The first consideration is described in detail in Section 11.3.2, but we review it here. Because of the lock table present in current RNICs, physical contention between independent latches can arise. We demonstrate that this is not only easily demonstrated by microbenchmarks that highlight the behavior but also have an important role in the scalability of a real system (i.e., NAM-DB). Hence, we advocate that system designers pay close attention to their data alignment when leveraging RDMA atomic operations to avoid this physical contention.

Consideration #2.
Optimistic synchronization performs best under high contention.

Conventional wisdom suggests that pessimistic synchronization pays off in high-contention write-heavy workloads because it eliminates excessive retries. However, our analysis demonstrates that this perspective does not hold for RDMA-based synchronization. Notably, in Section 11.4.5, we show that in write-heavy workloads, the optimistic approaches can match and even surpass pessimistic ones. While readers often retry in optimistic schemes, pessimistic approaches are subject to RNIC contention. Our results suggest that pessimistic approaches are beneficial when there is less skew, and the number of concurrent workers is small. Interestingly, this also applies in the read-only case since the sequential overhead is low compared to the optimistic strategies, which require validation. Therefore, optimistic synchronization should be preferred for highly skewed workloads to avoid RDMA atomic bottlenecks. As we established with the B-tree performance in Figure 11.11, this is particularly beneficial when there is a single point of contention, e.g., the root node of a B-tree.

Consideration #3.
Optimistic synchronization has non-negligible overheads.

Although they outperform pessimistic synchronization in many scenarios, there are computational and storage trade-offs among the various optimistic approaches, which we

discussed in Section 11.4.5. Versioning requires an additional RDMA read compared to the other techniques, which increases operation latency and is most evident for small data sizes. On the other hand, CRC is computationally expensive, becoming untenable at large data sizes ($\geq$ 4 KB). Cache line versioning generally performs well but has an increased storage cost, not to mention that software must either handle the embedded versions or copy the data out. While optimistic synchronization is beneficial in many scenarios, it is not a silver bullet as they often have more complex designs compared to pessimistic schemes.

Consideration #4.
Pessimistic synchronization is more "future proof".

As demonstrated, subtle hardware characteristics can have a profound impact on correctness. We highlight this using a widespread deployment but point out that disaggregated memory technologies are in active development. For example, advancements like CXL [198] – a protocol built on PCI-e to support memory coherence across devices – are poised to disrupt the status quo. Changes to intermediate components of RDMA communication may alter the behavior of concurrent RDMA reads and writes, and therefore optimistic synchronization schemes, in unpredictable ways. In contrast, RDMA Atomic operations implement a well-established higher-level abstraction independent of hardware implementation. Hence, system designers should lean toward simple pessimistic synchronization when prioritizing production stability until the community has successfully converged on well-defined memory semantics for RDMA reads and writes.

Conclusion. This paper is the first paper that holistically (1) highlights the subtleties of RDMA-based synchronization regarding the correctness, (2) provides a robust performance analysis of existing synchronization techniques and optimizations, (3) demonstrates pitfalls of existing designs, and (4) offers a set of anti-patterns and design considerations for enabling developers to design correct and high-performance RDMA-enabled system.

Acknowledgement. This work was partially funded by the German Research Foundation priority program 2037 (DFG) under the grants BI2011/1 & BI2011/2, the DFG Collaborative Research Center 1053 (MAKI), and the state of Hesse as part of the NHR Program. We also thank hessian.AI, DFKI, Mellanox, and 3AI for their support. We also like to thank Torsten Hoefler for an insightful discussion about the memory model of RDMA and RDMA's ordering guarantees.

12 RDMA Communciation Patterns

Abstract

Remote Direct Memory Access (RDMA) is a networking protocol that provides high bandwidth and low latency accesses to a remote node's main memory. Although there has been much work around RDMA, such as building libraries on top of RDMA or even applications leveraging RDMA, it remains a hard problem to identify the most suitable RDMA primitives and their combination for a given problem. While there have been some initial studies included in papers that aim to investigate selected performance characteristics of particular design choices, there has not been a systematic study to evaluate the communication patterns of scale-out systems. In this paper, we address this issue by systematically investigating how to efficiently use RDMA for building scale-out systems.

Bibliographic Information

The content of this chapter was previously published in the peer-reviewed work 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. The contributions of the author of this dissertation are summarized in Section 6.2.

12.1 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 performance and storage capacity by simple adding further processing nodes. However, a typical bottleneck in scale-out systems is the network which often slows down the computation if communication is in the critical path or, even worse, degrades overall performance when adding more nodes [183].

With the advent of high-speed networks such as InfiniBand and its networking protocol RDMA this changed. Network latencies dropped by orders of magnitude while throughput increased [20] making scale-out solutions more competitive. However, this improvement does not simply come by switching the network technology stack, but requires the usage of RDMA's low-level communication primitives, e.g., SEND or WRITE.

Although there has been much work around RDMA, such as building libraries on top of RDMA [5, 64] or even applications leveraging RDMA [14, 55, 56, 103, 134, 239], it remains a hard problem to identify the most suitable RDMA primitives and their combination for a given problem. While there have been some initial studies included in papers that aim to investigate selected performance characteristics of some design choices [20, 102, 104], there has not been a systematic study to evaluate the communication patterns of scale-out systems.

*.Contribution: We address this issue by systematically investigating how to efficiently use RDMA for building scale-out systems. For this study, we model communication between nodes using a request-response pattern (i.e., an RPC-style communication). This request-response communication pattern can be used for many data-centric communication scenarios such as a key-value stores or transactions in distributed databases.

For implementing a request-response communication pattern, the combination of RDMA verbs and other design considerations (e.g. the communication model or the communication topology) provide a huge design spectrum one has to navigate. In our study, we combine all of the above options and evaluate different performance characteristics with varying message sizes of 64B to 16KB to simulate workloads of different applications (e.g., a different tuple width in a distributed database). As a result, we shed light on important metrics such as the bandwidth, latency but also important CPU statistics.

To evaluate this design space, we perform an extensive analysis and experimental evaluation of the following dimensions:

1. *RDMA Verbs:* RDMA verbs allow an application to specify whether one-sided communication primitives (RDMA READ/WRITE) or two-sided communication primitives (RDMA SEND/RECEIVE) should be used. While there have been already almost "religious" fights which communication primitive is better in existing papers, the usage of these primitives has only been evaluated in particular settings (e.g., only in a distributed database using an M-to-N communication topology) but not in the full design space. Moreover, there are many low-level optimizations such as inlining, selective signalling, DDIO and many other low-level configuration options (e.g., if RDMA is run in single-threaded communication or not) that have not been investigated.

2. *Communication Model*: A second dimension, we aim to analyze in the design space is how threads are connected. In this paper we analyze two typical connection models: One option is to use a so called communication *Dispatcher* where worker threads delegate the communication to one dedicated thread. Another option is that each worker thread directly executes all communication by using a *Private Connection* to each other worker thread resulting in a much higher number of overall connections (i.e., not just between dedicated communication threads).

3. *Communication Topology:* The last dimension is the communication topology between requesters and responders. In this paper, we analyze different options: *One-to-One*, *N-to-One*, *N-to-M*. For instance, while a key-value-store typically uses an N-to-One communication topology (i.e., N requesters, 1 responder), whereas a distributed database often uses an M-to-N communication topology. In addition with the communication model, the communication topology significantly influences the number of connections and thus the contention on internal data structures of high-speed network components (e.g., connection queues in InfiniBand).

We believe that our evaluation framework is an interesting analysis tool for other research groups or industry that plan to leverage RDMA for building distributed data processing systems. To foster this and allow follow-up research to use our findings, we made the code available on GitHub [216].

***.**Outline: The rest of this paper is structured as follows: We first introduce the relevant background w.r.t RDMA in Section 2 and present related work in Section 3. In Section 4 we then describe our methodology before we start our study with Section 5 in which we evaluate single threaded performance in a One-to-One topology. We continue with the scale-out scenarios in Section 6 and 7, where we look at N-to-One and N-to-M topologies,

respectively. In Section 8 we finally discuss the effect of further RDMA optimizations before concluding in Section 9.

12.2 RDMA Background

Remote Direct Memory Access (RDMA) is a networking protocol that provides high bandwidth and low latency accesses to a remote node's main memory. RDMA achieves low-latency by using zero-copy transfer from the application space to bypass the OS kernel. A number of RDMA implementations are available — most notably InfiniBand and RDMA over Converged Ethernet (RoCE) [219].

RDMA implementations provide several communication primitives (so called verbs) that can be categorized into the following two classes: (1) one-sided and (2) two-sided verbs.

- *One-sided verbs:* One-sided RDMA verbs (READ/ WRITE) provide remote memory access semantics, in which the host specifies the memory address of the remote node that should be accessed. When using one-sided verbs, the CPU of the remote node is not actively involved in the data transfer.
- *Two-sided verbs:* Two-sided verbs (SEND/RECEIVE) provide channel semantics. In order to transfer data between a host and a remote node, the remote node first needs to publish a RECEIVE request before the host can transfer the data with a SEND operation. In contrast to one-sided verbs, the host does not specify the target remote memory address. Instead, the remote host defines the target address in its RECEIVE operation. Consequently, by posting the RECEIVE, the remote CPU is actively involved in the data transfer.

To setup the connection between requesters and responders, RDMA uses so called send/receive queues: (1) While a *send queue* is used by the requester to issue operations such as READ, WRITE, SEND as well as ATOMICS (2) the *receive queue* is used by the responder to issue RECEIVE requests. With RDMA, a connection between a requester and a responder bundles these two queues and is therefore called queue pair (QP). More precisely, to initiate a connection between a requester and a responder, the application needs to create queue pairs on both ends and connect them. To actually then issue communication operations, a client creates a *work queue element* (WQE). The WQE specifies parameters such as the verb to use but also other parameters (e.g., the remote memory location to be used, if the element is sent signaled/unsignaled see Section 12.8).

For sending the WQE, the requester then puts the WQE into a send queue and informs the local RDMA NIC (RNIC) via Programmed IO (PIO) to process the WQE. For a signaled WQE, the local RNIC pushes a completion event into a requester's completion queue (CQ) via a DMA WRITE once the WQE has been processed by the remote side.

12.3 Related Work

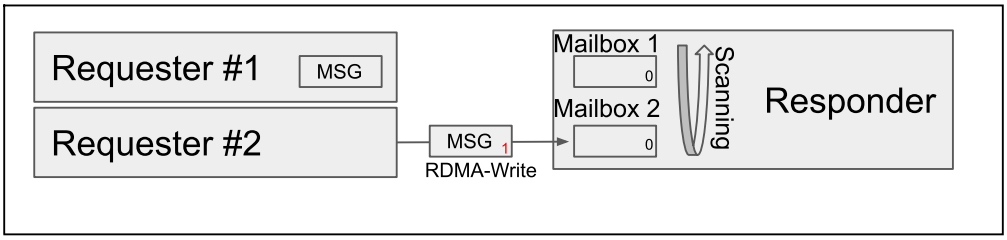
Research and industry have widely adopted high-speed networks [128, 175] to improve scale-out systems. In the following, we compare our evaluation with related work and provide a broader overview of work done in the database community.

***.**RDMA Evaluation: In this paper we aim at providing a systematic evaluation of communication patterns. Therefore, an important body of work are existing low-level RDMA evaluations [20, 102, 104]. Kalia et al. [102] discuss which RDMA operations should be used and how to use them efficiently. The resulting guidelines provide a low-level analysis on the RDMA verbs and how they can be optimized. In this evaluation we build on this findings but focus on a more higher-level evaluation using a request-response pattern to simulate various use-cases in a data processing system.

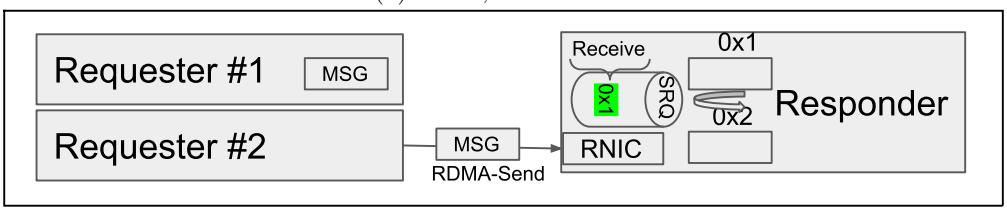
***.**High Performance RDMA Libraries and Systems: To simplify the use of RDMA, Alonso et al. [5] propose a high-level API for data flows for data intensive applications. Fent et al. [64], in contrast, present a library to accelerate ODBC-like database interfaces and propose a message-based communication framework.

Several recent systems adopted RDMA to speed-up their performance. There are many RDMA-enabled key/value stores [104, 114, 126, 138, 149, 166] and distributed database systems based on RDMA [14, 55, 56, 103, 134, 239]. Depending on their design, these systems use different communication patterns. For instance, FaSST [103] discusses how remote procedure calls (RPCs) over two-sided RDMA operations can be implemented efficiently. FaRM [55, 56], in contrast, leverages the benefits of one-sided verbs to implement distributed transactions. Further, several researchers optimized specific algorithms of a database, including distributed joins [15, 16, 183], RDMA-based shuffle operations [132], and indexes [256].

While all these papers include some micro benchmarks to evaluate individual design options, there has not been a systematic study of the broader design space in one unified setup to evaluate the communication patterns of scale-out systems.



(a) Write, *Mailboxes*



(b) Send, *Shared Receive Queue*

Figure 12.1: Building blocks: *Write with mailboxes and send with shared receive queue.*

12.4 Methodology

In this section, we describe the evaluation methodology used to evaluate the design space of RDMA across the different dimensions discussed before. To isolate the fundamental properties of different communication primitives, we implemented a *request-response framework* that allows us to evaluate all design dimensions (i.e., verbs, communication patterns, communication topology). For a fair comparison and to avoid potential overhead that stems from configurability (which would potentially falsify the measurements), the framework uses C++ templates to generate code for one design option (selected by template parameters). Furthermore, to focus on the communication aspects, we avoid executing application logic on the responder side such as a key-value lookup or a remote procedure call (RPC).

In the following, we describe the building blocks of our framework, the workload and the experimental setup used for our evaluation.

***.**Building Blocks of Framework:

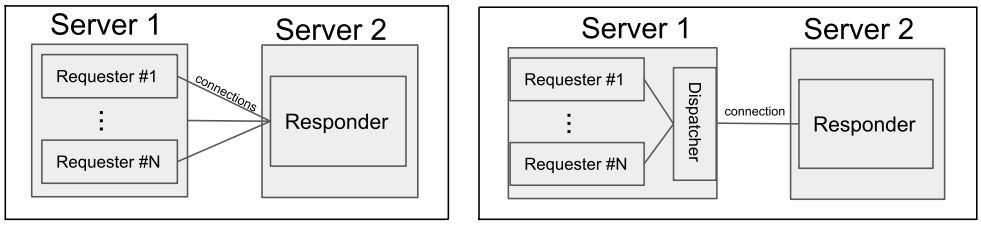
In a request-response communication pattern, the request or the response can either be transmitted by using an RDMA WRITE or an RDMA SEND operation. In the following, we first explain the basic building blocks of how a WRITE / SEND operations can be used to implement a request or a response and then how these basic building blocks can be combined to implement the request/response communication pattern.

Basic Building Blocks using RDMA WRITE: As mentioned in Section 12.2, WRITE is a one-sided verb, meaning it directly writes to remote memory and bypasses the remote CPU. Yet, the responder needs to know when new messages arrive, which is challenging with CPU-bypassing. A common protocol to detect incoming requests, is to write messages to a specific memory region, i.e., into *mailboxes*. Each requester knows the memory address to its dedicated mailbox as depicted in Figure 12.1a. The responder constantly iterates the mailboxes to detect new messages. Special care must be taken to (1) avoid reading partially transmitted messages and (2) avoid that the optimizing compiler removes code. To avoid reading partially transmitted messages, the responder needs to verify if a message is completely written into RDMA memory. We can exploit the increasing address order of WRITE to check the last transmitted bit (flag). Although neither the RDMA nor the InfiniBand specification mentions the order, most hardware vendors implement WRITE with increasing address order [56, 104]. Scanning mailboxes is often implemented as a simple loop, which might be removed by the optimizing compiler. Because the optimizing compiler is not aware that a third party writes to the mailboxes, such a loop is considered unnecessary. In C++ the `volatile` construct ensures the optimizing compiler leaves the code for the loop unchanged.

The transmission process initiated by the requester begins with setting the last bit in the message as depicted in red in Figure 12.1a. Afterwards, the message is written to the corresponding mailbox with WRITE. The responder will detect the incoming message by scanning the mailboxes. Finally, when processing is done the last bit in the mailbox is cleared to receive a new message. This approach is also applicable on the requester side, if the response is delivered by WRITE as well.

The mailbox design is very static and thus has some limitations. One needs to allocate the mailboxes and therefore decide in advance how many messages each requester can transmit. In our synchronous implementation, one requester is assigned to one mailbox. In the context of our evaluation, the WRITE verb always implies that the above mailbox architecture is set up on the remote side.

Basic Building Blocks using RDMA textscsend: In contrast to WRITE, SEND requires a RECEIVE posted to a Receive Queue beforehand. As mentioned in Section 12.2, the RECEIVE contains the information to which memory location an incoming SEND is copied to. Because it is not always known when to expect a SEND, a common pattern to prepare for multiple incoming SEND is to post multiple RECEIVES beforehand. When a SEND consumes one of the RECEIVES, the RNIC will notify the application that a message was received. To know which memory location, i.e., RECEIVE was used a common misconception is to exploit the order of the RECEIVES. For instance, the application



(a) Private Connection, *N Connections* (b) Dispatcher, *Only 1 Connection*

Figure 12.2: Designs — *N Requester and one Responder*

registers two RECEIVES in the Receive Queue one with address *0x1* and afterwards one with *0x2*. Since the RECEIVES will be taken in order from the Receive Queue one might suspect that the first message is written to *0x1*, but hardware parallelism in the RNIC does not guarantee which memory location is used first. For instance, if two RECEIVES are consumed directly after each other the second memory location *0x2* could be written first. The receiver would now look into the first memory address, in which in the worst case nothing, partially transmitted messages or old messages are located. A better approach is to tag the RECEIVES with ids and when a SEND consumes a RECEIVE the RNIC returns the id, which can be mapped to a memory location. To detect incoming SEND messages, the Receive Queue has to be polled. This polling is expensive for the responder that typically has multiple incoming connections. These connections require that the responder constantly switches between the different receive queues. To mitigate this overhead, RDMA offers a Shared Receive Queue to which multiple incoming connections can be mapped. The Shared Receive Queue allows the responder to only poll a single queue to handle all requests as depicted 12.1b. We use the Shared Receive Queue always on the responder side, when requester transfers messages with SEND.

Combining the Basic Building Blocks: These two basic building blocks encapsulate one and two sided RDMA communication and can be freely combined to implement a request-response pattern:

Design 1 — Private Connection per Thread: The first design is fairly simple: each requester is directly connected to all responders (it has N QPs for N receivers, as shown in Figure 12.2a). If the application has multiple requesters, the amount of QPs increases linearly. This, in turn, can negatively impact performance because the RNIC needs to switch between connection states. In the worst case, a large number of connections can lead to swapping some state to the host memory over PCIe [102]. However, because each requester owns its state, no synchronization or delegation is required.

Design 2 — Communication Dispatcher: The second design, which is shown in Figure 12.2b, aims to minimize direct connections by introducing a single communication dispatcher. The dispatcher solely handles communication for every requester on the same server. There is only one connection to every receiver, in contrast to the first design the number of QPs is decoupled from the number of requesters. This can lead to a higher RNIC utilization, but the communication dispatcher can also end up being the bottleneck. The dispatcher can only transfer messages from its own RDMA pinned memory. Hence, it is important that workers already prepare the messages in this memory region, otherwise an additional memory copy is needed. Furthermore, in high-contention scenarios an efficient method of delegating messages to the dispatcher is necessary. In our implementation, we use the same mailbox approach as for writes, but in a local setting. Requesters prepare the message in the RDMA memory and then set the last bit. The last bit again signals the dispatcher that the message can be transferred.

***.**Workload of Framework:

As mentioned before, we use different message sizes to simulate different application workloads. While the request in a request-response pattern typically transfers the request parameters (that are typically rather static in size), the response message can vary significantly. For example, in a key-value store the response depends on the width of the value. Whereas, in a distributed database, the response can vary between a few bytes and larger tuples grouped together in pages of multiple KB.

In our evaluation framework, we hence simulate the request-response pattern using the following setup:

- The requester always transmits 64 byte messages to the responder to simulate the request parameters. We use 64 byte messages since this allows us already to simulate typical request messages without giving up generality that we aim for: (1) We did not use smaller messages for the request, since this would not change the results since latency and bandwidth up to 64 byte messages is pretty stable [20]. (2) We did not use larger messages for the request since 64 byte messages allows us to already encode a large-enough number of parameters in the request.

- The responder replies with a response message, which we vary from 64 until we hit the bandwidth limit of our RNICs. Typically, as mentioned before, a response is very application specific and involves design decisions, such as memory allocation for each response or in advance. To avoid the effects of those decisions in our benchmark and concentrate on the effects of the communication, all response

messages are pre-allocated in a fixed-size memory region of multiple GB from which a requester can retrieve items from.

*.Experimental Setup:

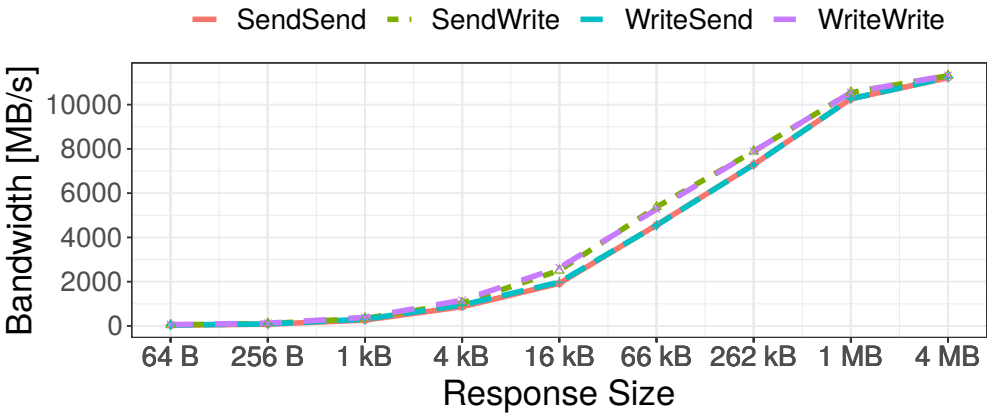
For all experiments, we use a cluster with 6 nodes featuring two Mellanox ConnectX5 cards connected to a single InfiniBand EDR switch. Each node has two Intel(R) Xeon(R) Gold 5120 Skylake processors (each with 14 cores) and 256GB RAM per NUMA node. Moreover, each node is equipped with two RDMA NICs (i.e., one per NUMA region). However, we will only use the second RNIC to create more load, i.e., more receivers, for our M-to-N scenario in Section 12.7. The reason is that M responders share all requests, therefore to get a similar load and ot make the numbers comparable to the previous experiments we utilize the second RNIC. The nodes run Ubuntu 18.04 Server Edition (kernel 4.15.0-47-generic) as their operating system and use the Mellanox $OFED - 5.0 - 1.0.0.0$ as the network driver. The benchmark is implemented with C++ 17 and compiled using GCC 7.3.0.

12.5 Evaluation: One-to-One

In the first part of our evaluation, we initially focus on evaluating the One-to-One communication topology. Different from the following experiments (N-to-One, M-to-N), we use a single thread to isolate fundamental performance differences of alternative request-response designs. Hence, the main focus is on evaluating the following combinations of verbs for implementing the request-response pattern where the first verb stands for request and the second verb for the response:

- Write/Write (denoted as WriteWrite)
- Send/Send (denoted as SendSend)
- Write/Send (denoted as WriteSend)
- Send/Write (denoted as SendWrite)

Moreover, since we run in single-threaded mode, we only use the design with a private connection per thread. We excluded the dispatcher design from this experiment because it is not necessary to minimize communications (queue pairs) in a one-to-one single threaded scenario. To evaluate the performance we requested 4 million responses and plotted the mean of 10 runs.



(a) Bandwidth, *One-to-One*

(b) Latency, *One-to-One*

Figure 12.3: Bandwidth and Median Latency — *For different response sizes in single threaded execution.*

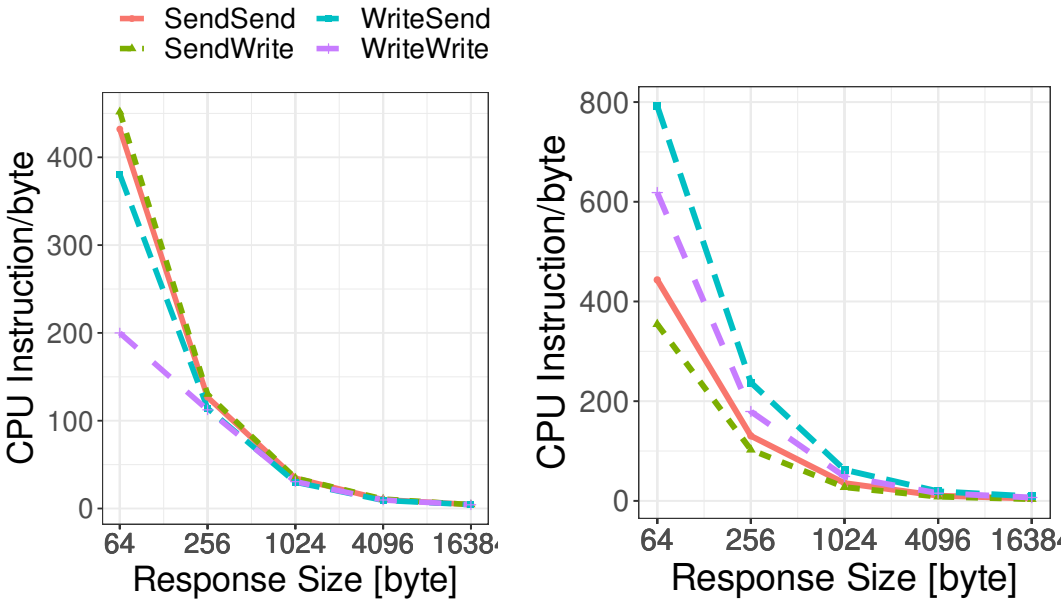
Figure 12.3a compares the bandwidth in MB/s for different response sizes. For the 64 and 256 byte sized responses, only a minor difference in bandwidth is observable. However, the relative difference between SendSend (worst) and WriteWrite (best) is still around 40%. In general, all combinations using a WRITE to implement the response outperform the versions in which SEND is used. This performance difference is even more pronounced for larger messages. For example, when using 16 KB messages, the performance differs by 700 MB/s between WriteWrite and SendSend. Finally, when reaching the bandwidth limit with 4MB response sizesthe gap closes again.

Next, we evaluate the corresponding median latency as shown in Figure 12.3b. The Y-axis shows the median latency in microseconds and the x-axis the response sizes. For small response messages up to 256 byte all combinations achieve a latency below 4 microseconds. Nevertheless, SendWrite and WriteWrite are again more efficient with below 3 microseconds. The difference again becomes even more pronounced for 16 KB responses as there is almost a 2 microsecond difference between SendWrite/ WriteWrite and SendSend/ WriteSend. However, as in the bandwidth experiment the difference between all combinations disappears with increasing messages sizes. It is important to note that 4 KB is the maximum transmission unit (MTU) which increases the latency for packets larger than 4 KB.

In order to determine the cost for one byte transferred, we additionally analyzed the CPU instructions normalized per byte. Figure 12.4a shows the instructions/byte on the requester and Figure 12.4b for the responder. We observe that for 64 byte messages there is an instruction overhead of up to 800 instructions for each byte received and respectively 450 instructions on the requester. However, with increasing response sizes the instruction overhead is amortized. For brevity we only show up to 16 KB because instructions are not further amortized. In other words, to save CPU cycles per byte one can increase the message size and trade-off higher latency.

12.6 Evaluation: N-to-One

So far, we investigated the effect of increasing response sizes in a One-to-One setup using only single-threaded communication. In a distributed system, however, a requester often has multiple incoming connections. Therefore, in this experiment we next examine a N-to-One scenario. Due to space limitations we will only show bandwidth in the following experiments. For the responder in this experiment, we only use a single thread to show the effects of scaling the requesters in isolation. Using multiple threads for responders



(a) Requester, *One-to-One* (b) Responder, *One-to-One*

Figure 12.4: Instructions per Byte for Requester and Responder — *For different response sizes in single threaded execution.*

is evaluated next in the M-to-N evaluation. To show the effects on bandwidth of using multiple requester threads, we gradually increase the number of servers used for requesters from 1 to 5. On each requester server we run 8 threads, therefore the maximum scale out with 5 server results in 40 requester threads. Additionally, we evaluate both design patterns (1) private connection and (2) dispatcher. The dispatcher design significantly reduces the connections needed, namely in the maximum scale-out from 40 connections to 5 only.

Figure 12.5 shows the bandwidth and the number of requesters used for the private connection per thread design. We again show the effect of different response sizes from 64 byte to 16 KB. Interestingly, the pattern which is observable in the 4 KB plot also applies for smaller sizes. For instance, 64 and 256 byte responses SendSend achieves 500 MB/s WriteWrite 1.2 GB/s with 16 requesters. When scaling further up the performance remains stable for all verbs and is hardly affected. This performance gap also leads to the fact that WriteWrite and WriteSend reach the maximum possible bandwidth per NIC with 16 requesters and 4 KB responses. SendSend and SendWrite, in contrast, stagnate after 16 requesters and only reach between 9.5 and 8 GB/s. This performance difference diminishes when using 16 KB messages, because SendWrite and WriteWrite were already

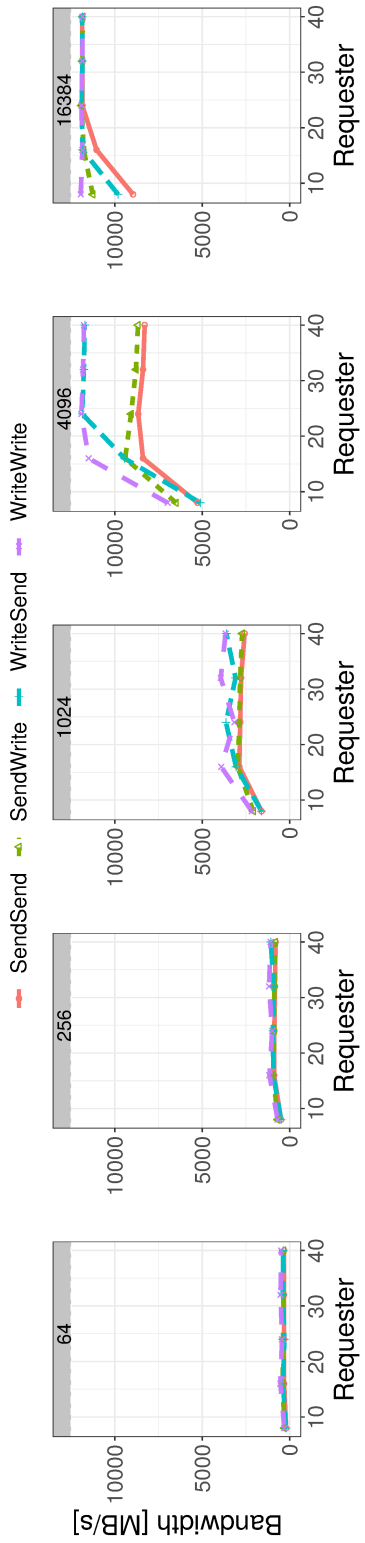


Figure 12.5: Bandwidth — *Private Connection, For different response sizes in N-to-One scenario.*

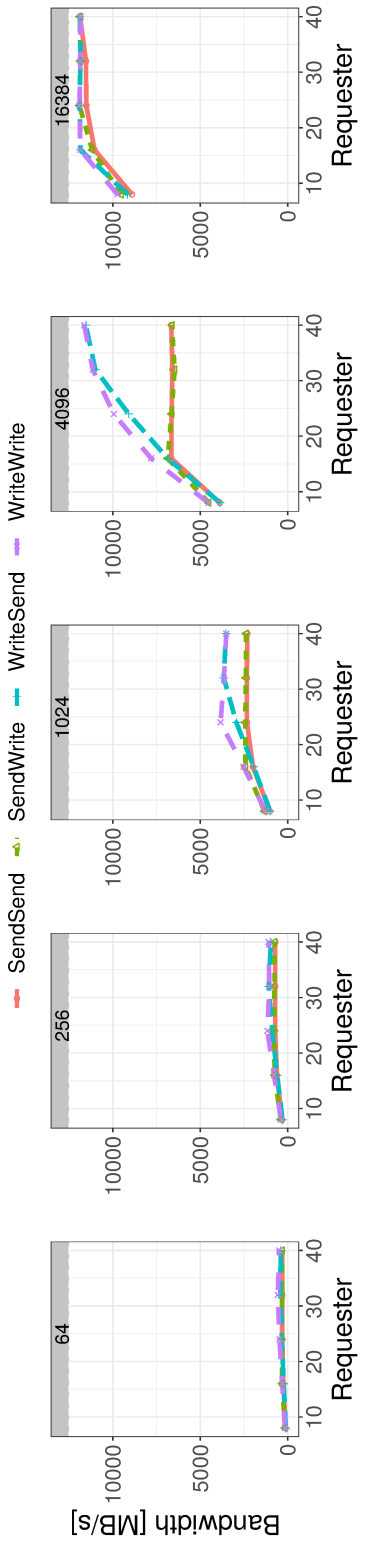


Figure 12.6: Bandwidth — *Dispatcher, For different response sizes in N-to-One scenario.*

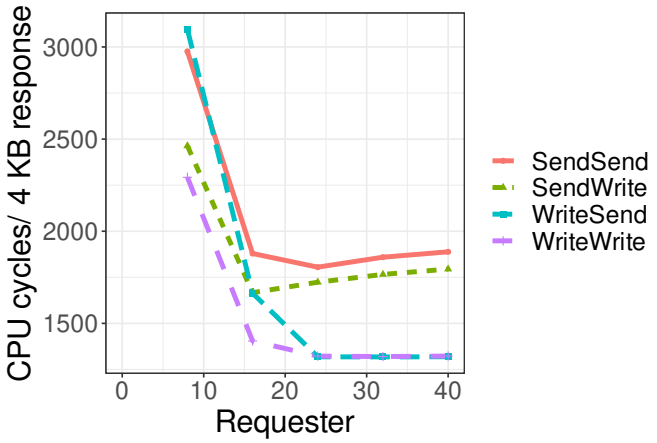


Figure 12.7: CPU Cycles spent — *4 KB responses, N-to-One*

network bound with 4 KB SendWrite catches up with 16 requesters. SendSend, however, is only bandwidth bound with 40 requesters and clearly not as efficient as the other verbs.

The decline in bandwidth for SendSend and SendWrite with 4 KB responses is quite surprising when comparing the numbers with the one-to-one experiment. In the previous experiment especially SendWrite appeared to be very efficient. A possible explanation might be the RNIC's incapability to handle many incoming send connections as described in [102]. Consequently, we will test this hypothesis with the dispatcher design which reduces queue pairs to a minimum. From Figure 12.6 we observe that the dispatcher has similar performance characteristics as the private connection per thread. We see the same pattern for smaller response sizes. However, for 4 KB we need 40 requester, instead of 16, to leverage the full bandwidth. Although, the dispatcher sends the messages asynchronously, the hand-over protocol (as mentioned in 12.4) from the requester thread to the dispatcher adds overhead. This overhead leads to a higher latency which in turn reduces bandwidth. Yet, we still have the decline in SendSend and SendWrite. Therefore, the number of connections did not cause the decline in our rack-scale experiment.

However, polling on the Shared Receive Queue could be more expensive than scanning over mailboxes in memory. Therefore we investigate the CPU cycles per 4 KB message measured on the responder shown in Figure 12.7. When the responder only serves 8 requesters polling incurs a high CPU cycle overhead, because the responder spins some time until a message is received. This holds true for iterating the mailboxes as well for polling the Shared Receive Queue. The more requesters (16) participate the less time is spent polling which decreases the cycles to around 1700 for all verbs. With even more requesters (> 20) the verbs diverge significantly, polling on the Shared Receive Queue

Method	Cycles %	Library
pthread_spin_lock	61%	libpthread-2.27.so
mlx5_poll_cq_1	17%	libmlx5.so.1.0.0

Table 12.1: Low-level metrics for SendSend (4 KB responses, 40 Requester threads, 1 Responder thread)

Responder	Avg. Req./Resp.	Avg. % Empty Mailboxes
6	15	83.3
12	7.5	91.6
18	5	94.4

Table 12.2: Average number of requesters threads per responder thread in different scenarios

always consumes around 1700 cycles and even increases again. In contrast, the cycles for scanning mailboxes is amortized with more requesters and continue to fall until the cycles per message stabilize at around 1100. When analyzing the Shared Receive Queue we observed that most cycles are spent in `phtread_spin_lock` and `mlx5_poll_cq` as shown in Table 12.1. Thus we conclude that polling on the Shared Receive Queue has a constant overhead which does not diminish with more requesters, i.e., messages.

12.7 Evaluation: N-to-M

Finally, in the last part of our evaluation, we analyze an N-to-M scenario by fixing the number of requesters but scaling the number of responders. However, different from the previous experiments we use both Mellanox cards and NUMA nodes per server to be

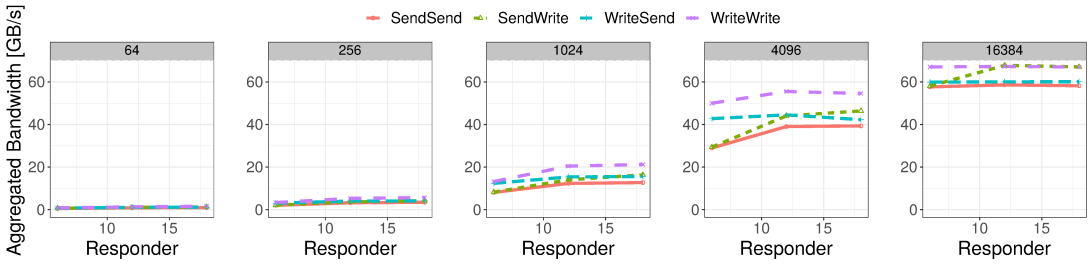
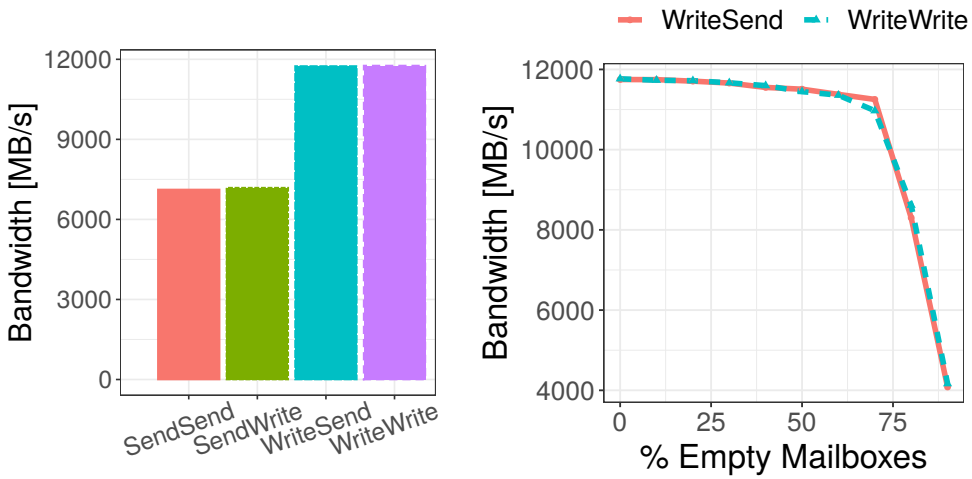


Figure 12.8: Private Connection: Aggregated BW for increasing number of responders – *different response sizes, M-to-N*



(a) Bandwidth, *4 KB response, 90 to One* (b) Effect of Empty Mailboxes, *90 to One*

Figure 12.9: Bandwidth and Effect of Empty Mailboxes — *4 KB responses*

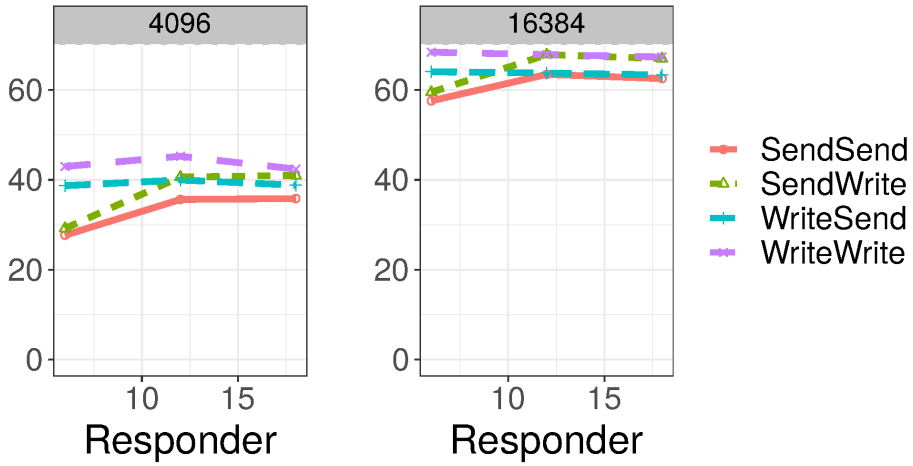


Figure 12.10: Dispatcher: Aggregated BW for increasing number of responders – *different response sizes, M-to-N*

able to run a requester / a responder per NUMA domain (i.e., in total we can simulate a cluster with 12 nodes).

From the previous experiment, we have seen that the network can be saturated with 16 requester threads per responder thread, therefore we scale responder threads in this experiment up to this ratio as well. More precisely, we use 9 nodes for requesters and run 10 threads per requester node (resulting in a total of 90 requester threads) while we use 3 servers for responders. The responder threads are scaled from 2, 4, and finally 6 per responder node resulting in 18 responder threads in total maximally.

For running the workload, each requester thread randomly chooses a responder thread for processing the request. Hence, with the setup that uses 6 responder threads in total, on average 15 requester threads need to be handled per responder thread while the other setups reflect a lower load per responder thread (as shown in Table 12.2). Moreover, we use the same setup as in the previous experiment and vary the message sizes while using both design patterns for the communication (private connections and dispatchers).

Figure 12.8 shows the aggregated cluster bandwidth and total number of responder threads used. For small messages up to 1 KB we see that the aggregated bandwidth rises with additional responders. As seen in Figure 12.4, small messages have a high instruction overhead causing a single core (responder) to be CPU bound. When adding new responders, i.e., CPU cores we thus can increase the throughput. Additionally, each requester needs to process fewer messages at the time. This effect can be observed in the 4 KB plot, in which SendWrite, surprisingly when looking at the N-to-One experiment, outperforms WriteSend with well over 40 GB/s. When comparing the aggregated bandwidth of WriteWrite with the one achieved in the N-to-One experiments its staggering that the bandwidth is far from the possible aggregated bandwidth of 66 GB/s. This effect can be explained with Table 12.2, when scaling the number of responders, the mailboxes are often empty and the threads thus simply poll without doing any actual work. Hence, the full aggregated bandwidth cannot be achieved.

To investigate this effect in more depth, we conducted the following microbenchmark: We evaluate a 90-to-1 setup to determine if the amount of mailboxes for the alternatives that use a Write operation (to implement the request) or connections mapped to the Shared Receive Queue for alternatives that use a Send operation to implement the request (i.e., SendWrite and SendSend) were limiting performance. Figure 12.9a shows the result of this microbenchmark using 4Kb responses. These results show that the alternatives that use a Write for the request (i.e., WriteWrite and WriteSend) are able to leverage the full bandwidth despite the high mailbox count. The alternatives that use a Send for

the request (i.e., SendWrite and SendSend), however, do not achieve a higher bandwidth, which is comparable to the findings from the N-to-One experiment.

Thus, focusing on Figure 12.8 with 4 KB responses we can see that the SendSend and SendWrite alternative with 6 responders achieve the expected performance. The difference between the 90-to-1 microbenchmark and the M-to-N Experiment is the number of average requesters per responder. In the 90-to-1 setup every requester was constantly transmitting messages, but in the M-to-N scenario only a fraction of the requesters send to the same responder resulting in many empty mailboxes.

From Table 12.2 we can observe that when having 6 responders an average of around 15 requesters need to be served, which corresponds to around 83 percent of empty mailboxes. Therefore the next microbenchmark, Figure 12.9b, shows the effect of empty mailboxes. We used the 90-to-1 setup but only a certain percentage of requesters transmit a message. The rest waited as long as it would take to contact another responder, i.e, 8 microsecond latency. We observe that the bandwidth is quite stable until 70 percent. After 70 percent the performance drops significantly with 80 percent empty mailboxes around 8 GB/s can be achieved and with 90 only around 4 GB/s. The achieved bandwidth of 8 GB/s correspond to the bandwidth each responder achieves in the M-to-N scenario.

To summarize, the effect of empty mailboxes is quite high when having more than 80 percent empty. When focusing again on Figure 12.8 we observe when increasing the number of receivers (see Table 12.2), i.e., reducing the average number of requester per responder, the bandwidth increases. For alternatives that use a Write operation to implement the request (i.e., WriteWrite and WriteSend) we observed that when having high empty mailboxes the instructions/message increases. Therefore, it takes longer to get a response on the wire. With more responders, this latency can be hidden, but only until enough messages are received, which explains why the bandwidth drops again after 12 responders.

In contrast to WriteWrite and WriteSend, SendWrite does not decline after 12 Responder threads instead it inclines. When looking at Table 12.2 we can see that only 5 requesters needs to be served on average. Therefore, the scenario is closer to the One-to-One scenario in which SendWrite outperformed WriteSend and SendSend. Furthermore, from Figure 12.7 we observed that the cycles of other verbs get amortized with new requesters. In this scenario, we effectively reduce the number of requesters per responder. This diminishes the amortization advantage of other verbs and in turn SendWrite is more competitive.

Lastly, we analyzed the dispatcher design in the N-to-M scenario. Figure 12.10 again shows the aggregated cluster bandwidth and total number of responder threads. For brevity we only show the measurements 4 KB and 16 KB as they are most interesting. For

smaller messages up to 4 KB we can see that the private connection design from Figure 12.8 outperforms the dispatcher design. Interestingly, with 16 KB messages WriteSend and SendSend in the dispatcher design achieve up to 5 GB/s more aggregated bandwidth compared to the private connection design. In contrast, SendWrite and WriteWrite are performing equally well in both designs.

12.8 RDMA Optimizations

In the previous sections, we evaluated various communication patterns in a holistic manner. This section, in contrast, discusses several low-level RDMA optimizations we analyzed in our benchmark code. Additionally, we present some optimization techniques we have not yet implemented but are worth mentioning.

We have seen that RDMA can achieve low latency and high bandwidth. In fact, the latency is so low that small inefficiencies directly translate to decreased bandwidth and increased latency. For instance, this impact has been observed in Section 12.7 when the number of empty mailboxes were high and the polling became expensive. Therefore, we implemented techniques like *selective signaling* and *inlining* [20, 102] for small messages up to 220 byte.

Selective Signaling. As described in Section 12.2, a signaled WQE creates a completion event once processing finished. However, the completion event is generated by the RNIC which incurs overhead and reduces throughput as shown in [102]. To avoid completion event generation all together, the application can specify the WQE to be *unsignaled*. However, even though the completion event is not constructed, it still consumes completion queue resources. Because the completion queue is bounded, the application needs to periodically post a signaled WQE and process the generated completion to avoid depleting completion queue resources. Selective signaling is a technique to reduce completion events and to prevent depletion of resources [20], thereby improving performance. The application posts n-1 unsignaled WQE and then the n-th WQE signaled.

Inlining is another technique, applicable to messages up to 220 byte, to reduce work inside the RNIC. In general, once the RNIC processes a WQE it fetches the payload of the message over the PCIe bus. Inlining allows the application to directly attach the payload to the WQE, which eliminates the need to fetch data over the PCIe bus. Consequently, the message can be faster transmitted aqnd latency decreases, as shown by Kalia et al. [102].

Another important aspect for an efficient usage of RDMA is **memory allocation and preparation**. In particular, the NUMA architecture has to be considered carefully when allocating RDMA memory to ensure that the RNIC fetches NUMA-local data. Otherwise, data is fetched from remote NUMA regions via the QPI/UPI bus incurring a high overhead. To allow the RNIC to write and read memory, the region must be pinned and registered on the RNIC first. This registration should be avoided on the hot path as it is quite expensive. Furthermore, to translate virtual to physical memory addresses, the RNIC caches virtual to physical address translations. To reduce address translation cache misses often huge pages are used [102].

A feature we implicitly exploited is **Data Direct I/O (DDIO)**, which is provided by recent Intel CPUs (Sandy Bridge and later). With DDIO, the DMA executed by the RNIC to read (write) data from (to) remote memory, places the data directly in the CPU L3 cache. DDIO supports two modes of operation [94]: (1) *Write Update* if the memory address is resident in the cache an in-place update is performed, and (2) *Write Allocate* causing allocation in the L3-cache if data is not yet cached. To reduce cache thrashing, DDIO's Write Allocate is limited to 10% of the L3 cache [94]. We measured the effect of DDIO during our experiments, especially when using WRITE in combination with the mailboxes. In our benchmark, the responder polled mailboxes, which placed them in the CPU cache. Incoming messages are then directly written to the L3-cache with DDIO. This saves memory bandwidth and reduces latency by avoiding a full cache miss.

An optimization for the mailbox design that we have not implemented, but should be mentioned, is exploited by L5 [64]. On the responder site, L5 uses a dense mailbox buffer, in which each client occupies only a single byte. Additionally, a message buffer is used, similarly to our mailbox design described in Section 12.4, in which the payload is written. To transfer a message, the requester writes the payload to the message buffer and then issues a second write of a single byte to the dense mailbox buffer to signal completion. The responder benefits from decreased cache misses and instructions, because only the dense region is polled. For instance, with 64 clients only one cache line needs to be polled. Especially, our N to M experiment has shown that scanning empty mailboxes is expensive and the dense mailbox design would have mitigated that overhead. However, the drawback of this design is that the requester needs to write two messages instead of one, which in turn influences the latency and increases the operations per second in the RNIC.

Finally, for completeness, we used the following performance relevant settings in all experiments: MLX5_SINGLE_THREADED=1, MLX_QP_ALLOC_TYPE="HUGE", and MLX_CQ_ALLOC_TYPE="HUGE". The first setting disables locking when used

in a single threaded application, and the two remaining variables allocate the queue pair and the completion queue on huge (2MB) pages.

12.9 Discussion and Conclusions

In this paper, we have presented a systematic evaluation of RDMA communication patterns for various use-cases. Our main findings are: (1) In the One-to-One experiment, we have seen that WriteWrite and SendWrite perform best. There was a 40% difference in performance between WriteWrite and SendWrite compared to SendSend and WriteSend. We further observed that the CPU cycles per byte decline with increasing messages. This fact is quite interesting as the trade-off between latency and CPU overhead can be tailored to the application needs. (2) In the N-to-One experiment, WriteWrite maintained the performance observed in the first experiment. Surprisingly, however, WriteSend outperformed SendWrite. We have seen that the reason why SendSend and SendWrite did not scale is that the cycles spent do not get amortized with additional requesters as it is the case for WriteWrite and WriteSend. Therefore, the constant overhead leads to a stagnation in bandwidth. (3) The results from the M-to-N experiment confirmed the previous findings, but added another interesting insight: WriteWrite and WriteSend performance is determined by the number of empty mailboxes.

12.10 Acknowledgements

This work was funded by the German Research Foundation (DFG) under grant BI2011/1.

13 ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA

Abstract

In this paper, we propose ScaleStore, a novel distributed storage engine that exploits DRAM caching, NVMe storage, and RDMA networking to achieve high performance, cost-efficiency, and scalability at the same time. Using low latency RDMA messages, ScaleStore implements a transparent memory abstraction that provides access to the aggregated DRAM memory and NVMe storage of all nodes. In contrast to existing distributed RDMA designs such as NAM-DB or FaRM, ScaleStore stores cold data on NVMe SSDs (flash), lowering the overall hardware cost significantly. The core of ScaleStore is a distributed caching strategy that dynamically decides which data to keep in memory (and which on SSDs) based on the workload. The caching protocol also provides strong consistency in the presence of concurrent data modifications. Our evaluation shows that ScaleStore achieves high performance for various types of workloads (read/write-dominated, uniform/skewed) even when the data size is larger than the aggregated memory of all nodes. We further show that ScaleStore can efficiently handle dynamic workload changes and supports elasticity.

Bibliographic Information

The content of this chapter was previously published in the peer-reviewed work 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 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. The contributions of the author of this dissertation are summarized in Chapter 6.

Table 13.1: Hardware landscape in terms of cost, latency, BW.

	Price [$/TB]	Read Latency [μs/4 KB]	Bandwidth [GB/s]
DRAM	5000	0.1	92.0
Flash SSDs	200	78.0	12.5
RDMA (IB EDR 4x)	-	5.0	11.2

13.1 Introduction

In-memory DBMSs. Decades of decreasing main memory prices have led to the era of in-memory DBMSs. This is reflected by the vast number of academic projects such as MonetDB [22], H-Store [105], and HyPer [110] as well as commercially-available in-memory DBMSs such as SAP HANA [62], Oracle Exalytics [67], and Microsoft Hekaton [52]. However, while in-memory DBMSs are certainly efficient, they also suffer from significant downsides.

Downsides of in-memory DBMSs. An inherent issue of in-memory DBMSs is that all data must be memory resident. In turn, this means if data sets grow, larger memory capacities are required. Unfortunately, DRAM module prices do not increase linearly with the capacity, for instance, a 64 GB DRAM module is 7 times more expensive than a 16 GB module [1]. Therefore, scaling data beyond a certain size results in an "explosion" of the hardware cost. More importantly, since 2012 main memory prices have started to stagnate [78] – while data set sizes are constantly growing. This is why research proposed two directions to handle very large data sets.

NVMe storage engines. As a first direction, a new class of storage engines [118, 161] has been presented that can leverage NVMe SSDs (flash) to store (cold) data. As Table 13.1 (second row) shows, the price per terabyte of SSD storage is about 25 times cheaper than the price of main memory. The key idea behind such high performance storage engines is to redesign buffer managers to cause only minimal overhead on modern hardware in case pages are cached in memory. This is in stark contrast to a classical buffer manager that suffers from high overhead even if data is cache resident [80]. Recent papers [118, 161] have shown that when the entire working set (aka hot set) fits into memory, the performance of such storage engines is comparable to pure in-memory DBMSs. Unfortunately, when the working set is considerably larger than the memory capacities, the system performance significantly degrades. This is because the latency of SSDs is still at least two orders of magnitude higher than DRAM (see Table 13.1 second row). This latency cliff mainly affects latency-critical workloads such as OLTP.

In-memory scale-out systems. A second (alternative) direction to accommodate large data sets is to use scale-out (distributed) in-memory DBMS designs on top of fast RDMA-capable networks [20, 55, 103, 128, 175]. The main intuition is to scale in-memory DBMSs beyond the capacities of a single machine by leveraging the aggregated memory capacity of multiple machines. This avoids the cost explosion that typically arises in scale-up designs. The main observation is that scale-out systems execute latency-critical transactions efficiently via RDMA. In fact, as shown in Table 13.1 (third row), the latency of remote memory access using a recent InfiniBand network (EDR 4×) is one order of magnitude lower than NVMe latency. As a result, systems such as FaRM [55, 56, 196] and NAM-DB [239] provide high performance even for latency-sensitive OLTP workloads. However, such distributed in-memory DBMS designs still require that all data must reside in the collective main memory of the cluster. This causes unnecessarily high hardware costs when the hot set is smaller than the complete data set.

ScaleStore. Given the two directions — single-node storage engines and in-memory scale-out systems — the question remains if one can combine the best of both worlds. In this work, we propose a novel distributed storage engine called ScaleStore (code available at [40]) that provides low-latency access to the aggregated memory of all nodes via RDMA while seamlessly integrating SSDs to store cold data. In contrast to single-node storage engines, this allows ScaleStore to accommodate the hot set in the aggregated memory to avoid the latency gap. However, unlike distributed in-memory systems, ScaleStore evicts cold data to cost-efficient storage.

Challenges. While combining distributed memory with SSDs appears intuitive, it triggers several non-trivial design questions: (1) Deciding on the optimal data placement in ScaleStore is challenging due to the various storage locations (i.e., local memory, SSDs, remote memory, and remote SSDs). Clearly, a naïve static allocation scheme could be used in which the storage locations are determined upfront (in an offline manner) to support a given workload. However, this prevents efficient support for shifting workloads where the hot set is changing over time [108] or for elasticity which is a key requirement for modern scalable DBMSs, especially in the cloud. (2) Another challenge is to synchronize and coordinate data accesses. Since data in ScaleStore is distributed across storage devices and nodes it is non-trivial to achieve consistency efficiently.

Distributed cache protocol. To address these challenges, as a first core contribution, ScaleStore implements a novel distributed caching protocol based on RDMA that operates on fixed-size pages. Our distributed caching protocol provides transparent page access across machines and storage devices. As such, worker threads can access all pages as in a non-distributed system. Furthermore, an important aspect is that the distributed caching

protocol dynamically handles shifts in the workload. For example, if the access pattern (i.e., which node requires which page) changes, the in-memory cache is dynamically repopulated. This means that pages are migrated from the cache of one node to another node. To enable high performance for workloads with high access locality ScaleStore dynamically caches frequently-accessed pages in DRAM on multiple nodes simultaneously. Finally, a last important aspect is that the distributed caching protocol coordinates page accesses across a cluster of nodes. This ensures a consistent view of the data despite concurrent modifications even when multiple copies are cached by several nodes.

High performance eviction. Because ScaleStore caches pages and handles workload shifts, the local DRAM buffer may fill up at very high rates, and thus unused (cold) pages have to be evicted efficiently. Existing strategies such as LRU or Second Chance are either too slow or not accurate enough. ScaleStore, therefore, employs a novel distributed high performance replacement strategy to identify cold pages and evict them efficiently. Importantly, our eviction strategy is generally applicable to arbitrary data structures, rather than being hard-coded to any particular data structure, which makes ScaleStore a general-purpose storage engine.

Easy-to-use programming abstraction. Despite being a complex system, ScaleStore offers a programming model that allows developers to implement distributed data structures in a simple manner. Typically, creating distributed data structures such as distributed B-trees is a very complex and tedious task. For instance, it is not uncommon for specialized RDMA data structures to have thousands of lines of code [256], even with hard-coded caching rules and no SSD support. The programming model of ScaleStore, in contrast, hides all this complexity and makes distributed data structure design as easy as local data structure design.

13.2 System Overview

In this section, we illustrate the main concept behind our system using a motivating example and introduce the main components.

13.2.1 A Motivating Example

In ScaleStore, page access is transparent: any node can access any page using its page identifier (PID) and the system takes care of page placement. Consider the B-tree as shown in Figure 13.1 I, which consists of a root page (P1) and four leaf pages (P2-P5). Figure 13.1 II illustrates how the pages might initially be distributed across a cluster of

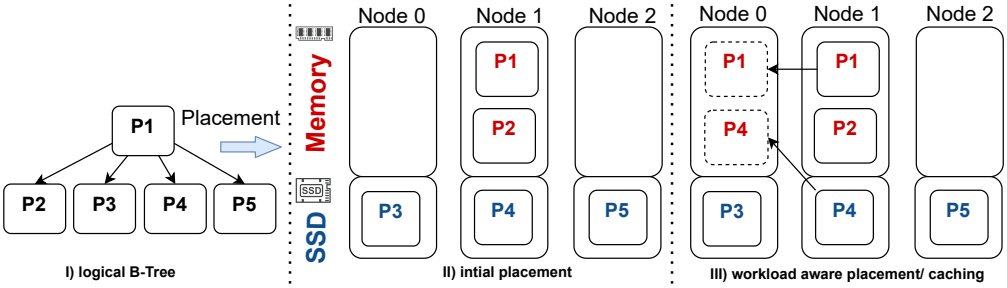


Figure 13.1: Overview of ScaleStore using a distributed B-Tree. II The B-tree pages can be spread across local/remote memory and SSDs. III The caching protocol optimizes how pages are laid out across machines

three nodes: Pages P1 and P2 are cached by Node 1, while pages P3, P4, and P5 are not cached and only reside on the SSDs. P1 and P2 also have a copy on SSDs, but for brevity these pages are not shown. Note that this placement is only a snapshot, and during operation, ScaleStore dynamically re-adjusts which pages are cached based on the workload. For example, as Figure 13.1 III shows, if Node 0 performs a lookup that involves pages P1 and P4, it will replicate P1 from the remote main memory of Node 1 and P4 from the remote SSD of Node 1. Note that Node 1 did not automatically cache P4. At this point, P1 will be cached by two nodes (Node 0 and Node 1), which is highly beneficial for frequently-accessed pages like the root of a B-tree.

If Node 2 accesses page P4, then in principle, Node 2 could either obtain the page from the SSD of Node 1 or the main memory cache of Node 0. Given current hardware latencies (see Table 13.1), ScaleStore always chooses the latter option. As a consequence, ScaleStore is effectively capable of combining the main memories of all nodes into a combined DRAM cache connected through a low-latency RDMA network. Furthermore, since page placement is dynamic and workload-driven, other placement scenarios are possible as well, and ScaleStore will dynamically adapt to those: For example, if the read-only working set is small and fits into the cache of each node, all data will eventually be fully replicated on each node and therefore enabling high performance by avoiding the network overhead of purely distributed systems. Another possibility occurs when the workload is partitioned, i.e., each node mainly works on a different subset of the data. In this situation, over time, each node will cache specifically its subset and thus exploit locality. Finally, note that our approach naturally adapts to workload shifts at runtime.

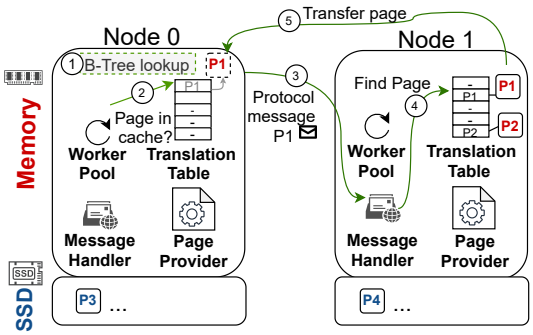


Figure 13.2: ScaleStore 's main components.

13.2.2 Main Components

As Figure 13.2 shows, each ScaleStore node consists of four main components: (1) *Worker Pool*, (2) *Translation Table*, (3) *Message Handler*, and (4) *Page Provider*. In the following, we briefly illustrate how these components interact at runtime.

Consider again the example from Figure 13.1 III where Node 0 wants to access the root node (page P1) from Node 1. Figure 13.2 illustrates how the page access is implemented. The first step from the application perspective is to invoke a worker thread ① to execute an operation on a data structure (i.e., B-Tree lookup). The worker then consults the local translation table to check if page P1 is already in its cache ②. If the page is indeed in the local cache, the page ID of P1 is translated to the memory address of the cached page and returned to the application. If the page is not in the local cache, it has to be fetched from a remote node. For example, in order to request a page from Node 1, Node 0 invokes the distributed page coherence protocol. For this, a page request ③ is sent to Node 1 to request page P1. When the message handler ④ on Node 1 finds that the page is already loaded into the remote memory using its translation table, the page is ⑤ directly transferred to Node 0. Otherwise, the page is loaded from SSD into the temporary memory of Node 1 before transferring it to Node 0.

13.3 Distributed Page Coherence

In this section, we describe our distributed page coherence protocol.

13.3.1 Protocol Overview

The basic idea of our protocol is inspired by cache coherence protocols like MESI that are used by multi-core CPUs to provide the illusion of a single unified main memory despite having multiple per-core caches and therefore duplicated copies of cache lines.

The MESI protocol. In MESI, each cache line has one of four eponymous states: (1) Modified (the cache line has been modified), (2) Exclusive (only a single copy exists), (3) Shared (there are multiple copies of this cache line), and (4) Invalidated (the cache line is out-dated). The protocol ensures coherence by using appropriate invalidation messages. For example, if a cache line is in the *Shared* state and a core wants to write to it, invalidation messages are sent to all cores that hold this cache line in *Shared* state.

Our protocol. In contrast to MESI, to work in a distributed setting, our protocol has several important differences. Instead of cache lines (usually 64 bytes), our protocol uses pages (e.g., 4 KB) as the unit of coherency to amortize network overhead and enable SSD support. These pages can be fixed in the local cache, which prevents page invalidation until an ongoing operation is finished. In a distributed cache coherence system this is required to avoid that the same page has to be fetched multiple times from a remote node due to invalidations which could quickly lead to a significant increase in overall latency. We further ensure robustness and fairness with anticipatory chaining, a technique that orders conflicts and thus ensures fairness and avoids starvation. Finally, our protocol implements sequential consistency, whereas multi-core CPUs typically implement lower memory consistency guarantees such as Total Store Order. Our protocol is separated into two distinct paths: (1) the local hot path and (2) the remote invocation. Figure 13.3 gives a high-level overview of the decisions in those paths. With this, ScaleStore follows the design principle "*make the common case fast*" and therefore, the local hot path is an important optimization to avoid unnecessary network messages when the page is already in the local page cache.

13.3.2 Local Hot Path

The local path in our protocol is a fast path that does not involve any remote messages – all decisions can be made locally. Especially for frequently-accessed pages, this obviously provides significant performance benefits. For instance, inner B-Tree pages that are rarely modified but frequently accessed are very likely to be cached on multiple nodes and can be accessed without any networking overhead. In the following, we explain the individual steps of the local hot path as shown in Figure 13.3.

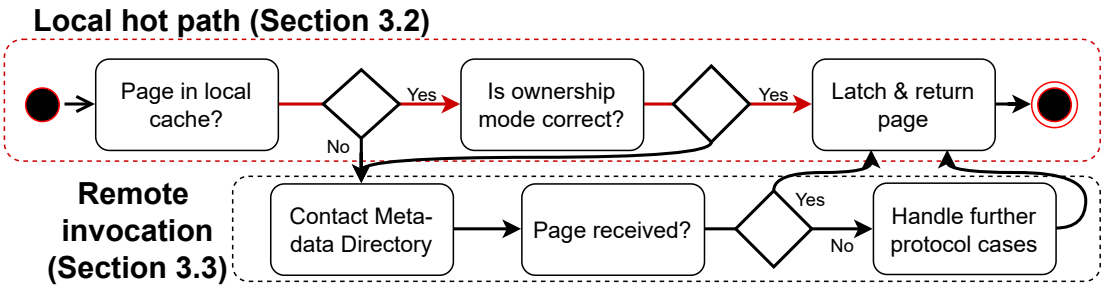


Figure 13.3: Protocol Flow.

Check ownership mode. The first step of every page access is to check the ownership mode. For this, the page translation table is used, which translates the page identifier (PID) to cache frames (similar to buffer frames in buffer managers). Besides the page data, we store the page latch, eviction information, and ownership metadata inside a cache frame. The ownership metadata describes the ownership mode for a page (i.e., what operations are allowed): (1) *node-exclusive* or (2) *node-shared*.

Before a worker thread accesses a page, it checks if the page is in the correct ownership mode. For example, the page has to be in the *node-exclusive* ownership mode for modifications (such as an update). Conversely, multiple nodes can access a page simultaneously only if the page is in the *node-shared* ownership mode. Note that node-exclusive and node-shared ownership regulate accesses on a *node-level basis*. That is, if a node owns a page in node-exclusive mode, then all worker threads can exclusively access this page.

Latch and return page. If the ownership mode is correct, the second step is that the page has to be latched for the concrete worker thread that wants to access the page. To efficiently synchronize worker threads within a node, we provide a hybrid latch [23] that combines a standard mutex with the option for optimistic access:

- *Exclusive:* Acquires cluster-wide exclusive access to a page for a worker thread. Note that this first requires a transition to the node-exclusive ownership mode for this page. Once the latch is acquired the page is fixed, i.e., it cannot be evicted from the local cache until it is unlatched.

- *Shared:* Acquires shared access to a page where multiple threads (and nodes) can access it. The ownership mode for this latch can be either node-shared or node-exclusive. As before, the page is fixed in the local cache.

- *Optimistic:* Allows an optimistic page read without acquiring the latch. The ownership mode for this latch can be again either node-shared or node-exclusive. In optimistic mode, the page is not fixed and can be evicted from the local cache.

While shared and exclusive modes use a traditional OS-supported read/write mutex, the optimistic mode sidesteps the mutex. This avoids cache line invalidations for latch acquisition and is crucial for making reads scalable on multi-core CPUs. Optimistic mode relies on a version counter, which is incremented for every page modification, to detect concurrent page modifications or ownership changes. If the version has changed (or the page was evicted), the reader must restart its read operation. If restarts keep happening, one can easily fall back to the shared latching mode, which will ensure forward progress.

13.3.3 Remote Invocation

In cases where the page is not in the local cache, ScaleStore triggers the remote invocation path. The remote path consists of several steps as shown in Figure 13.3 (lower part).

To query the current ownership mode and the location of a page, we first need to contact the *directory* node. To avoid a single directory node from becoming a performance bottleneck, every node is a directory for some of the pages. This also ensures that the SSD capacities are equally utilized because pages are only persisted at the directory nodes. In Figure 13.1, for example, the directory node of page P3 is Node 0. The directory has full knowledge about the state of the page such as which other nodes currently cache the page and in which ownership mode.

Which node is the directory? In our current implementation, the directory is the node on which the page is initially created at. During the allocation of the page, a unique page id (PID) is assigned to it. The directory node id is encoded in the first 8 bits of the PID to identify the directory node from the page ID. The remaining 56 bits indicate its page slot on the SSD at the directory node. For instance, `0x0100000000000000F` encodes node id 1 as the directory, and this page would be written to slot 15 on the SSD of Node 1.

Base case. Now that we know how to identify the directory node of a page from its page ID, we can request the page as shown in Figure 13.4 (which illustrates the base case). Node 0 – the requester – sends an *ownership request* ① to the directory. The ownership request describes to the directory what page is needed and whether *node-exclusive* or *node-shared ownership* is required. In the example of our B-Tree lookup in Section 13.2.1, Node 0 needs node-shared ownership for page P1. The directory then checks the ownership metadata ②. This is necessary because conflicts could happen, but for now, let us assume there is no conflict. Subsequently, the ownership metadata in the cache frame is updated ③ on the directory node to reflect where the page is located. Finally, an *ownership response* ④ is sent and the page is copied to Node 0 which is then

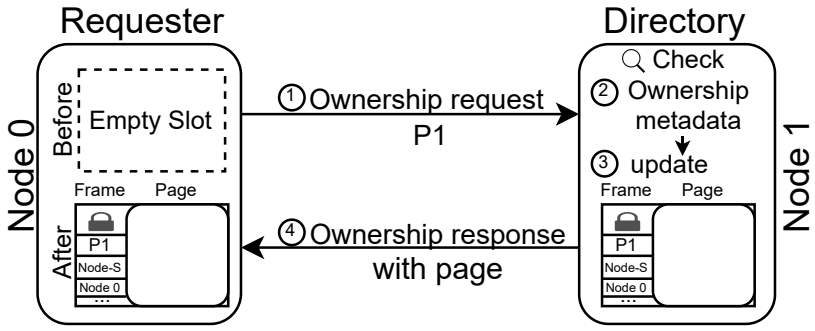


Figure 13.4: Base case of the Remote Invocation.

called a *node-shared owner*. Note, the node-shared owner stores the ownership mode in its local cache frame to support the local hot path as discussed before.

Conflict cases. As mentioned in step ② in Figure 13.4, two types of conflicts can happen during an ownership request: (1) exclusive and (2) shared conflict. A conflict in our protocol is always related to incompatible ownership modes, i.e., a requesting node (Requester) wants an ownership mode and another node already holds the page in an incompatible mode as shown below:

Requester wants:	Other Node has:	
	Node-exclusive	Node-shared
Node-exclusive	exclusive conflict	shared conflict
Node-shared	exclusive conflict	no conflict

In the following, we explain the two conflict cases.

Handling exclusive conflicts. An exclusive conflict occurs if one node requests a page of which another node is a node-exclusive owner, as Figure 13.5 illustrates. Regardless of whether the requester needs node-exclusive or node-shared ownership, both are incompatible with a node-exclusive ownership of another node. The first three steps are identical to the base case except that in step ② the directory will detect the conflict. Note that the directory does not necessarily have the up-to-date page content but only the metadata. In our example, Node 2 owns the page in node-exclusive mode, which implies that the page has been modified. Therefore, the directory does not store the old-page content and replies in ④ with the conflicting node id. The requester then sends an ⑤ *ownership transfer request* to Node 2, the current node-exclusive owner of the page. Node 2 transfers the page with the response ⑥ to the requester. Afterwards, Node 2 ⑦ removes the page from its cache. Notably, no acknowledgment message to the directory

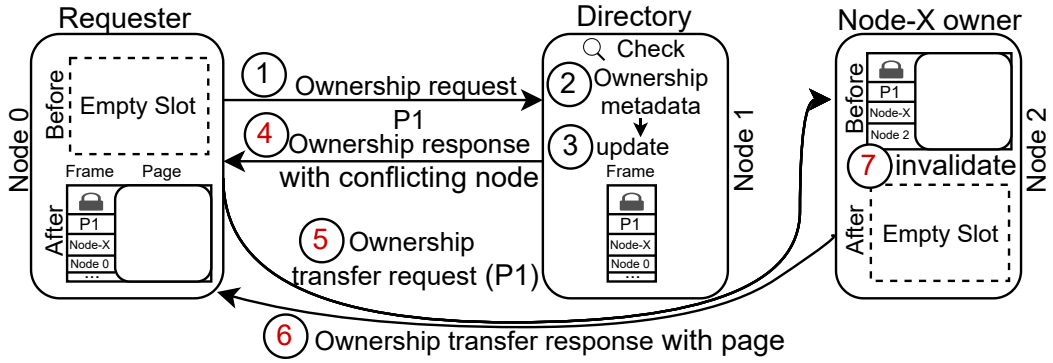


Figure 13.5: Exclusive Conflict and its Resolution.

is required because we employ a technique called *immediate metadata updates* which we will discuss in Section 13.3.4.

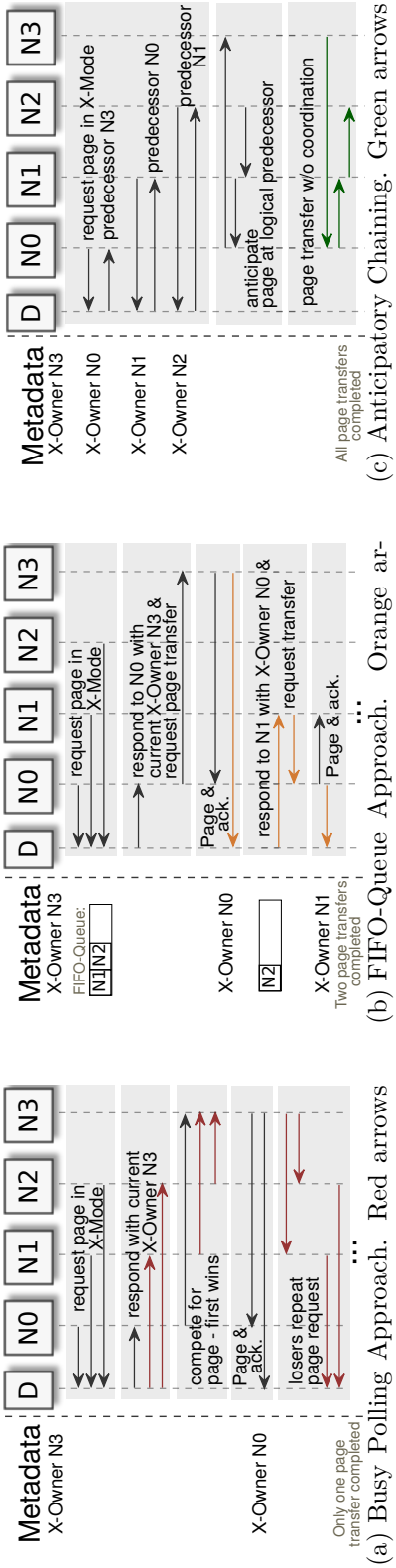
Handling shared conflicts. In shared conflicts, a page that is in node-shared mode is requested in node-exclusive mode by another node. To handle this case, the directory node detects the conflict and sends a list of node-shared owners to the requester. The requester then chooses one of the node-shared owners (at random) and sends a transfer ownership request to transfer the page to its own cache. Afterwards, the requester sends ownership invalidation requests to all other node-shared owners, which invalidate the page and ensure that the requester is the node-exclusive owner. Handling this conflict could seem expensive due to the invalidation messages, but often there are only a few node-shared owners of a page. Additionally, invalidation messages are sent in parallel to multiple nodes via low-latency messaging, as we will explain in Section 13.3.5.

13.3.4 Robustness and Fairness

After discussing the basic conflict resolution, we now describe how our protocol can efficiently handle conflicting requests coming from multiple nodes at the same time. For instance, when the B-Tree root is split or contended pages are accessed, multiple nodes compete for the same page. Such scenarios are not uncommon, and the challenge is to ensure fairness, avoid starvation, and keep latencies low.

13.3.4.1 Busy Polling.

The naïve approach for handling multiple conflicts is to let the nodes compete through busy polling on the directory node. As illustrated in Figure 13.6a, if three nodes ($N0$, $N1$, and $N2$) want to access a page in exclusive mode, all three nodes send ownership



(a) Busy Polling Approach. Red arrows highlight messages which are optimized in the FIFO approach.

(b) FIFO-Queue Approach. Orange arrows highlight messages that are optimized in Anticipatory Chaining.

(c) Anticipatory Chaining. Green arrows highlight the page transfer w/o coordination.

Figure 13.6: Handling multiple conflicts: Nodes N0, N1, and N2 request a Page in Exclusive (X-)Mode from Directory D.

requests to the directory (D). In the example, D responds with the current exclusive owner ($N3$) to all three nodes, which in turn send ownership transfer requests to $N3$. However, only the first of the three requests can succeed (exclusive mode). Therefore, the page is sent to $N0$ and D is informed, which triggers a metadata update to reflect that $N0$ is the new exclusive owner. The other requests fail, i.e., $N1$ and $N2$ back off and contact the directory again to get the new owner ($N0$). The above sequence is then repeated until all requesters finally get the page. As Figure 13.6a shows, this approach suffers from two issues: (1) many unnecessary messages are sent (highlighted in red), and (2) starvation can happen since a node might always lose to another concurrent request.

13.3.4.2 FIFO-Queue.

To avoid flooding the system with repeated requests for the same page, the requests can instead be queued at the directory as discussed in [27]. The requests are then served one at a time as determined by the queue. Figure 13.6b illustrates this approach in the same setting as before. The second request ($N1$) is dequeued and continued after the previous request of $N0$ succeeded, i.e., the metadata has been updated. The FIFO-queue might give a false impression that requests are ordered and fairness is achieved. However, for efficiency reasons, local workers can bypass the queue as discussed before (to enable the hot path). Therefore, as soon as the page can be accessed, the local worker threads can "steal" the page from the remote requester. For instance, in Figure 13.6b, if the directory node also accesses the same page, it can intercept the page requests of the other nodes $N0 - N3$ leading to an unfair access pattern. On the other hand, enqueueing the local workers also in the FIFO-queue may solve this problem but incurs high overhead on the hot path which is prohibitively expensive.

13.3.4.3 Anticipatory Chaining

To provide high efficiency and fairness at the same time, we propose *anticipatory chaining*. In this approach, a requester anticipates where the page will be just before its own turn (i.e., it anticipates its logical predecessor).

Immediate metadata updates. The key idea is that the metadata on the directory is directly updated once a page is requested. That way, nodes that request a page receive the anticipated page owner who will hold the page before them.

Figure 13.6c shows the general flow of our approach. Here, we see that the directory is updated immediately when the first request from $N0$ arrives at D. Moreover, when the second request from $N1$ arrives, D directly forwards $N1$ to the predecessor $N0$ (i.e., the

anticipated owner who holds the page before) even though the page might not have been physically moved yet. The same holds for the third requester $N2$ which is directed to $N1$. An important aspect of our protocol is that the page is moved strictly according to the order of the requests, and the directory cannot interfere; instead, the directory is treated like any other node, and thus we guarantee fairness. For instance, if D wants to access the page after the ownership request from $N3$, it is guaranteed that $N3$ receives the page before the directory. Overall, the approach is thus in stark contrast to the previous approaches (busy polling and FIFO-queue), which update the metadata only once the page is actually transferred to the new owner. To achieve this, the other approaches send acknowledgment messages once the page is transferred. Our approach saves this coordination step entirely. Additionally, because we eschew the FIFO-queue altogether, the local hot path is very efficient without sacrificing fairness.

Owner stability. While immediate metadata updates help to achieve fairness, they can become very costly if the page is not where the requester expects it to be. For instance, if $N1$ expects the page at the predecessor $N0$ but $N0$ evicts the page in the meantime, $N1$ would need to retreat to D to get the new page location. This would require additional communication, which results in increased latencies not only for $N1$ but for all chained requests. Therefore, an important aspect that anticipatory chaining provides is what we call *owner stability*; i.e., it guarantees that the page will be at the predecessor node where the requester expects it to be.

To decide when owner stability must be guaranteed we track a *conflict epoch* per page. This conflict epoch is incremented whenever the directory detects a conflict. Every node that requests a page remembers the current conflict epoch for this page. Let us consider the following example, initially, the conflict epoch for the page, which is requested by $N0$ is 5. Therefore, when $N0$ requests the page exclusively it remembers the conflict epoch at that time, i.e., 5. When $N1$ requests the page in exclusive mode, D detects the exclusive conflict, increments the conflict epoch to 6, and responds with the current exclusive owner ($N0$). If $N0$ then tries to evict the page, an eviction request is sent to D with the conflict epoch 5. D can thus detect that the current conflict epoch does not match the epoch from the eviction request. When such a mismatch is detected, owner stability must be guaranteed. Therefore, in this case, D declines the eviction request and $N0$ simply waits until the ownership transfer request from $N1$ invalidates the page as part of the exclusive conflict resolution. This guarantees that when $N1$ expects $N0$ to be the owner, this assumption holds true.

Deadlock avoidance. Lastly, our protocol also ensures that no deadlocks occur. For example, one scenario which may lead to a deadlock is if a node upgrades from shared

to exclusive ownership for a page. In this example, one node might want to copy the page from the upgrading node, but the upgrading node wants to invalidate the copying node at the same time (due to the exclusive conflict resolution). Therefore, both nodes are waiting for each other to finish and unlatch the page. In ScaleStore, we resolve such scenarios using the conflict epoch mentioned before. In the example, the upgrading node has the higher conflict epoch, and thus we detect the potential deadlock and the node with the lower conflict epoch needs to back off. Other edge cases, which we cannot describe due to space constraints, can be solved using conflict epochs as well.

Micro-benchmark. To show the effect of anticipatory chaining we execute a micro-benchmark with five nodes and one worker thread per node. In the micro-benchmark, workers either access multiple pages uniformly (*uncontended*) or all workers access a single page (*contended*). Figure 13.7 compares anticipatory chaining and the FIFO-queue approach. Anticipatory chaining performs generally better even in the uncontended scenario because fewer messages are sent. Moreover, in the contended scenario, anticipatory chaining achieves a 30% higher system throughput. This is because we achieve fairness between all participating nodes. In contrast, the FIFO approach has an unfair access schedule for the directory has two times higher throughput than the remote nodes. That is because the directory worker intercepts the remote requests as described in Section 13.3.4.2. This essentially leads to starvation and performance degradation, which is also reflected in the latency plot on the right-hand-side. Furthermore, we can observe that the latencies for anticipatory chaining are lower and the variance is significantly smaller compared to the FIFO-queue approach.

13.3.5 Low-Latency RDMA Messaging

We use Remote Direct Memory Access (RDMA) for all inter-node communication – including protocol messages and page transmissions. RDMA offers low latency and high bandwidth between nodes but requires careful engineering to achieve its potential. A number of RDMA implementations are available – most notably InfiniBand and RDMA over Converged Ethernet (RoCE) [219]. We use InfiniBand and Reliable Connection (RC), which guarantees that packets are delivered in order and without any loss.

RDMA implementations provide several communication primitives (so-called verbs) that can be categorized into the following two classes: (1) One-sided verbs (read/write) provide remote memory access semantics, in which one node accesses a remote node's memory over the network. The CPU of the remote node is not actively involved in the data transfer and thus is often used to save CPU cycles on the receiver. (2) Two-sided

verbs (send/receive), in contrast, provide channel semantics. In order to transfer data between two nodes, the receiving node first needs to publish a receive request; thus the remote CPU is actively involved.

Can we access pages via one-sided verbs directly? There are several systems (e.g., FaRM [55] or NAM-DB [239]) that use one-sided verbs to read or even write remote data directly. This often requires careful engineering to keep data consistent when multiple updates are applied concurrently. For instance, NAM-DB uses RDMA atomic fetch-and-add operations to lock and unlock remote objects. This operation allows one to atomically modify 8 byte values from remote memory. Unfortunately, compared to CPU atomics, the network atomics are very slow and even affect other non-atomic RDMA operations [102].

An even larger problem is that RDMA atomics and CPU atomics are not compatible [9, 102, 233]. While RDMA atomics work reasonably well in NAM-DB, which has been designed with decoupled storage and compute in mind, our design provides fast local accesses, which requires synchronization between remote and local accesses. The only option would be to use RDMA atomic operations for local accesses as well. However, this incurs a latency penalty in the order of 1 μs, making local accesses slow. Consequently, ScaleStore relies on remote procedure calls (RPC) for most operations. We exploit one-sided RDMA to (1) implement efficient RPC-based message passing and (2) transfer pages between nodes.

One-sided RPC. Using one-sided RDMA to build an RPC framework is very common [5, 55, 64, 213, 253]. In ScaleStore, we use a mailbox system very similar to the one from L5 [64]. In L5 every incoming message is written to a pre-specified memory area. This area is called the mailbox and incoming requests are detected when they are written to this region. To reduce the number of connections in ScaleStore, every worker is connected to a message handler on every remote node. The message handler provides a private mailbox for every worker which is continuously monitored to detect incoming requests. When a new request arrives, the message handler processes this request and replies to the worker's thread-local mailboxes. In ScaleStore all messages are cache-line size (64 byte) which makes this design very efficient as we do not need to handle variable-length messages. To avoid that the message handler becomes the bottleneck, we carefully optimized our design. For instance, we offload conflict resolution as seen in Section 13.3 to the worker threads of the requester instead of doing this inside the directory message handler. More specifically, the workers on the requester side implement the logic for conflict resolution. We also tried to resolve conflicts at the directory, however, it turned out that this impacts the overall performance negatively.

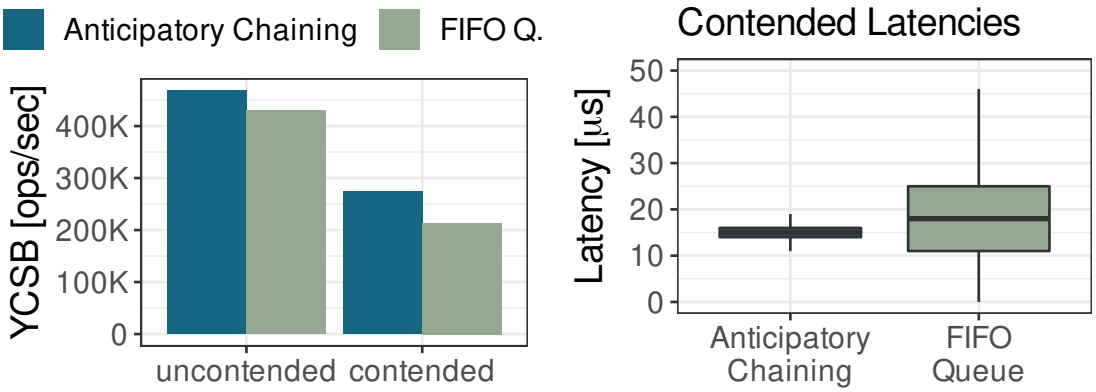


Figure 13.7: Effect of Anticipatory Chaining on throughput (contended and uncontended workload) and latencies (contended workload).

Page transfer using RDMA writes. The ultimate objective of our protocol is to transfer a page to the local cache. For the page transfer itself, we analyzed two possible strategies, one using one-sided reads and another using one-sided writes: (1) For one-sided reads, the message handler responds with a remote memory pointer to the page. The worker then reads (RDMA read) the page into its local cache. The benefit is that with one-sided reads the CPU of the remote message handler is not involved and that the page transfers are spread across all workers, i.e., threads. (2) The message handler directly writes the page to the cache of the worker.

Both strategies provide similar performance for the page transfer. However, we decided to use the second one for two reasons: First, the message handler can efficiently transfer pages in the background with RDMA since RDMA writes can be executed asynchronously. Second, the message handler can detect when a page transfer has been completed and immediately unlatch the page. In the first strategy, the worker however needs to send an additional message to indicate that the page transfer has finished.

Interestingly, one would assume that a single message from the worker would be enough, i.e., half roundtrip. However, due to the mailbox design, the message handler also needs to acknowledge this message. Otherwise, the previous message from the same worker could be overwritten before the message handler actually processes the first message and only sees the second message. All in all, the second strategy saves a full roundtrip.

13.4 High Performance Page Eviction

Modern RDMA networks are fast – which means that new pages may be added to the caches at very high rates. To avoid the overall performance from being throttled due to the lack of free pages, ScaleStore needs to evict cold pages quickly while making sure that hot pages stay in the cache. To achieve these goals, we separated our eviction process into two components: (1) a low-overhead strategy to track page accesses, and (2) a dedicated background thread – the page provider – which utilizes the access information to actually evict pages. This separation of concerns allows worker threads to focus solely on query processing while the page provider handles the eviction in the background.

13.4.1 Epoch-Based LRU Approximation

Downsides of existing strategies. To distinguish between hot and cold pages, the access pattern has to be tracked in some way. A well-known eviction strategy is Least Recently Used (LRU), which orders pages based on their access recency by maintaining an LRU-list. Cold pages are stored at the end of the list while hot pages are at the front, and thus one can reliably classify cold pages. While LRU would evict the right pages, maintaining an LRU list incurs high overhead for every page access and can easily become a scalability bottleneck in multi-core systems. An LRU-approximation such as *Second Chance* (or Clock) may be employed to avoid this overhead. Unfortunately, *Second Chance* only classifies hot pages well (since they are typically accessed very often), but it often evicts *warm* pages instead of cold pages. Evicting warm pages may lead to a significant slowdown in ScaleStore because pages that are required need to be read from remote memory or SSD, or even worse, pages may bounce between nodes. To identify cold pages more reliably, we need a more fine-grained distinction between the degree of hotness without incurring the overhead of LRU.

Epoch-based LRU approximation. The basic idea of our LRU approximation is to use a periodically-growing global epoch counter. This global epoch counter is used to track the access time for each page. When accessing a page the current epoch is determined from the global epoch counter and stored in the cache frame. This conceptually clusters pages that are similarly cold, warm, or hot and thus creates equivalency classes. In other words, some pages may share the same last accessed epoch and hence are treated equally by the eviction strategy. For instance, the B-tree root was likely accessed in the current epoch, whereas some leaf pages might have been accessed in an earlier epoch. This approach significantly reduces the cost to track access information because the global

epoch is rarely incremented and the check if a page has been accessed in the current epoch is a mere `if`-statement:

```
if(gEpoch > frame.lastEpoch) // if avoids unnecessary
    frame.lastEpoch = gEpoch; // cache-line invalidations
```

Compared to LRU, our approximation results in much higher performance (and multi-core scalability) while still accumulating enough access history to identify cold pages.

13.4.2 Page Provider

The main goal of the page provider is to maintain a sufficient number of free cache frames per node even under workload changes.

Sampling-based approach to find cold pages. With our epoch-based LRU approximation, the workers track access information efficiently, but what is left to discuss is how the page provider utilizes this information to find cold pages. To identify cold pages, we apply a sampling-based approach that iterates over the translation table (thus randomized). We sample N pages, sort them and determine a configurable epoch eviction threshold, e.g., 10% smallest epochs from the sample. Based on this epoch eviction threshold, pages are evicted from the cache. The sampling phase is repeated when the epoch changes or if the rate at which free pages are needed cannot be satisfied.

Evicting pages. When the page provider evicts a page, it is important to know whether it is dirty (modified) or not. As we know from Section 13.3, only the directory has full knowledge about the state of the page and, inter alia, if it is dirty. Therefore, there are two cases when the page provider evicts a page: (1) the evicting node is the directory, (2) or not. When the current ScaleStore node is the directory, it knows if the page is dirty. Typically, dirty pages are persisted to SSD, but when the page is replicated on other nodes, the directory can evict the page immediately. When the current ScaleStore node is not the directory of the selected page, we need to inform the (remote) directory node: An eviction request is sent to the directory, which then latches the page exclusively and checks if there are ongoing concurrent page requests with the conflict epoch. In the case of a concurrent page request, the eviction process for this page is deferred to maintain the owner stability mentioned in Section 13.3.4.3. When the directory discovers that the page is dirty, it copies the page to its cache and then eventually evicts it to SSD.

13.4.3 RDMA and NVMe Optimizations

The page providers communicate via RDMA messages with each other. To achieve high performance, we use the same setup as in the message handler, except that every page provider has a private mailbox on the remote page provider to ensure efficient communication. This allows one page provider to contact the page provider of the directory node directly.

RDMA optimizations. In contrast to the protocol messages for which low latency was required, we now optimize for high throughput. Therefore, we batch eviction candidates and send the batch to the directory. A batch has up to 100 pages that are all managed by the same directory. The batches are filled opportunistically in that the page provider tries to fill the batch, but if it does not find enough pages the batch is sent out earlier. Batching reduces the number of messages drastically and additionally enables another important optimization: For every page in a batch, the page provider on the directory checks if the page is dirty. If that is indeed the case, then the page must be persisted, i.e., first copied to the local cache of the directory. To optimize this copy request, we (1) use one-sided reads to directly copy the message from the remote node back to the directory node, and (2) we link multiple RDMA reads together. Instead of registering every single RDMA read with the NIC, we register a linked list of RDMA reads once. With that technique, we save precious CPU cycles because every operation which is explicitly registered with the NIC costs CPU cycles and the NIC can be better utilized (see doorbell batching in [102]).

NVMe optimizations. We use `libaio`, the asynchronous I/O API of the Linux Kernel, to saturate the bandwidth of high-speed NVMe SSDs when evicting pages to SSDs. To avoid OS caching effects, we open the database file with `O_DIRECT`.

13.5 Programming Abstraction

Designing and implementing efficient and scalable distributed data structures is a difficult task. Our abstraction allows programmers to port single-node data structures with minimal changes and ensures that all operations to the same page are sequentially ordered.

13.5.1 Interface

The key abstraction that ScaleStore's interface offers for application developers are different latch guards.

Latch guards for page access. In ScaleStore, we introduce latch guards that wrap around page accesses in a way that distribution is fully transparent and sequential consistency is guaranteed. As such, page guards act as a proxy for the translation from PID to the memory address and the acquisition of the correct ownership and latch modes. For each latch mode, we provide a guard:

- `ExclusiveGuard(PID)`: Latches the desired page in exclusive mode and ensures that the page is in node-exclusive ownership.
- `SharedGuard(PID)`: Latches the desired page in shared mode and ensures that the page is in node-exclusive or node-shared ownership mode (as both are compatible with reads, see Section 13.3).
- `OptimisticGuard(PID)`: Latches the desired page in optimistic mode, i.e., saves the version, and ensures the same ownership modes as the `SharedGuard(PID)` guard. Due to the optimistic nature a programmer needs to ensure that no concurrent changes occurred. Therefore, this guard provides a `hasChanged` method which indicates if a restart is necessary.

We further provide update and downgrade guards, e.g., an existing `SharedGuard` can be passed to a new `ExclusiveGuard` in order to upgrade from shared to exclusive. All guards have in common that they provide a `data` method to conveniently access the underlying page and the object encapsulated in that page, e.g., a B-Tree node. Moreover, with the destruction of the guards, the pages are unlatched.

13.5.2 Example: B-Tree Lookup

We now explain how the abstractions are applied in practice to develop the lookup operation in a distributed B-tree. In total, our B-tree code has only about 900 lines and thus is comparable to a single-node implementation. More importantly, the implementation is as easy as for a non-distributed B-tree. In our evaluation in Section 13.6 we use this distributed B-Tree implementation. The abbreviated C++ lookup code looks as follows:

```
0 bool lookup(KeyType key, ValueType& returnValue) {
1   restart:
2   OptimisticGuard g_parent(catalogPID); // get catalog
3   // get rootPID from catalog ...
4   OptimisticGuard g_node(rootPID);
5   if (g_parent.hasChanged()) goto restart;
6   auto node = g_node.data<NodeBase>();
7   if (g_node.hasChanged()) goto restart;
8   while (node->type.isInner()) { // traverse inner nodes
9     auto& inner = reinterpret_cast<Inner&>(node);
10    if (g_parent.hasChanged()) goto restart;
11    PID nextPid = inner.children[inner.lowerBound(key)];
12    if (g_node.hasChanged()) goto restart;
13    g_parent = std::move(g_node); // node becomes parent
14    g_node = OptimisticGuard(nextPid);
15    node = g_node.data<NodeBase>(0); // get next node
16    if (g_node.hasChanged()) goto restart;
17  }
18  auto& leaf = reinterpret_cast<Leaf&>(node);
19  SharedGuard sg_node(std::move(g_node));
20  // leaf latched; search key and return it ...
21 }
```

Synchronization is done using optimistic lock coupling [119, 120], and consequently we traverse pages using two optimistic guards: For the parent page `g_parent` (Line 2) and `g_node` for the current page (Line 4). The root node is stored in a catalog page, which is buffer-managed the same way as the B-tree itself. After every optimistic page access (Lines 5, 7, 10, 16), we validate whether that node was modified concurrently and restart if it was. Lines 8-17 traverse the inner nodes, at each level replacing the parent with the current node. Once we arrive at the leaf page (Line 18), we upgrade its optimistic guard to a shared guard.

Other data structures. As long as data is stored on fixed-size pages, the programming interface can be used to implement arbitrary data structures, e.g., hash table, barrier, or columnar storage.

13.6 Evaluation

In this section, we investigate the performance and scalability of ScaleStore (available at [40]) and compare it with other systems.

13.6.1 Experimental Setup

We conducted our experiments on a 5-node cluster running Ubuntu 18.04.1 LTS, with Linux 4.15.0 kernel. Each node is equipped with two Intel(R) Xeon(R) Gold 5120 CPUs (14 cores), 512 GB main-memory split between both sockets, and four Samsung SSD 980 Pro M.2 1 TB SSDs connected via PCIe by one ASRock Hyper Quad M.2 PCIe card. We use the Linux md software RAID 0 implementation and use direct block device access. The nodes are connected with an InfiniBand network using one Mellanox ConnectX-5 MT27800 NICs (InfiniBand EDR 4x, 100 Gbps) per node.

When not noted otherwise, we configured ScaleStore as follows: Every node has a 150 GB in-memory cache, and we use 4 KB pages. We use 20 worker, 4 message handler, and 2 page provider threads (pinned to NUMA 0). We use an optimistically-latched B-Tree implemented on top of ScaleStore using the programming abstractions from Section 13.5. In all experiments, the benchmark drivers are implemented in C++ and compiled together with ScaleStore into one binary.

Workloads. We use YCSB, a widely-used OLTP-style benchmark [43]. For all experiments, the key-value pairs use 8 byte keys and the values are randomly generated strings of 128 bytes. We use the following workloads: *100% Reads*, *95% Reads & 5% Writes*, *50% Reads & 50% Writes*, and *5% Reads & 95% Writes*. Reads are *point lookups* and writes are *point updates*. While the default distribution is uniform for most experiments, we also evaluate skewed workloads with a C++ Zipf generator [41]. Furthermore, the number of clients is defined by the number of worker threads, i.e., 20 clients per node. We execute one operation on a single client until completion, i.e., we do not batch operations nor execute them asynchronously. We only limit the throughput to achieve varying target throughput in our latency experiment in Section 13.6.7.3. We used a 30 second warm-up phase to measure the steady-state performance followed by three experiment phases of 30 seconds each. When not noted otherwise, we report the average system throughput, i.e., the accumulated average performance of every node. Finally, we use a distributed B-Tree which is implemented on top of ScaleStore for serving all operations.

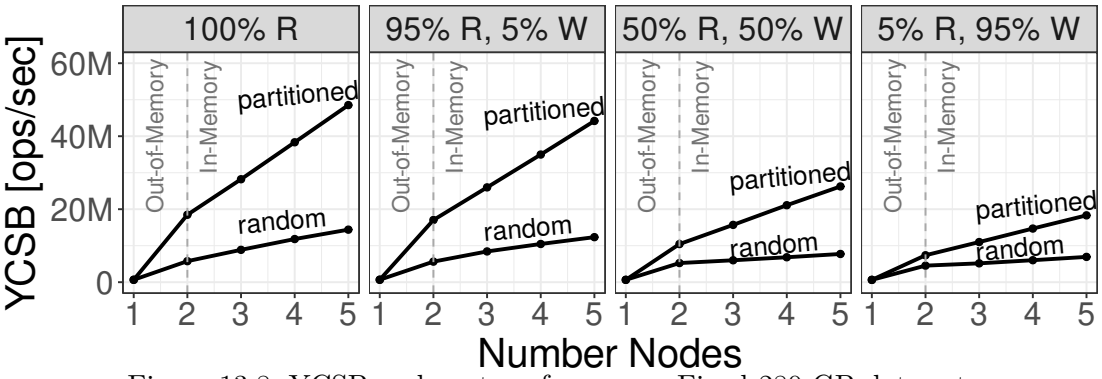


Figure 13.8: YCSB scale-out performance. Fixed 280 GB data set.

13.6.2 Scale-Out

We first conduct a scale-out experiment with the different YCSB workloads where we scale from 1 to 5 nodes and keep the data set size, i.e., hot set, constant at 280 GB. Hence, we show the different scenarios ScaleStore is designed for: if only one node is used, the data does not fit in the cache (150 GB). From 2 nodes on, however, the data can be fully cached. Furthermore, we use two access patterns: (1) a partitionable workload where nodes only cover distinct key ranges (and thus only parts of the data need to be cached per node) and a (2) random workload where workers access the full key range. This random access pattern is certainly extreme since all data is requested by all nodes. For that access pattern, the data set is too large that it can be fully cached at each node and therefore the performance is bound by the network latency.

Figure 13.8 shows the results of the experiment. When the workload is partitionable, ScaleStore achieves its peak performance with 5 nodes of up to $50M$ ops/sec in the read-only workload and $20M$ ops/sec in the update heavy workload. When the accesses are randomly spread across the cluster ScaleStore still scales and achieves around $15M$ ops/sec for read-only and $8M$ ops/sec for the update heavy workload. An interesting finding is that the performance for the update heavy workload (95% writes) is strictly bound by network latency. In fact, this is already the case for the 50% writes workload, which is why both workloads have similar performance. However, the relative performance difference between read-only and more write-heavy workloads can be observed in both access patterns partitioned and random. For random accesses it is slightly higher due to the average cache utilization, i.e., more cache misses. With 5 nodes the average cache utilization in the update heavy workload is just 35.1% compared to 98.7% in the read-only workload which implies that pages are frequently invalidated.

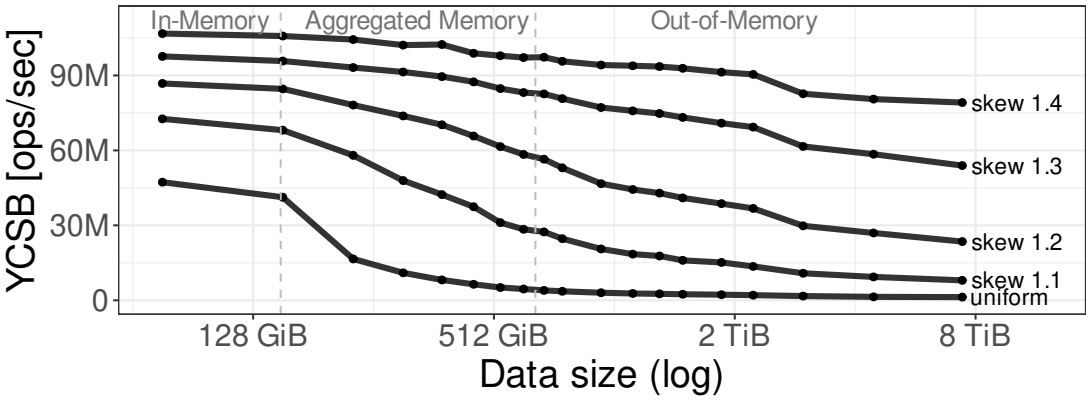


Figure 13.9: Data scalability with varying skew.

When looking at the speedup between single-node performance of $800K$ ops/sec and 5 nodes we can observe that this is an increase between 10 and 60 times. This shows that when the hot set out-grows the memory, the performance can be considerably increased when scaling-out to avoid the latency cliff of SSDs.

Ablation study. To better understand how ScaleStore achieves its performance, we now dissect how each optimization affects the performance of ScaleStore. For the experiment, we use the random read-only performance with 5 nodes from Figure 13.8. First, we disable all optimizations and then enable them step-by-step. The baseline system implements the high-level ideas of the protocol without any RDMA optimizations, a classical queue-based LRU-eviction strategy, a traditional single latched translation table, and a pessimistic lock-coupled B-Tree. The baseline system only achieves $1M$ ops/sec which already increases to around $3M$ when enabling the RDMA and message handler optimizations. Interestingly, already under this load, the standard translation table became a bottleneck due to the cache-line invalidations with a single reader-writer latch. Therefore, we designed our own optimistically-latched translation table which increased performance to $8M$ ops/sec. We resolved the next bottleneck, LRU-eviction, with our epoch-based LRU-approximation to increase performance to around $12M$ ops/sec. We finally partitioned our translation table to remove the single latch when inserting and deleting pages and used the B-Tree implementation as shown in Section 13.5 to achieve the final performance of around $15M$ ops/sec. All in all, to build a high performance distributed storage system many bottlenecks needed to be resolved before efficiently leveraging RDMA.

13.6.3 Data Scalability

Let us now focus on data scalability. We use all 5 nodes and analyze the read-only performance when increasing the YCSB data set size from 75 GB to 7.5 TB. Furthermore, we examine the effect of varying degrees of locality (skew). The results are shown in Figure 13.9. Overall, the results demonstrate that with ScaleStore we can gracefully bridge the latency cliff when data spills out from the local to aggregated cluster memory and then to SSDs.

We now look in more depth at the uniform (random) access pattern. What we can see is that with a data size of 150 GB and smaller, the performance is at its peak because the data set can be fully cached on all nodes. When increasing the data size, the performance degrades as expected. If the data does not fit into the local caches, pages need to be fetched from remote nodes (either from remote memory or SSDs) which increases page access latency as shown in the following table:

Median	99.9th Percentile	Data size	Storage
1.7 μs	3.0 μs	75 GB	in-memory
12.1 μs	30.6 μs	350 GB	aggregated memory
79.3 μs	178.8 μs	7500 TB	out-of-memory

Finally, ScaleStore benefits tremendously from locality in the access pattern as shown in Figure 13.9. This is because the hot path of our protocol can be exploited more frequently and our epoch-based eviction can reliably find cold pages. We see that even with a low skew of 1.2, ScaleStore achieves around $22M$ ops/sec with a data set size of 7.5 TB. The more locality the higher the performance resulting in around $85M$ ops/sec with 1.4 skew and 7.5 TB data size.

13.6.4 ScaleStore vs. GAM

Closest to the functionality of ScaleStore is GAM [27] (available at [37]), which is a state-of-the-art Distributed Shared Memory system using an RDMA-based cache coherence protocol. Different from ScaleStore, GAM is a pure in-memory system that provides a unified address space across a cluster of nodes but it does not provide the possibility to evict data to SSDs. Unfortunately, we needed to reduce the total data set size to 30 GB since GAM's hash-table performs suboptimally with larger data sets. Furthermore, we split the evaluation in a single-node scale-up and a single-thread scale-out experiment. We did not use a multi-threaded scale-out experiment as before, since GAM did not work in this setup even after thorough investigations (possible reasons are the different

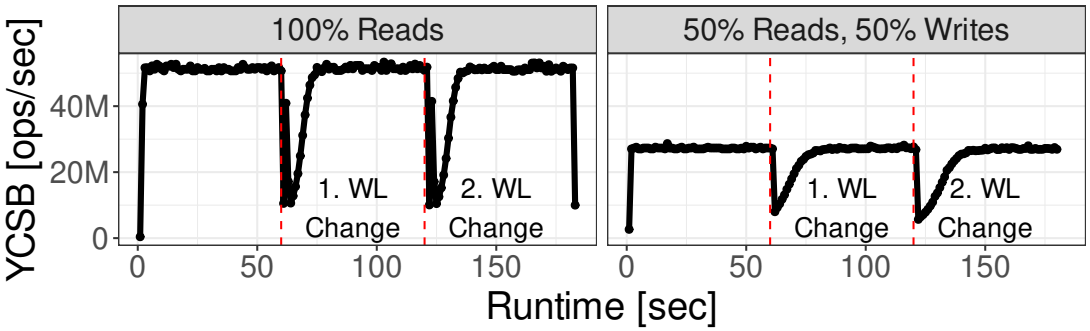


Figure 13.10: In-memory performance with workload shift.

Mellanox drivers and compiler versions). The results of the experiment are shown in the following table:

Configuration		100% Reads ScaleStore	100% Reads GAM	50% Reads/Writes ScaleStore	50% Reads/Writes GAM
Scale-up (1 Node)	1 Thread	**712K**	558K	**690K**	534K
	20 Threads	**12,465K**	2,654K	**11,566K**	2,752K
Scale-out (1 Thread)	2 Nodes	**1,449K**	513K	**396K**	250K
	4 Nodes	**3,075K**	850K	**625K**	546K

When looking at the results, we can observe that ScaleStore outperforms GAM in all workloads, most notably is ScaleStore's performance with 20 threads. One important aspect is that GAM uses a slightly modified YCSB runner with a hard-coded value for updates instead of randomly generating 128-byte values; thus, we use the same setup (see here).

13.6.5 Workload Change

Next, we show ScaleStore's ability to adapt to changing access patterns. Therefore, we partition the data uniformly across our 5-node cluster, such that every partition is 120 GB.

Initially, every node accesses only data from its local partition which results in a situation where the partition can be kept in the local cache. Then, in regular intervals, we reassign the partitions. For instance, we reassign Node 0 to access data in the partition of Node 1 instead of accessing its local data. This procedure is repeated multiple times to show that ScaleStore can reliably handle workload changes. While this workload may not be realistic, it shows the extreme case in which the cache needs to be re-filled completely while the eviction strategy handles this sudden workload shift.

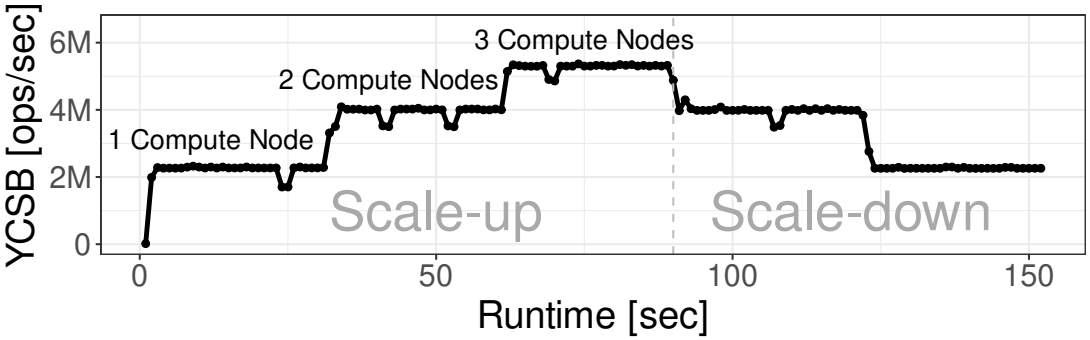


Figure 13.11: Elasticity in a decoupled architecture.

The results are shown in Figure 13.10 for the YCSB read-only and mixed workload. As we can see, ScaleStore recovers extremely fast already after a couple of seconds to the baseline performance of $50M$ respectively $28M$ ops/sec.

13.6.6 Elasticity

In the previous section, we showed how our protocol handles workload changes but with a fixed cluster size. In the following, we evaluate the elasticity in a decoupled storage and compute architecture which is typically used by cloud DBMSs today. We use two storage nodes that accommodate a 250 GB data set. To show the elasticity, we scale the number of compute nodes at runtime. Every compute node only has 10 GB of cache to avoid that most data is replicated to the compute layer. We start with a single compute node and add another compute node every 30 seconds until we reach three nodes, after which we disable them one by one again. In Figure 13.11 we can see that ScaleStore leverages the additional compute resources and that performance scales. Overall, the adaption happens in a few seconds after a new compute node has been added.

13.6.7 System Comparison

In this section, we provide a system comparison to specialized systems in the following settings: single-node in-memory, single-node out-of-memory, and distributed in-memory.

13.6.7.1 Single-Node In-Memory.

In the single-node in-memory scenario, we compare ScaleStore to a state-of-the-art in-memory B-tree [39, 229] that uses optimistic-lock-coupling (OLC). We use $400M$ records (120 GB) to ensure that the entire B-Tree fits in memory. Moreover, both the pure in-memory B-tree and ScaleStore's B-tree have the same page layout and synchronization

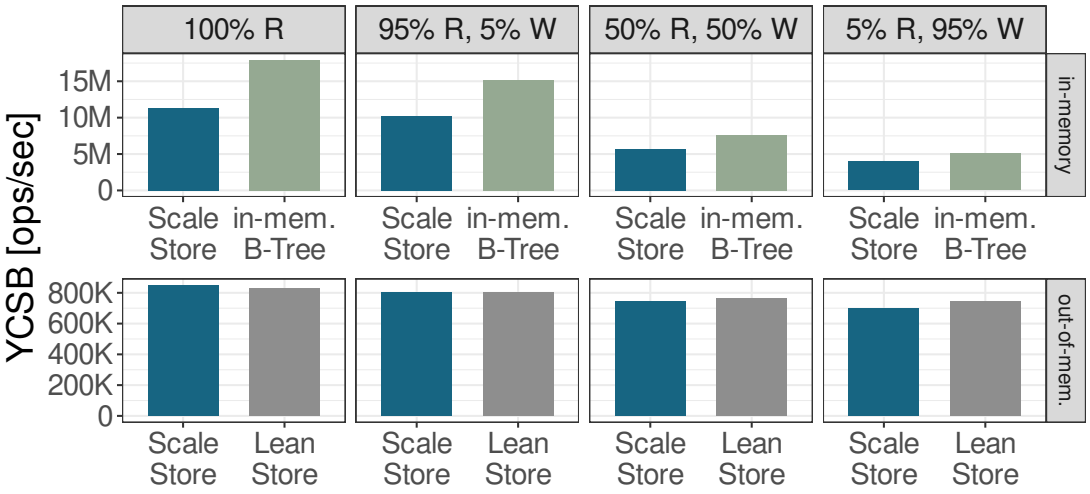


Figure 13.12: Performance comparison on a single node system.

protocol. This allows us to quantify the overhead of ScaleStore being able to scale-out and handle out-of-memory workloads.

The results of the comparison are shown in Figure 13.12 (upper row). We can observe that the overhead in the read-only workload is more prominent than in the write-heavy workloads. This is because the optimistic scheme for read-only workloads is very efficient and thereby highlights the extra work of ScaleStore. In ScaleStore, for every page access, a page identifier is translated to the corresponding memory pointer. While this indirection adds some overhead, it is necessary to be able to scale-out and handle out-of-memory workloads (i.e., pages on SSDs). The overhead diminishes with higher write ratios, for which CPU cache line invalidations and lock contention dominate.

13.6.7.2 Single-Node Out-of-Memory

Next, we focus on experiments with data sets that are larger than memory and compare ScaleStore with LeanStore (available at [38]), a recent high performance storage engine. For LeanStore and ScaleStore, the cache size is set to 150 GB while we use $1B$ YCSB records (300 GB). As such, the data size is twice the in-memory capacity. Figure 13.12 (lower row) shows that both systems perform nearly identical because both systems are I/O bound. This shows that ScaleStore integrates NVMe SSD efficiently.

13.6.7.3 Distributed In-Memory

Moving beyond a single-node deployment to a distributed setting.

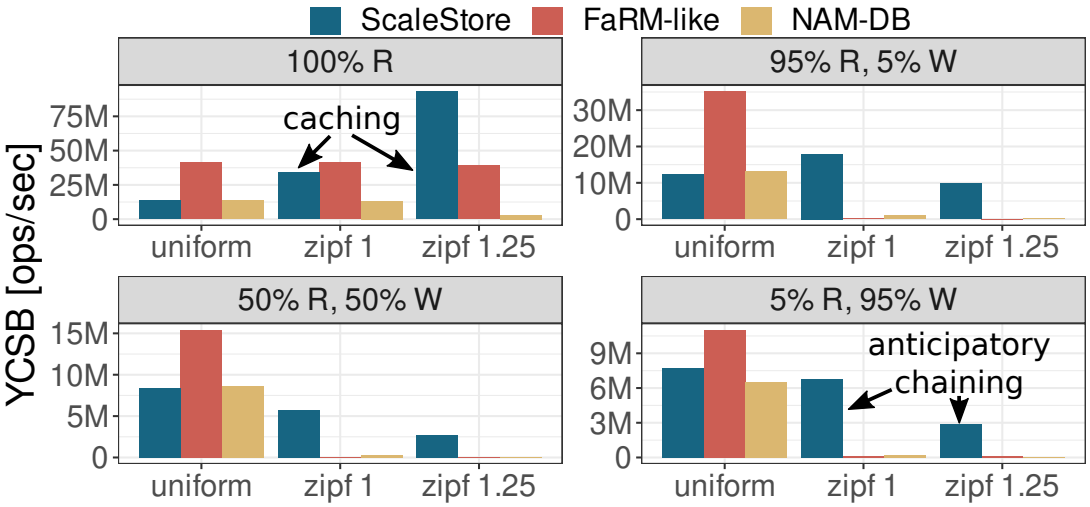


Figure 13.13: Distributed in-memory throughput comparison.

Competitors. In this section, in contrast to ScaleStore, all competitors are pure distributed in-memory systems that do not come with the overhead of integrating SSDs. One key feature of ScaleStore is that we do not require static partitioning but dynamically adapt on runtime. For this reason, we compare with *NAM-DB* [239]. NAM-DB does not rely on static partitioning either as it reads remote data with one-sided reads and updates remote data with one-sided writes respectively. To be fair we disabled transactions to cleanly compare the protocol overhead with ScaleStore.

Different from ScaleStore, NAM-DB uses shared and exclusive latches which are (un)latched with RDMA atomics, i.e., one-sided. The second competitor is *FaRM* [55] which in contrast to NAM-DB uses an optimistic synchronization scheme. Since FaRM is not publicly available, we re-implemented a FaRM-like approach based on lock-free one-sided reads as described in [55]. The lock-free one-sided read retrieves the remote value and to check if the read was consistent, we validate the versions stored in every CPU cache line. For updates, the lock-free one-sided read is used to copy the record to a thread-local buffer, the record is then modified and sent back to the owner. The owner checks the version(s) of the modified record against the local version(s), if they are equal the modified record is installed. Finally, for both competitors, we use a pre-allocated array (key-value pairs collocated), i.e., no collisions, which allows them to use a single one-sided read to fetch remote data. This is different from ScaleStore which uses a distributed B-tree that also supports range queries.

Throughput. For this experiment, we use $1B$ YCSB-records, distributed across 5 nodes. Figure 13.13 shows the throughput for uniform and skewed accesses. First, we

focus on the uniform access pattern. When looking at the read-only workload, one can see that FaRM outperforms NAM-DB and ScaleStore. This is because only a single very efficient one-sided lock-free read is needed to read a remote record, while NAM-DB requires 3 operations, i.e., two RDMA atomics and one one-sided read, and is thus bound by the latency of those operations. Similar to NAM-DB, ScaleStore is also latency bound. This is because only 50% of the data can be cached in this workload, and thus pages are read from remote nodes. Additionally, hot pages cannot be predicted with a uniform workload reliably.

With the skewed access patterns, the results look different. For the read-only workload, we can observe that with more locality (skew) the performance of ScaleStore increases. With a light skew of 1, the performance is comparable to FaRM, and with 1.25 we outperform them. In contrast to FaRM, NAM-DB suffers from locality even in the read-only scenario. The reason is that RDMA atomics limit the performance drastically; the higher the skew the more requests are routed to a subset of the nodes which cannot sustain the number of RDMA atomic-operations [102]. Finally, when looking at the skewed write-heavy workloads, we can see that ScaleStore outperforms NAM-DB and FaRM and also provides a robust performance for higher skews (contention). In the write-heavy workload (95% writes), the performance of NAM-DB and FaRM drops significantly whereas the performance of ScaleStore degrades gracefully. While FaRM and NAM-DB compete in a busy polling manner (with exponential back-off), we use anticipatory chaining as discussed in Section 13.3.4.

Latency. In addition to throughput, we additionally analyzed the latencies for the read-only and the write-heavy workload with uniform and skewed (1.25) access patterns. For this experiment, we increase the target throughput until the system cannot sustain the required throughput anymore. Based on the target throughput, every worker is assigned a schedule when it needs to send the next operation. This schedule is generated based on a Poisson process, where the time between two operations is exponentially distributed. When a worker misses a scheduled operation, we send it as soon as possible and include the wait time, i.e., we correct the latencies. This means that latencies spike as soon as the target throughput is not sustainable anymore.

We show the median and the 99th percentile latencies in Figure 13.14. As expected in the uniform distribution the latencies of NAM-DB and ScaleStore spike before FaRM since FaRM can sustain the target throughput longer. ScaleStore's latencies are comparable with NAM-DB even though NAM-DB has a very simple access scheme, i.e., two RDMA atomics and one RDMA read, whereas our protocol is more complex and uses larger transfer units (4 KB pages due to the SSD integration), yet it only adds a marginal

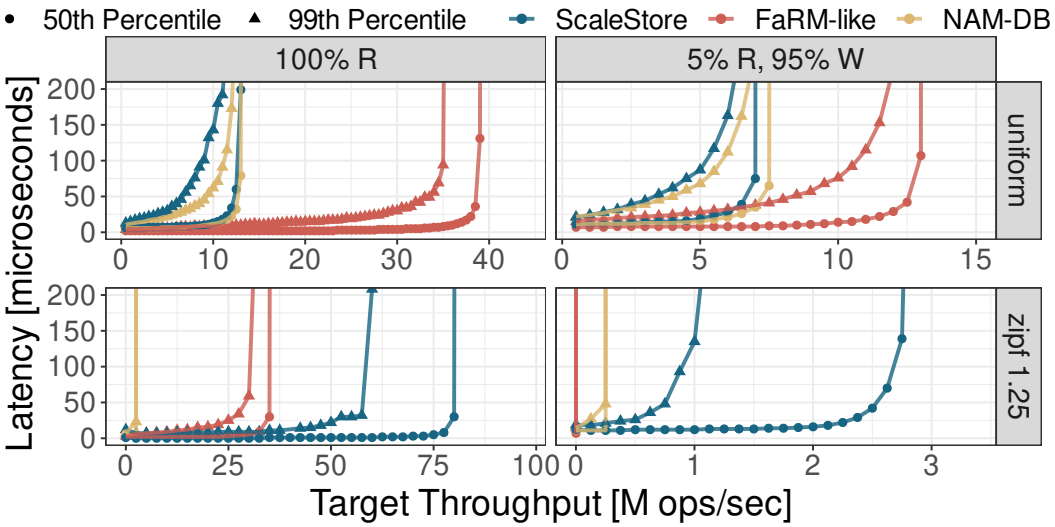


Figure 13.14: Distributed in-memory latency comparison.

overhead. Moreover, as mentioned before, in the read-only workload we only cache 50% of the workload and constantly fetch new pages from remote nodes. The eviction batches pages to sustain the rate of incoming pages and to maintain enough free space. Batching increases the 99th percentile latency because a worker may wait for a page that is currently in the eviction process. However, the median latency is not affected and thus comparable to NAM-DB.

In the skewed read-only workload, we can observe the benefit of caching in ScaleStore. Whereas in the skewed write-heavy workload, anticipatory chaining allows ScaleStore to achieve a higher target throughput than both competitors.

Discussion. To summarize, the results show that our system provides robust performance across different workloads and even outperforms pure distributed in-memory systems for skewed workloads due to caching and anticipatory chaining. This is especially prevalent in the write-heavy contended scenarios in which the performance degrades gracefully, compared to our baselines.

Let us mention that we handle out-of-memory workloads and as a result require an additional indirection. This prevents ScaleStore to use FaRM's one-sided lock-free reads and thus have a slightly higher latency in the uniform workloads. Moreover, important to note is that we measured the best case scenario (upper-bound) for NAM-DB and FaRM, where we know exactly where data needs to be read; i.e., they only require a single one-sided read to fetch the data. In the case where deletes and inserts need to be supported, typically direct reads can not be used in NAM-DB and FaRM since additional indexes are required to find the memory location. Hence, we argue that the benefits

of our implementation which also supports such cases outweigh the slight performance overhead.

Another interesting finding was that ScaleStore can benefit from the locality. In this case, our system can leverage caching and exploit local DRAM. In contrast, NAM-DB cannot fully use locality, even when data is fully partitioned, because the RDMA atomics are not compatible with local CPU-atomics, which means that NAM-DB requires two RDMA atomics even when accessing local data [239]. While FaRM can fully exploit locality they do not provide a general-purpose caching solution. FaRM has hard-coded rules for some data structures, i.e., cache inner nodes of their B-Tree implementation. Since they do not have a general caching protocol, they can only exploit locality if such hard-coded rules are provided. ScaleStore instead provides a general-purpose caching solution combined with a high performance eviction that works for arbitrary data structures.

13.7 Related Work

This paper is related to the following distinct lines of work:

RDMA-enabled Systems. Recent work on RDMA-enabled in-memory DBMSs [16, 55, 56, 103, 132, 134, 196, 239–241, 247, 256] and key-value stores [104, 129, 149, 166] achieve unprecedented performance for distributed systems. However, most do not support high performance NVMe SSDs to economically store cold data as ScaleStore.

In a similar spirit as ScaleStore, some of the other systems [55, 56, 196, 239] expose a distributed shared address space in which every node can access every data item. To exploit the shared address space, one-sided RDMA operations are used to directly access remote data and traverse data structures such as B-Trees or hash tables. To prevent multiple round-trips such systems rely on different approaches: Either data is merged with the index [55, 56, 196] or the index is fully or partially replicated [34, 150] at all servers, which has other downsides such as keeping data consistent under replication. Kalia et al. [103] thus avoid replication of the index and partition the data set. However, when workload changes they need to re-partition the data to exploit locality or fall back to expensive distributed transactions. In contrast to these approaches, ScaleStore instead provides a dynamic caching protocol that automatically adapts the placement and replication of data to the workload. Redy [247] is a cloud service that provides high performance caches by extending the local memory with remote memory of unutilized remote machines. Besides these research proposals, several industrial-strength products

have adopted RDMA in existing DBMSs [89, 163, 175]. Oracle RAC [175], for example, has RDMA support, including the use of RDMA atomic primitives. Microsoft SQL Server [128] uses RDMA to extend the buffer pool of a single node. They leveraged the lower latencies of RDMA to evict pages to remote memory instead of SSD but do not discuss the effect on distributed databases at all.

Distributed Shared Memory (DSM). Distributed shared memory exposes the memory of a collection of machines as a shared memory space. There are two flavors of DSM: (1) systems using coherence protocols [27, 109, 197] and (2) systems based on partitioning the global memory space (PGAS) [159]. From the first category, GAM is closely related to ScaleStore because it builds an RDMA-based cache coherence protocol. However, while the protocol may be related we employ multiple optimizations and outperform GAM in our evaluation. Additionally, unlike ScaleStore, GAM does not support transparent access to local or remote NVMe SSDs. In contrast to ScaleStore which uses sequential consistency, GAM uses a weaker consistency model – partial store order (PSO) – which in turn requires explicit placement of fences to achieve sequential consistency. Furthermore, ScaleStore scales better and its protocol enables anticipatory chaining. Pröbstl et al. [172] extended GAM to enable file I/O. The second system category (PGAS) does not abstract away remote data accesses, instead, remote accesses are explicit. However, this makes programming much more challenging compared to ScaleStore's abstraction. Additionally, PGAS systems often do not employ caching and thus cannot profit from locality.

Dynamic Re-partitioning & Live Reconfiguration. To dynamically re-partition data a number of papers [108, 130] focused on software transactions with ownership semantics. Zeus [108], for example, acquires all objects involved in a transaction and first moves them to the same server. This allows Zeus to avoid complex distributed transaction protocols and instead execute a single-node transaction on the re-partitioned objects. A key part of Zeus' contribution is an ownership protocol that bears similarities to our protocol, thus Zeus' underlying idea is related to ScaleStore. However, while ScaleStore handles NVMe SSDs and provides a general caching strategy, Zeus is in-memory only and does not discuss eviction strategies at all. Instead, they focus on transactions and their custom commit protocol. Because NVMe SSDs require different design decisions, such as fixed-size pages or asynchronous I/O, ScaleStore's protocol implementation is, despite some similarities, quite different. Live reconfiguration systems [59, 205, 232] only re-partition the shards online when (severe) workload imbalances are detected. This process is done to balance load at runtime while minimizing the impact on running transactions.

Cloud DBMSs. Cloud DBMSs such as Snowflake or Aurora [47, 218] often decouple the storage from the compute layer. To improve performance both systems deploy a caching solution. For instance, Snowflake uses consistent hashing to assign queries to the nodes which already cached the data. Queries are executed against a fixed set of immutable blocks and thus the cached content is not modified. Instead, a new block with the updated data is created and old blocks are evicted by an LRU-scheme. Crystal [58] improves caching in disaggregated architectures by reusing computations across multiple queries. We think of ScaleStore as a distributed storage engine that simplifies the development of disaggregated systems. ScaleStore can include NVMe SSDs on the storage layer and with our unified caching strategy speed up the compute layer.

Single-node Storage Engines. Now moving from distributed settings to modern single-node storage engines [2, 18, 118, 122, 124, 161, 182, 249]. Some systems leverage NVMe SSDs while others focus on non-volatile memory (NVM) to efficiently handle larger-than-memory workloads. NVM performance is quite different from NVMe SSDs and closer to DRAM performance, however, NVM is also more expensive [78] thus we focus on NVMe SSDs in ScaleStore. Single-node high performance storage engines with NVMe SSDs suffer from performance degradation if the hot set outgrows the memory due to the high latencies of NVMe SSDs.

13.8 Conclusion and Future Work

This paper introduced ScaleStore, a novel distributed storage engine for scalable DBMSs designed for DRAM, NVMe, and RDMA. As we have shown in our evaluation, ScaleStore can scale to large data sets efficiently even if the data set outgrows the aggregated main memory capacity of the full cluster. Furthermore, due to its carefully-designed distributed caching and eviction strategies, ScaleStore can not only efficiently adapt to workload changes but also support very different distributed DBMS architectures and requirements such as elasticity. In addition, we provide an easy-to-use programming abstraction for implementing distributed data structures managed by ScaleStore.

We think that ScaleStore can serve as the foundation for a distributed DBMS. In the following, we thus sketch ideas for future work. For example, a two-phase-locking concurrency control scheme can be implemented by physically storing locks within the tuples. This way, just like the data itself, locks are automatically distributed and managed through cache coherency. Another relevant observation is that because our protocol always moves the data to the processing node, we do not need a two-phase commit –

even though we are a distributed system. Instead of a two-phase commit, each node would have its own distributed ARIES-style WAL for recovery [225]. To avoid having to flush all distributed logs on commit, a technique called Remote-Flush Avoidance [81] can be used to commit non-overlapping transactions without any inter-node synchronization. Finally, a distributed setting also raises the question of high availability when nodes fail. We leave the modification of our cache protocol to support replicas for future work.

13.9 Acknowledgments

This work was partially funded by the German Research Foundation priority program 2037 (DFG) under the grants BI2011/1 & BI2011/2, the DFG Collaborative Research Center 1053 (MAKI), the BMBF and the state of Hesse as part of the NHR Program. We also thank hessian.AI for the support.

Bibliography

[1] Industry Perspectives | Nov 12. *Don't forget about Memory: DRAM's Surprising role in the high cost of data centers*. Nov. 2015. URL: https://www.datacenterknowledge.com/archives/2015/11/12/%20dont-forget-memory-drams-surprising-role-high-cost-data-%20centers.

[2] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. "Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia." In: *Proc. VLDB Endow.* 6.14 (2013), pp. 1714–1725. DOI: 10.14778/2556549.2556556. URL: http://www.vldb.org/pvldb/vol6/p1714-kossmann.pdf.

[3] Adnan Alhomssi and Viktor Leis. "Contention and Space Management in B-Trees." In: *11th Conference on Innovative Data Systems Research, CIDR 2021 , Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. URL: http://cidrdb.org/cidr2021/papers/cidr2021%5C%5C_paper21.pdf.

[4] Adnan Alhomssi and Viktor Leis. "Scalable and Robust Snapshot Isolation for High-Performance Storage Engines." In: *Proc. VLDB Endow.* 16.6 (2023), pp. 1426–1438. URL: https://www.vldb.org/pvldb/vol16/p1426-alhomssi.pdf.

[5] 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.

[6] 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.

[7] ARM. *Arm CoreLink CCI-550 Cache Coherent Interconnect Technical Reference Manual*. https://developer.arm.com/documentation/100282/0100/?lang=en. 2018. URL: https://developer.arm.com/documentation/100282/0100/?%20lang=en.

[8] ARM. *Introducing the AMBA Coherent Hub Interface*. 2021. URL: https://developer.arm.com/documentation/102407/0100.

[9] InfiniBand Trade Association. *InfiniBand Architecture Specification, Release 1.0, 2000*. http://www.infinibandta.org/specs. 2000.

[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. "Workload analysis of a large-scale key-value store." In: *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*. Ed. by Peter G. Harrison, Martin F. Arlitt, and Giuliano Casale. ACM, 2012, pp. 53–64. ISBN: 978-1-4503-1097-0. DOI: 10.1145/2254756.2254766. URL: https://doi.org/10.1145/2254756.2254766.

[11] *Scaling out with Azure SQL Database*.
url{ https://azure.microsoft.com/en-us/documentation/articles/sql-database-elastic-scale-introduction/ }.

[12] 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.

[13] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. "Tango: distributed data structures over a shared log." In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. Ed. by Michael Kaminsky and Mike Dahlin. ACM, 2013, pp. 325–340. DOI: 10.1145/2517349.2522732. URL: https://doi.org/10.1145/2517349.2522732.

[14] Claude Barthels, Gustavo Alonso, and Torsten Hoefler. "Designing Databases for Future High-Performance Networks." In: *IEEE Data Eng. Bull.* 40.1 (2017), pp. 15–26. URL: http://sites.computer.org/debull/A17mar/p15.pdf.

[15] 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.

[16] 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.

[17] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. "Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores." In: *Proc. VLDB Endow.* 12.13 (2019), pp. 2325–2338. DOI: 10.14778/3358701.3358702. URL: http://www.vldb.org/pvldb/vol12/p2325-barthels.pdf.

[18] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. "Viper: An Efficient Hybrid PMem-DRAM Key-Value Store." In: *Proc. VLDB Endow.* 14.9 (2021), pp. 1544–

1556. DOI: 10.14778/3461535.3461543. URL: http://www.vldb.org/pvldb/vol14/p1544-benson.pdf.

[19] Philip A. Bernstein and Nathan Goodman. "Concurrency Control in Distributed Database Systems." In: *ACM Comput. Surv.* 13.2 (1981), pp. 185–221. DOI: 10.1145/356842.356846. URL: https://doi.org/10.1145/356842.356846.

[20] 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.

[21] 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.

[22] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *CIDR*. 2005.

[23] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. "Scalable and robust latches for database systems." In: *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*. Ed. by Danica Porobic and Thomas Neumann. ACM, 2020, 2:1–2:8. DOI: 10.1145/3399666.3399908. URL: https://doi.org/10.1145/3399666.3399908.

[24] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Scalable Garbage Collection for In-Memory MVCC Systems." In: *Proc. VLDB Endow.* 13.2 (2019), pp. 128–141. DOI: 10.14778/3364324.3364328. URL: http://www.vldb.org/pvldb/vol13/p128-bottcher.pdf.

[25] 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*. ACM, 2021, pp. 228–242.

[26] Mengchu Cai, Martin Grund, Anurag Gupta, Fabian Nagel, Ippokratis Pandis, Yannis Papakonstantinou, and Michalis Petropoulos. "Integrated Querying of SQL database data and S3 data in Amazon Redshift." In: *IEEE Data Eng. Bull.* 41.2

(2018), pp. 82–90. URL: http://sites.computer.org/debull/A18june/p82.pdf.

[27] 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.

[28] David G. Campbell, Gopal Kakivaya, and Nigel Ellis. "Extreme scale with full SQL language support in microsoft SQL Azure." 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. 1021–1024. DOI: 10.1145/1807167.1807280. URL: https://doi.org/10.1145/1807167.1807280.

[29] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. "PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers." In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2021, pp. 2477–2489. DOI: 10.1145/3448016.3457560. URL: https://doi.org/10.1145/3448016.3457560.

[30] Benjamin Cassell, Tyler Szepesi, Bernard Wong, Tim Brecht, Jonathan Ma, and Xiaoyi Liu. "Nessie: A Decoupled, Client-Driven Key-Value Store Using RDMA." In: *IEEE Trans. Parallel Distributed Syst.* 28.12 (2017), pp. 3537–3552. DOI: 10.1109/TPDS.2017.2729545. URL: https://doi.org/10.1109/TPDS.2017.2729545.

[31] Sourav Chakraborty, Shulei Xu, Hari Subramoni, and Dhabaleswar K. Panda. "Designing Scalable and High-Performance MPI Libraries on Amazon Elastic Fabric Adapter." In: *2019 IEEE Symposium on High-Performance Interconnects, HOTI 2019, Santa Clara, CA, USA, August 14-16, 2019*. IEEE, 2019, pp. 40–44. DOI: 10.1109/HOTI.2019.00023. URL: https://doi.org/10.1109/HOTI.2019.00023.

[32] Sashikanth Chandrasekaran and Roger Bamford. "Shared Cache - The Future of Parallel Databases." In: *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*. Ed. by Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman. IEEE Computer Society, 2003, pp. 840–850. DOI: 10.1109/ICDE.2003.1260883. URL: https://doi.org/10.1109/ICDE.2003.1260883.

[33] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. "Fast In-Memory Transaction Processing Using RDMA and HTM." In: *ACM Trans. Comput. Syst.* 35.1 (2017), 3:1–3:37. DOI: 10.1145/3092701. URL: https://doi.org/10.1145/3092701.

[34] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. "Fast and general distributed transactions using RDMA and HTM." In: *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*. Ed. by Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues. ACM, 2016, 26:1–26:17. DOI: 10.1145/2901318.2901349. URL: https://doi.org/10.1145/2901318.2901349.

[35] Steffen Christgau and Bettina Schnor. "Software-managed Cache Coherence for fast One-Sided Communication." In: *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2016, Barcelona, Spain, March 12-16, 2016*. Ed. by Pavan Balaji and Kai-Cheung Leung. ACM, 2016, pp. 69–77. DOI: 10.1145/2883404.2883409. URL: https://doi.org/10.1145/2883404.2883409.

[36] Yeounoh Chung and Erfan Zamanian. "Using RDMA for Lock Management." In: *CoRR* abs/1507.03274 (2015). URL: http://arxiv.org/abs/1507.03274.

[37] GAM Code. 2018. URL: https://github.com/ooibc88/gam.

[38] LeanStore Code. 2022. URL: https://github.com/leanstore/leanstore.

[39] OLC B-Tree Code. 2018. URL: https://github.com/wangziqi2016/index-microbench/blob/%20master/BTreeOLC/BTreeOLC.h.

[40] ScaleStore Code. 2022. URL: https://github.com/DataManagementLab/ScaleStore.

[41] Zipf Generator Code. 2021. URL: https://github.com/opencog/cogutil.

[42] Douglas Comer. "The Ubiquitous B-Tree." In: *ACM Comput. Surv.* 11.2 (1979), pp. 121–137. DOI: 10.1145/356770.356776. URL: https://doi.org/10.1145/356770.356776.

[43] 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.

[44] NVIDIA Coporation. *NVIDIA InfiniBand Adaptive Routing Technology*. Whitepaper WP-10326-001_v01. 2021.

[45] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. "Spanner: Google's Globally Distributed Database." In: *ACM Trans. Comput. Syst.* (2013).

[46] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. "Schism: a Workload-Driven Approach to Database Replication and Partitioning." In: *Proc. VLDB Endow.* 3.1 (2010), pp. 48–57. DOI: 10.14778/1920841.1920853. URL: http://www.vldb.org/pvldb/vldb2010/pvldb%5C_vol3/R04.pdf.

[47] Benot 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.

[48] Andrei Marian Dan, Patrick Lam, Torsten Hoefler, and Martin T. Vechev. "Modeling and analysis of remote memory access programming." In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. Ed. by Eelco Visser and Yannis Smaragdakis. ACM, 2016, pp. 129–144. DOI: 10.1145/2983990.2984033. URL: https://doi.org/10.1145/2983990.2984033.

[49] Peter B. Danzig, Jong Suk Ahn, John Noll, and Katia Obraczka. "Distributed Indexing: A Scalable Mechanism for Distributed Information Retrieval." In: *Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Chicago, Illinois, USA, October 13-16, 1991 (Special Issue of the SIGIR Forum)*. Ed. by Abraham Bookstein, Yves Chiaramella, Gerard Salton, and Vijay V. Raghavan. ACM, 1991, pp. 220–229. DOI: 10.1145/122860.122883. URL: https://doi.org/10.1145/122860.122883.

[50] Niv Dayan, Yuval Rochman, Iddo Naiss, Shmuel Dashevsky, Noam Rabinovich, Edward Bortnikov, Igal Maly, Ofer Frishman, Itai Ben Zion, Avraham, Moshe Twitto, Uri Beitler, Evgeni Ginzburg, and Mark Mokryn. "The End of Moore's Law and the Rise of The Data Processor." In: *Proc. VLDB Endow.* 14.12 (2021), pp. 2932–2944. DOI: 10.14778/3476311.3476373. URL: http://www.vldb.org/pvldb/vol14/p2932-dayan.pdf.

[51] Ananth Devulapalli and Pete Wyckoff. "Distributed Queue-Based Locking Using Advanced Network Features." In: *34th International Conference on Parallel Processing (ICPP 2005), 14-17 June 2005, Oslo, Norway*. IEEE Computer Society, 2005, pp. 408–415. DOI: 10.1109/ICPP.2005.34. URL: https://doi.org/10.1109/ICPP.2005.34.

[52] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. "Hekaton: SQL server's memory-optimized OLTP engine." In: *SIGMOD*. 2013.

[53] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases." In: *Proc. VLDB Endow.* 7.4 (2013), pp. 277–288. DOI: 10.14778/2732240.2732246. URL: http://www.vldb.org/pvldb/vol7/p277-difallah.pdf.

[54] 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.

[55] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. "FaRM: Fast Remote Memory." In: *NSDI*. 2014.

[56] 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: *SOSP*. 2015.

[57] Jingwen Du, Fang Wang, Dan Feng, Changchen Gan, Yuchao Cao, Xiaomin Zou, and Fan Li. "Fast One-Sided RDMA-Based State Machine Replication for Disaggregated Memory." In: *ACM Trans. Archit. Code Optim.* (Mar. 2023). Just Accepted. ISSN: 1544-3566. DOI: 10.1145/3587096. URL: https://doi.org/10.1145/3587096.

[58] Dominik Durner, Badrish Chandramouli, and Yinan Li. "Crystal: A Unified Cache Storage System for Analytical Databases." In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2432–2444. DOI: 10.14778/3476249.3476292. URL: http://www.vldb.org/pvldb/vol14/p2432-durner.pdf.

[59] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. "Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases." 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. 299–313. DOI: 10.1145/2723372.2723726. URL: https://doi.org/10.1145/2723372.2723726.

[60] Steve Abraham Eric Boutin. *AWS re:Invent: Amazon Aurora Multi-Master: Scaling out database write performanc*. 2019. URL: https://www.youtube.com/watch?v=p0C0jakzYuc.

[61] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. "High Performance Transactions via Early Write Visibility." In: *Proc. VLDB Endow.* 10.5 (2017), pp. 613–624. DOI: 10.14778/3055540.3055553. URL: http://www.vldb.org/pvldb/vol10/p613-faleiro.pdf.

[62] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornh övd, 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.

[63] Chen Feng, Chundian Li, and Rui Li. "Indexing Techniques of Distributed Ordered Tables: A Survey and Analysis." In: *J. Comput. Sci. Technol.* 33.1 (2018), pp. 169–189. DOI: 10.1007/s11390-018-1813-8. URL: https://doi.org/10.1007/s11390-018-1813-8.

[64] 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: *ICDE*. 2020.

[65] Michael J. Freitag, Alfons Kemper, and Thomas Neumann. "Memory-Optimized Multi-Version Concurrency Control for Disk-Based Database Systems." In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2797–2810. URL: https://www.vldb.org/pvldb/vol15/p2797-freitag.pdf.

[66] 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.

[67] Gabriela Gligor, Silviu Teodoru, et al. "Oracle exalytics: engineered for speed-of-thought analytics." In: *Database Systems Journal* 2.4 (2011), pp. 3–8.

[68] Google. *AlloyDB*. 2022. URL: https://cloud.google.com/alloydb.

[69] Goetz Graefe. "Modern B-Tree Techniques." In: *Found. Trends Databases* 3.4 (2011), pp. 203–402. DOI: 10.1561/1900000028. URL: https://doi.org/10.1561/1900000028.

[70] Jim Gray. "Notes on Data Base Operating Systems." In: *Operating Systems, An Advanced Course*. Ed. by Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle. Vol. 60. Lecture Notes in Computer Science. Springer, 1978, pp. 393–481. ISBN: 3-540-08755-9. DOI: 10.1007/3-540-08755-9_9. URL: https://doi.org/10.1007/3-540-08755-9%5C_9.

[71] Jim Gray and Leslie Lamport. "Consensus on transaction commit." In: *ACM Trans. Database Syst.* 31.1 (2006), pp. 133–160. DOI: 10.1145/1132863.1132867. URL: https://doi.org/10.1145/1132863.1132867.

[72] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. ISBN: 1-55860-190-2.

[73] Matthias Gries, Ulrich Hoffmann, Michael Konow, and Michael Riepen. "SCC: A Flexible Architecture for Many-Core Platform Research." In: *Comput. Sci. Eng.* 13.6 (2011), pp. 79–83. DOI: 10.1109/MCSE.2011.109. URL: https://doi.org/10.1109/MCSE.2011.109.

[74] OpenFabrics Interfaces Working Group. *Libfabric*. 2022. URL: https://ofiwg.github.io/libfabric/.

[75] OpenFabrics Interfaces Working Group. *Libfabric Provider*. 2022. URL: https://ofiwg.github.io/libfabric/master/man/%20fi%5C%5C_provider.7.html.

[76] OpenFabrics Interfaces Working Group. *Perftest*. 2022. URL: https://github.com/linux-rdma/perftest.

[77] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. "RDMA over Commodity Ethernet at Scale." In: *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*. Ed. by Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti. ACM, 2016, pp. 202–215. ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934908. URL: https://doi.org/10.1145/2934872.2934908.

[78] Gabriel Haas, Michael Haubenschild, and Viktor Leis. "Exploiting Directly-Attached NVMe Arrays in DBMS." 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/p16-haas-cidr20.pdf.

[79] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. "An Evaluation of Distributed Concurrency Control." In: *Proc. VLDB Endow.* 10.5 (2017), pp. 553–564. DOI: 10.14778/3055540.3055548. URL: http://www.vldb.org/pvldb/vol10/p553-harding.pdf.

[80] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. "OLTP through the looking glass, and what we found there." In: *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. Ed. by Michael L. Brodie. ACM / Morgan & Claypool, 2019, pp. 409–439. DOI: 10.1145/3226595.3226635. URL: https://doi.org/10.1145/3226595.3226635.

[81] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. "Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines." 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. 877–892. DOI: 10.1145/3318464.3389716. URL: https://doi.org/10.1145/3318464.3389716.

[82] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E. Grant, and Ron Brightwell. "sPIN: high-performance streaming processing in the network." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*. Ed. by Bernd Mohr and Padma Raghavan. ACM, 2017, p. 59. ISBN:

978-1-4503-5114-0. DOI: 10.1145/3126908.3126970. URL: https://doi.org/10.1145/3126908.3126970.

[83] Torsten Hoefler, Ariel Hendel, and Duncan Roweth. "The Convergence of Hyperscale Data Center and High-Performance Computing Networks." In: *Computer* 55 (7 July 2022), pp. 29–37. DOI: 10.1109/mc.2022.3158437. URL: http://dx.doi.org/10.1109/mc.2022.3158437.

[84] Torsten Hoefler, Duncan Roweth, Keith 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*. 2023. arXiv: 2302.03337 [cs.NI].

[85] 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%5C%5C_adms19.pdf.

[86] Chenchen Huang, Huiqi Hu, Xuecheng Qi, Xuan Zhou, and Aoying Zhou. "RS-store: RDMA-enabled skiplist-based key-value store for efficient range query." In: *Frontiers Comput. Sci.* 15.6 (2021), p. 156617. DOI: 10.1007/s11704-020-0126-6. URL: https://doi.org/10.1007/s11704-020-0126-6.

[87] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. "X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing." 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. 651–665. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3314041. URL: https://doi.org/10.1145/3299869.3314041.

[88] Ram Huggahalli, Ravi R. Iyer, and Scott Tetrick. "Direct Cache Access for High Bandwidth Network I/O." In: *32st International Symposium on Computer Architecture (ISCA 2005), 4-8 June 2005, Madison, Wisconsin, USA*. IEEE Computer Society, 2005, pp. 50–59. DOI: 10.1109/ISCA.2005.23. URL: https://doi.org/10.1109/ISCA.2005.23.

[89] IBM. *Moving from a TCP/IP protocol network to an RDMA protocol network*. URL: https://www.ibm.com/docs/en/db2/11.1?topic=tfsai-moving-%20from-tcpip-protocol-network-rdma-protocol-network.

[90] Yugabyte Inc. *YugabyteDB*. 2022. URL: https://www.yugabyte.com/.

[91] *InfiniBand Architecture Specification Volume 1*. Release 1.2.1. InfiniBand Trade Association. Nov. 2007.

[92] InfiniBand Trade Association. *RDMA Over Converged Ethernet (RoCE)*. https://cw.infinibandta.org/document/dl/7148. 2010.

[93] Intel. *Intel Data Direct I/O Technology (Intel DDIO): A P rimer*. Feb. 2012. URL: https://www.intel.com/content/dam/www/public/us/en/%20documents/technology-briefs/data-direct-i-o-technology-%20brief.pdf.

[94] Intel Corporation. *Intelő Data Direct I/O Technology: Technology Brief*. URL: https://www.intel.de/content/www/de/de/io/data-direct-i-o-%20technology-brief.html.

[95] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees A. Vissers. "A flexible hash table design for 10GBPS key-value stores on FPGAS." In: *23rd International Conference on Field programmable Logic and Applications, FPL 2013, Porto, Portugal, September 2-4, 2013*. IEEE, 2013, pp. 1–8. DOI: 10.1109/FPL.2013.6645520. URL: https://doi.org/10.1109/FPL.2013.6645520.

[96] 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 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.

[97] Eunyoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems." 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. 489–502. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[98] 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.

[99] Ashok M. Joshi. "Adaptive Locking Strategies in a Multi-node Data Sharing Environment." In: *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*. Ed. by Guy M. Lohman, Amílcar Sernadas, and Rafael Camps. Morgan Kaufmann, 1991, pp. 181–191. URL: http://www.vldb.org/conf/1991/P181.PDF.

[100] Jeffrey W. Josten, C. Mohan, Inderpal Narang, and James Z. Teng. "DB2's Use of the Coupling Facility for Data Sharing." In: *IBM Syst. J.* (1997).

[101] 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.

[102] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Design Guidelines for High Performance RDMA Systems." In: *login Usenix Mag.* 41.3 (2016).

[103] 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.

[104] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Using RDMA efficiently for key-value services." In: *SIGCOMM*. 2014.

[105] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. "H-store: a high-performance, distributed main memory transaction processing system." In: *Proc. VLDB Endow.* 1.2 (2008), pp. 1496–1499. DOI: 10.14778/1454159.1454211. URL: http://www.vldb.org/pvldb/vol1/1454211.pdf.

[106] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web." In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of*

Computing, El Paso, Texas, USA, May 4-6, 1997. Ed. by Frank Thomson Leighton and Peter W. Shor. ACM, 1997, pp. 654–663. DOI: 10.1145/258533.258660. URL: https://doi.org/10.1145/258533.258660.

[107] Tejas Karmarkar. *Availability of Linux RDMA on Microsoft Azure*. Online. July 2015. URL: https://azure.microsoft.com/en-us/blog/azure-linux-rdma-hpc-%20available/.

[108] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. "Zeus: locality-aware distributed transactions." In: *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. Ed. by Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar. ACM, 2021, pp. 145–161. DOI: 10.1145/3447786.3456234. URL: https://doi.org/10.1145/3447786.3456234.

[109] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. "Turning Centralized Coherence and Distributed Critical-Section Execution on their Head: A New Approach for Scalable Distributed Shared Memory." 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. 3–14. DOI: 10.1145/2749246.2749250. URL: https://doi.org/10.1145/2749246.2749250.

[110] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots." In: *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*. Ed. by Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan. IEEE Computer Society, 2011, pp. 195–206. DOI: 10.1109/ICDE.2011.5767867. URL: https://doi.org/10.1109/ICDE.2011.5767867.

[111] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. "Farview: Disaggregated Memory with Operator Off-loading for Database Engines." In: *12th Conference on Innovative Data Systems Research, CIDR 2022 , Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL: https://www.cidrdb.org/cidr2022/papers/p11-korolija.pdf.

[112] Donald Kossmann. "The State of the art in distributed query processing." In: *ACM Comput. Surv.* 32.4 (2000), pp. 422–469. DOI: 10.1145/371578.371598. URL: https://doi.org/10.1145/371578.371598.

[113] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. "Fast Updates on Read-Optimized Databases Using Multi-Core CPUs." In: *Proc. VLDB Endow.* 5.1 (2011), pp. 61–72. DOI: 10.14778/2047485.2047491. URL: http://www.vldb.org/pvldb/vol5/p061%5C%5C_jenskrueger%5C%5C_vldb2012.pdf.

[114] Chinmay Kulkarni, Aniraj Kesavan, Robert Ricci, and Ryan Stutsman. "Beyond Simple Request Processing with RAMCloud." In: *IEEE Data Eng. Bull.* 40.1 (2017), pp. 62–69. URL: http://sites.computer.org/debull/A17mar/p62.pdf.

[115] Cockroach Labs. *CockroachDB*. 2022. URL: https://www.cockroachlabs.com/product/.

[116] Avraham Leff, Joel L. Wolf, and Philip S. Yu. "Replication Algorithms in a Remote Caching Architecture." In: *IEEE Trans. Parallel Distributed Syst.* 4.11 (1993), pp. 1185–1204. DOI: 10.1109/71.250099. URL: https://doi.org/10.1109/71.250099.

[117] Philip L. Lehman and S. Bing Yao. "Efficient Locking for Concurrent Operations on B-Trees." In: *ACM Trans. Database Syst.* 6.4 (1981), pp. 650–670. DOI: 10.1145/319628.319663. URL: https://doi.org/10.1145/319628.319663.

[118] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. "LeanStore: In-Memory Data Management beyond Main Memory." In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018, pp. 185–196. DOI: 10.1109/ICDE.2018.00026. URL: https://doi.org/10.1109/ICDE.2018.00026.

[119] Viktor Leis, Michael Haubenschild, and Thomas Neumann. "Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method." In: *IEEE Data Eng. Bull.* 42 (2019), pp. 73–84.

[120] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. "The ART of practical synchronization." In: *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*. ACM, 2016, 3:1–3:8. DOI: 10.1145/2933349.2933352. URL: https://doi.org/10.1145/2933349.2933352.

[121] Edgar A. León, Kurt B. Ferreira, and Arthur B. Maccabe. "Reducing the Impact of the MemoryWall for I/O Using Cache Injection." In: *15th Annual IEEE Symposium on High-Performance Interconnects, HOTI 2007, Stanford, CA, USA, August 22-24, 2007*. Ed. by John W. Lockwood, Fabrizio Petrini, Ron Brightwell, and Dhabaleswar K. Panda. IEEE Computer Society, 2007, pp. 143–150. DOI: 10.1109/HOTI.2007.8. URL: https://doi.org/10.1109/HOTI.2007.8.

[122] Lucas Lersch, Wolfgang Lehner, and Ismail Oukid. "Persistent Buffer Management with Optimistic Consistency." In: *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*. Ed. by Thomas Neumann and Ken Salem. ACM, 2019, 14:1–14:3. DOI: 10.1145/3329785.3329931. URL: https://doi.org/10.1145/3329785.3329931.

[123] Justin Levandoski. *HPTS 2019: Aurora Multi-Master*. 2019. URL: http://www.hpts.ws/papers/2019/aurora-multimaster-hpts2019.pdf.

[124] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. "Identifying hot and cold data in main-memory databases." 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. 26–37. DOI: 10.1109/ICDE.2013.6544811. URL: https://doi.org/10.1109/ICDE.2013.6544811.

[125] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. "High Performance Transactions in Deuteronomy." In: *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015. URL: http://cidrdb.org/cidr2015/Papers/CIDR15%5C%5C_Paper15.pdf.

[126] 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.

[127] Feifei Li. "Cloud native database systems at Alibaba: Opportunities and Challenges." In: *Proc. VLDB Endow.* 12.12 (2019), pp. 2263–2272. DOI: 10.14778/3352063.3352141. URL: http://www.vldb.org/pvldb/vol12/p2263-li.pdf.

[128] 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.

[129] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage." 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. 429–444. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

[130] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. "Towards a Non-2PC Transaction Management in Distributed Database Systems." 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. 1659–1674. DOI: 10.1145/2882903.2882923. URL: https://doi.org/10.1145/2882903.2882923.

[131] 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.

[132] 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.

[133] 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.

[134] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. "On the Design and Scalability of Distributed Shared-Data Databases." In: *SIGMOD*. 2015.

[135] David Lomet, Rick Anderson, TK Rengarajan, and Peter Spiro. "How the Rdb/VMS data sharing system became fast." In: *DEC Cambridge Research Lab Technical Report CRL* (1992).

[136] David B. Lomet. "Replicated Indexes for Distributed Data." In: *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*. IEEE Computer Society, 1996, pp. 108–119. DOI: 10.1109/PDIS.1996.568673. URL: https://doi.org/10.1109/PDIS.1996.568673.

[137] 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.

[138] Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K. Panda. "Scalable and Distributed Key-Value Store-based Data Management Using RDMA-Memcached." In: *IEEE Data Eng. Bull.* 40.1 (2017), pp. 50–61. URL: http://sites.computer.org/debull/A17mar/p50.pdf.

[139] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. "Aria: A Fast and Practical Deterministic OLTP Database." In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2047–2060. URL: http://www.vldb.org/pvldb/vol13/p2047-lu.pdf.

[140] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. "Octopus: an RDMA-enabled Distributed Persistent Memory File System." In: *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. Ed. by Dilma Da Silva and Bryan Ford. USENIX Association, 2017, pp. 773–785. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu.

[141] Teng Ma, Kang Chen, Shaonan Ma, Zhuo Song, and Yongwei Wu. "Thinking More about RDMA Memory Semantics." In: *IEEE International Conference on Cluster Computing, CLUSTER 2021, Portland, OR, USA, September 7-10, 2021*. IEEE, 2021, pp. 456–467. DOI: 10.1109/Cluster48925.2021.00033. URL: https://doi.org/10.1109/Cluster48925.2021.00033.

[142] Teng Ma, Dongbiao He, and Gordon Ning Liu. "HybridSkipList: A Case Study of Designing Distributed Data Structure with Hybrid RDMA." In: *IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC 2021, Madrid, Spain, July 12-16, 2021*. IEEE, 2021, pp. 68–73. DOI: 10.1109/COMPSAC51774.2021.00021. URL: https://doi.org/10.1109/COMPSAC51774.2021.00021.

[143] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. "MaaT: Effective and scalable coordination of distributed transactions in the cloud." In: *Proc. VLDB Endow.* 7.5 (2014), pp. 329–340. DOI: 10.14778/2732269.2732270. URL: http://www.vldb.org/pvldb/vol7/p329-mahmoud.pdf.

[144] Mellanox. *Sockperf*. 2022. URL: https://github.com/Mellanox/sockperf.

[145] Microsoft. *Azure Comsmos DB*. 2022. URL: https://azure.microsoft.com/en-us/products/cosmos-db/.

[146] Microsoft. *HC-series virtual machine sizes*. 2021. URL: https://docs.microsoft.com/en-us/azure/virtual-machines/%20workloads/hpc/hc-series-performance.

[147] Microsoft. *High performance computing VM sizes*. 2022. URL: https://docs.microsoft.com/en-us/azure/virtual-machines/%20sizes-hpc.

[148] Umar Farooq Minhas, David B. Lomet, and Chandramohan A. Thekkath. "Chimera: data sharing flexibility, shared nothing simplicity." In: *15th International Database Engineering and Applications Symposium (IDEAS 2011), September 21 - 27, 2011, Lisbon, Portugal*. Ed. by Bipin C. Desai, Isabel F. Cruz, and Jorge Bernardino. ACM, 2011, pp. 152–161. DOI: 10.1145/2076623.2076642. URL: https://doi.org/10.1145/2076623.2076642.

[149] 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/%20presentation/mitchell.

[150] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. "Balancing CPU and Network in the Cell Distributed B-Tree Store." In: *USENIX ATC*. 2016.

[151] C. Mohan. *Modern Cloud DBMSs Vindicate Age Old Work on Shared Disk DBMSs Keynote at SMBDB*. 2022. URL: https://db.cs.pitt.edu/smdb2022.

[152] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging." In: *ACM Trans. Database Syst.* 17.1 (1992), pp. 94–162. DOI: 10.1145/128765.128770. URL: https://doi.org/10.1145/128765.128770.

[153] C. Mohan, Bruce G. Lindsay, and Ron Obermarck. "Transaction Management in the R* Distributed Database Management System." In: *ACM Trans. Database Syst.* 11.4 (1986), pp. 378–396. DOI: 10.1145/7239.7266. URL: https://doi.org/10.1145/7239.7266.

[154] C. Mohan and Inderpal Narang. "Efficient Locking and Caching of Data in the Multisystem Shared Disks Transaction Environment." In: *Advances in Database Technology - EDBT'92, 3rd International Conference on Extending Database Technology, Vienna, Austria, March 23-27, 1992, Proceedings*. Ed. by Alain Pirotte, Claude Delobel, and Georg Gottlob. Vol. 580. Lecture Notes in Computer Science. Springer, 1992, pp. 453–468. DOI: 10.1007/BFb0032448. URL: https://doi.org/10.1007/BFb0032448.

[155] C. Mohan and Inderpal Narang. "Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment." In: *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*. Ed. by Guy M. Lohman, Amílcar Sernadas, and Rafael Camps. Morgan Kaufmann, 1991, pp. 193–207. URL: http://www.vldb.org/conf/1991/P193.PDF.

[156] Jayanta Mondal and Amol Deshpande. "Managing large dynamic graphs efficiently." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24 , 2012*. Ed. by K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman. ACM, 2012, pp. 145–156. DOI: 10.1145/2213836.2213854. URL: https://doi.org/10.1145/2213836.2213854.

[157] Gordon E. Moore. "Cramming More Components Onto Integrated Circuits." In: *Proc. IEEE* 86.1 (1998), pp. 82–85. DOI: 10.1109/jproc.1998.658762. URL: https://doi.org/10.1109/jproc.1998.658762.

[158] Sundeep Narravula, A. Marnidala, Abhinav Vishnu, Karthikeyan Vaidyanathan, and Dhabaleswar K. Panda. "High Performance Distributed Lock Management Services using Network-based Remote Atomic Operations." In: *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), 14-17 May 2007, Rio de Janeiro, Brazil*. IEEE Computer Society, 2007, pp. 583–590. DOI: 10.1109/CCGRID.2007.58. URL: https://doi.org/10.1109/CCGRID.2007.58.

[159] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. "Latency-Tolerant Software Distributed Shared Memory." In: *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*. Ed. by Shan Lu and Erik Riedel. USENIX Association, 2015, pp. 291–305. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/nelson.

[160] Jacob Nelson and Roberto Palmieri. "Performance Evaluation of the Impact of NUMA on One-sided RDMA Interactions." In: *SRDS*. 2020.

[161] Thomas Neumann and Michael J. Freitag. "Umbra: A Disk-Based System with In-Memory Performance." 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/p29-neumann-cidr20.pdf.

[162] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. "Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems." 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. 677–689. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2749436. URL: https://doi.org/10.1145/2723372.2749436.

[163] NVIDIA. *Mellanox InfiniBand Helps Accelerate Teradata Aster Big Analytics Appliance*. 2012. URL: https://www.mellanox.com/news/press%5C%5C_release/mellanox-%20infiniband-helps-accelerate-teradata-aster-big-analytics-%20appliance.

[164] Oracle. *Oracle RAC*. 2022. URL: https://www.oracle.com/de/database/real-application-clusters/.

[165] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. "The case for RAMCloud." In: *Commun. ACM* 54.7 (2011), pp. 121–130. DOI: 10.1145/1965724.1965751. URL: https://doi.org/10.1145/1965724.1965751.

[166] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. "The case for RAMClouds: scalable high-performance storage entirely in DRAM." In: *ACM SIGOPS Oper. Syst. Rev.* 43.4 (2009), pp. 92–105. DOI: 10.1145/1713254.1713276. URL: https://doi.org/10.1145/1713254.1713276.

[167] Xiangyong Ouyang, Sonya Marcarelli, Raghunath Rajachandrasekar, and Dhabaleswar K. Panda. "RDMA-Based Job Migration Framework for MPI over InfiniBand." In: *Proceedings of the 2010 IEEE International Conference on Cluster Computing, Heraklion, Crete, Greece, 20-24 September, 2010*. IEEE Computer Society, 2010, pp. 116–125. DOI: 10.1109/CLUSTER.2010.20. URL: https://doi.org/10.1109/CLUSTER.2010.20.

[168] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1991. ISBN: 0-13-715681-2.

[169] David A. Patterson. "Latency lags bandwith." In: *Commun. ACM* 47.10 (2004), pp. 71–75. DOI: 10.1145/1022594.1022596. URL: https://doi.org/10.1145/1022594.1022596.

[170] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24 , 2012*. Ed. by K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman. ACM, 2012, pp. 61–72. DOI: 10.1145/2213836.2213844. URL: https://doi.org/10.1145/2213836.2213844.

[171] PCI-SIG. "PCI Express Base Specification Revision 4.0." In: (2014).

[172] Magdalena Pröbstl, Philipp Fent, Maximilian E. Schüle, Moritz Sichert, Thomas Neumann, and Alfons Kemper. "One Buffer Manager to Rule Them All: Using Distributed Memory with Cache Coherence over RDMA." In: *International Workshop*

on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021, Copenhagen, Denmark, August 16, 2021. Ed. by Rajesh Bordawekar and Tirthankar Lahiri. 2021, pp. 17–26. URL: http://www.adms-conf.org/2021-camera-ready/probstl%5C%5C_adms21.pdf.

[173] DPDK Project. *DPDK*. 2022. URL: https://www.dpdk.org/.

[174] Francois Raab. "TPC-C - The Standard Benchmark for Online transaction Processing (OLTP)." In: *The Benchmark Handbook*. Ed. by Jim Gray. Morgan Kaufmann, 1993. ISBN: 1-55860-292-5. URL: http://dblp.uni-trier.de/db/books/collections/gray93.html#Raab93.

[175] Oracle. *Delivering Application Performance with Oracle's InfiniBand Technology*. 2012.

[176] Erhard Rahm. "Primary copy synchronization for DB-Sharing." In: *Inf. Syst.* 11.4 (1986), pp. 275–286. DOI: 10.1016/0306-4379(86)90008-6. URL: https://doi.org/10.1016/0306-4379(86)90008-6.

[177] Erhard Rahm. "Recovery Concepts for Data Sharing Systems." In: *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing, Montreal, Canada*. IEEE Computer Society, 1991, pp. 368–377. DOI: 10.1109/FTCS.1991.146687. URL: https://doi.org/10.1109/FTCS.1991.146687.

[178] Erhard Rahm and Robert Marek. "Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems." In: *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*. Ed. by Rakesh Agrawal, Seán Baker, and David A. Bell. Morgan Kaufmann, 1993, pp. 182–193. URL: http://www.vldb.org/conf/1993/P182.PDF.

[179] Mohammad J. Rashti and Ahmad Afsahi. "Improving Communication Progress and Overlap in MPI Rendezvous Protocol over RDMA-enabled Interconnects." In: *22nd Annual International Symposium on High Performance Computing Systems and Applications (HPCS 2008), June 9-11, 2008 , Québec City, Canada*. IEEE Computer Society, 2008, pp. 95–101. DOI: 10.1109/HPCS.2008.10. URL: https://doi.org/10.1109/HPCS.2008.10.

[180] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. *A Remote Direct Memory Access Protocol Specification*. Tech. rep. Oct. 2007. DOI: 10.17487/rfc5040.

[181] Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, and Thomas G. Robertazzi. "Design and performance evaluation of NUMA-aware RDMA-based end-to-end data transfer systems." In: *HiPC*. 2013.

[182] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. "Managing Non-Volatile Memory in Database Systems." 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. 1541–1555. DOI: 10.1145/3183713.3196897. URL: https://doi.org/10.1145/3183713.3196897.

[183] 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.

[184] 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.

[185] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. "AIFM: High-Performance, Application-Integrated Far Memory." In: *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020, pp. 315–332. URL: https://www.usenix.org/conference/osdi20/presentation/ruan.

[186] Abdallah Salama, Carsten Binnig, Tim Kraska, Ansgar Scherp, and Tobias Ziegler. "Rethinking Distributed Query Execution on High-Speed Networks." In: *IEEE Data Eng. Bull.* 40.1 (2017), pp. 27–37. URL: http://sites.computer.org/debull/A17mar/p27.pdf.

[187] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow." In: *CoRR* abs/1802.05799 (2018). arXiv: 1802.05799. URL: http://arxiv.org/abs/1802.05799.

[188] Amazon Web Services. *Amazon DynamoDB*. 2022. URL: https://aws.amazon.com/dynamodb/.

[189] Amazon Web Services. *Aurora Multi-Master Documentation*. 2022. URL: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-multi-master.html.

[190] Amazon Web Services. *AWS Nitro System*. 2021. URL: https://aws.amazon.com/ec2/nitro/.

[191] Amazon Web Services. *EFA is now mainstream, and thats a Good Thing*. 2021. URL: https://aws.amazon.com/blogs/hpc/efa-is-now-mainstream/.

[192] Amazon Web Services. *Elastic Fabric Adapter is officially integrated into Libfabric Library*. 2019. URL: https://aws.amazon.com/about-aws/whats-new/2019/07/elastic-%20fabric-adapter-officially-integrated-into-libfabric-library/.

[193] Amazon Web Services. *Placement Groups*. 2022. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/%20placement-groups.html.

[194] H. Shah, F. Marti, W. Noureddine, A. Eiriksson, and R. Sharp. *Remote Direct Memory Access (RDMA) Protocol Extensions*. Tech. rep. June 2014. DOI: 10.17487/rfc7306.

[195] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. "A Cloud-Optimized Transport Protocol for Elastic and Scalable HPC." In: *IEEE Micro* 40.6 (2020), pp. 67–73. DOI: 10.1109/MM.2020.3016891. URL: https://doi.org/10.1109/MM.2020.3016891.

[196] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. "Fast General Distributed Transactions with Opacity." 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. 433–448. DOI: 10.1145/3299869.3300069. URL: https://doi.org/10.1145/3299869.3300069.

[197] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. "Distributed shared persistent memory." In: *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 2017, pp. 323–337. DOI: 10.1145/3127479.3128610. URL: https://doi.org/10.1145/3127479.3128610.

[198] Debendra Das Sharma. *Compute Express Link*. Tech. rep. Compute Express Link, 2019.

[199] 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.

[200] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. "CliqueMap: productionizing an RMA-based distributed caching system." In: *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*. Ed. by Fernando A. Kuipers and Matthew C. Caesar. ACM, 2021, pp. 93–105. DOI: 10.1145/3452296.3472934. URL: https://doi.org/10.1145/3452296.3472934.

[201] Markus Sinnwell and Gerhard Weikum. "A Cost-Model-Based Online Method for Ditributed Caching." In: *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*. Ed. by W. A. Gray and Per-Åke Larson. IEEE Computer Society, 1997, pp. 532–541. DOI: 10.1109/ICDE.1997.582022. URL: https://doi.org/10.1109/ICDE.1997.582022.

[202] Michael Stonebraker. "The Case for Shared Nothing." In: *IEEE Database Eng. Bull.* 9.1 (1986), pp. 4–9. URL: http://sites.computer.org/debull/86MAR-CD.pdf.

[203] Sayantan Sur, Lei Chai, Hyun-Wook Jin, and Dhabaleswar K. Panda. "Shared receive queue based scalable MPI design for InfiniBand clusters." In: *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006. DOI: 10.1109/IPDPS.2006.1639336. URL: https://doi.org/10.1109/IPDPS.2006.1639336.

[204] Tyler Szepesi, Bernard Wong, Ben Cassell, and Tim Brecht. "Designing a low-latency cuckoo hash table for write-intensive workloads using RDMA." In: *First International Workshop on Rack-scale Computing*. 2014.

[205] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. "E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing." In: *Proc.*

VLDB Endow. 8.3 (2014), pp. 245–256. DOI: 10.14778/2735508.2735514. URL: http://www.vldb.org/pvldb/vol8/p245-taft.pdf.

[206] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. "CockroachDB: The Resilient Geo-Distributed SQL Database." 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. 1493–1509. DOI: 10.1145/3318464.3386134. URL: https://doi.org/10.1145/3318464.3386134.

[207] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. "Tailwind: Fast and Atomic RDMA-based Replication." In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. Ed. by Haryadi S. Gunawi and Benjamin C. Reed. USENIX Association, 2018, pp. 851–863. URL: https://www.usenix.org/conference/atc18/presentation/taleb.

[208] Dan Tang, Yungang Bao, Weiwu Hu, and Mingyu Chen. "DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance." In: *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010), 9-14 January 2010, Bangalore, India*. Ed. by Matthew T. Jacob, Chita R. Das, and Pradip Bose. IEEE Computer Society, 2010, pp. 1–12. DOI: 10.1109/HPCA.2010.5416638. URL: https://doi.org/10.1109/HPCA.2010.5416638.

[209] Konstantin Taranov, Fabian Fischer, and Torsten Hoefler. *Efficient RDMA Communication Protocols*. 2022.

[210] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. "CoRM: Compactable Remote Memory over RDMA." In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. 2021, pp. 1811–1824. DOI: 10.1145/3448016.3452817. URL: https://doi.org/10.1145/3448016.3452817.

[211] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. "Calvin: fast distributed transactions for partitioned database systems." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24 , 2012.*

Ed. by K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman. ACM, 2012, pp. 1–12. DOI: 10.1145/2213836.2213838. URL: https://doi.org/10.1145/2213836.2213838.

[212] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. "Distributed GPU Joins on Fast RDMA-capable Networks." In: *Proceedings of the ACM on Management of Data* 1.1 (2023), pp. 1–26.

[213] 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), pp. 15–22. DOI: 10.1145/3542700.3542705. URL: https://doi.org/10.1145/3542700.3542705.

[214] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. "Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores." In: *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. Ed. by Ada Gavrilovska and Erez Zadok. USENIX Association, 2020, pp. 33–48. URL: https://www.usenix.org/conference/atc20/presentation/tsai.

[215] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. "Speedy transactions in multicore in-memory databases." In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. Ed. by Michael Kaminsky and Mike Dahlin. ACM, 2013, pp. 18–32. DOI: 10.1145/2517349.2522713. URL: https://doi.org/10.1145/2517349.2522713.

[216] TU Darmstadt Data Management Lab. *RDMA Communication Patterns Code*. URL: https://github.com/DataManagementLab/%20RDMA_Communication_Patterns.

[217] 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.

[218] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. "Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes." In: *SIGMOD*. 2018.

[219] Jérôme Vienne, Jitong Chen, Md. Wasi-ur-Rahman, Nusrat S. Islam, Hari Subramoni, and Dhabaleswar K. Panda. "Performance Analysis and Evaluation of InfiniBand FDR and 40GigE RoCE on HPC and Cloud Computing Systems." In: *IEEE 20th Annual Symposium on High-Performance Interconnects, HOTI 2012, Santa Clara, CA, USA, August 22-24, 2012*. IEEE Computer Society, 2012, pp. 48–55. DOI: 10.1109/HOTI.2012.19. URL: https://doi.org/10.1109/HOTI.2012.19.

[220] Lukas Vogel, Daniel Ritter, Danica Porobic, Pnar Töz ün, Tianzheng Wang, and Alberto Lerner. "Data Pipes: Declarative Control over Data Movement." In: *13th Conference on Innovative Data Systems Research, CIDR 2023*. www.cidrdb.org, 2023. URL: https://www.cidrdb.org/cidr2023/papers/p55-vogel.pdf.

[221] Chao Wang and Xuehai Qian. "RDMA-enabled Concurrency Control Protocols for Transactions in the Cloud Era." In: *IEEE Transactions on Cloud Computing* (2021), pp. 1–1. DOI: 10.1109/tcc.2021.3116516.

[222] 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.

[223] 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 G. 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.

[224] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. "The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation." In: *CoRR* abs/2207.03027 (2022). DOI: 10.48550/arXiv.2207.03027. URL: https://doi.org/10.48550/arXiv.2207.03027.

[225] Tianzheng Wang and Ryan Johnson. "Scalable Logging through Emerging Non-Volatile Memory." In: *Proc. VLDB Endow.* 7.10 (2014), pp. 865–876. DOI: 10.14778/2732951.2732960. URL: http://www.vldb.org/pvldb/vol7/p865-wang.pdf.

[226] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. "Query Fresh: Log Shipping on Steroids." In: *Proc. VLDB Endow.* 11.4 (2017), pp. 406–419. DOI: 10.1145/3186728.3164137. URL: http://www.vldb.org/pvldb/vol11/p406-wang.pdf.

[227] Tinggang Wang, Shuo Yang, Hideaki Kimura, Garret Swart, and Spyros Blanas. "Efficient Usage of One-Sided RDMA for Linear Probing." In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2020, Tokyo, Japan, August 31, 2020*. Ed. by Rajesh Bordawekar and Tirthankar Lahiri. 2020, pp. 1–13. URL: http://www.adms-conf.org/2020-camera-ready/ADMS20%5C_06.pdf.

[228] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. "HydraDB: a resilient RDMA-driven key-value middleware for in-memory cluster computing." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*. Ed. by Jackie Kern and Jeffrey S. Vetter. ACM, 2015, 22:1–22:11. DOI: 10.1145/2807591.2807614. URL: https://doi.org/10.1145/2807591.2807614.

[229] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. "Building a Bw-Tree Takes More Than Just Buzz Words." 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. 473–488. DOI: 10.1145/3183713.3196895. URL: https://doi.org/10.1145/3183713.3196895.

[230] Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. "XStore: Fast RDMA-Based Ordered Key-Value Store Using Remote Learned Cache." In: *ACM Trans. Storage* 17.3 (2021), 18:1–18:32. DOI: 10.1145/3468520. URL: https://doi.org/10.1145/3468520.

[231] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. "Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!" In: *OSDI*. 2018.

[232] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. "Replication-driven Live Reconfiguration for Fast Distributed Transaction Processing." In: *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. Ed. by Dilma Da Silva and Bryan Ford. USENIX Association, 2017, pp. 335–347. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/wei.

[233] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. "Fast in-memory transaction processing using RDMA and HTM." In: *SOSP*. 2015.

[234] Shulei Xu, Seyedeh Mahdieh Ghazimirsaeed, Jahanzeb Maqbool Hashmi, Hari Subramoni, and Dhabaleswar K. Panda. "MPI Meets Cloud: Case Study with Amazon EC2 and Microsoft Azure." In: *Fourth IEEE/ACM Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware, IPDRM@SC 2020, Atlanta, GA, USA, November 13, 2020*. IEEE, 2020, pp. 41–48. DOI: 10.1109/IPDRM51949.2020.00010. URL: https://doi.org/10.1109/IPDRM51949.2020.00010.

[235] 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.

[236] 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.

[237] Xiangyao Yu, Hongzhe Liu, Ethan Zou, and Srinivas Devadas. "Tardis 2.0: Optimized Time Traveling Coherence for Relaxed Consistency Models." In: *PACT*. 2016.

[238] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. "Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System." In: *Proc. VLDB Endow.*

11.10 (2018), pp. 1289–1302. DOI: 10.14778/3231751.3231763. URL: http://www.vldb.org/pvldb/vol11/p1289-yu.pdf.

[239] 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.

[240] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. "Locality-aware Partitioning in Parallel Database Systems." 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. 17–30. DOI: 10.1145/2723372.2723718. URL: https://doi.org/10.1145/2723372.2723718.

[241] 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).

[242] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. "Rethinking Database High Availability with RDMA Networks." In: *Proc. VLDB Endow.* 12.11 (2019), pp. 1637–1650. DOI: 10.14778/3342263.3342639. URL: http://www.vldb.org/pvldb/vol12/p1637-zamanian.pdf.

[243] Steffen Zeuch, Sebastian BreSS, Tilmann Rabl, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, and Volker Markl. "Analyzing Efficient Stream Processing on Modern Hardware." In: *Proc. VLDB Endow.* 12.5 (2019), pp. 516–530. DOI: 10.14778/3303753.3303758. URL: http://www.vldb.org/pvldb/vol12/p516-zeuch.pdf.

[244] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. "Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes." 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. 1567–1581. DOI: 10.1145/2882903.2915222. URL: https://doi.org/10.1145/2882903.2915222.

[245] Jin Zhang, Xiangyao Yu, Zhengwei Qi, and Haibing Guan. "Falcon: A Timestamp-based Protocol to Maximize the Cache Efficiency in the Distributed Shared Memory." In: *2022 IEEE International Parallel and Distributed Processing Sym-*

posium, IPDPS 2022, Lyon, France, May 30 - June 3, 2022. IEEE, 2022, pp. 974–984. DOI: 10.1109/IPDPS53621.2022.00099. URL: https://doi.org/10.1109/IPDPS53621.2022.00099.

[246] 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.

[247] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. "Redy: Remote Dynamic Memory Cache." In: *Proc. VLDB Endow.* 15.4 (2021), pp. 766–779. DOI: 10.14778/3503585.3503587. URL: https://www.vldb.org/pvldb/vol15/p766-zhang.pdf.

[248] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. "Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation." In: *Proc. VLDB Endow.* 14.10 (2021), pp. 1900–1912. DOI: 10.14778/3467861.3467877. URL: http://www.vldb.org/pvldb/vol14/p1900-zhang.pdf.

[249] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. "Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory." 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. 2195–2207. DOI: 10.1145/3448016.3452819. URL: https://doi.org/10.1145/3448016.3452819.

[250] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. "Congestion Control for Large-Scale RDMA Deployments." 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. 523–536. ISBN: 978-1-4503-3542-3. DOI: 10.1145/2785956.2787484. URL: https://doi.org/10.1145/2785956.2787484.

[251] Tobias Ziegler, Philip A. Bernstein, Viktor Leis, and Carsten Binnig. "Is Scalable OLTP in the Cloud a Solved Problem?" In: *13th Annual Conference on Innovative*

Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023, Online Proceedings. www.cidrdb.org, 2023. URL: https://www.cidrdb.org/cidr2023/papers/p50-ziegler.pdf.

[252] 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 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.

[253] 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.

[254] Tobias Ziegler, Dwarakanandan Bindiganavile Mohan, Viktor Leis, and Carsten Binnig. "EFA: A Viable Alternative to RDMA over InfiniBand for DBMSs?" In: *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022*. Ed. by Spyros Blanas and Norman May. ACM, 2022, 10:1–10:5. DOI: 10.1145/3533737.3538506. URL: https://doi.org/10.1145/3533737.3538506.

[255] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. "Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA." In: *SIGMOD '23: International Conference on Management of Data, Seattle, WA, USA, June 18 - 23, 2023*. ACM, 2023.

[256] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. "Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks." In: *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.

[257] Pengfei Zuo, Qihui Zhou, Jiazhao Sun, Liu Yang, Shuangwu Zhang, Yu Hua, James Cheng, Rongfeng He, and Huabing Yan. "RACE: One-sided RDMA-conscious Extendible Hashing." In: *ACM Trans. Storage* 18.2 (2022), 11:1–11:29. DOI: 10.1145/3511895. URL: https://doi.org/10.1145/3511895.