HYPERSCALE DATA PROCESSING WITH NETWORK-CENTRIC DESIGNS

Qizhen Zhang

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2022

To my family

ACKNOWLEDGMENTS

This dissertation is possible because of many people. I take this chance to acknowledge their contributions to its successful completion.

I first thank Vincent Liu and Boon Thau Loo. I am beyond grateful for having both of them as my advisors. Boon took me under his wing when I decided to leave my previous graduate school six years ago. He told me my English was as good as his when I was worried about it before my first talk at SoCC 2017. Moments like these are countless. Boon has been an amazing advisor since the day we first met and has always been there to guide my research, encourage me in difficult times, and share my joy when I succeed. His interdisciplinary expertise and vision across different systems areas (e.g., databases and distributed systems) inspired this dissertation work and the spirit of my research. I feel blessed to have been working with Vincent since he joined Penn. He introduced me to networking research and taught me everything about data center networks—the "network" in the title of this dissertation. Vincent has been an example of both a great researcher who aims at challenging problems, designs innovative solutions, and pushes the boundaries of human knowledge, and a great person who is honest, kind, and true to others. I consistently benefit from his advice on thinking with first principles. Vincent and Boon have been generous with their time and patience in helping me finish this dissertation and become an independent researcher. I cannot think of a better way to spend these six years than growing under their guidance.

I also thank Phil Bernstein. It has been my privilege to be mentored by him in the data systems group at Microsoft Research and to have him on my dissertation committee. The collaboration with Phil is essential to this dissertation and my graduate studies. We worked on

disaggregated memory for remote caching in the summer of 2020, which resulted in the Redy system (Chapter 5). We built CompuCache the next year, which augments remote caches with computability using Spot VMs. The experience of working with Phil is undoubtedly invaluable. Phil's cares about both real-world impacts and technical details were crucial to these projects and touched me deeply. Our conversations have leveled up my knowledge of the history of the database community and Microsoft. Phil has been very supportive and provided numerous constructive comments on my research, talks, and writings. I am indebted to Phil for all his help.

I have also been fortunate to receive mentorship from Ang Chen. He is an exemplar of enthusiasm and kindness. His constant passion for systems research has been inspiring me. Ang has helped with my research since the first GraphRex system (Chapter 2). He is also a major contributor to the DDC project (Chapters 3 and 4). He has been heavily involved in every stage of all these projects, from brainstorming to experiment design and paper writing. Ang has also offered me suggestions about many other aspects of my research such as industrial internships. My life at Penn over the years would not have been the same without him.

I am also grateful to Sebastian Angel, Joe Devietti, and Zack Ives for serving on my dissertation committee and sharing their unique perspectives on this work. Sebastian has been critical to our DDC work (Chapters 3 and 4), and his expertise was a key to the success of that project. Joe has been an excellent chairman of my WPE-II exam and dissertation committees, providing me with valuable feedback after each defense. He also educated me in the literature on computer architecture. I enjoyed all the conversations with him. Zack has been providing insightful guidance on learning database systems. I am always impressed by his knowledge of recent advances in the field, considering his heavy administrative duties. Zack also helped me get prepared for my job search.

I have been fortunate to collaborate with a number of excellent researchers, who have been extremely helpful. They include Akash Acharya, Hongzhi Chen, and Simran Arora for GraphRex (Chapter 2); Yifan Cai, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, and Ke Zhong for the DDC project (Chapters 3 and 4); Daniel Berger and Badrish Chandramouli for Redy

(Chapter 5); Kelvin Ng, Charles Kazer, Shen Yan, and João Sedoc for MimicNet (Chapter 6). My appreciation goes on to the collaborators in other projects that I have worked on at Penn, including Shivani Agarwal, Meryem Essaidi, and Tengyuan Ye for the crowdfunding prediction project; James Cheng and Da Yan for the architectural study of graph systems.

I also deeply appreciate many others who have made significant contributions to this work. Chen Chen picked me up from the airport when I first arrived in the U.S. and later encouraged me to work on GraphRex and the DDC project. Susan Davidson read the first draft of the GraphRex paper and sent me detailed and positive comments. Nicolas Koh has been a close friend, and our chats sparked many interesting thoughts, especially when I worked on the DDC project and MimicNet. Yang Li and I brainstormed on join ordering for skewed graphs, which later helped the design of adaptive join ordering in GraphRex. Zhiyun Li and I exchanged thoughts about the implications of our DDC work for today's applications. Yucheng Lu and Jiacheng Wu made significant implementation efforts during the revision of GraphRex. Yizhou Shan helped us debug setup issues when we installed LegoOS for TELEPORT. Xujie Si discussed the mathematics in GraphRex with me. Val Tannen provided valuable feedback about the Datalog adaptation in GraphRex. Wentao Wu bolstered my interest in DDCs with his deep insights into the benefits of this new architecture for databases. Wenchao Zhou made encouraging comments on my DDC work and helped me get connected to industrial people for potential applications. I also thank many other Penn folks, including Mohammad Javad Amiri, Haoxian Chen, Leshang Chen, Max Demoulin, Soonbo Han, Yao Li, Yishuai Li, Tao Luo, Hui Lyu, Edo Roth, Lei Shi, Nik Sultana, Chenyuan Wu, Yinjun Wu, Meng Xu, Nofel Yaseen, Liangcheng Yu, Haoran Zhang, Yi Zhang, and Nan Zheng, for our discussions on various database and systems topics. During my internship at MSR in the summer of 2019, I chatted about my early-stage work on DDCs with Bailu Ding, Zhiwei Fan, Yao Lu, Tarique Siddiqui, and Cong Yan. The conversations were inspirational.

Last but not at all least, I thank my family. Their belief in the power of education and their support and love are the reasons I embarked on my Ph.D. journey and had the courage to finish it. I dedicate this dissertation to my mother, father, sister, uncles, aunts, and grandparents.

ABSTRACT

HYPERSCALE DATA PROCESSING WITH NETWORK-CENTRIC DESIGNS

Qizhen Zhang

Vincent Liu and Boon Thau Loo

Today's largest data processing workloads are hosted in cloud data centers. Due to unprecedented data growth and the end of Moore's Law, these workloads have ballooned to the hyperscale level, encompassing billions to trillions of data items and hundreds to thousands of machines per query. Enabling and expanding with these workloads are highly scalable data center networks that connect up to hundreds of thousands of networked servers. These massive scales fundamentally challenge the designs of both data processing systems and data center networks, and the classic layered designs are no longer sustainable.

Rather than optimize these massive layers in silos, we build systems across them with principled network-centric designs. In current networks, we redesign data processing systems with network-awareness to minimize the cost of moving data in the network. In future networks, we propose new interfaces and services that the cloud infrastructure offers to applications and codesign data processing systems to achieve optimal query processing performance. To transform the network to future designs, we facilitate network innovation at scale.

This dissertation presents a line of systems work that covers all three directions. It first discusses *GraphRex*, a network-aware system that combines classic database and systems techniques to push the performance of massive graph queries in current data centers. It then introduces data processing in disaggregated data centers, a promising new cloud proposal. It details *TELEPORT*, a compute pushdown feature that eliminates data processing performance bottlenecks in disaggregated data centers, and *Redy*, which provides high-performance caches using remote disaggregated memory. Finally, it presents *MimicNet*, a fine-grained simulation framework that evaluates network proposals at datacenter scale with machine learning approximation. These systems demonstrate that our ideas in network-centric designs achieve orders of magnitude higher efficiency compared to the state of the art at hyperscale.

TABLE OF CONTENTS

ACKNOWLEDGEMENT iv

ABSTRACT vii

LIST OF TABLES xii

LIST OF ILLUSTRATIONS xxii

CHAPTER 1 : Introduction 1

1.1 Hyperscale Data Processing 1

1.2 Principles of Network-centric Designs 3

1.3 Network-centric Systems 5

CHAPTER 2 : Hyperscale Graph Processing in Current Networks 12

2.1 Introduction 12

2.2 Background 16

2.3 GraphRex Query Interface 18

2.4 Query Planning 20

2.5 Global Operator Optimizations 28

2.6 Evaluation 39

2.7 Related Work 50

2.8 Summary 51

CHAPTER 3 : A Look at the Future—Understanding Data Processing in DDCs . . . 53

3.1 Introduction 54

3.2 Background 58

3.3 Overview of Executing DBMSs in DDCs 61

3.4 Setup and Methods for Extensive Evaluation 69

3.5 The Cost of Disaggregation 73
3.6 The Elasticity of DDCs 82
3.7 Analysis and Tuning 93
3.8 Proposals on Improving DBMS Performance in DDCs 101
3.9 Related Work 103
3.10 Summary 104

CHAPTER 4 : Achieving Optimal Data Processing Performance in DDCs 106
4.1 Introduction 107
4.2 Background 110
4.3 Design of TELEPORT 114
4.4 Data Synchronization 119
4.5 Applying TELEPORT 126
4.6 Implementation 131
4.7 Evaluation 132
4.8 Related Work 144
4.9 Summary 145

CHAPTER 5 : Realizing DDC Benefits in Today's Clouds 146
5.1 Introduction 147
5.2 Motivation 149
5.3 Redy Architecture 153
5.4 Remote Cache with RDMA 158
5.5 SLO-Driven Configuration 162
5.6 Remote Memory Management 168
5.7 Evaluation 171
5.8 FASTER with Redy 177
5.9 Related Work 182
5.10 Summary 184

CHAPTER 6 : Facilitating Hyperscale Network Innovation 185
6.1 Introduction 185
6.2 Motivation 189
6.3 Design Goals 191
6.4 MimicNet Overview 192
6.5 Internal Models 198
6.6 Feeder Models 205
6.7 Tuning and Final Simulation 206
6.8 Prototype Implementation 209
6.9 Evaluation 209
6.10 Related Work 223
6.11 Summary 225

CHAPTER 7 : Conclusion 226
7.1 Other Work 228
7.2 Future Work 229

BIBLIOGRAPHY 231

LIST OF TABLES

TABLE 1.1 : Hyperscale workloads are underpinning many services. 2

TABLE 2.1 : GraphRex's global operator optimizations and when they apply. . . . 29

TABLE 2.2 : Speedup of SHUFF and row-based compression in CC on *Twitter*. . . 32

TABLE 2.3 : An example of Hierarchical Deduplication with a single rack of two servers, with two workers per server. At each successive layer of the hierarchy, workers coordinate to deduplicate join results before incurring increasingly expensive communication 33

TABLE 2.4 : Hierarchical Deduplication in TC on *Twitter*. Dup % indicates received duplicates. 33

TABLE 2.5 : The percentage of tuples using each join order during the first four iterations of SG on *SynTw*. LR is the left-to-right join order and RL the right-to-left order. 37

TABLE 2.6 : Comparison of adaptive and static ordering. 37

TABLE 2.7 : Evaluation of `NV` on *Twitter*. 38

TABLE 2.8 : Evaluation of `CC` on *Twitter*. 39

TABLE 2.9 : Real-world, large-scale graphs in the evaluation. 41

TABLE 2.10 : Execution time and speedup for GraphRex (G.R.) compared to BigDatalog (B.D.), Giraph and PowerGraph (P.G.). This table presents results for CC and PR on four graph datasets (TW, FR, UK, CW). OOM indicates an out-of-memory error. B.D. does not support PR. 42

TABLE 2.11 : This table presents results for TC and REACH. 42

TABLE 3.1 : MonetDB buffer pool size tuning in Linux and LegoOS 98

TABLE 3.2 : PostgreSQL join algorithm tuning in Linux and LegoOS. Algorithms marked with * are what PostgreSQL selected. 99

TABLE 3.3 : Page faults in different set associativity configurations. 100

TABLE 4.1 : The components in executing a pushdown request. Grayed are factors on what TELEPORT has no control. 140

TABLE 5.1 : APIs provided by the cache client for applications. The underlined functions are for performing I/Os. 156

TABLE 5.2 : Variables balancing latency and throughput. 163

TABLE 6.1 : Basic set of scalable features. 201

TABLE 6.2 : Running time comparison for 20 s of simulated time of a 128 cluster, 1024 host data center. Benefits of MimicNet increase with simulated time and the size of the network as the first two values for MimicNet are constant. 215

LIST OF ILLUSTRATIONS

FIGURE 1.1 : A data center network connects hundreds of thousands of servers. . . 3

FIGURE 1.2 : Network-centric systems that this dissertation presents for addressing challenges in hyperscale data processing. 5

FIGURE 2.1 : Performance comparison (log scale) of SSSP between declarative systems: BigDatalog and GraphRex, and low-level graph systems: Giraph and PowerGraph on large graphs. All systems are run in a datacenter with 6 TB RAM and 1.6 thousand cores in aggregate. . . . 13

FIGURE 2.2 : The GraphRex architecture. A compiler generates a logical plan from a Datalog query. The static optimizer then constructs from the logical plan a datacenter-centric execution specification that is optimized before the final translation to and evaluation of the physical plan by workers. Gray lines describe dissemination of infrastructure configurations and black lines communication for query execution. 20

FIGURE 2.3 : Distributed Semi-Naïve in GraphRex 21

FIGURE 2.4 : Vertex states in GR-DSN. 22

FIGURE 2.5 : The logical plan of CC. 23

FIGURE 2.6 : The logical plan of SG. 23

FIGURE 2.7 : The execution specification of CC. 24

FIGURE 2.8 : The execution specification of SG. 24

FIGURE 2.9 : Two potential partitionings for TC. 27

FIGURE 2.10 : Static relation partitioning. 28

FIGURE 2.11 : Column-based organization for `r(V,A,B)`, where `V` is the partition key. Shaded is compressed data for reducing the number of bytes that we send in the network. 30

FIGURE 2.12 : An example hierarchical transfer. Each worker groups its tuples by partition key, and sends the them first within a server, then within a rack, and finally to their destinations. A naive system would send directly to other racks. Colors track where the tuple was generated; numbers indicate the partition. 31
FIGURE 2.13 : SG on an example graph. 35
FIGURE 2.14 : Hierarchical Global Aggregation in a rack. After worker-level aggregation, intermediate aggregates are shuffled (1) at a server-level, and then (2) at a rack-level. 38
FIGURE 2.15 : Aggregation query evaluation with CM. 43
FIGURE 2.16 : Multi-way join query evaluation with SG. 44
FIGURE 2.17 : Heat map of cross-server communication. 45
FIGURE 2.18 : Heat maps of traffic volume (number of bytes sent between servers, values are log 10 scale) for CC on *FR*. GraphRex (b) saves 94.8% traffic compared to the infrastructure-agnostic baseline (a). 46
FIGURE 2.19 : System performance when varying link degradations. 47
FIGURE 2.20 : System performance with varying #aggregation switches. 47
FIGURE 2.21 : The CDF of performance with random background traffic. 48
FIGURE 2.22 : Scalability with #Servers on *TW*. 49
FIGURE 2.23 : Scalability with #Servers on *FR*. 49
FIGURE 2.24 : Scalability with #Servers on *UK* 49

FIGURE 3.1 : An illustration of resource disaggregation. Same type of resources are centralized in a *resource pool*. Resource pools are disaggregated and connected by a fast network. 58

FIGURE 3.2 : DBMS execution in DDCs. DBMS workers are spawned on compute nodes with their small local memory acting as a cache. Buffer pools live in a remote memory pool; a storage pool stores and manages the database files. Workers send control messages to allocate and manage resources, and the data is transferred between memory and storage pool (loading and spilling) and the processing and memory pool (fetching and eviction). 62
FIGURE 3.3 : Figure (a) depicts a hash partitioning when the DBMS is running on LegoOS. Figure (b) shows the same operation but with additional primitives (§3.8). 65
FIGURE 3.4 : Query performance on hash indexes of TPC-H (scale factor 10) tables. LM stands for local memory. 67
FIGURE 3.5 : An optimized physical plan for TPC-H Query 5. Shaded are the operators that use hash tables that we built in Figure 3.4. 68
FIGURE 3.6 : Execution performance of hash joins and end-to-end TPC-H Query 5. 68
FIGURE 3.7 : Summary of parameters in MonetDB and PostgreSQL. 70
FIGURE 3.8 : Peak memory usage of TPC-H queries in MonetDB. 72
FIGURE 3.9 : Peak memory usage of TPC-H queries in PostgreSQL. 72
FIGURE 3.10 : MonetDB query execution time slowdowns with 4 GB local memory in LegoOS. The baseline is a single Linux server. 74
FIGURE 3.11 : MonetDB slowdowns with 1 GB local memory in LegoOS. 74
FIGURE 3.12 : MonetDB slowdowns with 64 MB local memory in LegoOS. 74
FIGURE 3.13 : The simplified execution plan for Q3 in PostgreSQL. Blue operators involve disk I/O and red operators are in memory. 76
FIGURE 3.14 : PostgreSQL cold execution time slowdowns with 4 GB local memory in LegoOS (Q20 excluded). The Baseline is a single Linux server. . 77
FIGURE 3.15 : PostgreSQL (cold) slowdowns with 1 GB local memory in LegoOS. 77
FIGURE 3.16 : PostgreSQL (cold) slowdowns with 64 MB local memory in LegoOS. 77

FIGURE 3.17 : PostgreSQL hot execution time slowdowns with 4 GB local memory in LegoOS (Q20 excluded). The Baseline is a single Linux server. . 78
FIGURE 3.18 : PostgreSQL (hot) slowdowns with 1 GB local memory in LegoOS. . 78
FIGURE 3.19 : PostgreSQL (hot) slowdowns with 64 MB local memory in LegoOS. 78
FIGURE 3.20 : The slowdowns of running distributed DBMSs in a cluster compared to a single machine of the same hardware. 79
FIGURE 3.21 : The slowdowns of LegoOS in the TPC-H throughput benchmark. Trends are similar to the observations for single query performance. 81
FIGURE 3.22 : MonetDB performance when varying local memory size for Query 16. 83
FIGURE 3.23 : Varying local memory size for Query 5. 83
FIGURE 3.24 : Varying local memory size for Query 9. 83
FIGURE 3.25 : PostgreSQL performance (cold) when varying local memory size for Query 6. 85
FIGURE 3.26 : Varying local memory size for Query 13. 85
FIGURE 3.27 : Varying local memory size for Query 4. 85
FIGURE 3.28 : PostgreSQL performance (hot) when varying local memory size for Query 6. 86
FIGURE 3.29 : Varying local memory size for Query 13. 86
FIGURE 3.30 : Varying local memory size for Query 4. 86
FIGURE 3.31 : MonetDB performance when varying dataset size for Query 16. . . 87
FIGURE 3.32 : Varying dataset size for Query 5. 87
FIGURE 3.33 : Varying dataset size for Query 9. 87
FIGURE 3.34 : PostgreSQL performance (cold) when varying dataset size for Query 6. 88
FIGURE 3.35 : Varying dataset size for Query 13. 88
FIGURE 3.36 : Varying dataset size for Query 6. 88
FIGURE 3.37 : MonetDB query execution performance in Linux and LegoOS with mixed workloads starting with cold memory. 89
FIGURE 3.38 : PostgreSQL performance with mixed workloads. 90

FIGURE 3.39 : The effect of prefetching in PostgreSQL. 92
FIGURE 3.40 : N_{RM} in MonetDB executions with 4 GB local memory. 94
FIGURE 3.41 : N_{RM} with 1 GB local memory. 94
FIGURE 3.42 : N_{RM} with 64 MB local memory. 94
FIGURE 3.43 : N_{RM} in PostgreSQL executions with 4 GB local memory. 95
FIGURE 3.44 : N_{RM} in PostgreSQL executions with 1 GB local memory. 95
FIGURE 3.45 : N_{RM} in PostgreSQL executions with 64 MB local memory. 95

FIGURE 4.1 : The benefits and cost of running DBMSs in DDCs. 108
FIGURE 4.2 : DDC performance overhead compared to a monolithic server. . . . 112
FIGURE 4.3 : A data-intensive relational operator example, selection, in DDCs. . 113
FIGURE 4.4 : TELEPORT architecture. 116
FIGURE 4.5 : Application performance in different systems and with different data sync approaches in TELEPORT. 119
FIGURE 4.6 : The benefit of a manual data sync with `syncmem` when false sharing occurs in the application. 119
FIGURE 4.7 : Preparation of the page tables before pushdown execution in the memory pool. *compute_pgs* is the transmitted list of pages from the compute pool. 121
FIGURE 4.8 : Handling compute-pool page faults during pushdown. 122
FIGURE 4.9 : Handling memory-pool page faults during pushdown. 123
FIGURE 4.10 : Performance breakdown of the query with the greatest cost of scaling in DDCs in each system. For every operator/phase, we show the times in both local and DDC executions and the remote memory accesses in the DDC. A common pattern is that there are one or two arbitrary operators/components dominating the overall query execution time. 126
FIGURE 4.11 : The flexibility of TELEPORT enables the pushdown of various memory-intensive operators in existing systems with minimal modification. 127

FIGURE 4.12 : Original projection code in MonetDB. 128
FIGURE 4.13 : TELEPORTing this memory-intensive operator is intuitive. 129
FIGURE 4.14 : The performance improvement of pushing the operators in Q_{filter} to the memory pool with TELEPORT. 133
FIGURE 4.15 : TELEPORT improves the performance of a wide range of data processing tasks. By removing expensive data movement, TELEPORT significantly reduces the overhead of memory disaggregation, up to an order of magnitude speedup compared to the baseline DDC. . . 133
FIGURE 4.16 : Query speedups with large disaggregated memory pools vs. SSDs. . 136
FIGURE 4.17 : The benefits of increasing physical memory for large workloads. . . 136
FIGURE 4.18 : Pushdown performance (Q_9) with different computation power settings in the memory pool. 138
FIGURE 4.19 : The benefits of parallelizing the processing of concurrent pushdown requests in the memory pool. 138
FIGURE 4.20 : The performance of different levels of pushdown. 139
FIGURE 4.21 : TELEPORT performance breakdown with different sync methods. . 141
FIGURE 4.22 : Application performance with varying levels of contention. 143
FIGURE 4.23 : The number of coherence messages in TELEPORT. 143

FIGURE 5.1 : The significance of stranded memory. 151
FIGURE 5.2 : The dynamics of stranding events. 151
FIGURE 5.3 : The impact of the RDMA configuration in Redy. 152
FIGURE 5.4 : Architecture of Redy. An application interacts with the *cache client*. The global *cache manager* asks the data center's VM allocator to reserve VMs to host the cache. *Cache servers* on those VMs coordinate with the client for cache accesses. 154
FIGURE 5.5 : A region table maps a cache to VMs. 157

FIGURE 5.6 : RDMA message flow in Redy. Ring buffers enable pipeline parallelism between adjacent threads. Message rings are only used when batch size is greater than one. 158
FIGURE 5.7 : Redy optimizations effectively decrease latency. 161
FIGURE 5.8 : Effectiveness on increasing throughput. 161
FIGURE 5.9 : Configuration performance modeling. 166
FIGURE 5.10 : Online SLO-based searching in the manager. 167
FIGURE 5.11 : The latency of Redy caches with latency-optimal configurations for different record sizes. On average, accessing records up to 4 KB sizes takes less than 5 μs, close to the raw RDMA hardware speed. 172
FIGURE 5.12 : The throughput of Redy caches with throughput-optimal and stranded-memory configurations. Batching small records improves the throughput by an order of magnitude. 173
FIGURE 5.13 : Satisfying latency SLOs. 175
FIGURE 5.14 : Satisfying throughput SLOs. 175
FIGURE 5.15 : The impact of region migration on reads. 176
FIGURE 5.16 : The impact of region migration on writes. 176
FIGURE 5.17 : FASTER with Redy. New records are appended to both tiers. Reads to records in Redy are only served by Redy. 178
FIGURE 5.18 : FASTER throughput with Redy, SMB Direct, and SSD respectively on the YCSB benchmark when the working set is larger than local memory.179
FIGURE 5.19 : FASTER throughput when varying local memory size. 180
FIGURE 5.20 : FASTER with various local memory sizes on uniform YCSB. 181
FIGURE 5.21 : Tiered store with various remote cache sizes. 181

FIGURE 6.1 : Accuracy for MimicNet's predictions of the FCT distribution for a range of data center sizes. Accuracy is quantified via the Wasserstein distance (W_1) to the distribution observed in the original simulation. Lower is better. Also shown are the accuracy of a flow-level simulator (SimGrid) and the accuracy of assuming a small (2-cluster) simulation's results are representative. 187
FIGURE 6.2 : OMNeT++ performance on leaf-spine topologies of various size. Even for these small cases, 5 mins of simulation time can take multiple days to process. Results were similar for ns-3 and other frameworks.190
FIGURE 6.3 : The end-to-end, fully automated workflow of MimicNet. 193
FIGURE 6.4 : Breakdown of traffic in a to-be-approximated cluster. MimicNet approximates all traffic that does not interact with the observable cluster (dotted-red lines) using the models in the referenced sections. . 195
FIGURE 6.5 : Ground truth and LSTM-predicted drops for a one-second test set using different loss functions. The y-axis is 1 for dropped, 0 for not. Ground truth has 0.3% drop rate and BCE loss has 0.01%. WBCE results in more realistic drop rates depending on the weight (w=0.6: 0.14%; w=0.9: 0.49%). 202
FIGURE 6.6 : Ground truth and LSTM-predicted latency (in seconds) for a one-second test set using different loss functions. With each, we report the output of the objective, MAE (listed in parentheses). Unfortunately, using MAE directly as the loss function fails to capture outliers. Instead, Huber produces more realistic results and a better eventual MAE score. 203

FIGURE 6.7 : The accuracy of MimicNet in the baseline configuration for 2 clusters and 128 clusters. Also shown are results from SimGrid and the assumption that small-scale results are representative. W_1 to ground truth is shown in parentheses. We annotate the 99-pct value of each metric for every approach at the tail in 128 clusters. 210
FIGURE 6.8 : Throughput Scalability. 212
FIGURE 6.9 : RTT Scalability. Flow-level simulation is too coarse-grained to provide this metric. 212
FIGURE 6.10 : Simulation running time speedup brought by MimicNet in different sizes of data centers. In a network of 128 clusters (256 racks), MimicNet reduces the simulation time from 12 days to under 30 minutes, achieving more than two orders of magnitude speedup. The speedups are consistent and stable across different workloads. . . 214
FIGURE 6.11 : Simulation latency with different network sizes. 215
FIGURE 6.12 : Simulation throughput with different network sizes. 216
FIGURE 6.13 : Simulation latency with different simulation lengths. 217
FIGURE 6.14 : Simulation throughput with different simulation lengths. 217
FIGURE 6.15 : Tuning the marking threshold K in DCTCP: the configuration that achieves the lowest 90-pct FCT is different between 2 clusters ($K = 60$) and 32 clusters ($K = 20$). MimicNet provides the same answer as the full simulation for 32 clusters, but it is $12\times$ faster. 218
FIGURE 6.16 : FCT distributions of Homa, DCTCP, TCP Vegas, and TCP Westwood for a 32-cluster data center. 220
FIGURE 6.17 : Comparison of throughput distributions. 221
FIGURE 6.18 : Comparison of packet RTT distributions. 222
FIGURE 6.19 : Compute consumption in different simulation approaches. 223

FIGURE 6.20 : The impact of the window size on modeling accuracy and speed. The BDP of the network is around 12 packets. More packets in the window help loss descent (through epochs), but can make the training slower (training latency is per batch in Python). 224
FIGURE 6.21 : The impact of the window size on modeling (validation) accuracy and inference latency per packet in C++. 224

CHAPTER 1

INTRODUCTION

One of the most fundamental tasks in computer science is to process data in a timely manner. It is critical for nearly every computing workload and affects aspects of our lives as diverse as health (e.g., tracking outbreaks in a pandemic [145]), finance (e.g., fast trading and quantitative analysis [203, 232]), education (e.g., large-scale resource sharing [229]), and entertainment (e.g., massive online services and recommendations [91, 269]). Data processing systems, such as database management systems, have been known for providing easy-to-use querying languages and achieving high performance on processing high-level queries. Unfortunately, unprecedented data growth has meant that achieving good performance is increasingly challenging for all of the components involved: the infrastructure must provide massive amounts of resources, data processing systems must utilize the resources efficiently, and even applications sometimes need different algorithm designs.

1.1 Hyperscale Data Processing

The largest data processing workloads are now hosted in cloud data centers. In recent years, data has been growing exponentially, but hardware performance advancement has not been able to keep up (e.g., the end of Dennard scaling and slowdown of Moore's Law). These two trends have forced cloud data processing workloads to balloon to the *hyperscale* level, where a single query (e.g., for database analytics, graph processing, MapReduce, or machine

Hyperscaler	**Workloads**
Amazon	*"Each service publishes datasets to Amazon's analytics infrastructure, including more than 50 petabytes of data and 75,000 tables, processing 600,000 user analytics jobs each day."* [2]
Google	*"The Google Search index contains hundreds of billions of webpages, which is well over 100,000,000 gigabytes in size."* [13]
Facebook	*"...our needs at Facebook with over 1.39B users and hundreds of billions of social connections..."* [79]
Alibaba	*"...analytical workloads from our clients...: 10PB+ data, hundred thousands of tables and trillions of rows..."* [279]
Databricks	*"The ability to execute rapid queries on petabyte-scale data sets using standard BI tools is a game changer..."* [10]
Snowflake	*"Largest single table: 45 trillion rows"* [23]
OpenAI	*"GPT-3 was trained on hundreds of billions of words"* [25]

Table 1.1: Hyperscale workloads are underpinning many services.

learning) can involve billions to trillions of data items and hundreds to thousands of servers. Table 1.1 lists a few examples of hyperscale workloads that support today's important services. Such massive workloads are enabled by data center networks, which also constantly expand in response to workload growth. These networks are highly scalable and connect up to hundreds of thousands of distributed machines. Figure 1.1 shows a network structure that is widely adopted in today's largest data centers [41, 84, 172, 245].

Many traditional design principles break down at hyperscale. A particular instance is the classic principle of layering, in which different components of a network, from applications to the transport and hardware, are built independently as layers and connected by well-specified protocols. Layering has allowed people working on different components to focus on their own systems with clear optimization goals. However, layered designs are no longer sustainable at hyperscale. For instance, it is difficult for applications to know how their data is transferred in the network with layering, so applications can make egregious decisions in their execution models; the cloud infrastructure also performs poorly without rethinking the interfaces and services exposed to its applications. In fact, most cloud providers already break layering in their current architectures. For example, cloud giants are creating custom hardware for their most common applications [102, 189, 268, 271], and their networking stacks

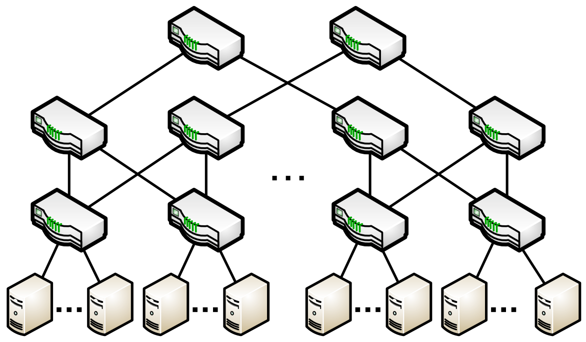


Figure 1.1: A data center network connects hundreds of thousands of servers.

are also moving towards the user space [174, 238]. The research community, however, has not kept up, due to the lack of systematic investigations on applying cross-layer designs to address hyperscale challenges in data processing.

1.2 Principles of Network-centric Designs

To overcome the weaknesses of layered designs, we bridge data processing systems and data center networks to achieve efficient data processing in cloud data centers. Our general finding is that for large-scale data processing, the network is often either the performance bottleneck or the leverage we can use to solve scale problems. Hence, we build systems with *network-centric designs*, which concern three questions and their associated challenges as follows.

- **How do data processing systems perform in current networks?** Developers of data processing systems treat the network as a black box that simply delivers messages from point A to point B. This assumption greatly simplifies the design of distributed data processing. For example, systems can ignore details like the physical placement of distributed workers in their execution models. Data center networks have also traditionally tried to support this assumption with a "one big switch" abstraction that it provides to applications. However, it incurs a great cost to maintain the abstraction at scale because a data processing job that requires hundreds of machines must necessarily span multiple racks or even clusters. Today's data center networks are commonly oversubscribed due to cost considerations [15, 245], meaning that the cross-rack and cross-cluster network performance (bandwidth and latency) is significantly worse than that within racks and clusters. Such data center network characteristics are important for large-scale data processing but rarely considered in prior systems. In consequence, network communications are becoming the primary performance bottleneck when data processing systems scale out.

- **How will they perform in future networks?** We envision that future data centers are disaggregated. Disaggregated data centers (DDCs) are a promising new cloud proposal that decouples different types of resources from monolithic servers into resource pools, e.g., compute/CPU pool, memory pool, and storage pool, and connects these pools by a high-speed network [109, 121, 235]. Compared to traditional server architectures, disaggregation offers vast operational benefits that are particularly attractive to hyperscale data centers. For instance, it solves the traditional bin-packing problem when assigning virtual machines (VMs) to physical machines by making resource allocation independent, potentially saving billion-dollar cost that is incurred by underutilized hardware resources in current data centers. DDCs also make data center expansion easier as different resources can be added and managed independently. In addition, DDCs provide hardware failure isolation and achieve better elasticity for applications.

  Despite these benefits, this architecture can potentially disrupt data processing systems. The memory disaggregation in DDCs completely separates compute and data, but data processing systems often cache large working sets in memory during query execution and have been designed with the assumption that memory accesses are cheap and can be random. Hence, processing queries with these systems incurs frequent data movement through the network that connects the compute and memory components, thereby resulting in high performance overhead of disaggregation.

- **How do networks evolve?** To enable network innovations such as resource disaggregation introduced above, we note that an essential part is to evaluate the performance of the proposals. Unfortunately, evaluating hyperscale networks is intractable because of their size and complexity. This is true for testbeds, where few, if any, can afford a

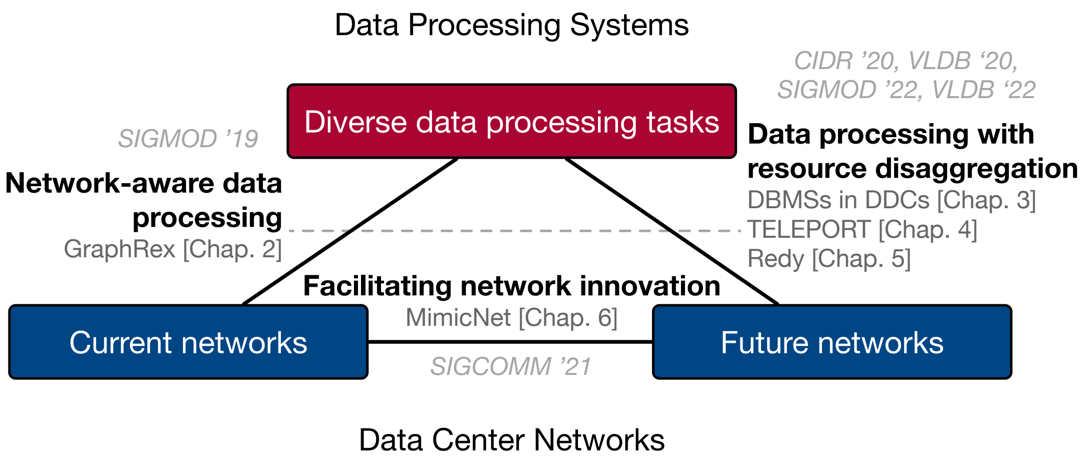


Figure 1.2: Network-centric systems that this dissertation presents for addressing challenges in hyperscale data processing.

dedicated, full-scale replica of a data center. It is also true for simulations, which while originally designed for at-scale network evaluation, have struggled to cope with today's hyperscale infrastructure. For example, simulating the TCP protocol for 60 seconds in a data center of a thousand machines takes 36 days to finish. Evaluation is becoming the main roadblock for data center network innovation.

The principles of network-centric designs correspond to these three questions. In current networks where the network has already been deployed and thus hard to change, we redesign data processing systems with network-awareness to minimize communication times. In future networks, we introduce new interfaces and services that the cloud infrastructure offers to applications to expose radical architectural changes; meanwhile, we codesign data processing systems to achieve optimal query processing performance by exploiting these new features. Finally, to facilitate network transformation from current to future designs, we must propose new network designs and be able to evaluate the performance of the proposals at scale.

1.3 Network-centric Systems

This dissertation improves the efficiency of hyperscale data processing with the systems that we build by following the network-centric design principles. It covers all directions involved and makes three thrusts: (1) *network-aware data processing* for addressing scaling bottlenecks in current networks, (2) *data processing with resource disaggregation* for fully understanding the behavior of data processing systems and overcoming their limitations in DDCs, a new cloud architecture, and (3) *facilitating network innovation* with at-scale network evaluation. Figure 1.2 maps out the thrusts and systems of this dissertation, which we describe as follows.

1.3.1 Network-aware Data Processing

The first thrust applies current data center domain knowledge to data processing, for which we have built GraphRex [286], the first system that processes hyperscale graph queries by systematically adopting network awareness.

Large-scale graph analytics is a popular example of hyperscale data processing as real-world graphs now scale up to billions of vertices and trillions of edges. Our study on massive workloads showed that network communication dominated the times in processing large graph queries. However, state-of-the-art systems were incapable of capturing modern data center network characteristics and thus suffered from substantial communication cost.

This motivated our GraphRex system [281], which we developed for processing graph queries at data center scale. GraphRex has three goals: ease of programming, querying efficiency, and robustness to network dynamics. These goals are required in practice but achieving all three together is difficult. We must carefully design the query language, execution, and optimizations with domain-specific knowledge.

GraphRex achieves the first goal with a declarative and easy-to-use query interface. The second and third goals require it to reduce overall network traffic, especially over bottleneck links. GraphRex introduces a set of new operators in its execution engine, called *global operators*, which consider data center network characteristics. For instance, as one of the global operators, our shuffle operator exchanges messages between workers in a topology-aware fashion, which consolidates messages when network cost is low and compresses them to minimize the amount of data sent through oversubscribed links. Combined, these operators substantially level up the performance and robustness of GraphRex in data centers.

1.3.2 Data Processing with Resource Disaggregation

The second thrust focuses on future networks. We introduced data processing in disaggregated data centers (DDCs) and conducted extensive research under this topic. Specifically, we pioneered the rethinking of data processing system designs in DDCs [284] and, for the first time, investigated the effect of DDCs on production systems [285]. We then proposed TELEPORT [287], a new DDC feature that allows for optimal data processing performance. We also developed Redy [283] with industry to harvest DDC benefits in today's clouds. We now introduce this line of work in greater detail.

Rethinking and understanding data processing systems in DDCs. We opened the topic with microbenchmarks on hash operations. We showed that the separation of compute and memory, i.e., memory disaggregation, causes significant overhead for data center applications. It happens because every data access to the main memory now translates to a network communication. This overhead is particularly felt by data processing systems, which hold large working sets in memory. We proposed a set of novel operators for reducing the overhead.

The largest public clouds are already in the transition to disaggregated architectures, including memory disaggregation. Fully understanding DDC implications on data processing is hence both urgent and important. We took the first step to investigate DDC effect on production systems. By studying DBMSs, which execute memory-intensive queries, we found that both the benefits and overhead of DDCs are substantial. On one hand, a large disaggregated memory pool can prevent the processing of memory-intensive queries from being spilled to secondary storage. On the other hand, network communications for remote memory accesses are expensive for large queries.

TELEPORT: achieving optimal data processing performance in DDCs. To overcome the overhead of DDCs and unlock all their benefits, we introduced TELEPORT, a compute pushdown framework that enables data processing systems to offload expensive operations close to data. It is based on disaggregated operating systems (OSes) that emulate traditional OS interfaces to provide backward compatibility in DDCs, so that current applications can directly run to harvest the benefits. With TELEPORT, applications are capable of executing light-weight but memory-intensive operations in the memory pool. In doing so, they eliminate costly remote memory accesses and hence achieve better performance.

TELEPORT is unique in its generality and efficiency. With a new system call, it allows applications to offload arbitrary pieces of computation by wrapping them as functions. Pushing a function down is as simple as providing the pointers of the function and its arguments to the memory pool. This is possible because applications' stack, heap, and code pages all live in the memory pool as a byproduct of disaggregated OSes. Data synchronization is critical for TELEPORT: the compute pool caches part of the main memory, so data copies in different pools can diverge before, during, and after a pushdown call. Without proper synchronization, concurrent threads in the two pools may access the same memory pages without observing each other's updates. TELEPORT employs specialized synchronization primitives that guarantee memory coherence. It only transfers data on applications' demands, which outperforms application-agnostic alternatives.

Redy: realizing DDC benefits in today's clouds. With TELEPORT, DDCs provide a radical solution to the limitations of current cloud infrastructure. Can we preharvest some of the DDC benefits in today's clouds? To seek the answer, we investigated the data centers of Microsoft Azure, one of the largest cloud providers. Azure data centers have massive amounts of unused memory, much of which is stranded because all local CPUs are allocated to VMs. Nevertheless, stranded memory can be productively employed by accessing it via Remote Direct Memory Access (RDMA). RDMA can access remote memory without involving remote CPUs and bypass OS kernels for low latency. Based on these insights, we developed Redy, a new cloud service that uses stranded memory as remote caches. It offers a lower-latency alternative to SSDs, using disaggregated memory resources that would otherwise go to waste.

Our use of RDMA leads to two challenges. The first is performance. Tuning RDMA requires complex, low-level optimizations to trade off network latency, throughput, and resource cost. There is no one-size-fits-all configuration. Second, stranded memory resources are highly dynamic. They come and go depending on VM allocations. Their availability can be as short as a few minutes. Providing reliable caches using dynamic memory is hard.

Redy addresses the first challenge with *SLO-based configuration*. It takes as input a user-defined performance service-level objective (SLO) and uses a dynamic optimization process to automatically find the configuration that satisfies the SLO with *minimal resource cost*. To solve the dynamic challenge, we developed a *dynamic memory manager* that migrates a cache to new stranded memory when the old memory is reclaimed by the cloud VM allocator. The migration occurs in a way that *minimizes the impact on cache performance*.

1.3.3 Facilitating Network Innovation

The final thrust of this dissertation concerns network evaluation. We focused on packet-level simulation, which was originally designed with three goals: providing performance results for *arbitrary scale* with *arbitrary network extensions* and *arbitrary user instrumentation*. Unfortunately, simulating data centers is prohibitive: parallelization barely works as the complexity of the network forces simulators to serialize all simulated events; approximations such as flow-level approaches lack accuracy and generality. We developed MimicNet [288], the first scalable simulator that predicts the performance of data center network proposals with high accuracy using machine learning techniques.

MimicNet assumes the popular FatTree topology, where racks of servers are connected by a cluster network, and clusters are connected by a set of core switches. Cluster is the unit for scaling—real-world data centers can have thousands of clusters. Based on this topology, MimicNet works as follows. It first runs a small simulation of two clusters in full fidelity. Using the simulation results, it trains machine learning (ML) models for approximating intra-cluster and inter-cluster behavior. Finally, it runs an N-cluster simulation by composing (1) a single 'observable' cluster for user instrumentation, regardless of the total number of clusters in the data center, and (2) $N-1$ clusters that are not directly observed. All components in the observable cluster and all of the remote components with which it communicates are simulated in full fidelity. All other behavior that is not directly observed by the user is approximated by the trained models. In essence, MimicNet *predicts the performance of a large network by* *observing only small subsets of it*. By removing the simulation of most clusters, it decreases the simulation time by orders of magnitude.

Network complexity makes achieving high accuracy in MimicNet approximations challenging. We address this by baking data center domain knowledge into the designs of the ML models such as their learning features and loss functions. Additionally, we allow users to trade accuracy off for higher simulation speed by training smaller models instead of more accurate larger ones.

1.3.4 Summary

With the above systems, this dissertation demonstrates that our ideas in network-centric designs can improve the efficiency of hyperscale data processing by orders of magnitude compared to the state of the art. Specifically, we evaluated GraphRex with large real-world graphs in a data center testbed what has several terabytes of memory and thousands of CPU cores, a scale much larger than that have been tested in prior declarative graph systems. GraphRex proved to be two orders of magnitude faster than state-of-the-art systems under various network conditions. We applied TELEPORT to three popular workloads: in-memory database, graph processing, and MapReduce. Compared to baseline DDCs, these workloads execute up to an order of magnitude faster with TELEPORT. In addition, TELEPORT requires little effort from its users. We integrated Redy with a production key-value store to demonstrate its ease of use and performance benefits. Results show that Redy caches are $20\times$ and $8\times$ faster than an SSD and an RDMA baseline respectively. For MimicNet, our experiments with real-world network traces showed that it simulates data center networks orders of magnitude faster than full-fidelity packet-level simulation, and its results closely mimic the ground-truth. For example, while it takes more than a month to simulate recent network innovations in a data center of a thousand servers in full fidelity, MimicNet finishes in a few hours and its predictions are within 5% of the true results.

The remainder of this dissertation is outlined as follows. The first part (Chapter 2) presents GraphRex. The second part (Chapter 3, Chapter 4, and Chapter 5) focuses on data processing in disaggregated data centers. It details our works on investigating database management systems in disaggregated data centers, TELEPORT, and Redy. The third part (Chapter 6) describes MimicNet. Finally, Chapter 7 summarizes the dissertation, mentions other works that are not covered, and discusses future directions.

CHAPTER 2

HYPERSCALE GRAPH PROCESSING IN CURRENT NETWORKS

In this chapter, we present GraphRex, an efficient, robust, scalable, and easy-to-program framework for graph processing on current datacenter infrastructure. To users, GraphRex presents a declarative, Datalog-like interface that is natural and expressive. Underneath, it compiles those queries into efficient implementations. A key technical contribution of GraphRex is the identification and optimization of a set of global operators whose efficiency is crucial to the good performance of datacenter-based, large graph analysis. Our experimental results show that GraphRex significantly outperforms existing graph analytics frameworks—both high- and low-level—in scenarios ranging across a wide variety of graph workloads and network conditions, sometimes by two orders of magnitude.

2.1 Introduction

Over the past decade, there has been a proliferation of graph processing systems, ranging from low-level platforms [80, 136, 176, 183] to more recent declarative designs [241]. While users can deploy these systems in a variety of contexts, the largest instances routinely scale to multiple racks of servers contained in vast datacenters like those of Google, Facebook, and Microsoft [225]. This trend of large-scale distributed data processing is likely to persist as data continues to accumulate.

These massive deployments are in a class of their own: their size and the inherent prop-

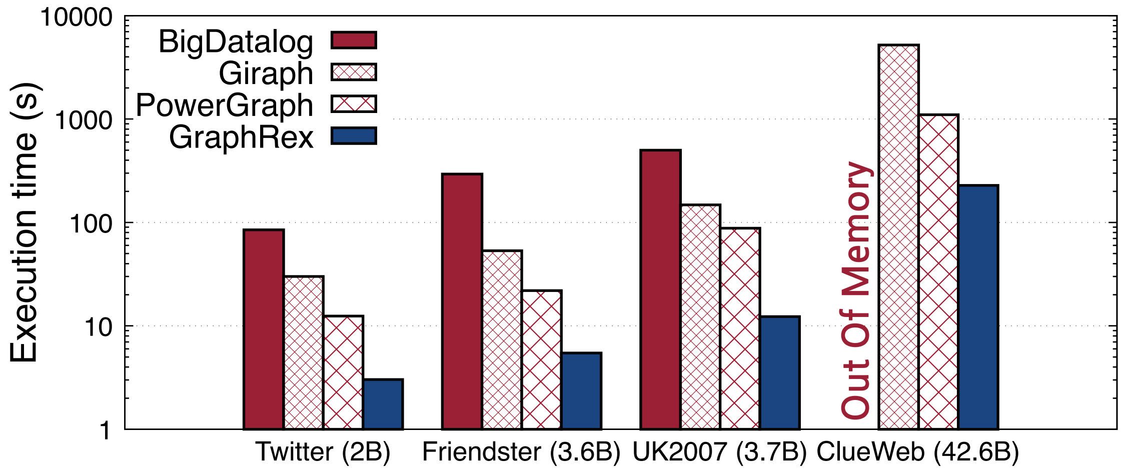


Figure 2.1: Performance comparison (log scale) of SSSP between declarative systems: BigDatalog and GraphRex, and low-level graph systems: Giraph and PowerGraph on large graphs. All systems are run in a datacenter with 6 TB RAM and 1.6 thousand cores in aggregate.

erties of the datacenter infrastructure present unique challenges for graph processing. To highlight these performance issues on practical workloads, Figure 2.1 illustrates, for multiple graph processing systems and billion-edge graphs, the running time of a single-source shortest path (SSSP) query on a representative datacenter testbed. We tested four systems: (1) BigDatalog [241], a recent system that provides a declarative interface to Spark; (2) Giraph [80], a platform built on Hadoop that powers Facebook's social graph analytics; (3) PowerGraph [114], a highly optimized custom framework; and as a sneak preview of the space of possible improvement (4) GraphRex, the system that this chapter presents for large-scale datacenter-based graph processing. As the results demonstrate, while the three existing systems are capable of scaling to billion-edge workloads, our approach leads to *up to two orders of magnitude* better performance.

The above results barely scratch the surface of optimization opportunities for large-scale graph queries in datacenters. We note two significant opportunities that are underexplored in previous work:

Opportunity #1: The impact of graph workload characteristics. Real-world graphs exhibit particular qualities that incur serious performance degradation if ignored. One example is a *power-law distribution with high skew*, where most vertices are of fairly low degree, but a few vertices have very high edge counts. Even within a single execution, the optimal query plan may then differ depending on which vertex is being processed. Another is a proclivity to produce redundant data, e.g., in the case of label propagation where nodes can often reach each other via many paths. Each of these presents opportunities for optimization.

Opportunity #2: The impact of datacenter architecture. Performance can also depend heavily on the underlying infrastructure. Consider the rack-based architecture of Facebook's most recent datacenter design [15]. Racks of servers are connected through an interconnection network such that a given server's bandwidth to another can *differ by a factor of four* depending on whether the other server is in the same rack or not. Though this type of structure is ubiquitous in today's datacenters due to practical design constraints [15, 245, 119], existing processing systems (e.g., [80, 114, 241]) have largely ignored these effects, typically assuming uniform connectivity that is not the case in modern datacenters.

The GraphRex system. To exploit these two opportunities, this chapter explores a suite of optimization techniques specifically designed to ensure good performance for massive graph queries running in modern datacenters. We have developed GraphRex (Graph Recursive Execution) that significantly outperforms state-of-the-art graph processing systems.

The performance of GraphRex stems, in part, from the high-level language it presents. It compiles Datalog queries into distributed execution plans that can be processed in a massively parallel fashion using distributed semi-naïve evaluation [175]. While prior work has noted that declarative abstractions based on Datalog are natural fits for graph queries [34, 241], these systems fall short on constructing efficient physical plans that (1) scale to large graphs that cannot fit in the memory of one machine, and (2) scale to a large number of machines where the network is a bottleneck. GraphRex goes beyond these systems by combining traditional query processing with *network-layer optimizations*. It aims to achieve the best of both worlds: ease of programming using a declarative interface and high performance on typical datacenter infrastructure. Our key observation is that these two goals exhibit extraordinary synergy.

We note that this synergy comes with a requirement: that the graph processing system be aware of the underlying physical network. In a private cloud datacenter where the operator has full-stack control of the application and infrastructure, visibility is trivial. In a public cloud, the provider would likely expose GraphRex "as a service" in order to abstract away infrastructure concerns from users.

Our contributions. This chapter makes the following contributions in the design and implementation of GraphRex:

(i) Datacenter-centric relational operators for large-scale graph processing. We have developed a collection of optimizations that, taken together, specialize relational operators for datacenter-scale graph processing. The scope and effect of these optimizations is broad, but their overarching goal is to reduce data and data transfer, particularly across "expensive" links in the datacenter. These techniques, applied using knowledge of the underlying datacenter topology and semantics of relational operators in GraphRex's declarative language, allow us to significantly outperform existing graph systems.

(ii) Dynamic join reordering. We also observe that graph queries may require changing join reorderings as join selectivity is heavily influenced by node degrees; and degrees can vary significantly across a graph. Inspired by prior work on pipelined dynamic query reoptimizations [59], we develop a distributed join operator that can dynamically adapt to changing join selectivities as the query execution progresses along different regions of a graph.

(iii) Implementation and evaluation. We have implemented a prototype of GraphRex. Based on evaluations on the CloudLab testbed, we observe that GraphRex has dominant efficiency over existing declarative and low-level systems on a wide range of real-world workloads and micro-benchmarks. GraphRex outperforms BigDatalog by factors of 11–109×, Giraph by factors of 5–26×, and PowerGraph by 3–8×. In addition, GraphRex is more robust to datacenter network practicalities such as cross-traffic and link degradation because our datacenter-centric operators significantly reduce the amount of traffic traversing bottleneck links.

2.2 Background

Today's graph processing processing systems span multiple layers. *Applications* are written in low-level languages like C++ or Java; they run on *frameworks* including GraphX, Giraph; which in turn run in large *datacenter deployments* like those of Google, Amazon, Microsoft, and Facebook. These systems are powerful, efficient, and robust, but difficult to program and tune [44, 241].

2.2.1 Declarative Graph Processing

GraphRex uses Datalog as a declarative abstraction, drawing inspiration from recent work [34, 241]. Datalog is a particularly attractive choice for writing graph queries because of its natural support for *recursion*—a key construct in a wide variety of graph queries [153, 233].

Datalog *rules* have the form *p* :- $q_1, q_2, ..., q_n$, which can be read informally as "q_1 and q_2 ... and q_n implies p." p is the *head* of the rule, and $q_1, q_2, ..., q_n$ is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* over *fields* (variables and constants), or functions (formally, *function symbols*) applied to fields. The rules can refer to each other in a cyclic fashion to express recursion, which is particularly useful for graph processing. We adhere to the convention that names of predicates, function symbols and constants begin with a lower-case letter, while variable names begin with an upper-case letter. We use predicate, table, and relation interchangeably.

```
cc(A,min<A>) :- e(A,_)
cc(A,min<L>) :- cc(B,L), e(B,A)
```

Query 2.1: Connected Components (CC)

Our example above shows a classical graph query that computes connected components in a graph. This query takes a set of edges `e` as inputs, with `e(X,Y)` representing an edge from vertex `X` to vertex `Y`, and computes a `cc` tuple for each vertex, where the first field is the vertex and the second is a label for the vertex. The first rule initializes the label of each vertex with its vertex id. In the second rule, `cc(A,min<L>)` means that the tuples in `cc` are grouped by `A` first, and in each group, the labels `L` are aggregated with `min`. The rule is recursively evaluated so that the smallest label is passed hop by hop until all vertices in the same connected component have the same label. An equivalent program in Spark requires upwards of one hundred lines of code.

Partitioning graph data. Distributed graph processing requires specification of how the graph data and relations are partitioned. *Graph partitioning* maps vertices (or edges) to workers, and is useful when queries have consistent and predictable access patterns over data. In this chapter, we assume a default graph partitioning where vertex id is hashed modulo the number of workers, although our optimizations are not restricted to, and indeed are compatible with, more advanced graph partitioning mechanisms. *Relation partitioning* refers to cases where an attribute of a relation is selected as *partition key* and all of its tuples with the same partition key are put in the same location. For example, in the `CC` query, `cc` has two attributes so it has two potential partitionings: `cc(@A,B)` and `cc(A,@B)`, where `@` denotes the partition key.

2.2.2 Graph Queries in Datacenters

A crucial component for performance is an understanding of the deployment environment, which in the case of today's largest graph applications, refers to a datacenter. Modern datacenter designs, e.g., those of Google [245], Facebook [15], and Microsoft [119], have coalesced around a few common features, depicted in Figure 1.1, which are necessitated by practical considerations such as scalability and cost.

At the core of all modern datacenter designs are *racks* of networked servers [84, 172, 245]. The servers come in many form factors, but server racks typically contain a few dozen standard servers connected to a single *rack switch* that serves as a gateway to the rest of the datacenter network [206]. The datacenter-wide network that connects those rack switches is structured as a multi-rooted tree, as shown in Figure 1.1. The rack switches form the leaves [41, 157].

The above architecture leads to several defining features in modern datacenter networks. One example: *oversubscription*. While recent architectures strive to reduce oversubscription [119, 41], fundamentally, cross-rack links are much longer and therefore more expensive (as much as an order of magnitude) [172, 293]. As such, the tree is often thinned immediately above the rack level, i.e., oversubscribed, and it may be oversubscribed even further higher up. This is in contrast to racks' internal networks, which are well connected.

The result is that servers can often overwhelm their rack switch with too much traffic. A 1:y oversubscription ratio indicates that the datacenter's servers can generate $y\times$ more traffic than the inter-rack network can handle.[1] In essence, these networks are wagering that servers either mostly send to others in the same rack, or rarely send traffic concurrently. In this way, network connectivity is not uniform. Instead, datacenter networks are hierarchical, and job placement within the network affects application performance. Ignoring these issues can lead to poor results (see Figure 2.1).

2.3 GraphRex Query Interface

The goal of GraphRex is to provide a high-level interface with the performance of a system tuned for datacenters. To that end, GraphRex presents a Datalog-like interface and leverages an array of optimizations that reduce data and data transfer. We illustrate our variant of Datalog with several graph queries, most of which involve recursion:

```
vnum(count<A>) :- e(A,B)
```

Query 2.2: Number of Vertices (NV)

```
deg(A,count<B>) :- e(A,B)
pr(A, 1.0) :- deg(A,_)
pr(A,0.15+0.85*sum<PR/DEG>)[10] :- pr(B,PR), deg(B,DEG), e(B,A)
```

Query 2.3: PageRank (PR)

```
sg(A,B) :- e(X,A), e(X,B), A!=B
sg(A,B) :- e(X,A), sg(X,Y), e(Y,B)
```

Query 2.4: Same Generation (SG)

[1]Typical rack-level oversubscription ratios can range from 1:2 to 1:10 [245, 15]. Some public clouds strive for 1:1, but these are in the minority [265]. Regardless, other datacenter practicalities can result in effects similar to oversubscription.

```
tc(A,B) :- e(A,B)
tc(A,B) :- tc(A,C), e(C,B)
```

Query 2.5: Transitive Closure (TC)

Query 2.2 counts the number of vertices in a graph (NV). It takes as input all edge tuples `e(A,B)` and does a count of all unique vertices `A`. Query 2.3 computes page ranks of all vertices in a graph (PR). Query 2.4 returns the set of all vertices that are at the same generation starting from a vertex (SG). Query 2.5 computes standard transitive closure (TC). The Datalog variant we use has similar syntax to traditional Datalog with aggregation, where aggregate constructs are represented as functions with variables in brackets (<>).

One extension we make to Datalog can be seen in PR: a stopping condition denoted as "[..]" in the rule head, for rules that may not converge to a fixpoint using traditional incremental evaluation of aggregates in recursive queries [108, 153, 252, 267]. For example, in PR, instead of stopping the query when no more new tuples are generated, we can impose a bound on the number of iterations, e.g., "[10]".

We also note that some of our queries involve multi-way joins. For example, SG is a "same generation" query that generates all pairs of vertices that are at the same distance from a given vertex. For example, given the root of a tree, SG generates a tuple for each pair of vertices which have the same depth. If the graph has cycles, a vertex can appear in different generations, significantly increasing query complexity. In existing distributed Datalog systems, the syntactic order is the sole determinant for the evaluation strategy of these joins—*they are simply evaluated "from left to right"* [241, 267]. This is because in a distributed environment, there is no global knowledge of relations and no easy way to find the optimal join order. As we will show later, this naive order is suboptimal in many cases, and GraphRex improves on this by dynamically picking the best join order. Note that PR also has a multi-way join, but there is no need of join reordering for this particular case, because the cardinalities of `pr`, `deg` and `e` never change in semi-naive evaluation.

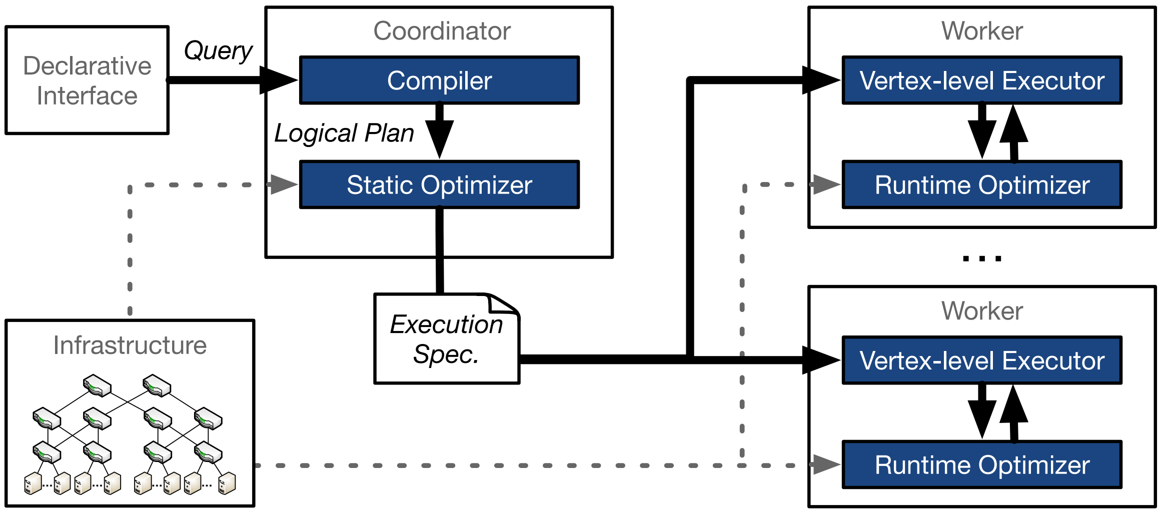


Figure 2.2: The GraphRex architecture. A compiler generates a logical plan from a Datalog query. The static optimizer then constructs from the logical plan a datacenter-centric execution specification that is optimized before the final translation to and evaluation of the physical plan by workers. Gray lines describe dissemination of infrastructure configurations and black lines communication for query execution.

2.4 Query Planning

Figure 2.2 shows the overall architecture of GraphRex, consisting of a centralized coordinator and set of workers. The coordinator first applies a graph partitioning, so that each worker has a portion of the graph. Then during query execution, the coordinator's Query Compiler translates queries into a **logical plan**.

A Static Optimizer then generates an **execution specification** from that logical plan. Execution specifications are similar to physical plans, but include our datacenter-centric **global operators**. The final translation of these operators to concrete physical operators is left until runtime, and depends on both the placement of workers in the datacenter (which is obtained through a configuration file that describes the datacenter infrastructure) and data characteristics. Each worker's physical plan may differ.

Finally, each worker runs the Distributed Semi-Naïve (GR-DSN) algorithm designed for very fine-grained execution, which is a distributed extension of the semi-naïve (SN) algorithm used in Datalog evaluation [175]. In SN evaluation, tuples generated in each iteration are used as input in the next iteration until no new tuples are generated. The distributed variant relaxes

```
1  Function Init($v$):
2      NewTuples$_v$ ← Eval(BaseRules, $\Gamma_v$)                # Evaluate the base Datalog rules
3      AllTuples$_v$ ← NewTuples$_v$                               # Generate initial tuples

4  Function Recur($v$):
5      NewTuples$_v$ ← Eval(RecurRules, $\Gamma_v$, NewTuples$_v$)  # Evaluate the resursive rules
6      NewTuples$_v$ ← NewTuples$_v$ − AllTuples$_v$                # Find new tuples
7      AllTuples$_v$ ← AllTuples$_v$ $\bigcup$ NewTuples$_v$         # Merge new tuples to all tuples

8  Function OnRecv($v$):
9      NewTuples$_v$ ← NewTuples$_v$ $\bigcup$ $v$'s received tuples  # Combine all tuples for $v$
10     NewTuples$_v$ ← NewTuples$_v$ − AllTuples$_v$
11     AllTuples$_v$ ← AllTuples$_v$ $\bigcup$ NewTuples$_v$

12 foreach each vertex $v \in V_i$ do
13     Init($v$)                                  # Initialize all vertices in local partition
14 while no termination signal from the coordinator do
15     foreach vertex $v \in V_i$ do
16         if the size of NewTuples$_v$ $> 0$ then
17             Recur($v$)           # If there is work, perform one iteration of execution
18         if the size of NewTuples$_v$ $= 0$ then
19             Sleep($v$)                         # Otherwise, deactivate the vertex
```

Figure 2.3: Distributed Semi-Naïve in GraphRex

the set operations by allowing for tuple-at-a-time pipelined execution. GR-DSN is designed for graph queries to allow massively parallel execution and tuple-level optimizations.

Specifically, the GR-DSN pseudocode is shown in Figure 2.3. Here, w_i represents a worker that stores the subgraph V_i, and each vertex v maintains its own vertex id id_v and the edge list Γ_v. The GR-DSN algorithm works as follows. Initially, w_i initializes each vertex with `Init` function (line 12-13). Specifically, w_i creates a local table $\texttt{r}_v$ for each vertex v and each relation `r` except edge relation. Recall that the logical plan already ensures that all relations are indexable by vertex. In the `Init` function (line 1-3), base rules are evaluated, which generates the initial tuple set NewTuples in each relation, and the entire tuple set AllTuples is initialized to be the same set. w_i then loops to iteratively evaluate recursive rules. In each iteration, w_i checks if new tuples were generated in last iteration (the Δ tuples in semi-naïve evaluation [175]) at vertex v and uses `Recur` function to evaluate recursive rules

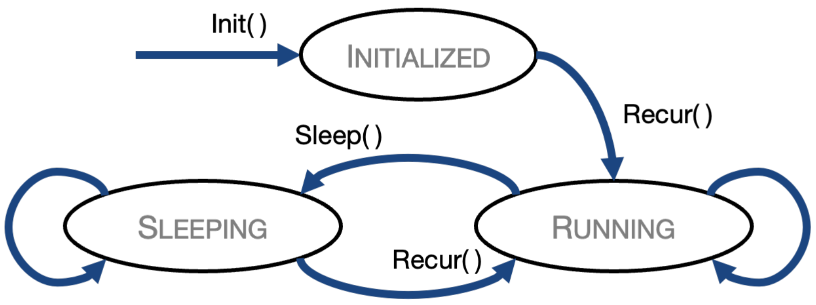


Figure 2.4: Vertex states in GR-DSN.

one time, otherwise calls `Sleep` to deactivate v (line 14-19). Inside `recur`, the recursive rules are evaluated based on Γ_v and NewTuples of last iteration to generate new NewTuples (line 5), and then the deduplication is performed to eliminate redundant evaluation (line 6) and the resulting tuples are merged to the entire tuple set (line 7). In the `Eval` function, the corresponding part of execution plan is evaluated; and the executor consults the dynamic optimizer to execute each global operator efficiently. In particular, A SHUFF operator sends around new tuples according to their partition key. If a vertex v receives tuples, the callback function `OnRecv` is invoked to handle the tuples. Specifically, the received tuples are merged to NewTuples$_v$ and deduplicated, and also added to AllTuples$_v$ (line 8-11).

A vertex in GraphRex could be in one of three states: *initialized*, *running* and *sleeping*. A vertex enters *initialized* after calling `init` to evaluate the base rules, and transitions to *running* on calling `recur`, where the recursive rules are iteratively evaluated in GR-DSN.

A significant difference from traditional, centralized semi-naïve evaluation is that when a vertex has no new tuples, it transitions to *sleeping*; if later, new tuples are received, the vertex will be activated again and transition into *running* again. This design ensures that the distributed evaluation converges globally rather than locally at a vertex level. The recursion reaches a fixpoint when: (1) all vertices in the graph are at the *sleeping* state, and (2) no tuples are being shuffled, i.e., no vertex received new tuples. The coordinator sends termination signal to workers when either the specified number of iterations or the fixpoint is reached.

The above process occurs directly at the workers, which receive the execution specification, generate a local physical plan, and execute it, all with the help of two components: (1)

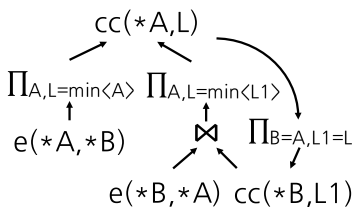


Figure 2.5: The logical plan of CC.

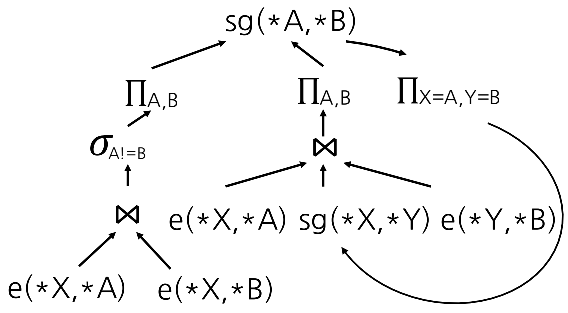


Figure 2.6: The logical plan of SG.

a *Vertex-level Executor* that uses GR-DSN to execute the specification until a fixpoint; and (2) a *Runtime Optimizer* that optimizes each global operator locally.

2.4.1 Logical Plan

From the query, the first step in processing it is to generate a logical plan. In GraphRex, a logical plan is a directed graph, where nodes represent relations or relational operators, and edges represent dataflow. Figures 2.5 and 2.6 show logical plans for `CC` and `SG`, respectively.

An important part of logical plan generation in GraphRex is a *Vertex Identification* phase, in which the compiler traverses the plan graph starting from the edge relations and marks attributes whose types are vertices with a `*` symbol. These attributes are candidates for being the partition key. As an example, in Figure 2.5, since both attributes in the input edge relation `e(A,B)` represent vertices, they are both marked with the `*` symbol. Likewise, all attributes that have a dependency to either vertex attribute `A` or `B` are also marked.

By the time we generate a physical plan, only one partition attribute will be chosen for every relation. Later, we will denote the selected attribute by prepending an `@` symbol. At this stage, we can make the decision for two simple cases. First, if a relation `r` only has one vertex attribute, then it is trivially partitioned by that attribute. Second, the edge table `e` is partitioned on the first key by default so that each vertex maintains the list of outgoing neighbors. This is a convenient placement for many practical graph applications, such as PageRank, SSSP, that only require each vertex to know its outgoing neighbors.

All other partitioning decisions are made during the placement of the SHUFF and ROUT

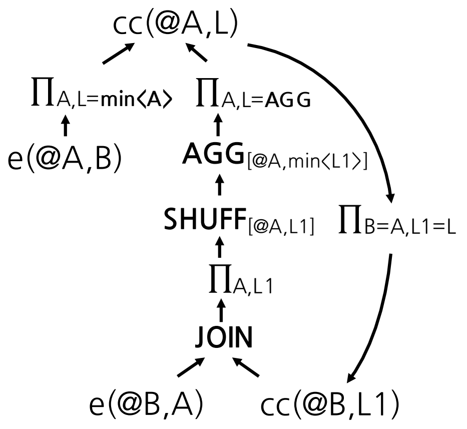


Figure 2.7: The execution specification of CC.

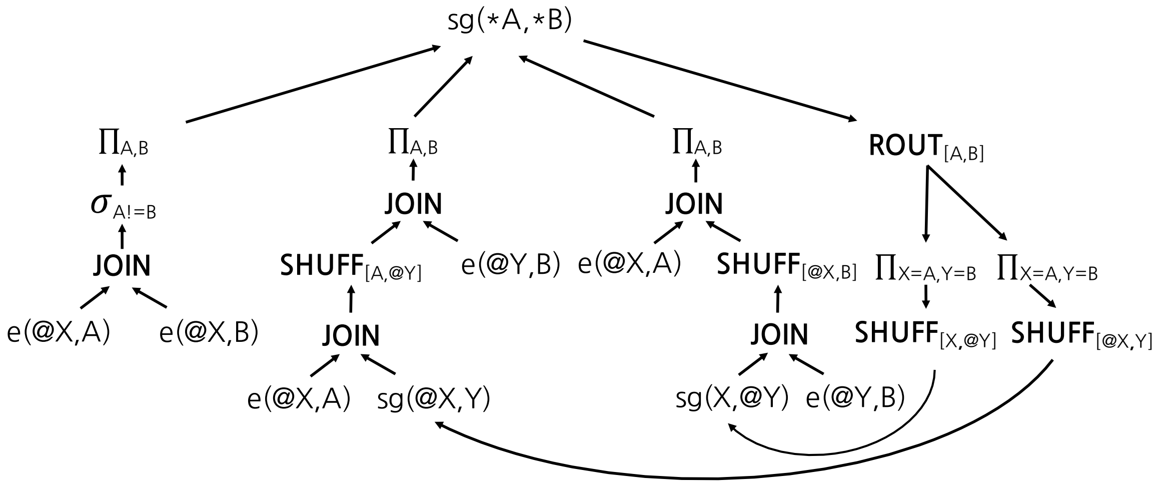


Figure 2.8: The execution specification of SG.

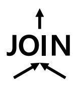
operators described in the following section.

2.4.2 Execution Specification

Traditional query planning proceeds directly from a logical plan to a physical plan. We identify opportunities for datacenter-centric optimization with an additional step. The core of this process is the addition of **global operators** to the logical plan to form what we term an *execution specification*. These operators are special in that they govern communication across workers; oversubscription, capacity constraints, and congestion mean that their efficient execution is a primary bottleneck in processing large graphs. We describe them below.

Join (JOIN)

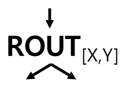
Joins are one such operation that frequently incurs communication in graph processing. In Datalog, (natural) joins are expressed as conjunctive queries. GraphRex evaluates them as binary operations; multi-way joins are executed as a sequence of binary joins. Graphically, we represent these as:



In the case of binary joins, we simply insert a JOIN in lieu of the logical operator $\bowtie$. Recursive joins, where one or more of the inputs are recursive predicates, are handled similarly to BigDatalog [241]. Namely, if the recursion is linear, the non-recursive inputs are loaded into a lookup table and streamed. If the recursion is non-linear, we load all but one of the recursive inputs into a lookup table and stream the remaining input. This enables us to reduce non-linear recursion to linear recursion from the viewpoint of a single new tuple. Figure 2.7 shows an example of a recursive join. Multi-way joins require additional handling, as different join orders can lead to drastically different evaluation costs. In GraphRex, multi-way joins are implemented as a sequence of binary joins, where the order is chosen at *runtime* and *per-tuple*. Existing distributed Datalog systems arbitrarily evaluate 'left-to-right' [241, 267]. We represent this choice in the execution specification by enumerating all possible decompositions of the multi-way join and routing between them dynamically with the next operator.

Routing (ROUT)

The ROUT operator enables the dynamic and tuple-level multi-way join ordering mentioned above. ROUTs take a tuple and direct it to one among multiple potential branches in the execution specification. This operator is only used in conjunction with multi-way joins, and is represented as:



For example, Figure 2.8 shows the specification for `SG` where the multi-way join `e ⋈ sg ⋈ e` in Figure 2.6 is broken into `(e ⋈ sg) ⋈ e` and `e ⋈ (sg ⋈ e)`. We generate plans for the two possible orderings and insert a ROUT operator that takes `A` and `B` as input to decide which will result in better performance.

Aggregation (AGG)

Another important global operator is AGG, which aggregates tuples. There are three types of aggregation in GraphRex, two of which are mapped to global operators. The one type of aggregation that is *not* mapped is purely local aggregation, which operates on tuples with the same partition key, for instance, in the left branch of Figure 2.7 (in the projection). This type of aggregation does not need its own global operator as its evaluation does not incur communication. The other two variants are represented as follows:

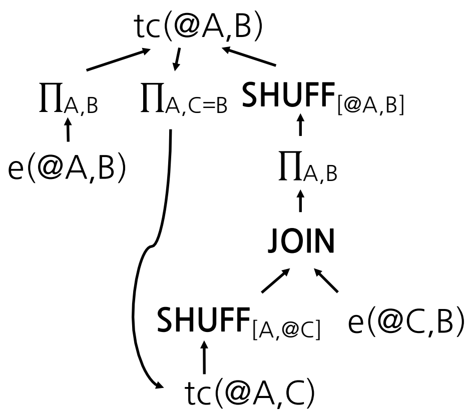
$$\overset{\uparrow}{\underset{\uparrow}{\mathbf{AGG}}}_{[@X,min<L>]} \qquad\qquad \overset{\uparrow}{\underset{\uparrow}{\mathbf{AGG}}}_{[min<L>]}$$

Left to right, (1) also operates at each vertex, but requires shuffling of inputs to compute the relation, and (2) covers global aggregation, where a single value is obtained across the entire graph. For (1), the semantics are similar to a purely local aggregation, but as communication is required, GraphRex will eventually rewrite the specification in order to reduce the data sent across the oversubscribed datacenter interconnect. The right branch of Figure 2.7 demonstrates this case. For (2), aggregation is instead finalized at the coordinator. For example, `NV` computes the number of vertices in the graph using a global aggregator. That value is eventually collected by the coordinator and potentially redistributed to all workers for subsequent use.

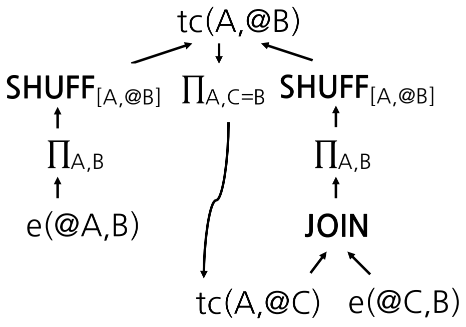
Shuffle (SHUFF)

Last, but arguably most important is the SHUFF operator that encompasses all network communication in GraphRex.

SHUFFs are inserted into the execution specification whenever it is necessary to move

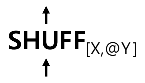


(a) By the first attribute of `tc`.



(b) By the second attribute of `tc`.

Figure 2.9: Two potential partitionings for TC.



tuples from one worker to another between relations. Their placement is therefore closely integrated with the process of relation partitioning, which instantiates the partition attribute (`@`) from the set of partition candidates (`*`) and inserts SHUFF operators where necessary.

Conceptually, there are two scenarios that require a SHUFF. The first is when the tuples of relation `r` are not generated in the location specified by `r`'s partition key. An example of this is shown in Figure 2.7. The JOIN operation generates `cc` tuples that have a distinct partition key (denoted by the `@` sign) from the join key B. This results in the insertion of a SHUFF operator after the join. The second scenario is when the input relations to an operator are not partitioned on the same attribute, such as the inputs to the join operator in Figure 2.9a. In the example, there is a join operator for `tc` and `e` on attribute `C`. If we partition `tc` on its first attribute, as in Figure 2.9a, a SHUFF is needed to repartition the tuples in `tc` on the second attribute so that the join can be evaluated.

In relation partitioning, the optimizer checks every possible partitioning and selects the one that incurs the minimum number of SHUFFs. As a heuristic, we assume that recursive rules are executed many times. To demonstrate this, Figure 2.9a shows the execution specification where `tc` is partitioned by the first key. The number of SHUFFs in the plan is $2K$, as there are two SHUFFs in each recursive rule evaluation. In comparison, the other partitioning

```
1 $v\_attrs \leftarrow$ GetMarkedAttrs($r$)                    # Get the list of marked attributes
2 if size of $v\_attrs = 1$ then
3     PartKey($v\_attrs[0]$)                  # Mark the only attribute as the partition key
4 else
5     $v \leftarrow \arg\min_{v \in v\_atts}$ NumSHUFF($v$)        # Find $v$ that minimizes SHUFF operators
6     PartKey($v$)                                          # Mark $v$ as the partition key
```

Figure 2.10: Static relation partitioning.

of tc shown in Figure 2.9b requires fewer SHUFFs, i.e., $K + 1$; there is a single SHUFF for the non-recursive rule as well as one for each recursion. Our evaluation later shows that the latter plan provides a greater than $2\times$ improvement.

Figure 2.10 shows the relation partitioning algorithm that we adopt in the Static Optimizer. For each relation r, if there is only one attribute being marked as '*', then r is partitioned by that attribute, because that is the only vertex attribute that can maintain the tuples of r; otherwise the static optimizer enumerates every possible partitioning and selects the one with the minimum number of SHUFFs. We assume the heuristic that recursive rules are executed many times. This assumption is reasonable as practical graph queries often run more than one iteration because of the dense connectivity between vertices in real-world graphs.

2.5 Global Operator Optimizations

Translation from the global operator described above depends on both context and the structure of the datacenter network. Refining these operators is important as they can incur significant performance costs in a large-scale datacenter deployment. We note that translation of the execution specification's *classic* logical operators into equivalent physical operators follows standard database plan generation, and we omit those details for brevity.

GraphRex introduces an array of synergistic optimizations (see Table 2.1), some of which can be used in combination, and some of which are intended as complements. Their benefits stem from a variety of reasons, but the overarching principle is to reduce data and data transfer, particularly across "expensive" links in the datacenter. Our results show that these techniques

	Optimization	Description
SHUFF	*Columnization & Compression*	Leverages workload characteristics to reduce data sent across the network on every SHUFF.
	Hierarchical Network Transfer	Further reduces data over 'expensive' links by applying above optimization hierarchically.
JOIN/ ROUT	*Join Deduplication*	To enforce distributed set semantics in JOINs, when a JOIN feeds into a SHUFF, we push deduplication into the SHUFF.
	Adaptive Join Ordering	To account for power-law degrees, we allow ROUT to dynamically order joins at tuple level. Only used when duplicates are rare.
AGG	*Hierarchical Global Aggregation*	Applies our datacenter-centric approach to global aggregation.
	On-path Aggregation	When SHUFF comes before a local AGG, we push the AGG down into the SHUFF to pre-aggregate values to reduce shuffled tuples.

Table 2.1: GraphRex's global operator optimizations and when they apply.

result in orders of magnitude better performance in typical datacenter environments.

2.5.1 Columnization and Compression

One important optimization in GraphRex applies to SHUFF. In SHUFF, tuples to be shuffled are stored in *message buffers*, which are then exchanged between workers. Rather than directly shuffling those buffers between workers, GraphRex (1) first sorts the data, (2) reorganizes (transposes) the tuples into a column-based structure, and (3) compresses the resulting data using the two techniques described below.

Although columnar databases are well-studied [31, 32, 33], their primary benefit in the literature has been in reducing storage requirements. Performance benefits, on the other hand, are traditionally dependent on access patterns [132, 182]. GraphRex instead sends columnar data by default due to its benefits to two techniques—column unrolling and byte-level compression—that are particularly effective on typical graph workloads.

The first technique, column unrolling, is a process where we elide columns of known low cardinality, C, by creating C distinct columnar data stores—one for each unique value. For

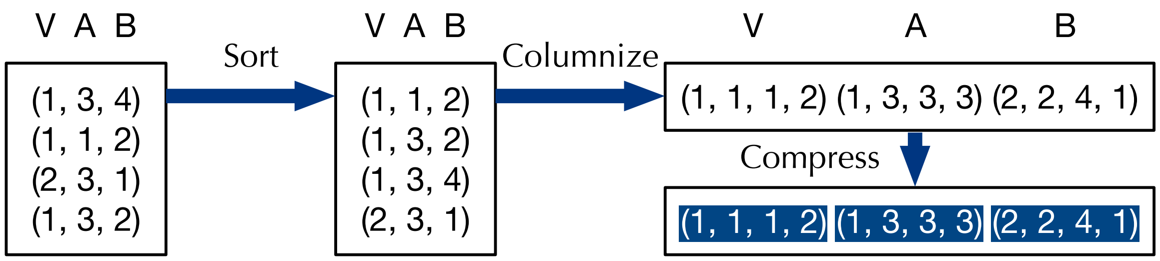


Figure 2.11: Column-based organization for `r(V,A,B)`, where `V` is the partition key. Shaded is compressed data for reducing the number of bytes that we send in the network.

instance, in an adaptively ordered multi-way join, each intermediate tuple must carry an ID that denotes the join order and its place in that ordering of binary joins. In this and many other queries, column unrolling can all but remove the storage requirement of those columns.

The second technique, byte-level compression, compresses sorted and serialized streams using the Lempel-Ziv-class LZ4 lossless and streaming compression algorithm [17]. This process is shown in Figure 2.11. Both sorting and columnization significantly increase the similarity of adjacent data in typical graph applications, resulting in higher compression ratios. More optimal algorithms exist, but LZ4 is among the fastest in terms of both compression and decompression speed. To further reduce the overhead of this optimization, we only sort over the partition key (`V` in the example of Figure 2.11). We also limit compression to large messages, directly sending messages that are under certain size. As typical message sizes are bimodal, any reasonable threshold will provide a similarly effective reduction of overhead (in our infrastructure, a threshold of 128 bytes was robust).

Once the shuffle operation is finished, each worker decompresses, deserializes and transposes the received data to access the tuples. We store the tuples in row form for access and cache efficiency. We also heavily optimize memory copies, buffer reuse, and other aspects of serialization and deserialization, but omit the details for space. Applying columnization and compression together at a worker level brings $\sim 2\times$ overall message reduction for the CC query. However, its effectiveness in typical datacenters can be magnified by the next optimization we propose to SHUFF operator.

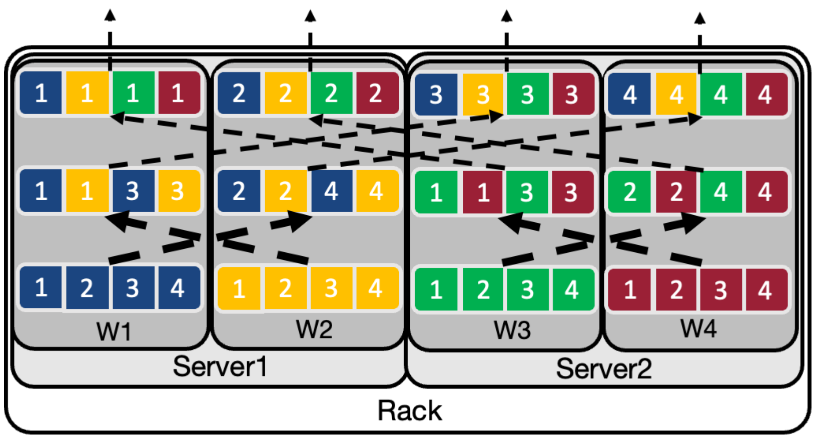


Figure 2.12: An example hierarchical transfer. Each worker groups its tuples by partition key, and sends the them first within a server, then within a rack, and finally to their destinations. A naive system would send directly to other racks. Colors track where the tuple was generated; numbers indicate the partition.

2.5.2 Hierarchical Network Transfer

GraphRex extends the benefits of the previous section by executing Hierarchical Network Transfers as part of SHUFF. This optimization reduces transfers over the network, particularly the oversubscribed portions.

Figure 2.12 depicts this process for a rack with two servers and two workers per server. Specifically, message transfers occur in three steps: *server-level shuffling*, *rack-level shuffling* and the final *global shuffling*. At each level, workers communicate with other workers in the same group, and split their tuples so that each partition key is assigned to a single worker in the group. At each step, tuples are efficiently decompressed, merge sorted, and re-compressed. The benefit of performing this iterative shuffling and compression is that, with every stage, the working sets of workers become increasingly homogeneous and thus more easily compressed.

To show the effect of this optimization, we present results for `CC` on a billion-edge *Twitter* dataset running in a 40-server, 1:5 oversubscription testbed (more results are in the evaluation section). Table 2.2 shows the communication/total speedup of two schemes: simple compression (directly on tuples) and SHUFF (column-based hierarchical compression).

They are compared against a baseline that does not implement compression or infrastructure-aware network transfer. Columnization combined with hierarchical network transfer creates more total traffic, but with less going over oversubscribed links and better load balancing. In this case, server-level shuffling reduces the data by $4.6\times$, and rack-level shuffling reduces the data by $6.2\times$ in our datacenter testbed running 20 workers per server. Together with our optimizations on memory management and (de)serialization, SHUFF achieves a $9.8\times$ speedup in communication time and $7.2\times$ in total execution time.

	Comm	**Total**
Only compression	1.02×	1.02×
SHUFF	9.84×	7.2×

Table 2.2: Speedup of SHUFF and row-based compression in CC on *Twitter*.

2.5.3 Join Deduplication

JOINs are among the most expensive operations in large graph applications. One reason for this is the prevalence of high amounts of duplicate data in real-world distributed graph joins. For example, with `TC` on a social graph, users may have many common friends and thus many potential paths to any other user.

In order to provide set-semantics for joins, previous systems perform a global deduplication on the generated tuples [241]. GraphRex instead introduces Hierarchical Deduplication, which takes advantage of datacenter-specific communication structures to decrease the cost of deduplication when it observes JOIN followed by a SHUFF. Note that when the results of a JOIN are used directly (without an intermediate SHUFF), local deduplication is sufficient.

To illustrate the process of Hierarchical Deduplication, consider again the deployment environment of Figure 2.12, where we have four workers in a single rack. Assume also that all four workers generate the same tuples {(1,2), (2,3), (3,4), (4,5)}, where the first attribute in the relation is the partition key. After the tuples are generated, workers insert them into a hash set that stores all tuples they have seen thus far. This results in the local state shown in the second column of Table 2.3. Workers on the same server then shuffle tuples among themselves, never traversing the network. The same is done at a rack level: servers deduplicate tuples without

Worker	**Worker Level**	**Server Level**	**Rack Level**
$W1$	(1,2),(2,3),(3,4),(4,5)	(1,2),(3,4)	(1,2)
$W2$	(1,2),(2,3),(3,4),(4,5)	(2,3),(4,5)	(2,3)
$W3$	(1,2),(2,3),(3,4),(4,5)	(1,2),(3,4)	(3,4)
$W4$	(1,2),(2,3),(3,4),(4,5)	(2,3),(4,5)	(4,5)

Table 2.3: An example of Hierarchical Deduplication with a single rack of two servers, with two workers per server. At each successive layer of the hierarchy, workers coordinate to deduplicate join results before incurring increasingly expensive communication

ever sending across the oversubscribed interconnect. In the end, of the 16 tuples generated in the rack, only 4 are sent to the other rack—a factor of 4 decrease in inter-rack communication. Queries on real-world graphs, e.g., social networks and web graphs, often exhibit even greater duplication because of dense connectivity: in the execution of TC over *Twitter*, for instance, 98.5% of generated `tc` tuples are duplicates.

	Dup %	**Comm**	**Total**
Baseline	98.5%	39.9 s	41.1 s
Hierarchical Deduplication	42.7%	2.7 s (14.8×)	4.3 s (9.6×)

Table 2.4: Hierarchical Deduplication in TC on *Twitter*. Dup % indicates received duplicates.

Table 2.4 presents the *Twitter*/TC result on the testbed used in the preceding section. We can see that, for workloads with many duplicates, hierarchical deduplication efficiently removes most of them. In comparison, push-down techniques at worker level and server level only reduce the duplication ratio to 96.3% and 90.7% respectively, which shows that deduplication should be performed at greater scale. The high deduplication rate of JOIN results in a 14.8× communication speedup and 9.6× total speedup. Even for workloads with few duplicates, the overhead of this optimization is low.

2.5.4 Adaptive Join Ordering

In the case of multi-way joins, GraphRex sometimes chooses a more aggressive optimization: Adaptive Join Ordering. To that end, the ROUT operator decides, for every tuple, how to order the constituent binary joins of a multi-way join. A key challenge here is predicting the performance effects of choosing one order over another. One reason this can be difficult is due to duplicates; different join orders may result in tuples that are generated on different workers, impacting the occurrence of duplicates in unpredictable ways for the current and future iterations.

For that reason, Adaptive Join Ordering is a complement to Join Deduplication: when the number of duplicates is high, the latter is effective, otherwise the optimization described here is a better choice. We rely on programmers to differentiate between the two when configuring the query. In practice, this is typically straightforward (and akin to the configuration of combiners in Hadoop/Spark), but techniques such as profiling and sampling could be adopted to automate the process in future work.

To illustrate a simple example of how join ordering can result in improved performance, consider the evaluation of `SG` over the graph in Figure 2.13. Starting at the root, vertices a and b are in the same generation, so a tuple (a, b) in `sg` is generated by the first rule. The evaluation of the second rule is decided by how `sg` is partitioned:

- If the relation is partitioned by the first attribute, then the join is evaluated from left to right ((`e` $\bowtie$ `sg`) $\bowtie$ `e`) where (a, b) is sent to a to join with Γ_a (the adjacency list of a) before the intermediate tuples are shuffled to b to finish the join.

- If partitioned by the second key, then the join ordering is from right to left (`e` $\bowtie$ (`sg` $\bowtie$ `e`)) where Γ_b is sent to a to finish the join, less cost than the first order.

For this iteration, the left-to-right ordering used by existing distributed Datalog systems results in a factor of three increase in intermediate tuples compared to right-to-left. The opposite is true for the third generation. Real-world graphs produce many such structural discrepancies due to their power-law distributions of vertex degree. This distribution can result in substantial performance discrepancies between different join orderings, even within a single relation. Thus, static ordering—any static ordering—can result in poor performance.

Optimization target. The goal of ROUT is as follows. Let T be the bag of tuples generated by GR-DSN query evaluation. T consists of tuples generated in every iteration, so we have

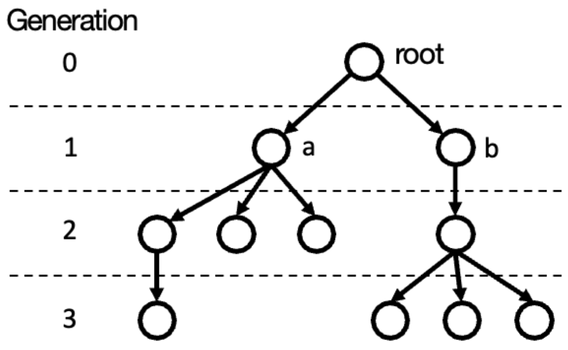


Figure 2.13: SG on an example graph.

$T = \sum_{k=0}^{K} T_k$ where T_k is the bag of tuples generated in iteration k and K is the iteration where a fixpoint is reached. ROUT's optimization objective is:

$$\min |T| = \min \sum_{k=0}^{K} |T_k|$$

Intuitively, more tuples mean increased cost of tuple generation and shuffling. More formally, let T_k^{α} be the bag of intermediate tuples—those that are generated in the intermediate binary joins in order to complete the multi-way join—and T_k^{β} be the bag of output tuples of the head relation (for example, sg in SG), so $T_k = T_k^{\alpha} + T_k^{\beta}$, and we have:

$$\min |T| = \min \sum_{k=0}^{K} (|T_k^{\alpha}| + |T_k^{\beta}|)$$

As mentioned previously, GraphRex makes an assumption that there are no duplicates in generated tuples. Formally, this simplifies optimization in two ways. First, if there are no duplicates, any ordering generates the same T_k^{β} (because of the commutativity and associativity of natural joins) so $|T_k^{\beta}|$ becomes a constant. Second, the ordering of one iteration does not affect another. This independence allows us to optimize each iteration without worrying about later ones. With this assumption, we now have

$$\min |T| = \sum_{k=0}^{K} \min(|T_k^{\alpha}|) + C \tag{2.1}$$

where C is a constant representing the number of output tuples generated in the evaluation.

Ordering joins. With the above, GraphRex picks a tuple-level optimal ordering using a precomputed index. For every newly generated tuple that goes through ROUT, GraphRex enumerates all possible left-deep join orders, computes the cost (i.e., the number of tuples in T_k^α that the order generates) for each order, and selects the order with the minimum cost. Then, GraphRex sets the partition key of this tuple based on the join order, and sends it to the destination for join evaluation. For example, in SG, for every new `sg` tuple (a, b), there are two possible join orders: ((`e` ⋈ `sg`) ⋈ `e`) and (`e` ⋈ (`sg` ⋈ `e`)). The cost for the first order is the degree of a because (a, b) is sent to a first for the first binary join and then Γ_a is sent to b for the second binary join. Similarly, the cost for the second order is the degree of b. The degrees of all vertices are precomputed as an index, and thus efficiently accessible at runtime.

Generality. For n-way joins, the possible options grow to $\binom{n-1}{i-1}$, where i is the position of the recursive predicate, e.g., `e` ⋈ `sg` ⋈ `e` is a 3-way join with `sg` in position 2. Note that the recursive predicate in position 0 or n leads to only 1 ordering. GraphRex scales efficiently by preloading necessary information as indexes whose total size grows as $O(n|V|)$. Regardless, typical values of n are small and there are only a small number of possible orders. We use a 4-way join example: `r(X,Y) :- e(X,A), r(A,B), e(B,C), e(C,Y)`. Given a new `r` tuple, there are three possible left-deep join orders: (1) (((`e` ⋈ `r`) ⋈ `e`) ⋈ `e`), (2) ((`e` ⋈ (`r` ⋈ `e`)) ⋈ `e`), and (3) (`e` ⋈ ((`r` ⋈ `e`) ⋈ `e`)). The costs (in terms of the numbers of intermediate tuples) of the three orders for `r(v1,v2)` are: (1) $C_1 = InDeg(\texttt{v1}) + InDeg(\texttt{v1}) \times OutDeg(\texttt{v2})$, (2) $C_2 = OutDeg(\texttt{v2}) + InDeg(\texttt{v1}) \times OutDeg(\texttt{v2})$, and (3) $C_3 = OutDeg(\texttt{v2}) + Out^2Deg(\texttt{v2})$, where $InDeg(v)$ is v's indegree, $OutDeg(v)$ is v's outdegree and $Out^2Deg(v)$ is v's two-hop outdegree. Therefore, the global information needed by GraphRex for this query is: the indegrees of all vertices, the outdegrees of all vertices and the two-hop outdegrees of all vertices, which is $O(|V|)$ where V is the set of vertices. When GraphRex enumerates the three orders for a tuple, the costs of the orders can be efficiently computed using the preloaded index, and GraphRex selects the order with minimum cost for this tuple. Similarly, the adaptive join ordering can be extended to other values of n for n-way joins.

	1st	**2nd**	**3rd**	**4th**
% of LR	77.47%	80.64%	87.65%	88.16%
% of RL	22.53%	19.36%	12.35%	11.84%

Table 2.5: The percentage of tuples using each join order during the first four iterations of SG on *SynTw*. LR is the left-to-right join order and RL the right-to-left order.

Table 2.5 shows the percentages of tuples in the optimal query plan of the first four iterations of SG on *SynTw*, a synthetic graph of Twitter follower behavior. For most tuples, LR ordering is optimal, but for a non-negligible fraction, it is not. Because of this variability, Table 2.6 shows that, compared to static ordering, Adaptive Join Ordering brings 2.7× and 2× speedup to communication and execution time respectively.

	Comm	**Total**
Static ordering	3.4 s	9.3 s
Adaptive Join Ordering	1.3 s (2.7×)	4.6 s (2×)

Table 2.6: Comparison of adaptive and static ordering.

2.5.5 Hierarchical Global Aggregation

There are three types of aggregations, two of which are translated to global operators. This section describes our optimizations for the global AGG, which is used to compute and disseminate a single global value to all workers via the coordinator. A naive implementation would create a significant bottleneck at the coordinator. A classic alternative is parallel aggregation, in which workers aggregate among themselves in small groups, then aggregate the sub-aggregates, and so on. GraphRex improves this by leveraging knowledge of datacenter network hierarchies.

Figure 2.14 shows an example of this process. First, each worker applies the aggregate function on its vertices and computes a partial aggregated value, then it sends its partial value to a designated aggregation master in the server. When the server master receives partial values from all workers in the same server, it again applies the aggregate function to update its partial value and then it sends the value to the rack master, which updates its own partial

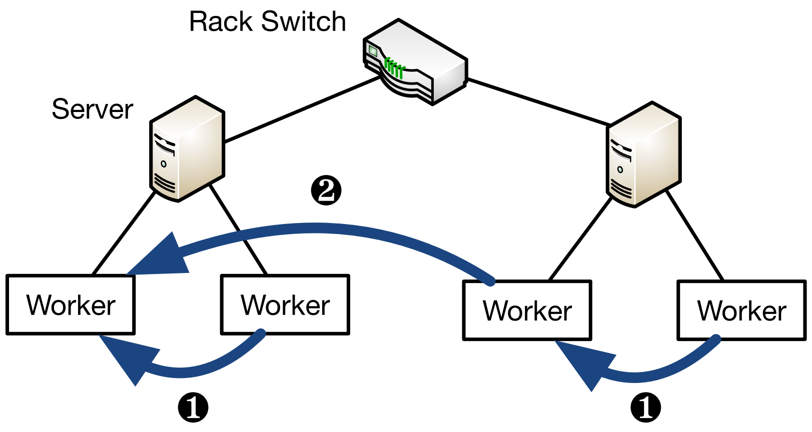


Figure 2.14: Hierarchical Global Aggregation in a rack. After worker-level aggregation, intermediate aggregates are shuffled (1) at a server-level, and then (2) at a rack-level.

value and finally sends that value to the global aggregation coordinator.

As in previous instances, hierarchical transmission significantly reduces traffic over the oversubscribed network. As the computations and communications of Hierarchical Global Aggregation are distributed at each network hierarchy, the overhead to the aggregation coordinator is also reduced. Table 2.7 shows the performance of Hierarchical Global Aggregation in the query of counting vertex number (NV) on *Twitter*. The baseline is infrastructure-agnostic, which means the global aggregation is implemented in an AllReduce manner where all workers send their partial aggregates to the coordinator. Hierarchical Global Aggregation results in 41× speedup in communication and reduces query processing latency from 2.26 s to 0.16 s.

	Comm	**Total**
Baseline	2.154 s	2.26 s
Hierarchical Global Aggregation	0.052 s (41.4×)	0.158 s (14.3×)

Table 2.7: Evaluation of `NV` on *Twitter*.

2.5.6 On-path Aggregation

Finally, the other AGG operator computes a value for each vertex, but requires a SHUFF first. In this case, GraphRex pushes AGG down into SHUFF so that every worker only sends aggregated tuples. The key insight is that tuples that are shuffled to the same vertex can be pre-aggregated. On-path Aggregation again leverages hierarchical shuffling: at each level in the network, it consolidates the tuples for the same vertices to efficiently and incrementally apply aggregation and reduce the number of shuffled tuples.

Table 2.8 shows the performance of On-path Aggregation in CC on *Twitter*, where the baseline is aggregation at the destination, which means that all tuples are shuffled through the network first, and then aggregated. On-path Aggregation brings a 10× speedup in the communication, and the end-to-end query processing latency is reduced by 7.8×.

	Comm	**Total**
Baseline	119.8 s	124.29 s
On-path Aggregation	11.997 s (10×)	15.97 s (7.8×)

Table 2.8: Evaluation of `CC` on *Twitter*.

2.6 Evaluation

In this section, we evaluate the performance of GraphRex with a representative set of real-world graph datasets and queries in order to answer three high-level questions:

- *How competitive is the performance of GraphRex?* We compare GraphRex with BigDatalog [7], which is shown to outperform other distributed declarative graph processing systems (such as Myria [267] and SociaLite [233]), Giraph [3], and PowerGraph [114], two highly-optimized distributed graph processing systems.

- *How robust is GraphRex to datacenter network dynamics?* We emulate typical network events that affect the connectivity between servers, vary network capacity, inject background traffic following typical traffic patterns in datacenters, and test systems under such dynamics.

- *How scalable is GraphRex?* We evaluate how GraphRex scales with additional datacenter resources for large-scale graph processing.

2.6.1 Methodology

Setup. Our CloudLab datacenter testbed consists of two racks with 20 servers per-rack. Each server has two 10-core Intel E5-2660 2.60GHz CPUs, 160 GB of DDR4 memory, and a 10 Gb/s NIC. In aggregate, the testbed has 6.4 TB memory and 1.6 K CPU threads. Mirroring modern datacenter designs [119, 245, 15], our testbed is connected using a 10 Gb/s leaf-spine network [41] with four spine switches by default, resulting in an oversubscription ratio of 1:5.

Queries. We have selected a set of representative queries to evaluate GraphRex. *General Graph Queries* include Connected Components (CC, Q2.1), PageRank (PR, Q2.3), Single Source Transitive Closure (TC, Q2.5), Single Source Shortest Path (SSSP, Q2.6), and Reachability (REACH, Q2.7). Among those queries, CC and PR are compute-intensive and TC, SSSP and REACH are more communication-intensive. We also evaluated local and global *Aggregation* queries (CM, Q2.8) (*sum* and *min* aggregators produced similar results) as well as *Multi-way Join* queries like Same Generation (SG, Q2.4).

```
sssp($ID,0) :- e($ID,_,_)
sssp(A,min<C1+C2>) :- sssp(B,C1), e(B,A,C2)
```

Query 2.6: SSSP (SSSP)

```
reach($ID) :- e($ID,_)
reach(A) :- reach(B), e(B,A)
```

Query 2.7: Reachability (REACH)

```
inout(A,count<B>) :- e(A,$ID), e($ID,B)
maxcount(max<CNT>) :- inout(_,CNT)
```

Query 2.8: CountMax (CM)

Datasets. As shown in Table 2.9, we have selected four real-world graph datasets, all of which contain billions of edges. *Twitter* and *Friendster* are social network graphs, and *UK2007* and *ClueWeb* are web graphs.

System configurations. We compare against the latest versions of in-comparison systems. We configured these systems to achieve the best performance in our datacenter testbed. We pro-

Graph	# Vertices	# Edges	Data Size
Twitter (TW)	52.6 M	2 B	12 GB
Friendster (FR)	65.6 M	3.6 B	31 GB
UK2007 (UK)	105.9 M	3.7 B	33 GB
ClueWeb (CW)	978.4 M	42.6 B	406 GB

Table 2.9: Real-world, large-scale graphs in the evaluation.

visioned them with sufficient cores and memory and optimized other parameters, such as the number of shuffle partitions in BigDatalog, the number of containers in Giraph, and partition strategies in PowerGraph. When possible, we used the query implementations provided by these systems, and implemented the remainder from scratch. Not all systems were able to support all queries easily/efficiently; we omit those as needed. BigDatalog, for instance, has difficulty supporting *PageRank* because it cannot limit the number of iterations. The original paper [241] also omits PR. Similarly, PowerGraph cannot easily support SG, because (1) vertex adjacency lists are not readily accessible, and (2) it forces message consolidation, which would be very inefficient for SG.

2.6.2 System Performance

We first evaluate the performance of GraphRex against state-of-the-art systems in terms of query processing times.

General graph queries. Table 2.10 and 2.11 compare the overall performance of GraphRex, BigDatalog, PowerGraph, and Giraph across different graphs and queries. CC and PR (Table 2.10) require more computation than TC and REACH (Table 2.11). Even in these cases, the oversubscribed network is enough of a bottleneck that GraphRex outperforms other systems by up to an order of magnitude. Against BigDatalog and CC, this order of magnitude improvement is consistent. PowerGraph and Giraph, due to their specialization to graph processing, perform better than BigDatalog, but they are still significantly slower than GraphRex, if they complete (between 3.2× and 17.3×). We note that the largest graph, *CW*, caused out-of-memory issues on both BigDatalog and Giraph; our deduplication and compression

		CC				PR			
		G.R.	B.D.	Giraph	P.G.	G.R.	B.D.	Giraph	P.G.
TW	Time	10.3s	119.8s	49.1s	35.6s	13.4s	-	68.6s	43.2s
	SpdUp		**11.6**×	**4.7**×	**3.4**×		**N/A**	**5.1**×	**3.2**×
FR	Time	15.3s	278.6s	79.3s	60.5s	18.5s	-	148.7s	60s
	SpdUp		**18.2**×	**5.2**×	**4.0**×		**N/A**	**8.1**×	**3.2**×
UK	Time	30.9s	452.8s	274.4s	164.6s	9.6s	-	149.9s	73.6s
	SpdUp		**14.7**×	**8.9**×	**5.3**×		**N/A**	**15.6**×	**7.7**×
CW	Time	472.6s	OOM	8159.5s	1808s	188.7s	-	OOM	668.8s
	SpdUp		**N/A**	**17.3**×	**3.8**×		**N/A**	**N/A**	**3.5**×

Table 2.10: Execution time and speedup for GraphRex (G.R.) compared to BigDatalog (B.D.), Giraph and PowerGraph (P.G.). This table presents results for CC and PR on four graph datasets (TW, FR, UK, CW). OOM indicates an out-of-memory error. B.D. does not support PR.

		TC				REACH			
		G.R.	B.D.	Giraph	P.G.	G.R.	B.D.	Giraph	P.G.
TW	Time	3.1s	336.8s	50.8s	11.8s	2.8s	90s	26.7s	11.5s
	SpdUp		**109.4**×	**16.5**×	**3.8**×		**32**×	**9.5**×	**4.1**×
FR	Time	5.1s	898.5s	81.8s	20.4s	5.2s	236.1s	49.01s	20.7s
	SpdUp		**176**×	**16**×	**4**×		**45.6**×	**9.5**×	**3.99**×
UK	Time	18.5s	866.3s	192.1s	86.1s	17.6s	361.02s	152.6s	87.1s
	SpdUp		**46.9**×	**10.4**×	**4.7**×		**20.5**×	**8.7**×	**4.9**×
CW	Time	207.4s	OOM	5395.2s	978.7s	187.1s	OOM	4909.7s	969.2s
	SpdUp		**N/A**	**26**×	**4.7**×		**N/A**	**26.2**×	**5.2**×

Table 2.11: This table presents results for TC and REACH.

alleviate some issues with working set size.

On more communication-intensive queries, i.e., TC, SSSP and REACH, GraphRex achieves even greater speedups. On these too, BigDatalog failed to complete on the largest graph, *CW*. For TC, GraphRex outperforms BigDatalog and Giraph by up to two orders of magnitude, and PowerGraph by more than 4× on average. Some of this stems from GraphRex's automatic relation partitioning. BigDatalog, by default, partitions by the first key, which happens to be a poor choice in this case. Manually partitioning by the second key leads to 2× better performance, but this is still much slower than GraphRex as it lacks our other optimizations. For SSSP (results in Figure 2.1), GraphRex outperforms BigDatalog by 28–54× on the workloads

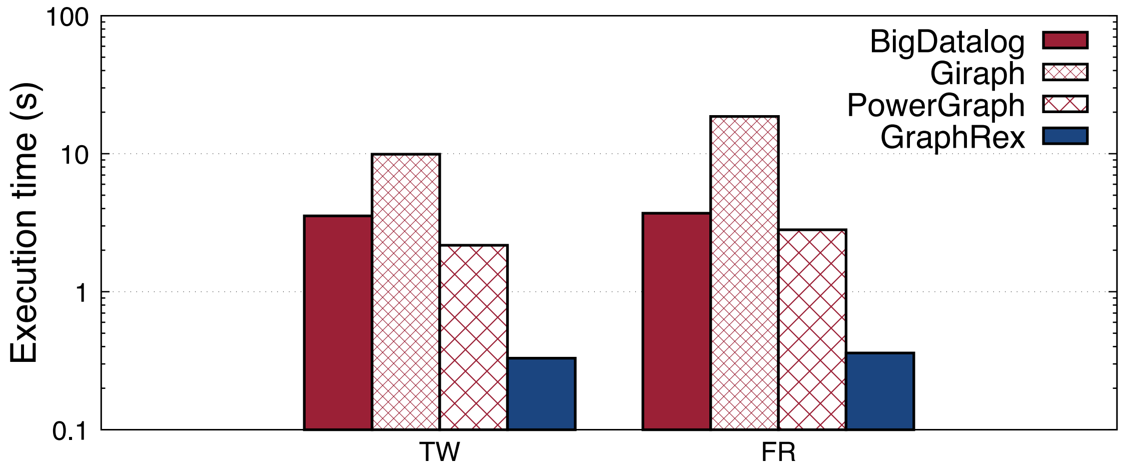


Figure 2.15: Aggregation query evaluation with CM.

BigDatalog could complete, and outforms PowerGraph and Giraph by an average of more than 5× and 10×. Finally, for REACH, GraphRex achieves up to 45.6× higher performance than BigDatalog and up to 26.2× speedup over PowerGraph and Giraph.

Aggregation queries. Figure 2.15 shows the results of an aggregation, `CM`, on *TW* and *FR*. Since we have found similar results on *UK*, and BigDatalog cannot handle *CW*, we have omitted these results. Here, BigDatalog performs better than Giraph, achieving 2.8× and 5× better performance on *TW* and *FR*, respectively, similar to PowerGraph. GraphRex is almost an order of magnitude faster than all of them as a result of our AGG global operator optimizations that avoid the traversal of the oversubscribed network. GraphRex finishes within only one second.

Multi-way join queries. Multi-way joins are challenging even on small social network and web graphs. Consider SG as an example: since such graphs are well-connected, all vertices will eventually be at the same generation. This would result in an output size of $|V|^2$, where $|V|$ is the number of vertices; so a small graph with 1 M vertices would result in 1 T `sg` tuples. Therefore, we have used three alternative datasets to evaluate SG: (1) *BiasedTree*, which amplifies the imbalance in Figure 2.13 by setting the degree of the high-degree vertices to 10K and increasing the depth of the tree to 10, (2) *SynTw*, a synthesized graph simulating follower behavior in Twitter but without cycles, and (3) *Citation*, which is a real-world graph of paper citation relationships that we collected from public sources. While numbers of edges are

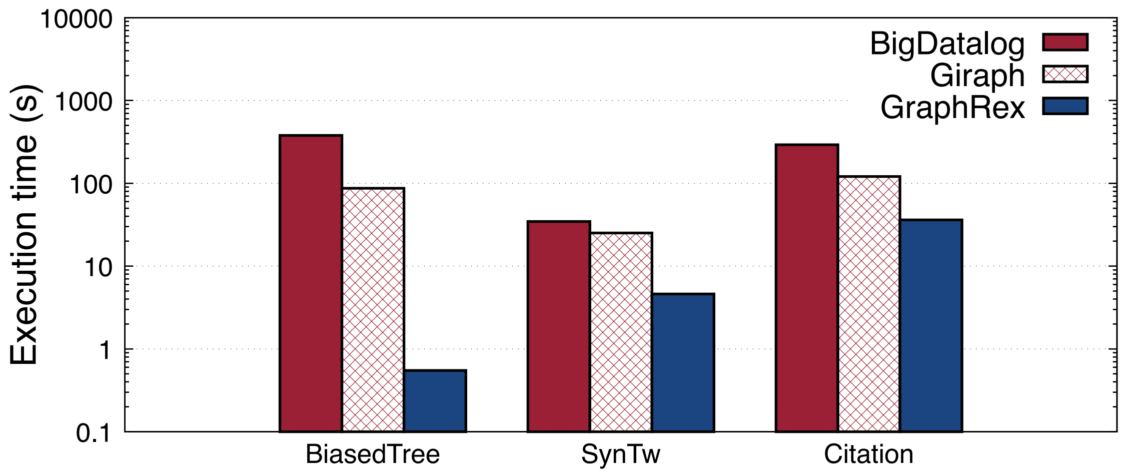


Figure 2.16: Multi-way join query evaluation with SG.

relatively small (0.1 M, 35.7 M, and 20.4 M, respectively), the generated tuple sets are large: 1 B, 70 M, 6 B tuples during the evaluation of SG when using the best static join order.

Figure 2.16 shows our results (PowerGraph is omitted as noted earlier). For fairness, we ensured that Giraph and BigDatalog used the best static join order for the query. Even so, GraphRex significantly outperforms both. Adaptive Join Ordering, by picking the most efficient join ordering for every tuple, reduces the number of generated tuples to 0.2 M, 17 M, and 3 B. The resulting performance improvement is 3.3× in the worst case, with an upper bound of 2–3 orders of magnitude in the extreme case (*BiasedTree*).

Summary: This set of experiments shows that, as a declarative system, GraphRex consistently and significantly outperforms existing systems—both declarative and low-level—particularly on large-scale graph workloads.

2.6.3 Communication Pattern

We now analyze the communication patterns and the benefits of GraphRex's datacenter-centric optimizations. Figure 2.17 shows the communication cost distribution in the datacenter, with three layers: (1) communications inside servers require no network traffic (the left diagonal in the server matrix), (2) communications between servers in the same rack require traffic to be sent intra-rack (the light blue areas), and (3) communications between servers in

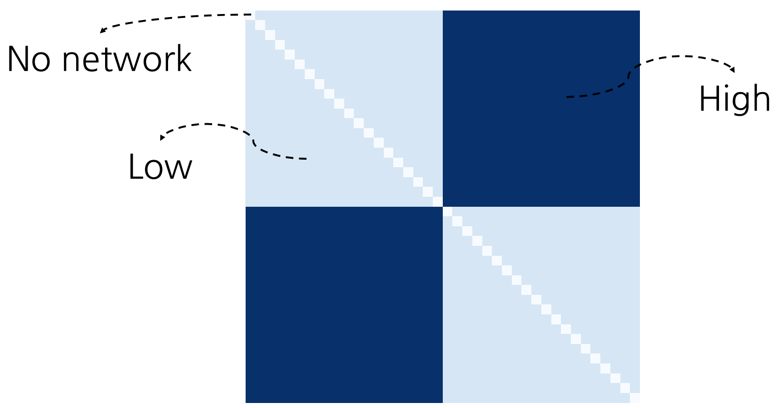


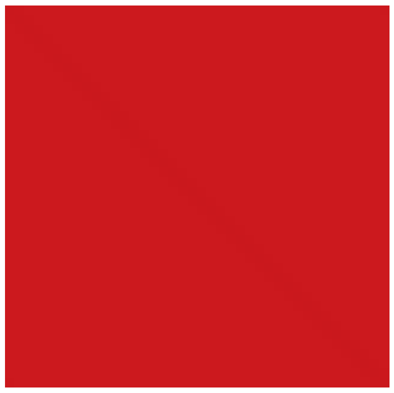
Figure 2.17: Heat map of cross-server communication.

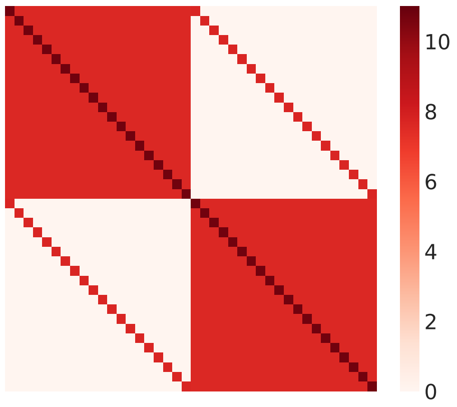
different racks, which incur the highest traffic cost.

Figure 2.18 compares GraphRex against the infrastructure-agnostic baseline in terms of the communication patterns. Although the baseline has server-level locality, i.e., each worker sends more traffic to the workers in the same server than the workers in other servers, it ignores the network structure and treats all other servers as the same. However, the communication pattern in GraphRex results in two benefits.

Reduced traffic: GraphRex prefers low-cost communications to reduce high-cost traffic due to its infrastructure-aware design, minimizing the amount of inter-rack traffic by incurring additional intra-rack communication. As a result, in this example, it reduces the traffic cost by 94.8% compared to the naïve approach.

Fewer connections: In the baseline, every worker directly builds $N-1$ connections with all other workers for shuffling, where N is the number of parallel workers. In GraphRex, each worker establishes $W-1$ connections with other workers in the same server first, where W is the number of workers in the same server; then, at the rack level, it establishes at most $S-1$ connections with other servers in the same rack, where S is the number of servers in the same rack. Finally, it establishes at most $R-1$ connections with other racks, where R is the number of racks in the datacenter. Therefore, the number of connections that each worker builds in GraphRex is $O(W+S+R)$. The naïve approach, assuming that all racks have the same server


(a) The baseline.



(b) GraphRex.

Figure 2.18: Heat maps of traffic volume (number of bytes sent between servers, values are log 10 scale) for CC on *FR*. GraphRex (b) saves 94.8% traffic compared to the infrastructure-agnostic baseline (a).

count and all servers have the same worker count, has $O(W \times S \times R)$.

Summary. The infrastructure-centric design in GraphRex minimizes traffic cost by reducing the traffic sent over bottleneck links and overall network connections.

2.6.4 Robustness to Network Dynamics

We next evaluate the robustness of GraphRex to network dynamics, which are common in datacenter networks.

Network degradation. One such class is link degradations, where the link capabilities can experience a sudden drop due to gray failures, faulty connections, or hardware issues [112, 292]. To emulate this, we randomly select a single rack switch uplink and throttle its capacity to 1/10, 1/50 and 1/100 of its original capacity. Note that a degradation of a single server's access link would decrease performance for all systems equally. We deploy five systems and test their performance with CC on *TW* (results are similar for other graphs and queries): GraphRex, BigDatalog, Giraph, PowerGraph, and 'GR-Baseline', a version of GraphRex with global operator optimizations disabled.

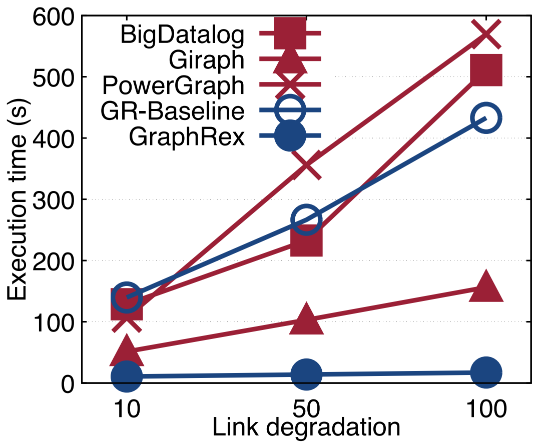


Figure 2.19: System performance when varying link degradations.

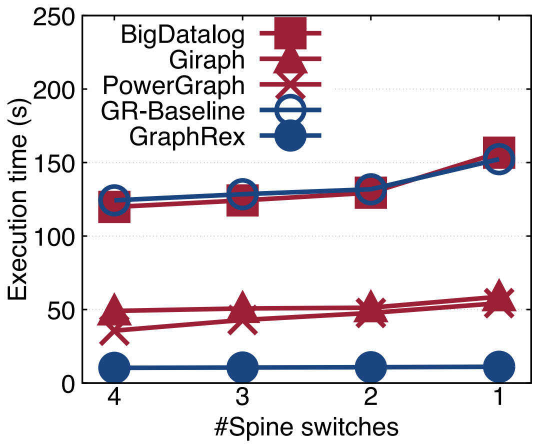


Figure 2.20: System performance with varying #aggregation switches.

Figure 2.19 shows performance under different degrees of link degradation. Because GraphRex minimizes traffic sent through bottleneck links, it is by far the most robust to degradations of those links. In fact, a 1/10 degradation shows almost no effect at all (10.61 s vs. 10.3 s); even in the 1/100 case, GraphRex finishes in 17.24 s. In comparison, GraphRex-baseline experiences significant delay, taking 140 s in the 1/10 case, and 433 s in the 1/100 case. Among all systems, PowerGraph is most sensitive to network dynamics (16× slower than normal for the 1/100 case. Other systems are also severely impacted.

Oversubscription variation. We next evaluate the effect of over-subscription. We emulate this by adding/removing spine switches from the testbed. Less spine switches means less inter-rack capacity and greater over-subscription. Due to hardware constraints, we only vary the number of switches in the spine layer from 4 to 1.

Figure 2.20 shows results for CC on *TW*. The over-subscription significantly degrades the performance of other systems: PowerGraph performance drops 52% (36 s to 54 s) between 4 and 1 spine switches, BigDatalog drops 31% (120 s to 157 s), Giraph 20% (49 s to 59 s), and GR-Baseline 23% (124 s to 152 s). For reasons similar to the prior section, GraphRex's performance only changes 7% (10.3s to 11.1s) over the same range.

Background traffic. Finally, since datacenters typically host multiple applications, applications often experience unpredictable "noise" in the network in the form of background traffic.

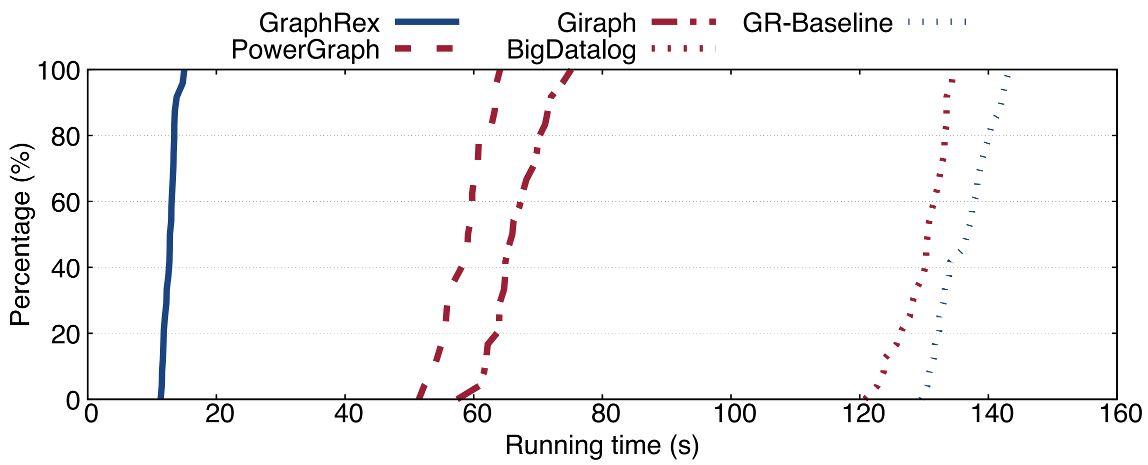


Figure 2.21: The CDF of performance with random background traffic.

To evaluate GraphRex and the other systems in its presence, we inject background traffic using a commonly used datacenter traffic pattern [45, 46, 141]. Following the existing methodology, we generate traffic flows from random sender/receiver pairs, with flow sizes and flow arrival times governed by the real-world datacenter workloads [46]. Overall, we generated five representative traffic traces, each with an average network utilization of 40%. We ran CC on *TW* in each system with background traffic, and note that other query workloads have similar findings. As Figure 2.21 shows, the performance variation is significant for other systems, with standard deviations (σ) of 3.6 (P.G.), 4.3 (Giraph), 3.9 (B.D.) and 4.2 (GR-Baseline). GraphRex, on the other hand, achieves $\sigma = 0.96$, which is much more robust, and its performance is significantly better than other systems, with average speedups of 4.6× (over P.G.), 5.2× (over Giraph), 10.1× (over B.D.), and 10.6× (over the baseline).

Summary: The datacenter-centric design in GraphRex increases robustness to network dynamics, even in harsh network conditions with significant link degradation, over-subscription, and random background noise.

2.6.5 Scalability Analysis

Finally, we evaluate scalability compared to other systems. We examine how adding servers to the job affects performance. Specifically, we vary the number of servers per rack in our two-

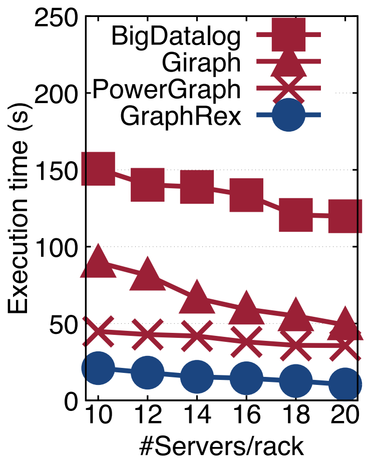


Figure 2.22: Scalability with #Servers on *TW*.

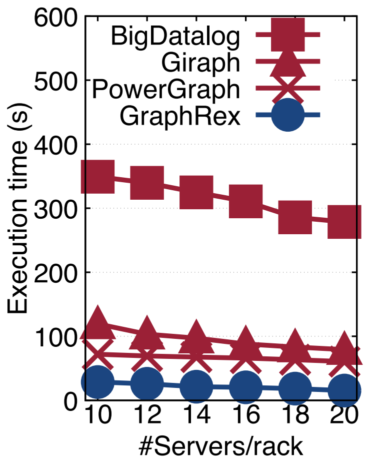


Figure 2.23: Scalability with #Servers on *FR*.

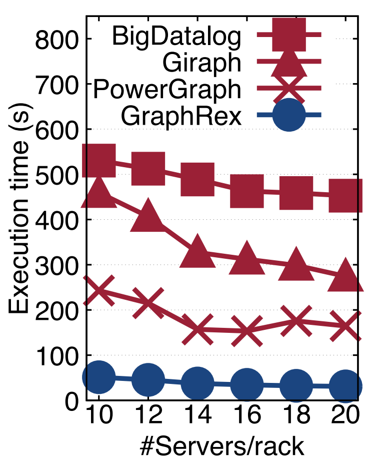


Figure 2.24: Scalability with #Servers on *UK* .

rack testbed from 10 to 20 with a step of 2. Figure 2.22 shows the result of running CC on *TW*. For all systems, the running times decrease when more servers are added. However, more servers per rack also leads to higher oversubscription, which poses scalability bottlenecks. As a result, BigDatalog and PowerGraph only achieve around 1.3× speedup when we double the number of servers; Giraph achieves a 1.8× speedup, yet it still has lower performance than PowerGraph. In contrast, GraphRex, in our representative datacenter configuration, scales almost linearly: 2× speedup when server count doubles.

Figures 2.23 and 2.24 present the performance of different systems for CC on *FR* and *UK*, respectively, when the number of servers in the cluster changes. GraphRex achieves the highest speedup when the number of servers in the datacenter doubles: 1.8× on *FR* and 1.7× on *UK*. Although PowerGraph always achieves the best performance among other systems, it does not scale well. Especially, on *UK*, its performance stops getting improved when the number of servers in each rack is higher than 16. Adding more machines improves the performance of Giraph and BigDatalog, but their scalability is not as good as GraphRex which minimizes the impact from network constraints, and scales better with more resources.

2.7 Related Work

The first part of this dissertation focuses on large-scale graph systems. Many graph processing systems have been proposed [286], including Pregel [183], Giraph [80], GraphX [115], PowerGraph [114], GPS [228], Pregelix [70], GraphChi [152], and Chaos [224]. GraphRex adopts a Datalog-like interface and computation model in order to explore the space of optimizations for large graph queries running on modern datacenter infrastructure.

Declarative data analytics. SociaLite [153] and Emptyheaded [34] are Datalog systems optimized for a single-machine setting. RecStep [101] and DCDatalog [272] focus on improving the performance of Datalog execution on multi-core machines. Hive [4] and SparkSQL [56] are distributed, but only accept SQL queries without recursion. BigDatalog [241] and Datalography [198] explore an intermediate design point (Datalog compiled to SparkSQL and GiraphUC); however, they ignore infrastructure-level optimizations and can be worse than the systems they are built on. GraphRex instead leverages Datalog for graph-specific and datacenter-centric optimizations, and outperforms existing systems significantly.

Adaptive query processing. The idea of adapting the optimal join plan at runtime according to the shape of the data in GraphRex is closely related to the literature of adaptive query processing [94]. Ripple joins [125] generalize nest-loop and hash joins to optimize online aggregation. It interleaves the inner and outer roles of tables to adapt the join orders for arriving tuples based on data properties. Eddies [59] route input tuples between operators at runtime to eliminate the need of an offline query optimizer. They can adapt the query plan with commutative operators to determine the optimal execution order in accordance with the continuous and dynamic workload. State Modules [222] store tuples for base tables and support insert and probe operations that facilitate adaptive query processing with, for example, eddies. RouLette [247] is a recent system that exploits reinforcement learning for runtime adaptation at sharing work across multiple queries. QOOP [179] re-plans a distributed query during its execution based on the resource availability in the cluster. Adaptive join ordering in GraphRex is distinct in its application on real-world graphs that are often skewed. Since our focus is on static graphs, the statistics that we need in order to make runtime decisions can be efficiently pre-computed.

Shuffle optimizations. Optimizing shuffle efficiency for improving overall system performance in large-scale data analytics has been gaining popularity recently. Camdoop [90] proposes to use direct-connect topologies and in-network aggregation for reducing network traffic for data-intensive applications, while others optimize shuffling for different scenarios, including NUMA [74, 162] and serverless computing [199, 215, 220].

Network-level optimizations. Several existing proposals [39, 81, 90, 127, 133] have explored the network-level optimization of groups of related network traffic flows. [66] also discusses the importance of modeling the network in parallel data processing. GraphRex is distinguished by its deep level of integration with the Datalog execution model and its optimizations for graph workloads.

Graph compression and deduplication. Recent work has used data compression on graphs. Blandford et al. [68, 67] propose techniques to compactly represent graphs. Ligra+ [243] further parallelizes these techniques. GBASE [139] and SLASHBURN [168] perform compression for MapReduce to reduce storage. GraphRex is mostly related to C-Store [31], a column-oriented database, and we have further proposed novel techniques like the compressed transpose data structure.

Prior work has also explored deduplication, e.g., via MapReduce combiners [92, 278] and mechanisms for distributed set semantics [82, 241]. Our system pursues the same goals, but our key contribution is to adapt these techniques to create datacenter-centric optimizations for relational operators.

2.8 Summary

This chapter proposes GraphRex, a framework that supports declarative graph queries by translating them to low-level datacenter-centric implementations that are optimized for current networks. At its core, GraphRex identifies a set of global operators (SHUFF, JOIN/ROUT, and AGG) that account for a significant portion of typical graph queries, and then heavily optimizes them based on the underlying datacenter, using techniques such as hierarchical deduplication, aggregation, data compression, and dynamic join orders. With a comprehensive evaluation, we demonstrate that GraphRex works efficiently over large graphs and outperforms state-of-the-art systems by orders of magnitude. Generalizing our techniques to not rely on graph-specific properties (e.g., the ability to preload join cardinalities for Adaptive Join Ordering) is left to future work.

CHAPTER 3

A LOOK AT THE FUTURE—UNDERSTANDING DATA PROCESSING IN DDCS

One recent trend of cloud data center design is *resource disaggregation*. Instead of having server units with "converged" compute, memory, and storage resources, a disaggregated data center (DDC) has pools of resources of each type connected via a network. While the systems community has been investigating the research challenges of DDCs by designing new OS and network stacks, the implications of DDCs for next-generation data processing systems like database management systems (DBMSs) remain unclear.

In this chapter, we take a first step towards understanding how DDCs might affect the design of DBMSs. DBMSs are an interesting case study for DDCs for two main reasons: (1) DBMSs normally process data-intensive workloads and require data movement between different resource components; and (2) disaggregation drastically changes the assumption that DBMSs can rely on their own internal resource management.

We first discuss the potential advantages and drawbacks of DDCs in the context of data processing, focusing on query execution performance. With a set of preliminary microbenchmarks, we show that DBMSs can experience significant performance degradation in DDCs caused by frequent remote memory accesses.

To thoroughly investigate the query execution performance of production DBMSs in disaggregated data centers, we evaluate two popular open-source production-grade DBMSs (MonetDB and PostgreSQL) and test their performance with the TPC-H benchmark in a recently released operating system for resource disaggregation. We evaluate these DBMSs with various configurations and compare their performance with that of single-machine Linux with the same hardware resources. Our results confirm that significant performance degradation does occur, but, perhaps surprisingly, we also find settings in which the degradation is minor or where DDCs actually improve performance.

Finally, we outline the research proposals for addressing the drawbacks of DDCs on supporting data-intensive systems, which call for TELEPORT (Chapter 4) and Redy (Chapter 5).

3.1 Introduction

Over the past few decades, we have witnessed a number of hardware inflection points that required rethinking the design of databases. An early example was the transition of relational database systems (DBMSs) from mainframes to networks of workstations [52, 95]. Since then, we have seen the rise of multicore machines, GPUs and FPGAs that augment existing compute resources, and recent interest in non-volatile memory.

In each of the these cases, the hardware enabled DBMSs to improve their performance, scalability, and/or reliability. We believe that we are approaching a new inflection point. One that is fundamentally different from past ones because the change in hardware is likely to *harm*—rather than improve—performance for DBMSs. This is the case with the *disaggregation* of cloud data center resources [58, 140, 143, 151, 163, 164, 235].

In a fully (resource) disaggregated data center (DDC), servers are no longer built as standalone machines equipped with sufficient compute, memory, and storage to process a single job. Instead, each resource node in a DDC is kept physically separate, with some nodes specialized for processing, others for memory, and others for storage. To complete a single task, a compute node will need to continually "page" memory from remote nodes into and out of its small on-board working set, write chunks to remote disks, or farm out tasks to remote CPUs or GPUs.

Disaggregating resources in this way provides substantial benefits to data center operators. It allows them to upgrade and expand each resource independently, e.g., if a new processor technology becomes available or if the workload changes require additional CPUs. It also allows them to prevent fragmentation and over-provisioning, e.g., if a customer requests an unusual balance between CPU cores, RAM, and GPUs that does not fit neatly into an existing machine. Finally, to users, disaggregation creates the illusion of a near-infinite pool of any resource for any program.

Disaggregation has fundamental implications on the performance of data-intensive applications, not all of which are positive. For example, recent work [53] and the preliminary study that we present with hash-based microbenchmarks in this chapter highlight the potential performance degradation that stems from moving storage and most of memory to a remote machine. These results, however, are not enough for understanding the effect of DDCs on real systems because they consider only synthetic workloads and simple applications. In contrast, DBMSs have complex software stacks; having a thorough understanding of their end-to-end performance in a DDC is therefore critical for the design, implementation, and optimization of DBMSs in future cloud architectures.

Further, to DDCs, database systems provide an interesting case study of the effects of disaggregation. At a basic level, query executions in DBMSs are typically data-intensive, involving frequent and repeated movement of large quantities of data between disk and memory (loading data from storage to main memory and spilling intermediate data to disk when memory is limited), memory and CPU (moving data between compute units and working sets in main memory), CPU and CPU (data shuffling between workers). In a DDC, each of these steps requires network communication, which can impact query performance. Even so, queries, each having unique access patterns, exhibit a great deal of diversity in their reaction to disaggregation. The case study also presents an opportunity to examine modern DBMS design in a different light. Specifically, decades of optimization and tuning on top of traditional servers and operating systems have resulted in a series of baked-in assumptions about memory access latency, buffer management, and paging strategies. Disaggregation exposes many of these fundamental assumptions.

Similarly, to DBMSs, disaggregation presents a unique set of challenges even when compared to the extensive literature on production DBMS performance in new architectures, e.g., disaggregated storage [43, 48, 194] and remote memory [62, 96, 160]. First, unlike traditional remote memory systems where the remote memory is treated as extra cache, disaggregation is typically accompanied by a corresponding decrease in local memory—remote access becomes a necessity rather than an optimization. Second and related, in DDCs, these accesses are mediated by the operating system and network infrastructure rather than controlled by the application. This means that the interactions between each layer of the stack are critical to the system's overall performance.

In this chapter, we first lay out a series of challenges to database design that are unique to DDC architectures, and present some preliminary benchmark results that urge the redesign of DBMSs in DDCs: naïve query execution on DDCs results in order-of-magnitude worse performance compared to running the same query on one of today's monolithic servers!

We next present the first characterization and analysis of modern production database systems running on a DDC. Enabling our study is a combination of recent hardware, network, and operating system advances that, for the first time, provide a complete disaggregated operating environment. This environment allows us to investigate the interactions between each layer in detail.

More specifically, we evaluate queries from the TPC-H benchmark in MonetDB [19] and PostgreSQL [218] in a variety of disaggregation settings. We find that PostgreSQL is less sensitive to disaggregation than MonetDB, but PostgreSQL is also incapable of adapting to varying levels of local memory since it delegates disk caching to the underlying OS (i.e., PostgreSQL achieves similar performance when the compute nodes' cache is very large and when it is small). We also observe that without modifications to either MonetDB or PostgreSQL, DDCs can enable these production databases to scale up and achieve *high* and *stable* performance. This is in contrast to traditional architectures that spill to disk and introduce significant performance variability. While RDMA-based DBMSs [62, 160] may achieve similar benefits, it comes at the cost of an extensive redesign of these DBMSs.

In summary, this chapter makes the following contributions.

- We provide an introduction to disaggregated architectures, and explore how existing DBMSs might be deployed in a DDC. We consider both single and parallel DBMSs.

- DBMSs have high data transfer bandwidth requirements from storage to memory, and from memory to compute. Using a case study of a single and a parallel join operator, we demonstrate how a naive implementation can result in multiple redundant round-trips of memory to memory copies. We validate the actual performance degradation by running query execution on LegoOS [235].

- For a more thorough evaluation, we use the complete TPC-H benchmark to validate that DDC degrades the performance of DBMSs due to expensive remote memory accesses (data movement between compute and memory components). We also identify several scenarios where DDCs can be a better alternative for DBMSs.

- We analyze the bottlenecks of executing DBMSs on DDCs and shed light on different ways to optimize the execution of future DBMSs in this new architecture.

- Based on our findings, we propose new hardware and OS primitives that DBMSs can use to perform memory copies more efficiently. These primitives are inspired by decades of work in *near data processing* [120, 207, 214, 273] where the memory has some small computational ability that can be leveraged for significant gains. For example, we propose new mechanisms that bypass the compute nodes entirely when partitioning data in preparation for a hash join. We show that these primitives (in conjunction with modifications to the database execution engine) apply to different parallel relational operators and can reduce the overheads introduced by DDCs.

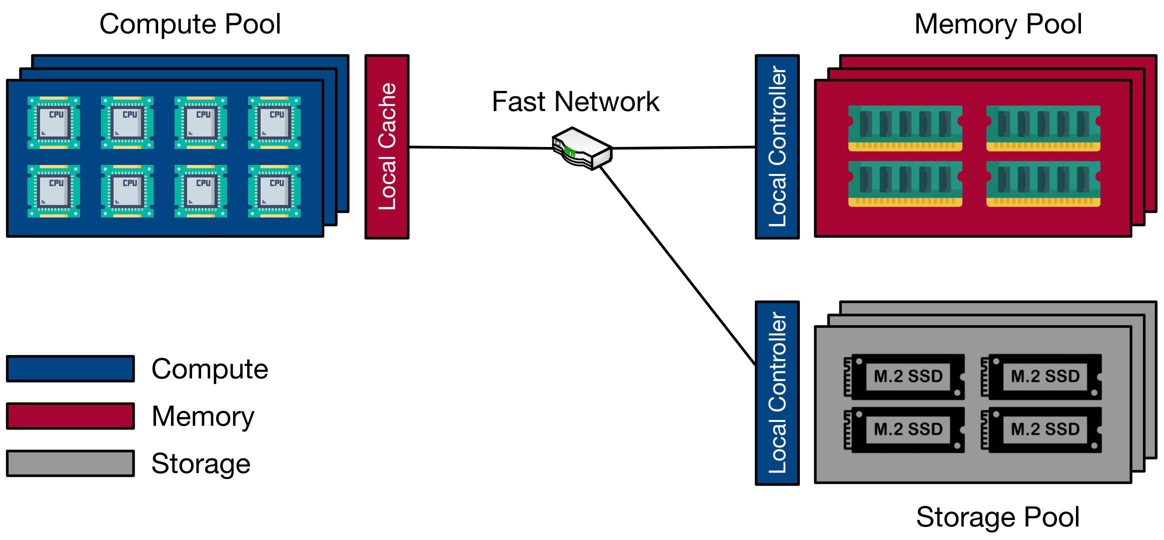


Figure 3.1: An illustration of resource disaggregation. Same type of resources are centralized in a *resource pool*. Resource pools are disaggregated and connected by a fast network.

3.2 Background

This section introduces the key architectural elements of recent DDC proposals, with a focus on their effect on DBMS operation.

3.2.1 Disaggregated Data Centers

Resource disaggregation is an architectural style in which the resources of a data center, traditionally spread across every server, are instead partitioned into physically distinct pools of resources connected with a fast network fabric such as RDMA over InfiniBand, as illustrated in Figure 3.1 (slight variants exist). While today's data centers already disaggregate storage, a defining feature of disaggregated data centers (DDCs) is the more complete disaggregation of resources including of memory. There are several core components in this architecture: individual resource pools with *compute elements*, *memory elements*, and *storage pools* connected over a low-latency *resource interconnect*. While pools hosting each type of resource may also contain a small amount of other resources (e.g., low-frequency CPUs in the memory/storage pools that manage local resources and process accesses, or a modest amount of DRAM in the compute pool that caches data), the expectation is that any computation of sufficient size will require coordination across pools spanning different resource types. We describe each of these components in more detail below.

Compute, memory, and storage pools. A compute pool consists of commodity processors with their associated memory hierarchy (including private and shared caches) and a small amount of local memory. This local memory is used primarily for the OS and as another cache to improve performance [109, 235].

A memory pool is composed of a dense array of DRAM or NVRAM chips, which are typically accompanied by a small computing element (processor, RNIC, FPGA, ASIC, etc.) that proxies communication with the compute pool and converts network requests into reads and writes operated on local memory nodes. This processor interacts with each memory node through a standard MMU, and is responsible for addressing and access control. Storage pool is structured similarly.

Resource interconnect. There are a few proposals for implementing the network that connects different resource pools:

- **Packet switching.** The compute pool interacts with the memory and storage pools by sending packets over a network of switches. The primary benefit of this approach is that all of its components—the Ethernet switches, RNICs, and Ethernet links—are readily available and commoditized. Compared to the other proposals, packet switches also typically have lower latency for small individual memory requests and higher utilization.

- **Circuit switching.** Researchers have argued that for the large port counts and throughput requirements of rack-scale disaggregation, packet switches will eventually become too demanding for typical rack-level power budgets. These physical requirements have led to the exploration of simpler circuit switches, which transmit optical or electrical signals at the physical layer rather than parsing, processing, and buffering packets. Scheduling, setting up, and tearing down circuits impose a performance cost to circuit switching, but systems like Shoal [242] propose potential solutions to compensate.

- **Direct connect.** Finally, the compute pool can be connected directly to memory and storage pools (e.g., using a 3D Torus network [89]), eliminating switches entirely. Direct connect topologies are cheap, and in some cases, also efficient and low-latency. Unfortunately, these properties depend on the provided workload as some messages may need to traverse multiple other nodes before reaching their destination.

Regardless of how the interconnect is instantiated, we assume that any node in the compute pool can access any memory node, and that accesses can be reconfigured at runtime at fine granularity.

Benefits of resource disaggregation. As mentioned in prior work [53, 75, 89, 109, 235, 242, 259], DDCs bring significant operational benefits over traditional architectures. These benefits include:

- **Independent expansion.** The hardware resources can be expanded and upgraded independently. For example, if a DDC is running low on memory, the operator can just hot-plug more memory in the memory pool. This is more flexible and cost-efficient than traditional data centers where an operator would need to add large servers with additional resources that are unnecessary.

- **Independent failures.** Since resources are decoupled, the failure of one resource does not signify the failure of all others. For example, it is possible for a memory node to fail, while the associated CPU remains alive. Prior work suggests ways to recover from these types of failures in DDCs [54].

- **Independent allocation.** For cloud operators, resource allocation becomes a simpler task: packing virtual machines to DDCs simply requires identifying the appropriate resource pools and creating the appropriate forwarding rules in the network fabric. In comparison, packing VMs to monolithic servers while maximizing utilization and minimizing resource fragmentation is an NP-hard bin-packing problem.

In exchange for those benefits, DDCs convert a subset of what used to be local memory and device accesses to remote accesses. While the latest InfiniBand networks are undoubtedly very fast (sub-600 ns latency at 200 Gb/s [28]) and some proposals have advocated for new network substrates [242], both are, nevertheless, much slower than accessing resources on the same motherboard.

Disaggregated operating systems. A critical piece of the above architecture is the disaggregated operating system. Fundamentally, the migration of memory away from compute means that, while the compute pool may have a nominal amount of memory to store a kernel, it may not have enough for the code segment, data segment, heap, and/or stack. In the same way, the memory pool may have enough compute to perform address translation and basic access control, but will not have enough to execute queries. Thus, the operation of a DDC will likely need to be mediated through a specialized operating system.

A state-of-the-art disaggregated OS is LegoOS [235], which takes a splitkernel approach to dividing kernel responsibilities over resource-disaggregated nodes. In LegoOS, the local kernel on a computation node, where DBMS instructions are expected to run, is in charge of configuring and negotiating access to external resources, and of managing a small amount of local memory that is attached to the CPU. This memory hosts the local kernel and serves as a cache for applications. LegoOS supports the Linux system call interface as well as an unmodified Linux ABI, allowing users—in principle—to run unmodified Linux applications.

In this work, we take advantage of LegoOS's interface. Unfortunately, while LegoOS is a working research prototype that highlights the complexities of building a distributed operating system that coordinates and manages disaggregated resources, it is not sufficiently complete to run a real production DBMS. One of the contributions of our work is, therefore, to extend LegoOS's codebase with several system calls and additional functionality that is needed to run these DBMSs. We discuss these efforts in Section 3.4.

3.3 Overview of Executing DBMSs in DDCs

How do modern DBMSs fare in a disaggregated environment? In this section, we first discuss the operation of these systems on DDCs, and then we run microbenchmarks with hash oper-

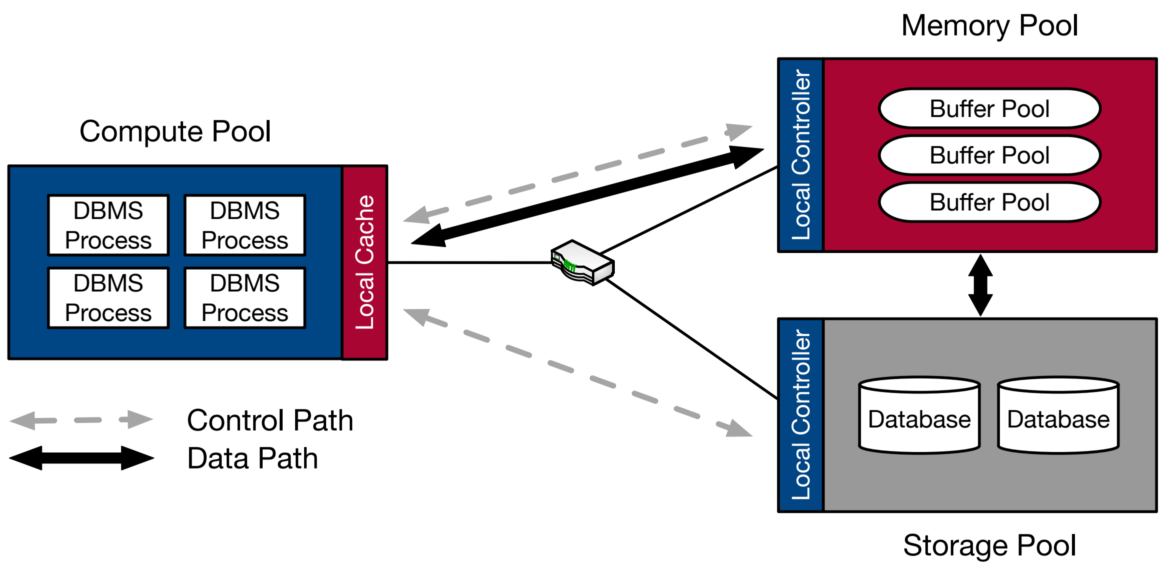


Figure 3.2: DBMS execution in DDCs. DBMS workers are spawned on compute nodes with their small local memory acting as a cache. Buffer pools live in a remote memory pool; a storage pool stores and manages the database files. Workers send control messages to allocate and manage resources, and the data is transferred between memory and storage pool (loading and spilling) and the processing and memory pool (fetching and eviction).

ators to have a preliminary view on the performance implications of DDCs, before measuring and analyzing production DBMSs more thoroughly in subsequent sections. Our discussion here focuses on three types of hardware used by a DBMS: CPU, random access memory, and disk storage. Like prior work, we assume that compute nodes have a limited amount of memory and that memory/storage nodes have a limited amount of compute. Otherwise, resources are decoupled and connected via a low-latency, high-bandwidth network.

Figure 3.2 depicts the typical execution of DBMSs when running in a DDC. A pool of *storage nodes* holds the database data in persistent storage, a pool of *memory nodes* holds the buffer pool of the DBMS in random access memory, and a pool of *compute nodes* runs the actual DBMS processes, with the local memory of the compute nodes serving as a cache of the buffer pool. The original copies of each process's virtual memory, therefore, reside entirely remotely, either in the remote memory pool, or paged onto remote disk. To execute a query, the database tables are scanned and loaded into the buffer pool; in-memory data will then be transferred to and from the processing and the memory pools during execution. The processing and storage nodes do not exchange data directly.

The OS chooses which pages to maintain in the local memory of compute nodes using well-known page eviction policies like LRU or FIFO—data is fetched from remote memory on a local memory cache miss and fetched from storage on a remote memory cache miss. We term the former *remote memory accesses* and the latter *disk page faults* to differentiate the two in this chapter. In both cases, if a query requires data beyond what is cached in its local memory, a kernel trap will block the execution of the query until the memory can be fetched from the memory pool.

The overall performance cost of this additional layer in the memory hierarchy depends on several factors. For instance, the relative size of local memory compared to the buffer pool will determine the frequency of accesses. The interplay between the buffer pool management strategy and the OS local memory eviction policy can also have a significant effect on performance, as can the interaction between remote memory accesses and disk page faults, and the pattern of accesses and the architecture of the DBMS. To illustrate one example of the complexities of this space, consider an LRU buffer pool on top of an LRU local memory eviction policy. When the DBMS evicts an item from the buffer pool, it might:

1. Bring a new item into a memory node from storage.
2. Bring the new item into local memory, evicting others.
3. Bring the LRU item from memory into local memory.
4. Finally, copy from local memory to the buffer pool.

Step 3 is due to the DBMS's replacement algorithm running in the compute node. This highlights how two in-memory buffers result in two sets of replacement policies whose interaction may be suboptimal, suggesting the need for the buffer pool to be aware of "cheaper" local memory and more expensive remote memory.

3.3.1 Single Join Operation

Now consider a simple single-pass, single-machine join on the above architecture. Assume two tables, A and B, are joined using a traditional hash join. Also assume that A is the bigger of the two tables and that we have an existing hash index of B built using the join key. In a traditional DBMS, we would scan table A, compute a hash on the join attribute for each tuple in A, and probe the hash index of B for matches, returning the matching tuples.

In a DDC, this operation would proceed as follows.

- **Initial scan of A.** To perform the initial scan, a query engine on a compute node will have to first request table A's blocks from storage. As mentioned, this process is recursive, involving multiple rounds of communication until the requested blocks are transferred to the local memory of the compute node. If local memory is scarce, the compute node may need to fetch a single block at a time; this is inefficient, but correct, as the algorithm requires only a single block of A as its working set.

- **Probing B's hash index.** When a block of A is in the compute node, the query processor iterates through every A-tuple to probe B's hash index. Again, a portion of B's hash index needs to be fetched from disk to remote memory, and then brought into the local memory of the query processor. Recent work [38] shows that fetching entries from a hash table stored in "far memory" is particularly expensive because of hash collisions that require multiple round trips between the compute node and remote memory/storage to traverse the hash table's buckets. While such collisions also exist in today's systems, the overhead is typically dominated by disk I/O.

The join operation leads to more inefficiencies if A or B are too large to fit in remote memory, requiring multiple passes over the data with either sort-merge or grace hash join.

3.3.2 Exchange and Symmetric Hash Join

We next consider the case of a parallel join operation. Since the local join operation is similar to the single-machine hash join described above, we focus on the *exchange operator* [21, 118]

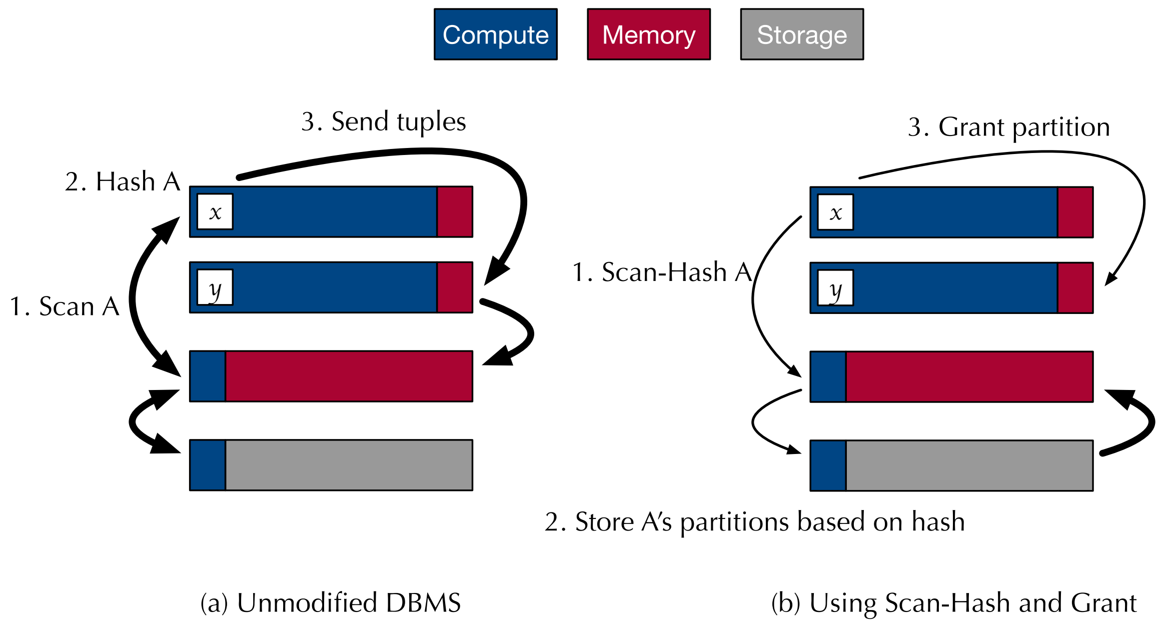


Figure 3.3: Figure (a) depicts a hash partitioning when the DBMS is running on LegoOS. Figure (b) shows the same operation but with additional primitives (§3.8).

followed by a parallel pipelined symmetric hash join. The hash exchange operator repartitions the data by hashing the join attributes. Consider a join of tables A and B executed in parallel on a number of compute nodes, each running a query processor with its own local memory. Assume that each compute node is responsible for a subset of each table.

- **Initial scan.** In the initial scan, each compute node does a scan of its assigned A and B blocks. As above, data needs to be transferred from storage to remote memory, then from remote to local, potentially over multiple iterations if cache is limited.

- **Parallel hash partitioning.** Each compute node iterates through A and B tuples in its local memory and applies a hash function on the join attribute value to determine the *destination* node performing the join. The repartitioned tuples (by join key) are then pushed to the destination node, which stores them in local memory. If needed (e.g., due to insufficient local memory), the destination node may need to copy these tuples *out* to remote memory before it can receive more tuples. This process is bandwidth intensive, as shown in Figure 3.3(a). Compute node x scans table A from remote storage by first attempting to fetch A from remote memory. When that fetch fails, the memory forwards that request to the remote storage. A is then scanned into the remote memory, and then into x's local memory. At this point, x can compute a hash on the join key, and determine the partitions. x then sends some of the partitions over the network (via ToR) to y, which is then responsible for those partitions. This forces y to copy those tuples from its local memory to its remote memory.

- **Pipelined symmetric hash join.** Finally, each destination compute node performs a pipelined symmetric hash join in which a local-memory hash table is built for each of the partitions, and incoming A and B tuples are probed against the hash tables. If the local memory is insufficient, the in-memory hash tables may also need to be continually fetched from and evicted to remote memory. In this case, there is another round trip to construct the hash tables in local memory (from the rehashed tuples that arrive in the previous step) and then transfer them to remote memory.

In each of the above steps, data is repeatedly transferred between storage, remote, and local memory—all to end up storing the data back in remote memory anyway! In Section 3.8 we show that with small modifications to the OS and the DBMS, many of these data transfers can be avoided. We note that the above issues (and our proposed solutions) apply to other parallel join algorithms as well.

3.3.3 Performance Challenges

To demonstrate the impact of running DBMSs over DDCs, we evaluate the performance of hash operators in Linux and LegoOS using the TPC-H benchmark. Our testbed consists of three RDMA-enabled CloudLab r320 machines [223] that emulate one compute node, one memory node, and one storage node running LegoOS. All of these nodes are connected via a 56 Gbps Infiniband network using Mellanox MX354A NICs and a Mellanox SX6036G switch. Compute nodes have access to a Xeon E5-2450 (8 cores, 2.1 Ghz) and memory nodes have 16 GB of RAM. As the amount of local memory on compute nodes is currently undetermined,

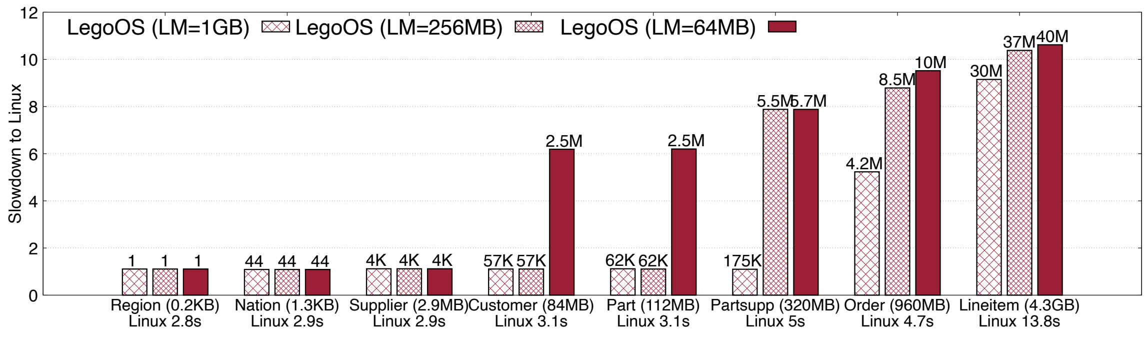


Figure 3.4: Query performance on hash indexes of TPC-H (scale factor 10) tables. LM stands for local memory.

we test a range of possible values: 64 MB, 256 MB, and 1 GB. We expect the eventual value, as a ratio to remote memory and dataset sizes, will trend lower in the spirit of disaggregation.

LegoOS currently supports a subset of the Linux system calls, so we extended the codebase as needed to implement a query execution engine for DBMS operations. Section 3.4 discusses more extensions that we made to LegoOS to support production DBMSs. For comparison, we also make our query engine compatible with Linux 3.11 and run it on a single machine with the same compute, memory, and storage resources as the disaggregated testbed. The engine supports hash join, hash aggregation, nested loop, filter, project, and sort operations, all of which are needed for the TPC-H queries.

Hash table performance. Hash table performance is crucial to the execution of many DBMS operators. Our first experiment evaluates the performance of querying hash tables in DDCs. Figure 3.4 describes the details of the experiment and shows the results. Specifically, we run 100 million random accesses to each hash table and compare the running times of LegoOS with different sizes of local memory (1GB, 256MB and 64MB) and Linux. The x-axis shows each TPC-H table along with its hash index size and the performance in Linux. The y-axis shows the slowdown in LegoOS compared to that in Linux. Each bar is labeled (above) with the number of remote memory accesses in LegoOS. When data is larger than local memory, the performance of accessing hash tables in LegoOS is an order of magnitude worse than in Linux. We observed that, when the size of the hash table is within the local memory capacity, the whole table can be cached locally and the performance slowdown of LegoOS (relative to

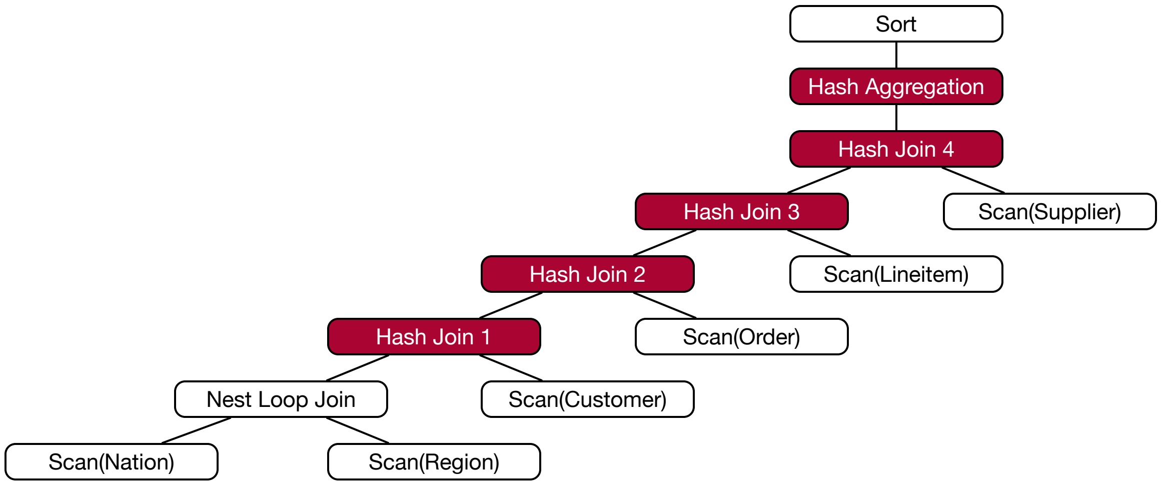


Figure 3.5: An optimized physical plan for TPC-H Query 5. Shaded are the operators that use hash tables that we built in Figure 3.4.

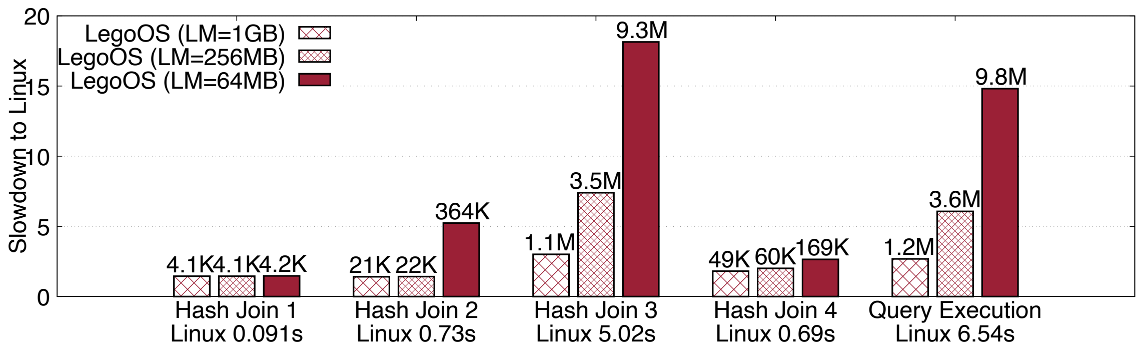


Figure 3.6: Execution performance of hash joins and end-to-end TPC-H Query 5.

Linux) is within 2×. However, when the table is larger than local memory, the corresponding slowdown ranges from 5× to 11×, showing severe performance degradation.

Query execution performance. We next evaluate the effect of the above slowdowns on an end-to-end TPC-H query—Query 5—the optimized plan of which (shown in Figure 3.5[1]) involves multiple hash joins. While we present results based on this specific query and plan, Section 3.5 shows that the insights are general to any other DBMS query execution.

Figure 3.6 compares the query execution performance in LegoOS with different sizes of local memory and Linux. Specifically, the x-axis shows the execution of four hash join operators in Figure 3.5, the end-to-end query execution, and their performance in Linux. The

[1]Adopted from the optimized plan in Microsoft SQL Server [24].

y-axis shows the slowdown of the performance in LegoOS compared to that in Linux. Each bar is labeled with the number of remote memory accesses in LegoOS. The results show that when the working set of an operator can fit in local memory, the performance slowdown can be controlled around 2×. As expected, the worst degradation is observed at Hash Join 3 where the two largest tables are joined: the slowdown relative to Linux is 3×, 7× and 18× for LegoOS with 1 GB, 256 MB and 64 MB local memory, respectively. *This degradation results in a slowdown of 2.7×, 6×, and 14.8×, respectively, in overall query execution.* We expect greater slowdowns in larger-scale executions.

Remote memory access. To confirm our hypothesis that the majority of the overhead in LegoOS (over Linux) comes from remote memory requests, we measure the number of remote memory accesses in LegoOS. As Figures 3.4 and 3.6 depict (in the label above each bar), LegoOS needs to constantly page remote memory for queries that require large working sets.

Summary. While limited in scope, the above experiments show that a naïve DBMS implementation on a DDC would suffer severe performance degradation.

The remainder of this chapter presents extensive exploration of the performance implications of DDCs on production DBMSs. Our results highlight the need to develop new techniques to make the performance of DBMSs palatable in this brave new world, which is the focus of Section 3.8 and the next chapter.

3.4 Setup and Methods for Extensive Evaluation

To extensively explore the implications of resource disaggregation for data processing, we now present an in-depth characterization and analysis of the performance of production DBMSs running on DDCs. This section details the setup of our performance measurements.

3.4.1 Testbed Setup

Our DDC testbed consists of three bare-metal servers connected by an Infiniband network in CloudLab [223]. Both the servers and the network are the same as what we described in

	MonetDB	**PostgreSQL**
Execution	In-memory	Out-of-core
Storage	Column-based	Row-based
Architecture	Client/Server	Client/Server
Buffer pool size	$\min(S_{\text{Capacity}}, S_{\text{Demand}})$	Customizable

Figure 3.7: Summary of parameters in MonetDB and PostgreSQL.

Section 3.3.3. At the time of this work, we were restricted to this hardware configuration due to LegoOS's limited driver support, and the low availability of compatible servers in CloudLab. To provide a fair baseline, we compare to a single Ubuntu Linux 3.11 server with the same compute, memory, and storage resources as our DDC testbed.

Local memory configuration. Vendors and cloud providers have not yet settled on the size of local memory in DDCs, but we expect the most cost-efficient nominal (i.e., per-CPU) sizes to be smaller than typical DBMS buffer pools. We evaluate DDC performance on a variety of local memory sizes ranging from low (64 MB) to high (4 to 6 GB) capacity to emulate different degrees of disaggregation.

Storage. Due to hardware availability, our testbed uses hard disk drives for storage. While SSDs would improve performance, we expect that the general trend of the disk being a bottleneck in some of our experiments would still hold as our Infiniband network significantly outperforms SSDs in both latency and throughput.

3.4.2 System Selection and Adaptation

We select two popular open-source DBMSs: MonetDB [19] (Version 11.33.11) and PostgreSQL [218] (Version 11.5)—both are the latest versions at the time of this evaluation. We select these two systems to represent different types of DBMSs: MonetDB is a column store, designed to be executed in-memory; PostgreSQL is a row-based system and it adopts an out-of-core execution model. We summarize and compare the technical parameters of MonetDB and PostgreSQL in Figure 3.7. One parameter of interest is the buffer pool size. In MonetDB, the system consumes as much memory as needed to match application demand (S_{Demand})

as long as it does not exceed the amount of physical memory (S_{Capacity}). In PostgreSQL, the buffer pool size is customizable. For both Linux and LegoOS, we tune the PostgreSQL buffer pool size to maximize performance.

We note that LegoOS currently supports only a subset of Linux system calls. Thus, to execute PostgreSQL and MonetDB, we spent significant effort adapting these two DBMSs to LegoOS (for reference, PostgreSQL has ~1.3M lines of C code, MonetDB has ~400K lines of C and MAL code, and the LegoOS kernel consists of ~300K lines of C code). We highlight three examples:

Socket bypass. LegoOS, which relies solely on RDMA for communication between nodes, currently does not support sockets, but the client and the server communications of both PostgreSQL and MonetDB are based on sockets. Thus, we bypass the client and directly start the server to execute the SQL queries and benchmark the query execution performance.

Read system call. Another example is a slight difference between the implementation of the `read` system call in LegoOS and Linux. When the application `read`s N bytes from a file, due to disaggregation, LegoOS allocates N bytes of memory in kernel space in the compute node to receive the data that is finally returned from the memory node (refer to the data paths in Figure 3.2). If N is large, the compute node can run out of memory, leaving other components of the system hanging. We added additional functionality to address this issue.

Relative paths. The original version of LegoOS could only support absolute paths while relative paths are extensively used in the selected DBMSs; we implemented two system calls (`getcwd` and `chdir`) in order to run MonetDB and PostgreSQL.

Additionally, we fixed several inconsistent behaviors in the way that LegoOS performs file system operations. For example, in LegoOS, `rename` always unlinks the old file on the storage node without detecting the existence of the new file, so if the new file does not exist, then the old file is deleted, while in Linux, the old file is still safe. We note that these issues are due to the immaturity of the current LegoOS codebase, rather than its higher-level design.

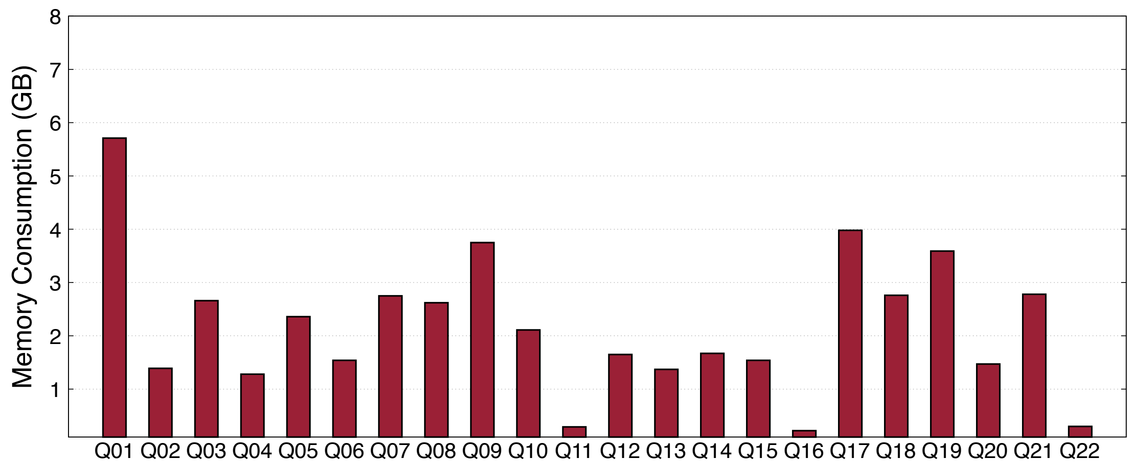


Figure 3.8: Peak memory usage of TPC-H queries in MonetDB.

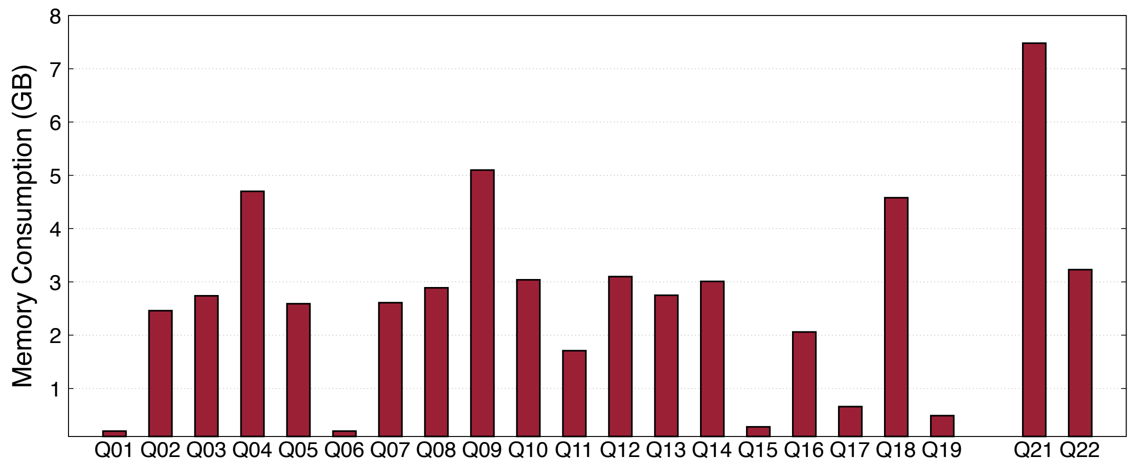


Figure 3.9: Peak memory usage of TPC-H queries in PostgreSQL.

3.4.3 Workload Selection and Characterization

To study the implications of disaggregation on the end-to-end query execution performance of real-world complex queries, we select the TPC-H benchmark and use all of its 22 queries, which represent a wide range of execution patterns. Unless otherwise specified, we use a scale factor of 10.

As discussed, memory disaggregation makes memory accesses a bottleneck for many applications in DDCs, so overall memory consumption is an important factor in this study. To that end, we provide a characterization of the memory demands of all 22 TPC-H queries. The exact demands of each query, of course, vary as each DBMS will select its own execution plan for each query depending on a number of factors; different plans will result in different memory usage patterns. Thus, we run the queries with the aforementioned scale factor in MonetDB and PostgreSQL in Linux and measure the memory consumption of each query. We note that the memory used by the OS for disk caching is not included here because it is determined by the OS, and the OS (for example, Linux) can aggressively use available memory for caching disk data as long as it does not affect the memory usage of the applications.

Figure 3.8 and Figure 3.9 show the measurement results for MonetDB and PostgreSQL respectively. Note that in the case of PostgreSQL, we configured the maximum buffer pool as 8 GB to allow for sufficient OS and disk cache space, and we excluded Q20 because it could not finish execution [213]. The memory consumption of different queries running on a single DBMS can vary substantially, as can the consumption of a single query on two different DBMSs. Even so, we can summarize a few patterns: (1) all queries consume more than 200 MB of memory; (2) most queries use around the average amount of memory (2.2 GB in MonetDB and 2.8 GB in PostgreSQL); (3) a few queries use significantly more memory (Query 1, 9, 17, 19 in MonetDB, Query 4, 9, 18, 21 in PostgreSQL) than others; and (4) a few queries use significantly less memory (Query 11, 16, 22 in MonetDB, Query 1, 6, 15, 17, 19 in PostgreSQL) than others. We will refer back to these two figures when we analyze experimental results in the next sections.

3.5 The Cost of Disaggregation

We evaluate the overhead of disaggregation by running both production DBMSs on LegoOS and a traditional standalone Linux server. We equalize the amount of compute, memory, and storage resources between LegoOS and Linux to ensure a fair comparison.

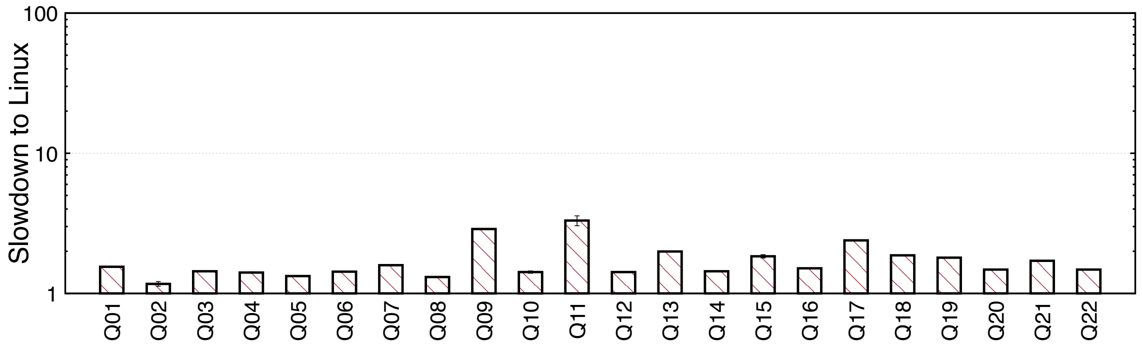


Figure 3.10: MonetDB query execution time slowdowns with 4 GB local memory in LegoOS. The baseline is a single Linux server.

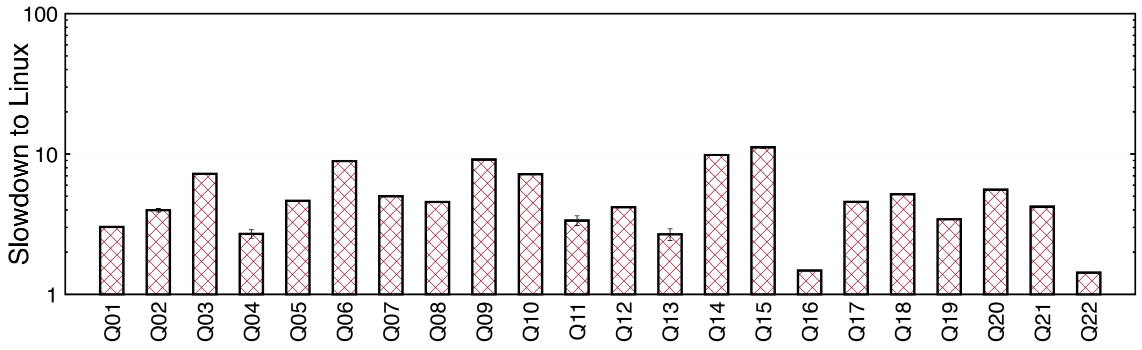


Figure 3.11: MonetDB slowdowns with 1 GB local memory in LegoOS.

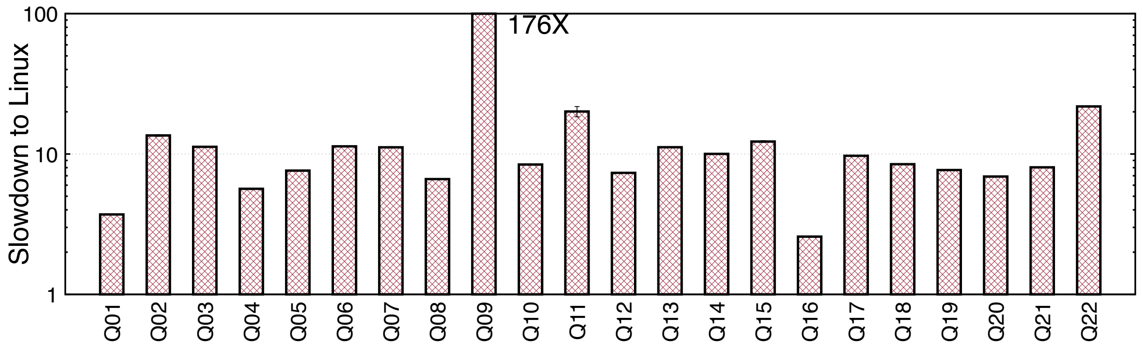


Figure 3.12: MonetDB slowdowns with 64 MB local memory in LegoOS.

3.5.1 In-memory Execution

We first evaluate MonetDB under three different local memory sizes (4 GB, 1 GB, and 64 MB). Before running each TPC-H query, we warm up the DB buffer pool, to remove the effect of disk paging. For each graph, we show the slowdown relative to Linux for all 22 TPC-H queries, where a slowdown ratio of 1 means performance is on par with Linux. In all figures, all bars are augmented with 95% confidence intervals, which show that the results are stable with low variance. We summarize our findings as follows.

4 GB local memory (Figure 3.10). The slowdown is moderate: an average of 1.7× and a median of 1.5×. The slowdown stems from fetching data from the remote buffer pool in the memory pool to the local memory. However, LegoOS's optimizations on memory prefetching and lazy memory allocation keep the slowdown moderate. Moreover, because CPU is the primary bottleneck, even though the working set of Q1 (Figure 3.8) does not fit into 4 GB, the slowdown is only 1.6×. Q9 and Q17 experience higher slowdowns (2.9× and 2.4× respectively) because their actual working sets (once kernel, stack, and instruction cache are included) well exceed 4 GB, resulting in thrashing of local memory. Q11 has the highest slowdown given that it is very short (it runs in 0.07 s) and a handful of memory stalls incur a high relative slowdown.

1 GB local memory (Figure 3.11). As local memory decreases, the average slowdown increases because most queries utilize more than 1 GB of memory. Queries with small memory footprint (Q16, Q22) are not affected by the local memory reduction, and Q11 is also less sensitive because, although it does spill data to remote memory, going from 4 GB to 1 GB does not exacerbate the effect.

64 MB local memory (Figure 3.12). The final configuration reduces local memory to only 64 MB. All queries are more than 2.5× slower than their non-disaggregated executions and ten of them have performance degradation larger than 10×. Q9 has the most extreme slowdown of 176×. This is because Q9 adopts nested loop joins for six tables, and together with an expression calculation, they result in frequent random accesses to the buffer pool. Those random accesses cause extreme inefficiency when the local memory is constrained. We analyze the slowdowns in greater detail by relating them to remote memory accesses in Section 3.7.

3.5.2 Out-of-Core Execution

We next evaluate PostgresSQL to understand the impact of out-of-core execution under two settings: (1) execution in a cold hardware/software cache scenario (*cold execution*); and (2)

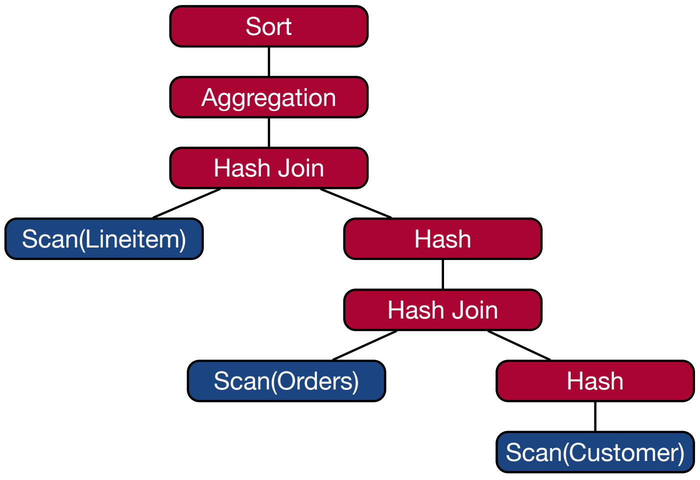


Figure 3.13: The simplified execution plan for Q3 in PostgreSQL. Blue operators involve disk I/O and red operators are in memory.

execution after the buffer pool and caches are warmed up by running the same query multiple times (*hot execution*). We differentiate between those two scenarios because PostgreSQL heavily relies on OS mechanisms to cache recent data.

Cold execution. Figures 3.14–3.16 show the cold execution performance in LegoOS with different sizes of local memory. In cold execution given 4 GB and 1 GB local memory (Figures 3.14 and 3.15), most queries have negligible slowdowns since disk I/O overshadows the additional network latency. Consider the plan of Q3 (Figure 3.13) which consists of a right-deep tree of a 3-way join, the tree is executed in a pipelined fashion: every time a tuple of `lineitem` is scanned, it is used to join with the rest of the tree. Given the size of the `lineitem` table, significant disk I/O incurred during the scan dominates the execution. This is still true when we migrate to a disaggregated environment—disk I/O takes a longer time than the memory stalls that fetch data from remote memory. This disk bottleneck closes the gap between LegoOS and Linux. In the 64 MB setting (Figure 3.16), the majority of queries continue to achieve similar performance to their non-disaggregated executions, as shown in Figure 3.16. The queries that experience higher than 2$\times$ slowdowns (e.g., Q13), do so because of unmasked memory stalls (e.g., executions that are not pipelined or that perform many random accesses).

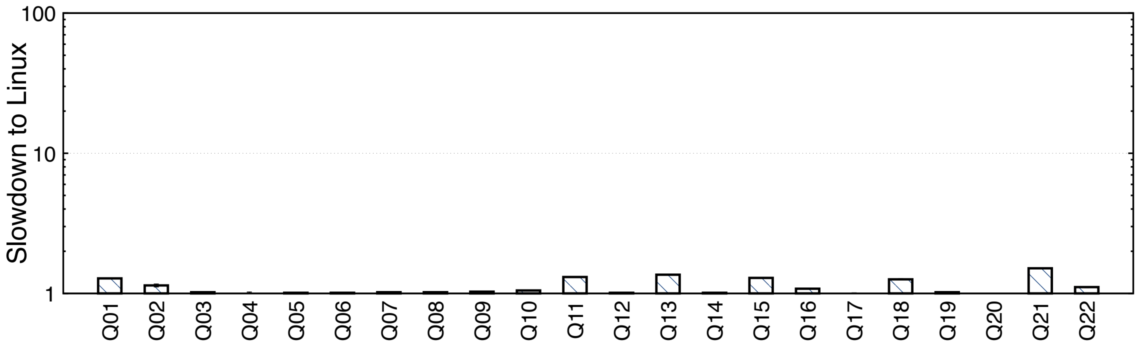


Figure 3.14: PostgreSQL cold execution time slowdowns with 4 GB local memory in LegoOS (Q20 excluded). The Baseline is a single Linux server.

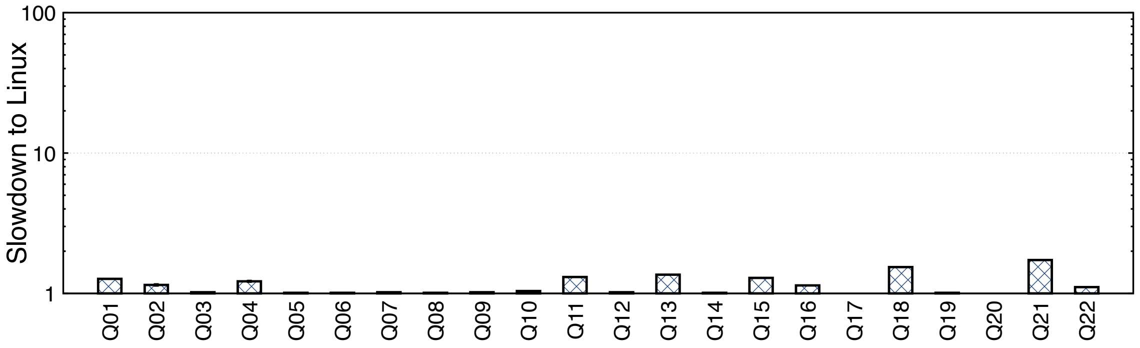


Figure 3.15: PostgreSQL (cold) slowdowns with 1 GB local memory in LegoOS.

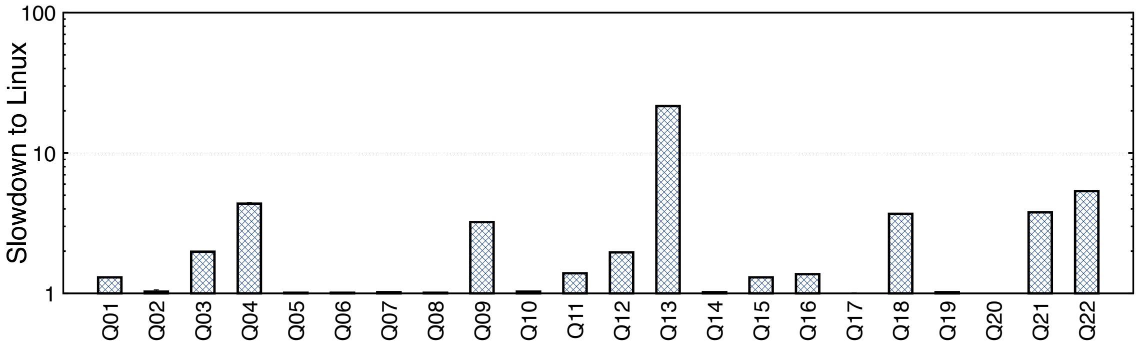


Figure 3.16: PostgreSQL (cold) slowdowns with 64 MB local memory in LegoOS.

Hot execution. Figure 3.17 shows that given 4 GB local memory, average and median hot execution slowdowns are 2× and 2.1×, respectively. At 1 GB memory (Figure 3.18), the average slowdown increases only slightly to 2.4×, indicating that the performance is still largely bottlenecked by I/O.

These two results show an interesting effect. Although in-memory and hot out-of-core execution both bypass disk I/O, they perform very differently in DDCs when local memory is

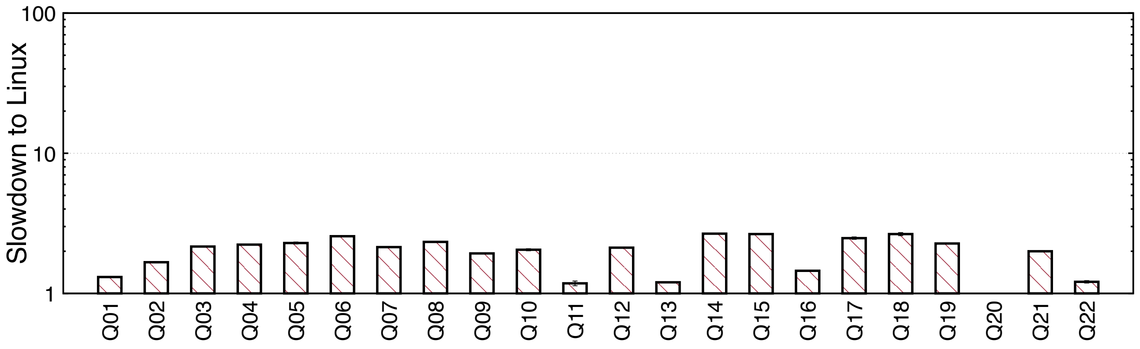


Figure 3.17: PostgreSQL hot execution time slowdowns with 4 GB local memory in LegoOS (Q20 excluded). The Baseline is a single Linux server.

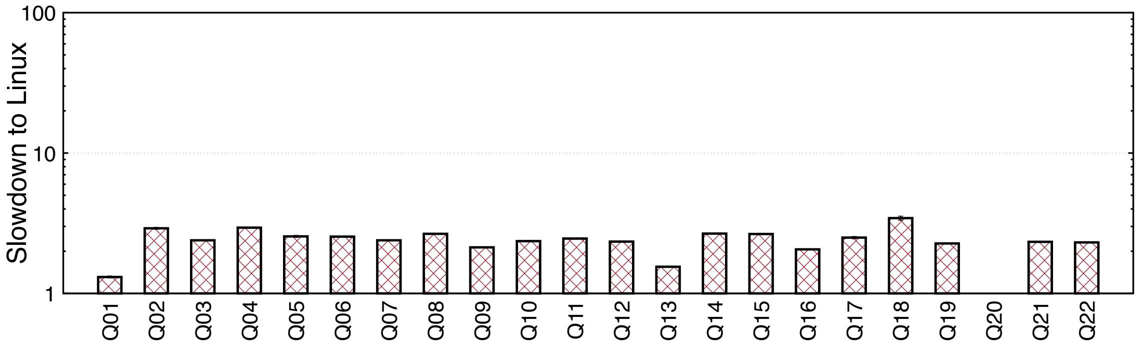


Figure 3.18: PostgreSQL (hot) slowdowns with 1 GB local memory in LegoOS.

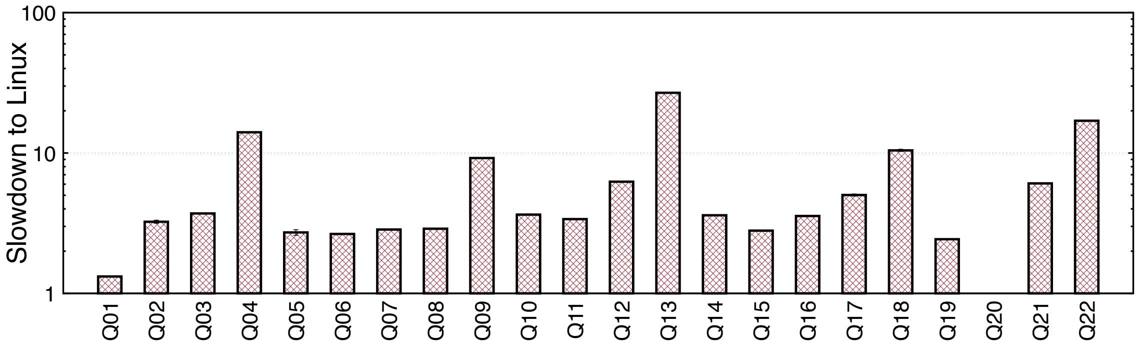


Figure 3.19: PostgreSQL (hot) slowdowns with 64 MB local memory in LegoOS.

sufficient: the latter still suffers from I/O bottlenecks while the former does not. The reason is a gap between the efficacy of application-based disk cache management (as done by MonetDB) and LegoOS's disk management (as outsourced by PostgreSQL). The difference is that while the application data can be cached locally in the processing pool, LegoOS stores its disk cache remotely in the memory and storage pool. Consequently, MonetDB's manual management of the disk cache results in much better data reuse and pipelining.

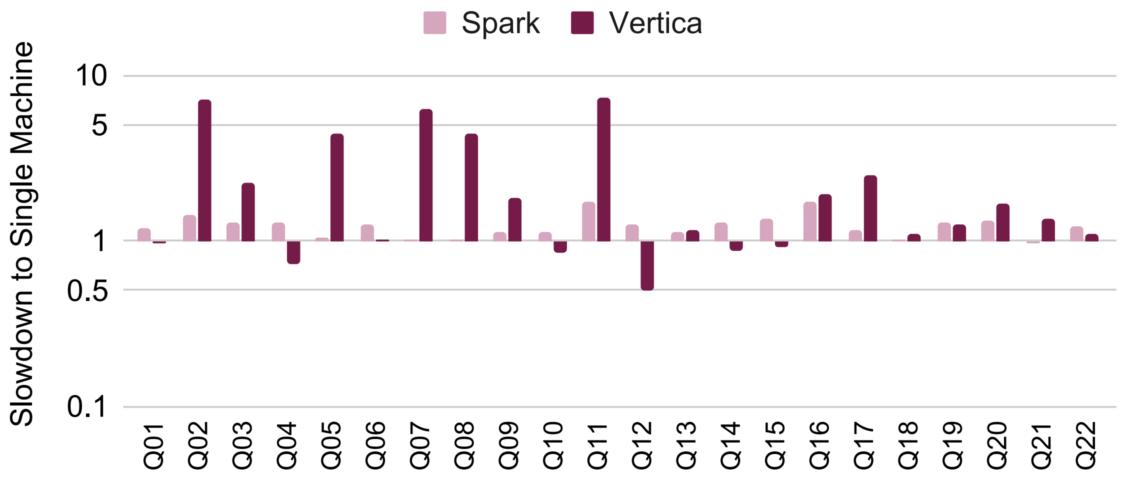


Figure 3.20: The slowdowns of running distributed DBMSs in a cluster compared to a single machine of the same hardware.

A further reduction of local memory to 64 MB results in a significant slowdown for a subset of queries (Figure 3.19). Memory intensive queries (Q4, Q9, Q18, and Q22, cf. Figure 3.9) experience 10× slowdowns, and Q13 has a worst-case 27× slowdown. The slowdown is larger in hot executions because they eliminate the disk I/O that masks the performance degradation in cold executions.

3.5.3 Distributed Baseline

Next, we study how scale-out setups affect the performance of traditional, distributed DBMSs, relying on these results to put DDC performance slowdowns into perspective. We have chosen two highly-optimized DBMSs: Apache Spark SQL [5] v2.4.5 and Vertica [6] v9.3.0. We first ran these systems using TPC-H (scale factor 10) in a single Linux machine, and then set up a distributed environment using three "smaller" machines that collectively provide equivalent hardware resources, including CPU cores, memory, and storage. We configured Spark SQL to use NFS to access a remote storage server, ensuring the same disk I/O performance. Vertica, however, does not support NFS, so we configured it to use the local storage on each machine. We note that this gives the distributed setup of Vertica a slight advantage in aggregate disk

I/O throughput. Nevertheless, this setup paints a useful picture of the contrast of DDC and distributed DBMS slowdowns.

Figure 3.20 shows the results for performance slowdowns of these DBMSs due to distributed execution. Overall, the distributed setup led to an average slowdown of 1.2× in Spark and 2.3× in Vertica. Spark performs better because it has a higher sensitivity to computation than network communication [209]. Vertica, on the other hand, has more performance variance. It is more sensitive to network communication in some queries; for example, in Q2, Q7, and Q11, the execution incurs heavy communication between workers. In Q12, the distributed setup is even better than the single-machine setting because of good partitioning and higher aggregate disk bandwidth. Comparing these results with DDC slowdowns (Figures 3.12, 3.16, and 3.19), we see that the overhead of scaling out is more significant in DDCs, highlighting the need for optimizations.

3.5.4 Query Throughput

So far, we have focused on quantifying the slowdown of query completion time; we have similar findings in query throughput. As discussed, the main bottleneck in the DDC setting stems from memory stalls not compute parallelism—in fact, DDCs can spawn as many compute workers as the resource pools allow. We, therefore, observe similar trends for query throughput as we did for individual query completion times.

We evaluate the impact on TPC-H throughput by feeding two streams of TPC-H queries to MonetDB and compare the respective throughput of Linux and LegoOS. Figure 3.21 shows slowdowns in LegoOS with different local memory sizes, with the highest-level takeaway that the trends match those in Figures 3.10–3.12 for individual queries. The DDC setting, of course, provides new opportunities for rethinking how parallel/concurrent executions can be further optimized. This requires redesigning the underlying OS abstractions and compute models, which we leave for the next chapter.

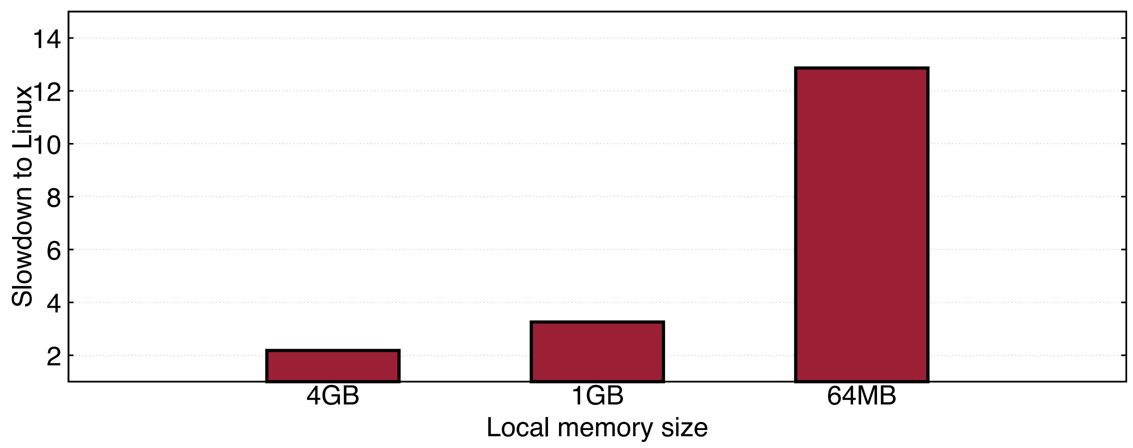


Figure 3.21: The slowdowns of LegoOS in the TPC-H throughput benchmark. Trends are similar to the observations for single query performance.

3.5.5 Summary

The overhead of DDCs is moderate for in-memory query executions if each query's working set fits into the processing pool's local memory. However, as query memory requirements exceed the local memory, the communication overhead can result in a significant degradation in query execution times. The degradation is even worse under frequent random accesses. In both cases, the interaction between the OS and DBMS-level memory access patterns can heavily influence the effect of disaggregation.

There are significant differences in disaggregation slowdowns in out-of-core vs in-memory systems. Even within out-of-core systems, hot and cold executions vary in slowdowns as well. Cold executions are dominated by disk I/O and hence less sensitive to the network overheads introduced by disaggregation. Hot executions rely too heavily on default LegoOS disk cache management, which stores the cache in remote memory. Overall, out-of-core executions are generally less sensitive to the degree of disaggregation than in-memory executions because they are bottlenecked by other factors, though we note that significant slowdowns can still occur when the degree of disaggregation is extreme (for instance, Q13 in LegoOS with 64 MB local memory).

Moreover, distributed DBMSs set a good baseline for DDCs on the cost of scaling out and highlight the need for codesigning DDCs and DBMSs to avoid redundancy and mismatched execution policies, especially on data movement.

3.6 The Elasticity of DDCs

While disaggregation can introduce new overheads, a key advantage of DDCs is their elasticity—a DDC can provision an almost arbitrary amount of resources to each process, and this provisioning can expand beyond the resources contained in any one server. This elasticity can have concrete performance benefits, preventing the DBMS from needing to spill data to disk when it overwhelms a single machine's capacity.

To evaluate these effects, we compare LegoOS's efficiency to that of a monolithic server across varying local memory capacities and working set sizes. For some of the more constrained local memory sizes, we note that it is unlikely that monolithic servers will be built with such limited memory; instead, the goal of the experiments is to isolate the implications of having a pool of remote memory that is orders of magnitude larger than local memory.

3.6.1 Versus a Constrained Monolithic Server

We begin by matching and scaling down the local memories of both the LegoOS processing node and a monolithic server in order to emulate a case where today's monolithic servers are augmented with a large pool of remote memory. This is in contrast with the previous section in which we matched the total amount of *remote* memory in the DDC to the memory of the monolithic server. In some ways, the latter represents a lower bound on the relative performance of DDCs. This subsection represents an upper bound.

As before, we fix the scale factor of the TPC-H workload at 10 and set the memory pool capacity to 16 GB, large enough for this particular workload.

In-memory execution. We select three representative queries with which to explore these effects: Q16, Q5, and Q9. These three queries represent three different levels of sensitivity to local memory capacity: low, medium, and high, respectively (cf. Figure 3.12). Other queries

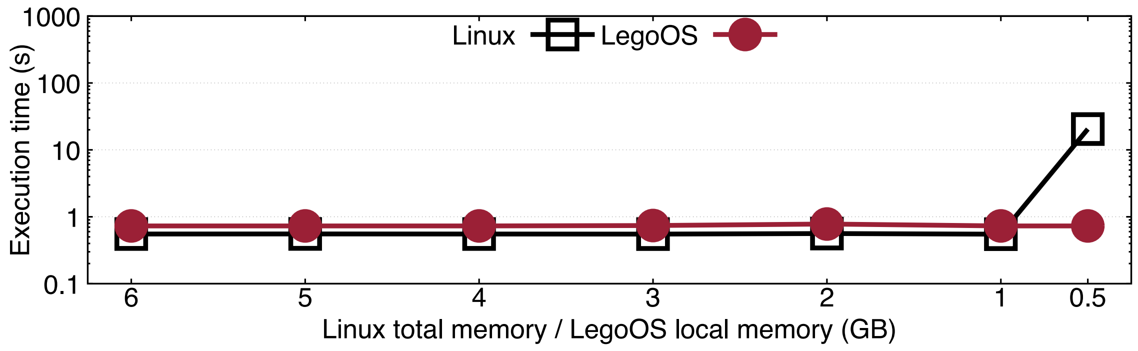


Figure 3.22: MonetDB performance when varying local memory size for Query 16.

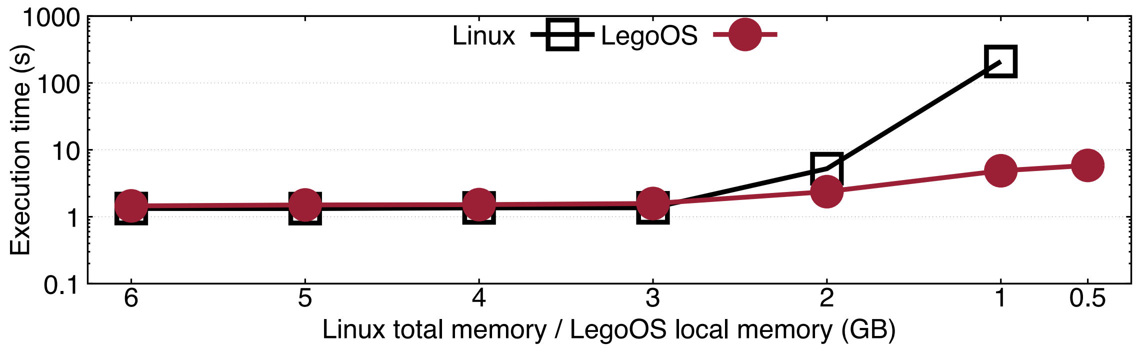


Figure 3.23: Varying local memory size for Query 5.

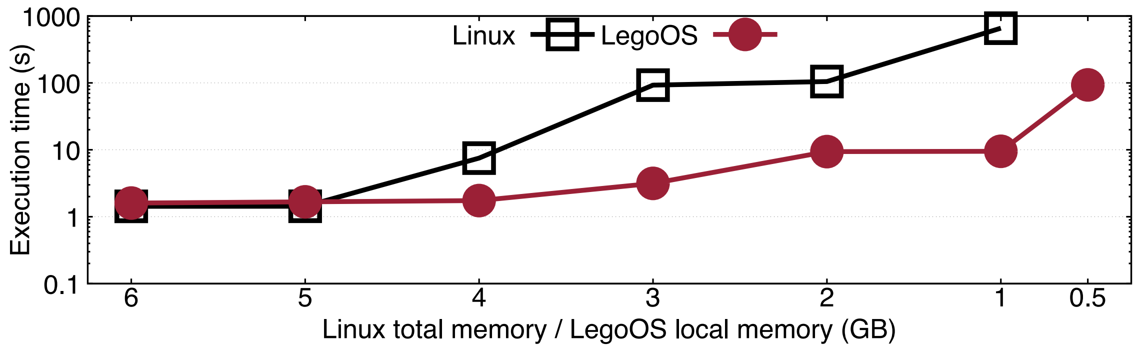


Figure 3.24: Varying local memory size for Query 9.

with similar sensitivity exhibit similar results. Figures 3.22, 3.23, and 3.24 show the execution times of all three queries against local memory capacity in both a single server and a DDC.

As expected, the low-sensitivity query, Q16, maintains its performance across different local memory capacities in the DDC. The monolithic server also retains its (slightly better) performance across most memory capacities; however, when memory is very constrained, performance suffers greatly as data is spilled to disk, with a $\sim$37$\times$ slowdown when memory is constrained to 512 MB. LegoOS is 28× faster than the monolithic server in this scenario. The two more sensitive queries, Q5 and Q9, exhibit similar effects except that the monolithic server slows down much earlier. In fact, for both queries, the 512 MB case fails in the monolithic server with an out-of-memory error. The DDC is still able to execute the queries with a more graceful degradation in performance (but with similar scalability trends). Execution time does rise as memory becomes very constrained, but even in that case, the DDC is consistently 1–2 orders of magnitude faster as data is spilled to remote memory rather than disk. For instance, the most sensitive query in the monolithic server at the smallest capacity that completes, 1 GB, experiences a 460× slowdown compared to LegoOS on the DDC.

Out-of-core execution. In evaluating PostgreSQL, we selected another three representative queries: Q6, Q13, and Q4 for low, medium, and high sensitivity, respectively. These three queries are different from the three queries chosen for MonetDB as the two DBMSs generate different plans with different sensitivities.

Figures 3.25–3.27 show the results of cold executions in PostgreSQL for the three queries. Unsurprisingly, the DDC performance on the low-sensitivity query is again very stable across local memory capacities. Also like the in-memory case, the monolithic server begins to fail in low-memory situations. For both environments, these graphs provide a fine-grained record of performance degradation versus local memory size, showing exactly where local memory becomes the bottleneck of the execution.

Overall, LegoOS performance is significantly more stable across local memory sizes. There are two main reasons for this. The first is related to how query planning is done in a DDC versus a traditional server. One of the key inputs to a query planner is the size of memory—different memory sizes can result in significantly different plans and performance, and a wrong choice in a plan can have bad consequences. When creating a plan for a DDC, LegoOS presents to the DBMS the size of remote memory, rather than local memory. Second is the aforementioned conversion of disk spills to remote memory spills. Disaggregation thus provides an easy to understand scaling model: *When disk I/O dominates the time of a pipelined execution, disaggregation causes no harm; when memory becomes stringent in a*

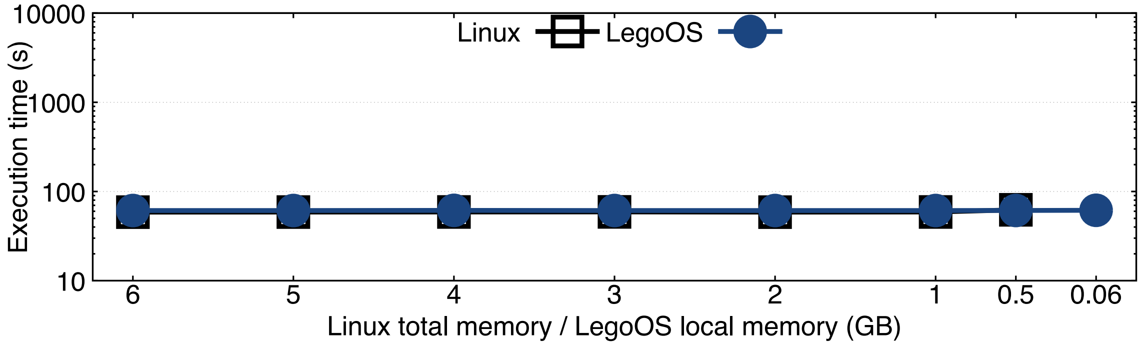


Figure 3.25: PostgreSQL performance (cold) when varying local memory size for Query 6.

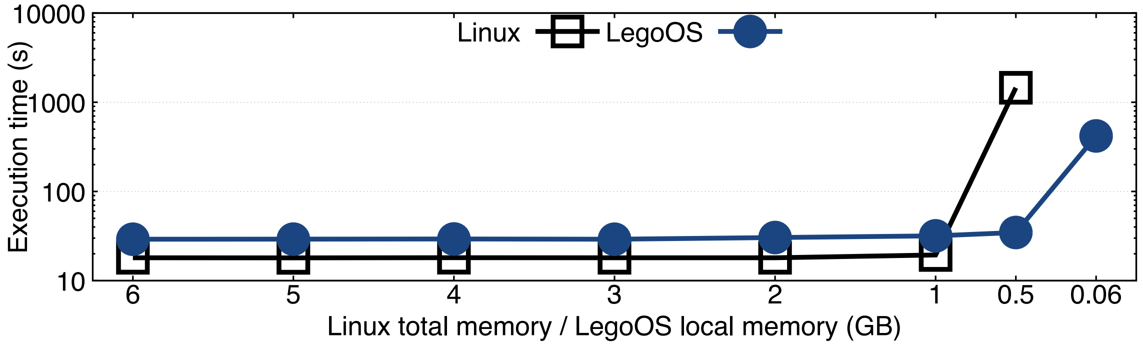


Figure 3.26: Varying local memory size for Query 13.

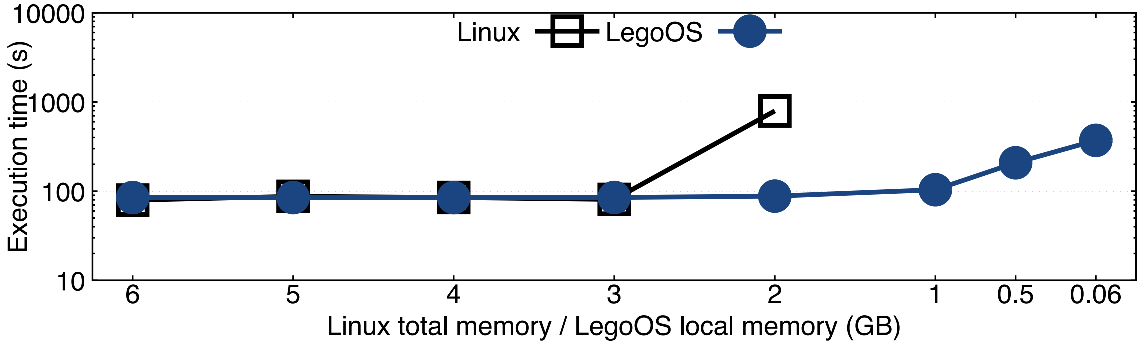


Figure 3.27: Varying local memory size for Query 4.

single server, disaggregation provides better performance and more graceful degradation.

Figures 3.28–3.30 present results for hot executions, which show similar trends to the cold executions. One notable difference is that LegoOS benefits from hot executions of all three queries due to its use of the OS disk cache for repeated loading of the same data. In contrast, the monolithic deployment fails to show a similar improvement because there is insufficient memory in the monolithic server to cache the largest tables used in each query.

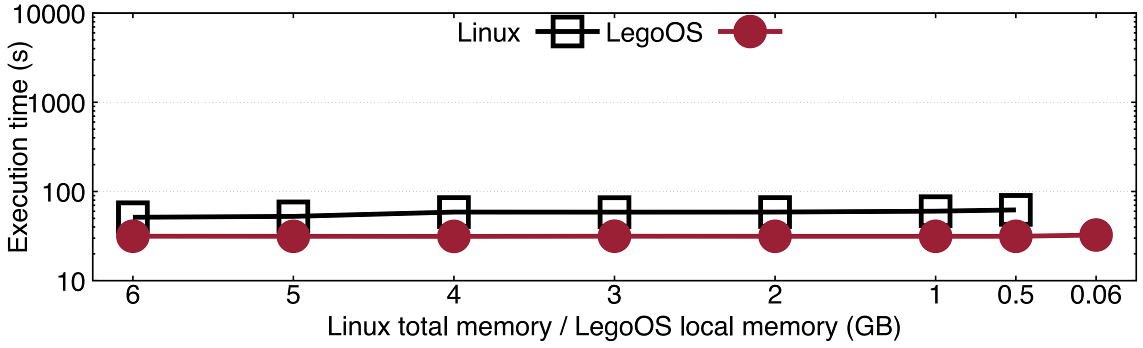


Figure 3.28: PostgreSQL performance (hot) when varying local memory size for Query 6.

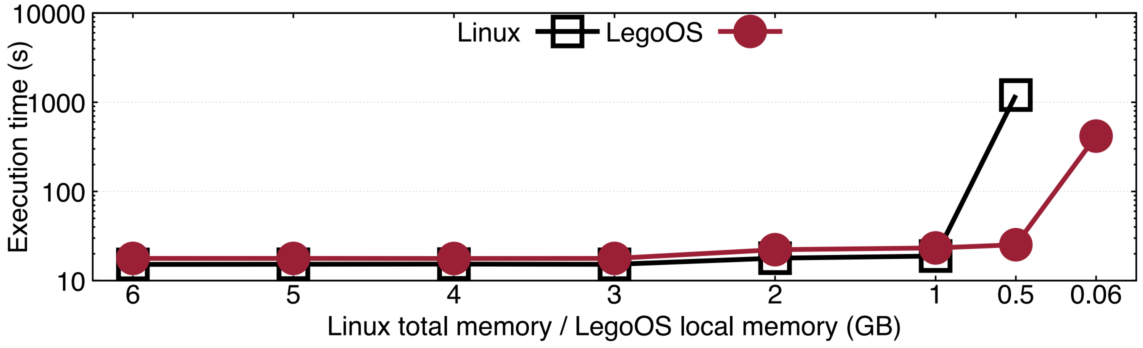


Figure 3.29: Varying local memory size for Query 13.

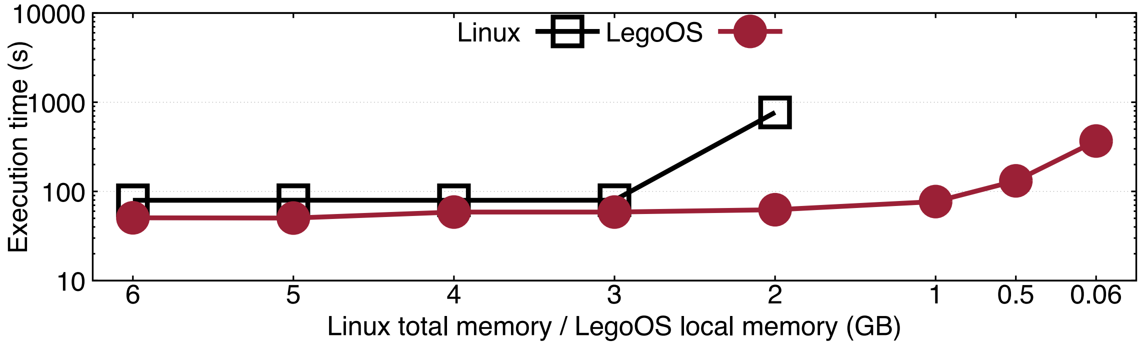


Figure 3.30: Varying local memory size for Query 4.

3.6.2 The Impact of Dataset Size

Next, we compare how monolithic servers and DDCs scale with their workload. To do this, we fix the memory capacity of both the monolithic server and the DDC processing node to 4 GB, and we vary the scale factor (SF) of TPC-H from 2 to 20 with a step of 2.

Figures 3.31–3.33 shows the query execution times for Q16, Q5, and Q9, the same three queries used in the previous subsection for MonetDB. As MonetDB does not require much

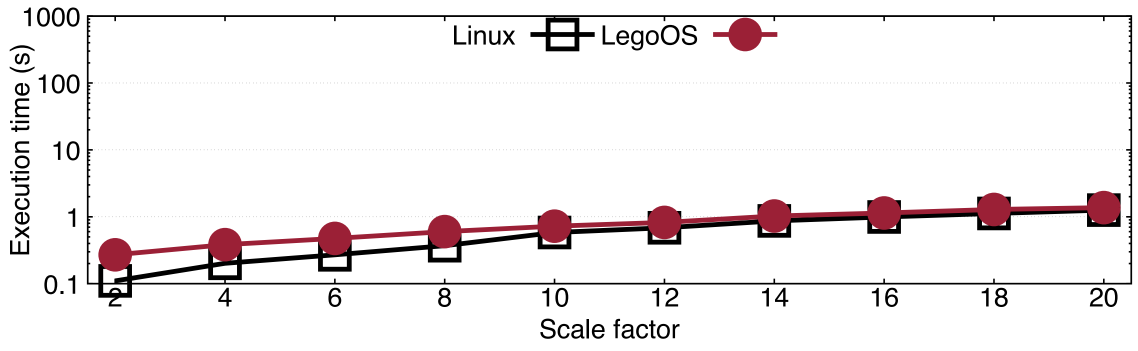


Figure 3.31: MonetDB performance when varying dataset size for Query 16.

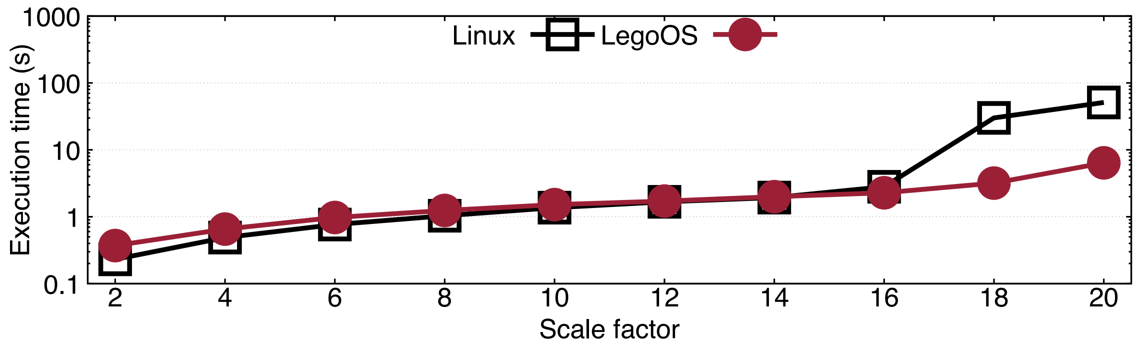


Figure 3.32: Varying dataset size for Query 5.

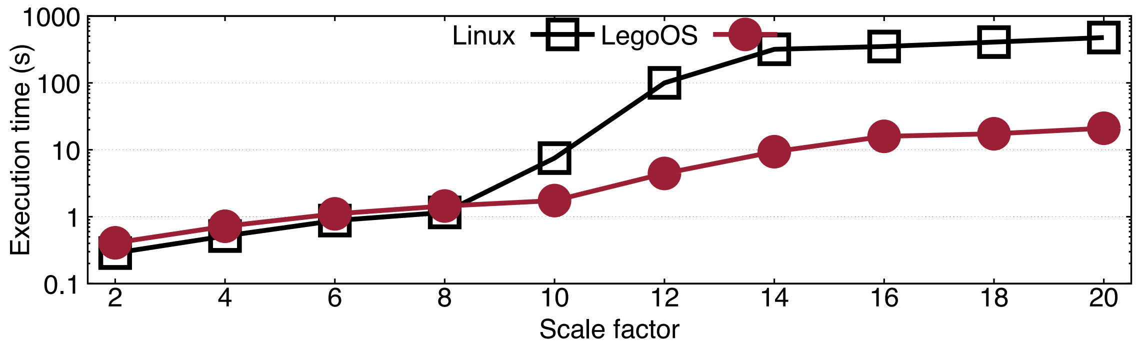


Figure 3.33: Varying dataset size for Query 9.

memory to execute Q16, which joins three small tables (`part`, `part supp`, and `supplier`), 4 GB memory is enough for even SF 20. Execution time, therefore, grows slowly with the size of the data set in both disaggregated and non-disaggregated environments.

For the queries with higher memory sensitivity, LegoOS significantly outperforms the monolithic server on large data sets, just as it did when we decreased local memory in Section 3.6.1. For example, in Q5, when the SF is 16 or above, i.e., when the input size exceeds

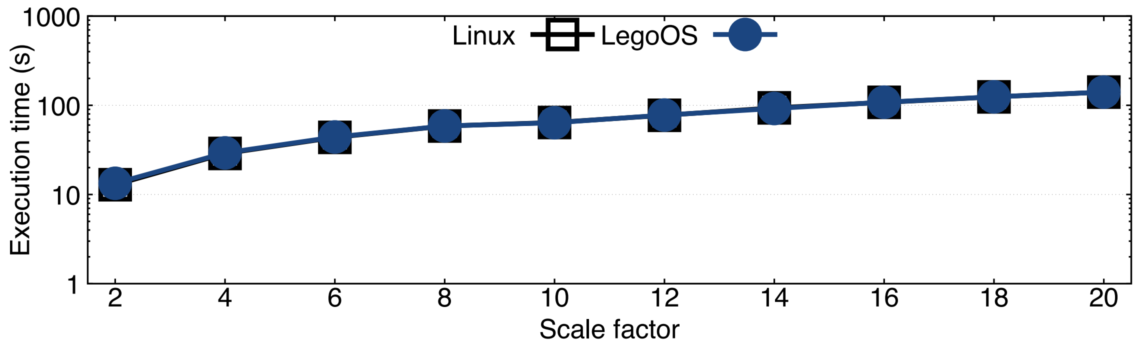


Figure 3.34: PostgreSQL performance (cold) when varying dataset size for Query 6.

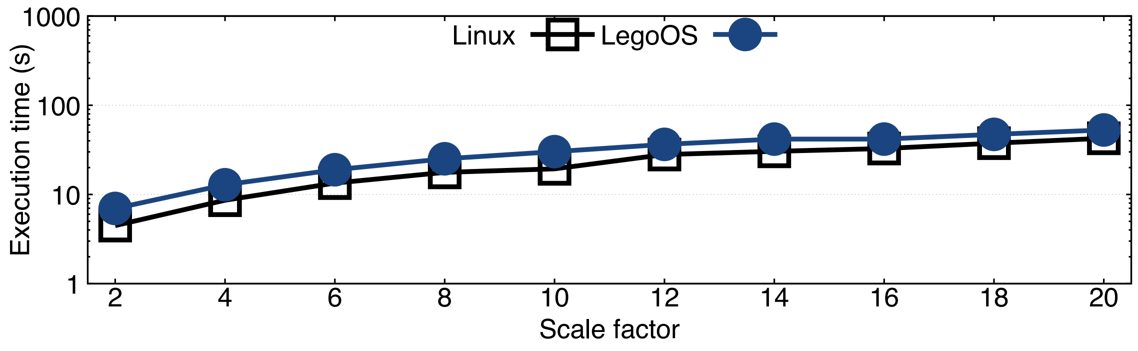


Figure 3.35: Varying dataset size for Query 13.

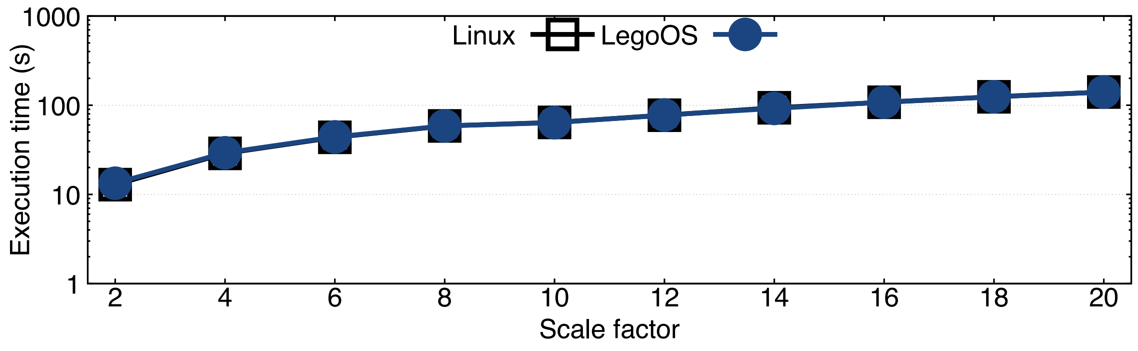
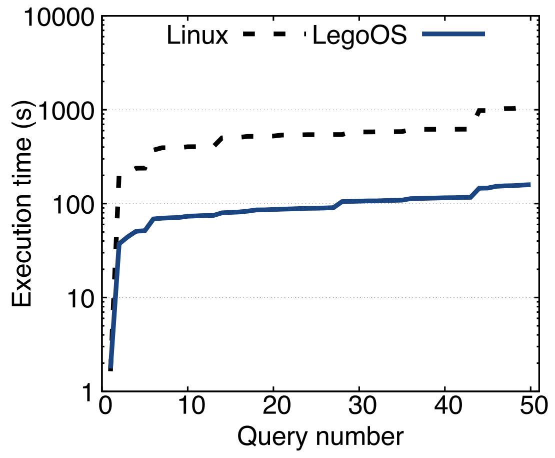


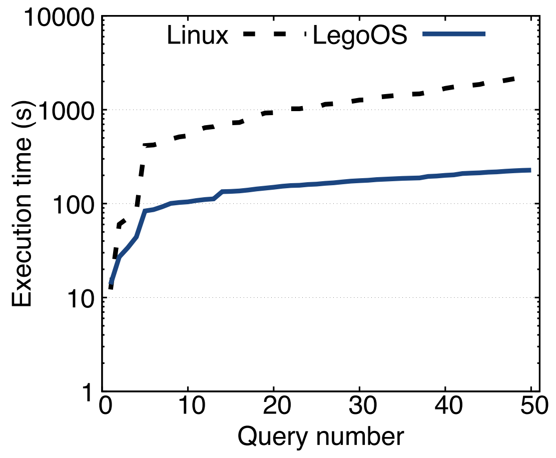
Figure 3.36: Varying dataset size for Query 6.

physical memory on the monolithic server, MonetDB runs faster in the DDC. At SF 20, the speedup is 6.8×. This effect occurs much earlier for Q9, where SF 12 already results in DDCs having a 27× speedup compared to the monolithic server.

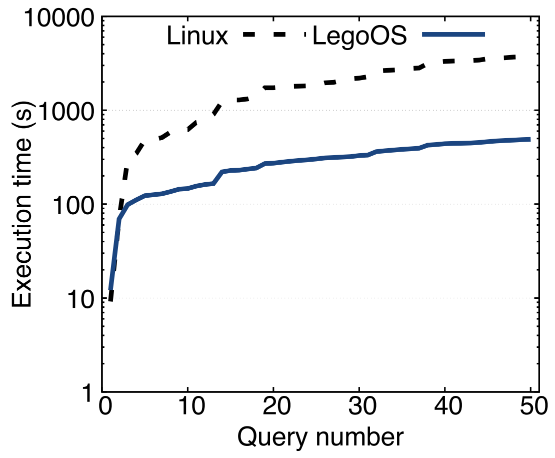
PostgreSQL's cold execution performance, shown in Figures 3.36–**??**, also reflects the results of Section 3.6.1. We omit hot executions as the trends are similar. In each case, LegoOS demonstrates comparable performance to the monolithic machine, except for mem-



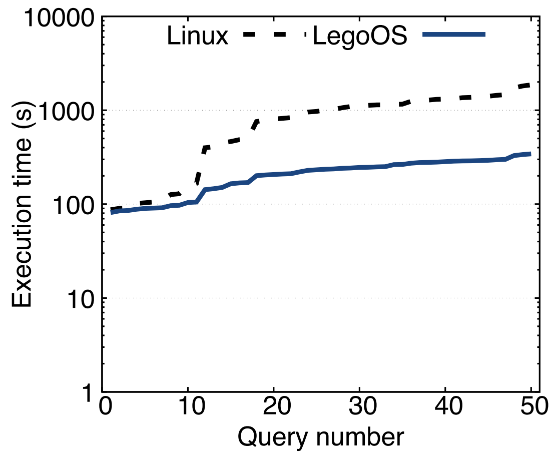
(a) Short-heavy workload.



(b) Medium-heavy workload.



(c) Long-heavy workload.



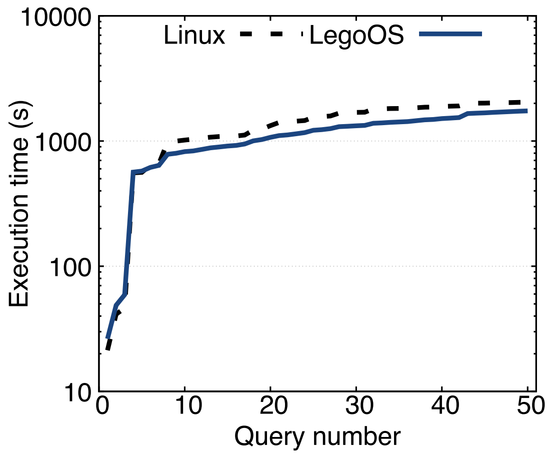
(d) Random mix workload.

Figure 3.37: MonetDB query execution performance in Linux and LegoOS with mixed workloads starting with cold memory.

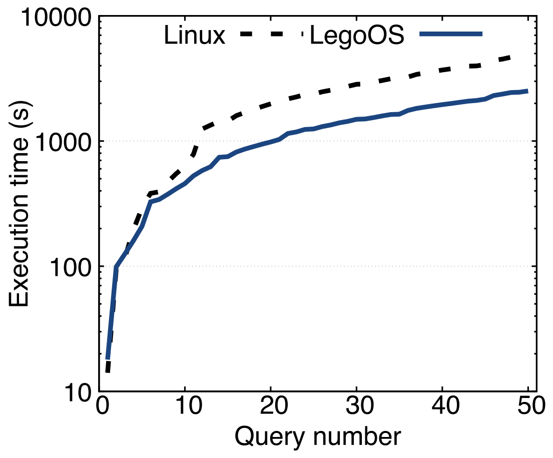
ory-sensitive workloads and large data sets. In these cases, monolithic server performance degrades quickly as soon as local memory is insufficient for the working set. Finally, we note that, in both deployments, PostgreSQL has better overall scaling effects than MonetDB, partially due to the role of disk I/O as the bottleneck.

3.6.3 Large, Compound Workloads

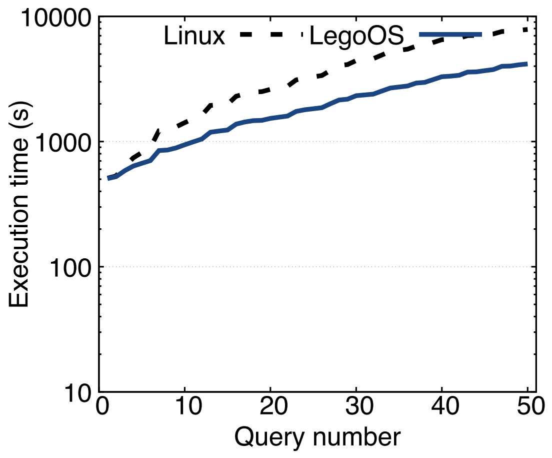
Finally, we extend the above experiments to cases where the workload is large and involves multiple query workloads. Specifically, we fix the physical memory of the monolithic server



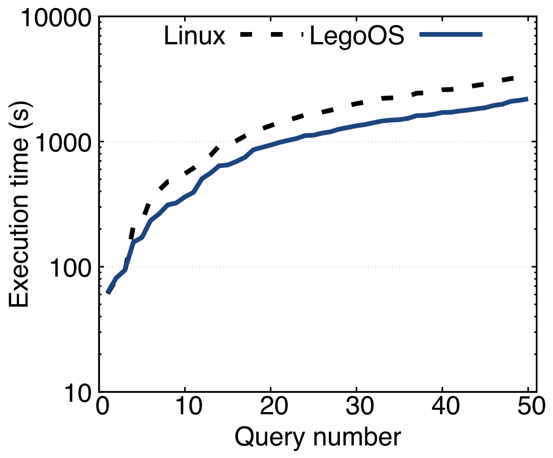
(a) Short-heavy workload.



(b) Medium-heavy workload.



(c) Long-heavy workload.



(d) Random mix workload.

Figure 3.38: PostgreSQL performance with mixed workloads.

and local memory of the DDC processing nodes to 4 GB, and we fix the TPC-H scale factor to 10. In this environment, we randomly draw 50 queries from the set of all 22 TPC-H queries. We classify the queries into three categories by their execution times: short, medium, and long queries. We control the portions of short queries (S), medium queries (M), and long queries (L) to create four configurations: (1) *short-heavy workload:* 80% S, 10% M, and 10% L; (2) *medium-heavy workload:* 10% S, 80% M, and 10% L; (3) *long-heavy workload:* 10% S, 10% M, and 80% L; and (4) *random mix:* each query has equal probability. After the queries are drawn, we permute and evaluate them sequentially in MonetDB and PostgreSQL starting from cold buffer pools, i.e., no prior cached data.

The results are presented in Figure 3.37 and 3.38. The x-axis denotes the query progress within the 50-query trace. The y-axis is the cumulative execution time up to and including that query. In the monolithic server, due to limited memory, MonetDB selects more memory-constrained and less efficient execution plans, while in the DDC, it can take advantage of enough memory in the memory pool to execute the queries more efficiently. For the first few queries, MonetDB has similar performance in both deployments because of the data loading, which is dominated by disk I/O. When more queries have been executed and the buffer pool warms, the DDC becomes increasingly effective compared to today's systems due to its additional remote memory.

The effect is most pronounced in MonetDB, where the short-, medium-, long-heavy, and mixed workloads exhibit total speedups of 6.7×, 9.9×, 7.7×, and 5.4×, respectively. These speedups manifest quickly. For long-heavy workloads, for instance, the speedup is 3.1× at only 5 queries and 6.1× at 25. The relative speedup of PostgreSQL is lower because it uses out-of-core execution. The speedups range from 1.2×–1.9× across all workload mixes.

3.6.4 The Effect of Prefetching

Prefetching the data to be used in the execution from disk can mitigate the I/O bottleneck for out-of-core systems. We evaluate this effect in PostgreSQL through the `pg_prewarm` module, which allows the user to preload specified tables into either the OS cache or the buffer pool. Figure 3.39 shows the results of applying this module in LegoOS with 4 GB local memory. To ease the comparison, we normalized the times of prefetching, execution after prefetching, and hot execution to the cold execution time, and we stacked the first two to show the overhead of prefetching. There are a few interesting findings. Overall, prefetching can effectively cache necessary data: the performance of executions after prefetching matches the performance of hot executions. Although the total times of prefetching and the following execution are generally higher than cold execution times, we can leverage a large memory pool in a DDC to make the prefetching a one-time overhead: prefetching all tables in the memory pool for arbitrary queries.

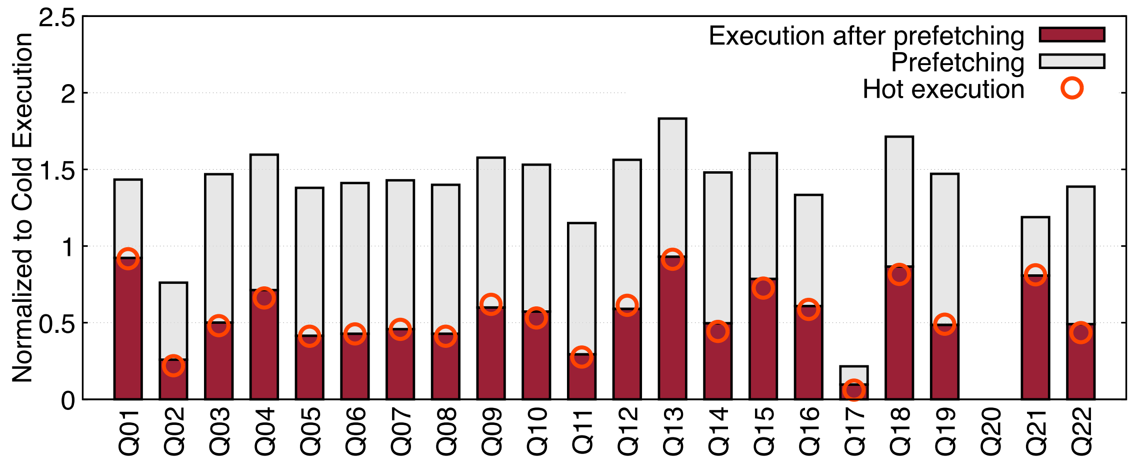


Figure 3.39: The effect of prefetching in PostgreSQL.

3.6.5 Summary

For both in-memory and out-of-core executions, the resource consolidation of DDCs can provide applications with much more memory than a single server. For that reason, resource disaggregation provides better and more stable performance than a single server for DBMSs when the execution reaches the memory limit in the server and they have to spill data to disk.

The experiments of mixed workloads further validate the advantage of disaggregation in having more resources to provision for a single program. This advantage potentially enables DBMSs to scale to much larger workloads than when in a single server.

The disaggregated deployment can also cache additional data in the memory pool, through either historical queries or prefetching. Better caching leads to higher performance.

In addition, we note that the advantage of DDCs does not require any changes in DBMSs. DBMSs can be directly run in a DDC to utilize more resources, while alternative approaches to acquiring more memory, e.g., distributed [85, 217] or RDMA-based [62, 96, 160] DBMSs typically require drastic architectural changes.

3.7 Analysis and Tuning

Section 3.5 shows that with the same resource capacity, DBMS executions are slower in a DDC than in a single server due to higher memory access latency. It also shows that this overhead depends on both the DBMS workload and the degree of disaggregation. In this section, we analyze this overhead through the profiling statistics we acquire in LegoOS and consider how we might tune DBMS performance in DDCs based on this analysis. The goal here is to gain insight into potential ways to modify the behavior of DBMSs to reduce (or mask) the overhead of disaggregation.

3.7.1 Remote Memory Access Analysis

We measure the hardware counters for page faults and InfiniBand communication volume between local memory and remote memory in LegoOS to estimate N_{RM}—the amount of remote memory data transferred (in bytes) during an execution. We first present the results for in-memory executions in MonetDB.

In-memory execution. Figures 3.40–3.42 show N_{RM} for the experiments in Section 3.5.1, where we configured different local memory sizes. Figure 3.40 shows the statistics for the setting where local memory is enough for most queries. We make two key observations: (1) all queries have non-zero N_{RM}; and (2) most queries have N_{RM} smaller than 1 GB.

The first observation suggests that the overhead of disaggregation is inevitable: there are remote memory accesses even when the local memory is larger than what an application demands. Those remote memory accesses include program data and the initial transfer of data into processing nodes' local memory.

The second observation (combined with the $< 2\times$ slowdowns in Section 3.5.1) suggests that, for most queries, the extra latency due to this overhead is smaller than the execution time in a single server. There are, however, a few exceptions: Q17 transfers almost 10 GB data and its execution time is inflated by $2\times$ (shown in Figure 3.10); the N_{RM} of Q9 is 500 MB, which incurs a $2.9\times$ slowdown. For those queries, remote memory accesses dominate the execution

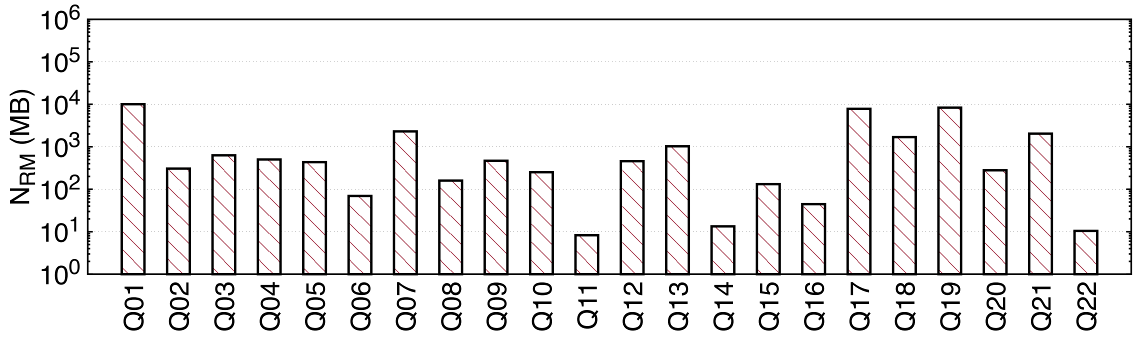


Figure 3.40: N_{RM} in MonetDB executions with 4 GB local memory.

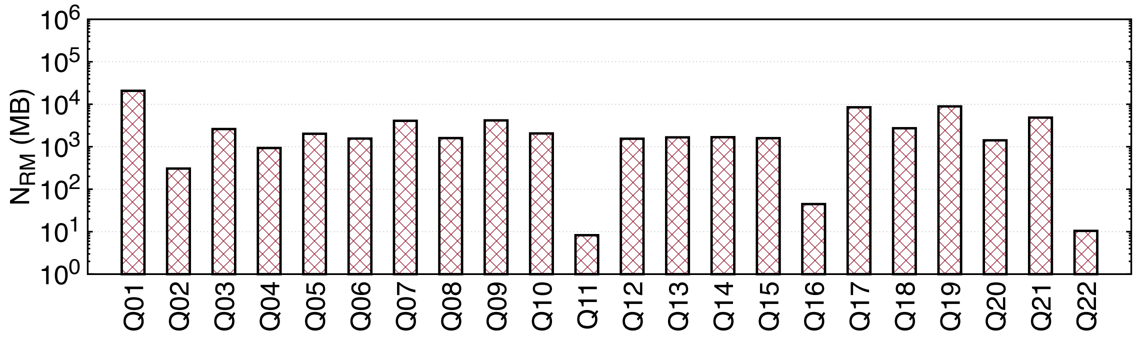


Figure 3.41: N_{RM} with 1 GB local memory.

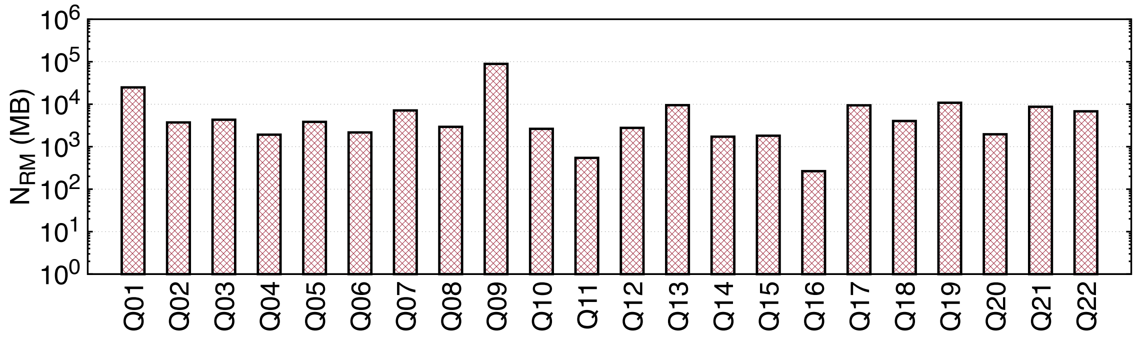


Figure 3.42: N_{RM} with 64 MB local memory.

times. In fact, the impact of memory stalls on the execution time is highly dependent on queries. As examples, Q1 and Q7 transfer 10 GB and 2.3 GB data, respectively, but both of them cause a 1.6× slowdown because computation dominates the execution time. In comparison, Q11 transfers only 8 MB data, but it has a 3.3× slowdown—this suggests that low-latency queries are more sensitive to memory stalls.

Figure 3.41 shows the results when LegoOS has 1 GB local memory. The average and

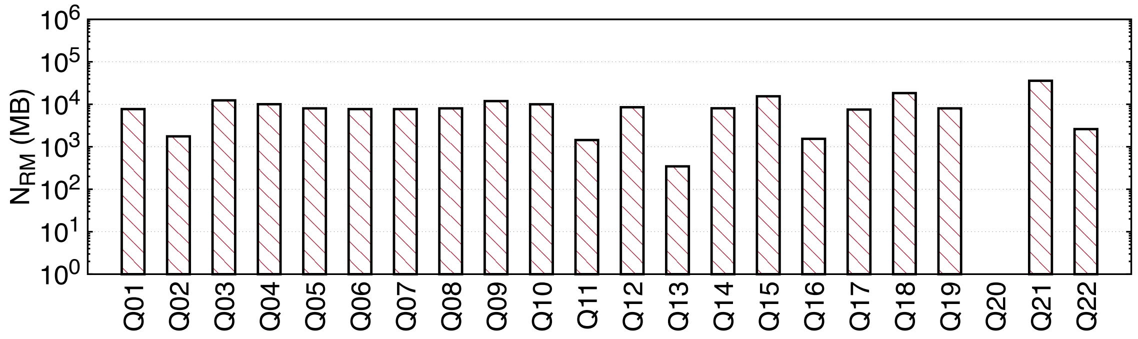


Figure 3.43: N_{RM} in PostgreSQL executions with 4 GB local memory.

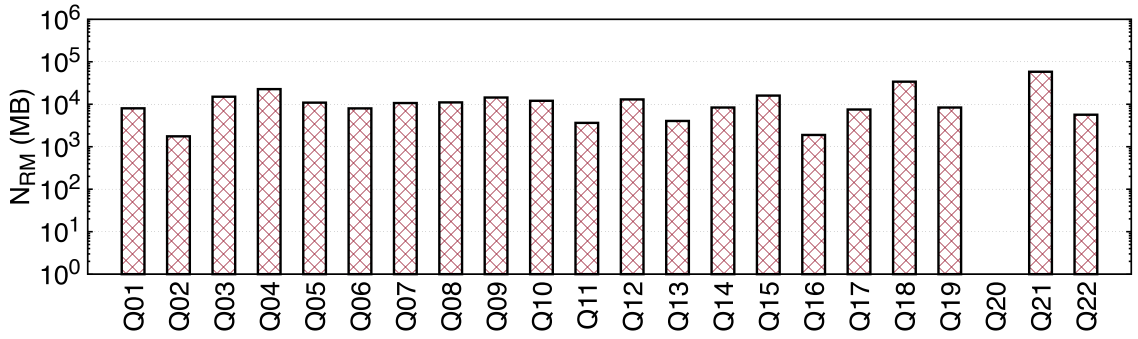


Figure 3.44: N_{RM} in PostgreSQL executions with 1 GB local memory.

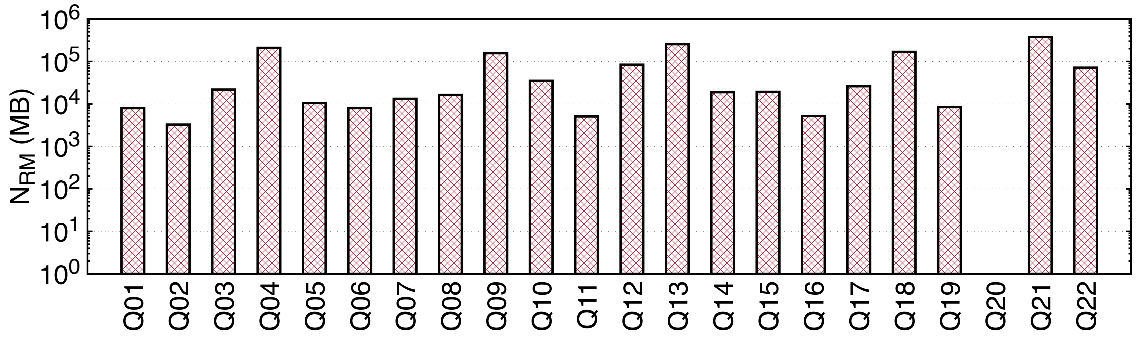


Figure 3.45: N_{RM} in PostgreSQL executions with 64 MB local memory.

median N_{RM} has increased to 3.3 GB and 1.7 GB respectively, significantly higher compared to the case with 4 GB local memory. This is because most TPC-H queries consume more than 1 GB memory, as shown in Figure 3.8, and therefore across the execution, virtually all cached data has to be transferred from the memory pool. This shows the mismatch between DBMS execution and the current OS caching mechanism that uses LRU or its variants (e.g., FIFO). This mismatch results in serious performance degradation: $\sim 5\times$ on average as shown in Figure 3.11. When the local memory size is as low as 64 MB (Figure 3.42), the average N_{RM} increases to 9.1 GB and the median to 3.8 GB, showing that the buffer pool data is accessed multiple times. Those multiple rounds of data transfers cause an order of magnitude performance degradation (Figure 3.12). As an extreme case, Query 9 transfers 89 GB data, causing two orders of magnitude degradation.

Out-of-core execution. For out-of-core executions in PostgreSQL, although the performance slowdowns of cold and hot executions are very different (Figures 3.14–3.16 versus Figures 3.17–3.19), the remote memory accesses are similar. This is because PostgreSQL relies on the OS to cache the raw input data when it reads from files, rather than loading/storing it to virtual memory or its buffer pool directly. LegoOS caches this disk data in memory nodes and storage nodes and not in local memory. As a result, even for hot executions, the DBMS needs to access remote memory for the cached data.

Figure 3.43 plots the results for hot executions in PostgreSQL in a DDC processing node with 4 GB local memory. The average N_{RM} is 9 GB, which is much higher than the peak memory usage derived in Figure 3.9. The reason we did not observe an associated slowdown in the cold executions of Section 3.5.2 is pipelined execution, which hides these accesses by the time spent in disk I/O. The absence of good pipelining is why we do see a slowdown of $\sim 2.2\times$ in the hot executions of Section 3.5.2. Moving to limited local memory sizes (Figures 3.44 and 3.45) increases the average N_{RM}, resulting in a $1.2\times$ and $6.3\times$ increase in the average slowdown, respectively. In the latter case, the PostgreSQL buffer pool is frequently evicted. The queries with the largest N_{RM} (Queries 4, 9, 13, 18, 21, and 22) also have the worst performance slowdowns (cf. Figures 3.14–3.19).

Summary. This set of experiments quantitatively evaluates the overhead of resource disaggregation that comes from remote memory accesses, which, being synchronous, stall the DBMS execution. The results also reveal two mismatches between the current OS design and the DBMS data access patterns: (1) LRU-like local memory eviction policies are a poor fit for DBMSs when the local memory in the processing pool is too small to cache the working set; and (2) out-of-core executions that rely on the OS to cache raw input data from the disk cannot take advantage of processing units' local memory since the cached data is still remote to the processing pool.

We note that the absolute numbers in this analysis are from running existing DBMSs directly in LegoOS—neither the DBMS nor LegoOS is aware of the other side. More flexible data access granularity that leverages the patterns of DBMS workloads can improve data transfer efficiency in LegoOS for DBMS executions, but we leave this optimization as future work.

3.7.2 Plan Optimality

Plan selection is an important function of the query planner and optimizer. We measure the impact of different execution plans on performance. In particular, we focus on two aspects: (1) the size of the buffer pool, and (2) the join algorithm.

Buffer pool size. The size of the buffer pool is a key determining factor for the execution plan chosen by the DBMS. To measure the impact of this choice, we focus on MonetDB for two reasons: (1) as an in-memory DBMS, it is more sensitive to memory size, and (2) the main bottleneck of PostgreSQL is either disk I/O (in cold executions) or network communication overhead incurred by fetching cached data from remote memory (in hot executions), so the size of the buffer pool is not a dominant factor. We study three representative queries: 16, 5, and 9, and evaluate them with the workload configured to a scale factor of 10. We vary the buffer pool size between 16 GB (enough memory), 4 GB (reasonably large) and 1 GB (small), and test two LegoOS configurations: one with 4 GB local memory and one with 64 MB of memory in each processing node. The baseline is a monolithic server with 16 GB memory.

Table 3.1 shows the results. In the monolithic server, a larger buffer pool results in better performance. The DDC results are similar with one exception: Q16 performs marginally better with 1 GB memory than it does with 4 GB (0.7 s vs. 0.73 s), attributed to noise. We also observed that MonetDB's query planner had some difficulty in planning for intermediate buffer pool sizes in Q16.

A somewhat surprising result is that, when moving to smaller buffer pool sizes compared to available memory, the penalty to LegoOS is outsized. When data needs to be spilled out

	Buffer Pool Size	16 GB	4 GB	1 GB
Query 16	Linux	**0.5 s**	0.75 s	0.53 s
	LegoOS (4 GB)	0.73 s	10.78 s	**0.7 s**
	LegoOS (64 MB)	**1.29 s**	11.13 s	1.37 s
Query 5	Linux	**1.14 s**	1.14 s	5.08 s
	LegoOS (4 GB)	**1.44 s**	1.44 s	18.11 s
	LegoOS (64 MB)	**8.72 s**	8.9 s	52.74 s
Query 9	Linux	**1.11 s**	2.2 s	9.65 s
	LegoOS (4 GB)	**1.7 s**	10.55 s	40.81 s
	LegoOS (64 MB)	**178.55 s**	190.8 s	257.53 s

Table 3.1: MonetDB buffer pool size tuning in Linux and LegoOS

of the buffer pool, one might think that the monolithic server would need to spill to disk and that LegoOS, spilling to the remote memory pool would gain an advantage. In fact, it is the opposite. If the buffer pool is less than the total memory of the system, the monolithic server can spill data to memory. That memory is local, unlike the memory to which LegoOS spills data, which is on a remote machine. Thus, spilling data out of the buffer pool comes at a significantly higher cost in a DDC than a traditional system.

Join algorithm. We next evaluate the performance characteristics of join algorithms. We use PostgreSQL because it allows the user to select from three different join algorithms: `Nested-Loop Join`, `Merge Join`, and `Hash Join`. We disable two of them to ensure that PostgreSQL selects the remaining one. For example, to use `Hash Join`, we set `enable_nestloop` and `enable_mergejoin` to `off`. Since Q6 does not involve joins, we evaluate only Q4 and Q13. We use Linux with 16 GB memory as the baseline, configure LegoOS to use a 16 GB memory pool and vary the local memory size between 4 GB and 64 MB.

Table 3.2 shows the results. For Q13, nested-loop join cannot finish within 1 hour in both Linux and any LegoOS setting; merge join has slightly better performance than hash join in this query. Since merge joins incur less random accesses, they are much more efficient than hash joins when local memory is small: merge join incurs 62 GB of remote memory accesses when the local memory is 64 MB, but hash join incurs 250 GB. This is an interesting observation

	Join Algorithm	Nested-Loop	Merge	Hash
Query 13	Linux	>1 h	**14.45 s**	15.69 s*
	LegoOS (4 GB)	>1 h	**18.34 s**	18.51 s*
	LegoOS (64 MB)	>1 h	**122.84 s**	373.49 s*
Query 4	Linux	**3.97 s**	30.55 s	26.5 s*
	LegoOS (4 GB)	**15.95 s**	59.34 s	57.42 s*
	LegoOS (64 MB)	**18.82 s**	351.78 s	348.58 s*

Table 3.2: PostgreSQL join algorithm tuning in Linux and LegoOS. Algorithms marked with * are what PostgreSQL selected.

because PostgreSQL suboptimally selects hash joins when all join algorithms are enabled. Q4 is different: nested-loop join is the best algorithm (i.e., uses the least amount of memory) in both Linux and LegoOS. When running in LegoOS with 64 MB of local memory, nested-loop join, hash join, and merge join incur 8 GB, 134 GB, and 149 GB of remote memory accesses, respectively. Unlike in Q13, hash join performs slightly better than merge join in this query. Again, PostgreSQL suboptimally chooses hash join (over nested-loop join). These experiments show that join algorithms with small memory footprint work better in DDCs, but current DBMSs (PostgreSQL) do not make this choice. It might be possible (and even profitable) to design join algorithms that are tailored for DDCs.

3.7.3 OS Configuration

We now examine how changing different parameters in the underlying disaggregated OS can affect the performance of DBMS. In the previous sections, we observe that the bulk of the overhead comes from accessing remote memory, which could in principle be improved with better caching. We, therefore, experiment with two key choices: the cache eviction and the cache placement policies.

Cache eviction policy. LegoOS supports two eviction policies: FIFO and LRU. Neither one favors DBMSs workloads. We evaluate how these policies affect the execution of MonetDB and PostgreSQL and find that there is little difference in terms of the number and size of remote memory accesses for both eviction policies. As one example, consider the setting

Set Associativity	**1-way**	**256-way**	**8K-way**
MonetDB	22,763,177	22,762,688	22,721,776
PostgreSQL	35,227,319	35,216,064	35,108,257

Table 3.3: Page faults in different set associativity configurations.

where the local memory size is 64 MB (where eviction frequently happens) and where we use 256-way set associativity using the most memory-intensive queries in both MonetDB (Q9) and PostgreSQL (Q4). In this setting, MonetDB with LRU incurs 22,763,745 page faults that trigger remote memory accesses (each page has 4 KB size) while FIFO incurs 22,763,053 page faults. Similarly, in PostgreSQL, LRU is roughly the same as FIFO (35,216,064 vs. 35,216,712) and this is consistent between cold and hot executions. However, as described in LegoOS [235], LRU introduces lock contention on the LRU list; Query 9 in MonetDB finishes in 179.16 s with FIFO and 181.76 s with LRU.

Cache placement policy. To evaluate the effect of cache placement policies, we vary the set associativity for the local memory. We study 1-way, 256-way, and half of fully associative (8K-way for 64 MB local memory, the highest associativity that LegoOS supports). As in the previous experiment, we use the most memory-intensive queries in MonetDB and PostgreSQL and a 64 MB local memory in LegoOS to magnify the effect of caching mechanisms. The results are in Table 3.3.

We find that increasing the set associativity improves the hit rate of the local memory so that the remote memory accesses are reduced in both MonetDB and PostgreSQL. However, the reduction on the remote memory accesses is not significant from the lowest to the highest associativity, and this benefit is offset by the cost of maintaining high set associativity so we do not observe a significant difference between the execution times of different configurations.

In conclusion, switching between current configurations without resource capacity change has little to no effect on data-intensive executions. We leave the codesign of the OS and the DBMSs, which we believe can improve this state of affairs, for the next chapter.

3.8 Proposals on Improving DBMS Performance in DDCs

Despite the elastic benefits of DDCs (Section 3.6), Sections 3.3.3 and 3.5 confirm the significant overhead of disaggregation on query processing. Hence, DDCs prompt us to rethink many aspects of DBMS design, from concurrency control to caching and buffer pool architectures. In this chapter, we focus on a small subset of these issues that affect performance.

We start with the example in Figure 3.3(a). Recall the two main issues: (1) process x's scan of table A leads to two network round-trips: one between x and its memory, and one between the memory and storage. (2) process x then sends the corresponding partition tuples to process y via the network, which must then store them into its own memory. This is particularly problematic because communication over the network can be expensive, and because all the data ends up in the same memory blade anyway! We make the following observation: if x could somehow push the partitioning task to the storage node, so that the storage node could directly place the partitions in the appropriate memory elements, we could cut most data movement.

To achieve this, we draw inspiration from decades of work on *near data processing* [120, 207, 214, 273]. At a high-level, near data processing enhances memory (and in our particular case also storage) with the ability to perform simple operations on the data. In our disaggregated architecture (Figure 3.1) this is possible thanks to the small processing unit (CPU, FPGA, ASIC, or even programmable NIC) attached to storage and memory blades that mediates all accesses. Given this functionality, we propose a new operation that runs at the storage node called Scan-Hash.

Scan-Hash. Scan-Hash streams through a particular table while computing a hash function on the records' join-key. Note that this is simple enough that it can be performed by the CPU or ASIC on the memory and storage blades at high speed. Given this operator, process x would issue a Scan-Hash request to its remote memory, which then does two things: (1) forwards the request to the storage node; (2) places the results provided by the storage node in a memory location based on the returned hash.

At the end of the Scan-Hash operation (Figure 3.3(b), Step 2), the corresponding partitions are stored in memory, all under the ownership of process x (which initiated the Scan-Hash). Note that x is not given the data: when x sends a Scan-Hash to the memory node, the memory node simply stores the result in the appropriate memory locations and notifies x when this task is done.

At this point, x can use the *memory grant* operation proposed in prior work [54]. A memory grant is essentially a way for process x to "gift" some of its remote memory pages to some other process. It consists of x telling the memory controller to change the permissions at the memory node to allow some other process (in this case y) to access those pages. The memory controller then notifies y's OS that new pages have been added to its address space, and the OS propagates this information to y via a signal. The result is that the data is moved exactly once during a partitioning (instead of 4 times): from the storage node into the memory node. This technique generalizes to multiple storage nodes, multiple memory nodes, and other partition strategies. If range partition is preferred over hash partitioning, we can generalize Scan-Hash into a Scan-Partition operation where the application can pass in a partition function.

Collision-avoidance. At the end of the above partitioning protocol, the remote memory contains partitions of records that can be used to build corresponding hash indexes. These indexes can then be probed with records from another table to compute the join. However, as we mention in Section 3.3, accessing hash indexes in remote memory is expensive due to the possibility of collisions. In particular, every time there is a collision, the query processor (x in the above example) must traverse the corresponding bucket by issuing a series of requests to the memory controller. To avoid these costly collisions, we adopt two techniques recently introduced by Aguilera et al. [38]: *indirect addressing* and *HT-Trees*.

Indirect addressing allows the remote memory element to automatically dereference a pointer to determine the corresponding address to load or store, saving one network round trip in our context. Without indirect addressing, a query processor would first have to fetch the memory address stored at the pointer's address, and then fetch the data (leading to two RTTs). In an HT-Tree, leaf nodes store base pointers to hash tables (but not the hash tables themselves).

A query processor can cache the (small) HT-Tree locally, and leave the (large) hash indexes in remote memory. The query processor can then: (1) retrieve a key by traversing the local HT-Tree to obtain the base pointer of the target hash index; (2) apply the hash function to calculate the bucket number; (3) fetch the appropriate value from the target hash and bucket using indirect addressing to follow the pointer in the bucket.

Remote-memory aggregation. Like joins, aggregation can be potentially expensive due to global reshuffling. In a DDC architecture, there are opportunities for in-memory aggregation given that data is consolidated within memory blades. For example, the Scan-Hash mechanism above can be enhanced to hash tuples into buckets based on group-by keys, and a reduction phase applied to each bucket to generate aggregation results. This avoids expensive round-trip times to compute nodes, and in fact, allows us to avoid extensive data shuffling within a rack. If data resides across memory blades on different racks, one can compute intermediate aggregates at the rack level before combining across racks.

3.9 Related Work

A significant part of this dissertation is about resource disaggregation, which offers great benefits for cloud operators, but we speculate that significant changes to disaggregated OSes and applications are crucial to achieve reasonable performance. Our work in Chapter 3 provides the first empirical backing for such claims, with a comprehensive evaluation of production DBMSs on a disaggregated data center setup. A key distinction from others' work is our focus on DBMSs, our experimental setup, and results. We remark, however, that our general call to action—that codesign is not only desirable but necessary in this case—shares a similar ethos to decades-old proposals to codesign OSes and DBMSs [111, 250].

Operating systems for disaggregated data centers. With the advent of DDC hardware, the first wave of software innovation has focused on operating systems support. For example, LegoOS [235] introduces an operating system that decouples hardware resources and connects them with a fast network, while still providing the abstraction of a single machine to applications, which can be run without modifications. Other proposals include new architectures [165, 166], network architectures [75, 110, 242], and data structures for remote memory [38]. Our work is the first to explore application-level optimizations, in this case how DBMS performance is impacted and can be improved by disaggregation.

Remote memory and distributed shared memory. Prior work has revisited the idea of remote memory in fast data center networks [37], proposing a standard API for remote memory access with exported files [36]; implementing generic data structures like vector, map, and queue for remote memory by customizing hardware primitives [38]; and designing a new paging system to avoid application modification [121]. Previous work has proposed novel memory management for DBMSs to utilize remote memories [62, 104, 160] and even provided distributed shared memory abstraction [71, 96, 234]. While remote memory and distributed shared memory, and disaggregation overlap in spirit, the ideas differ in that previous work assumes that significant resources remain coupled with the compute components while disaggregated data centers target completely separating computation and memory. This fundamental difference has implications for buffer management, cost estimation, physical executions, and other important components in DBMSs.

Hierarchical storage management. DDCs have a richer memory and storage hierarchy than traditional distributed environments. Existing work has investigated improving DBMSs on hierarchical storage, such as cache-aware DBMS execution [184, 200], caching [173, 274], and memory management for new hardware [57, 260]. We believe that the existing work would serve as further inspiration for DDC optimizations while noting that the separation between compute and memory represents a radical change unique to DDCs.

3.10 Summary

This chapter initiates the investigation of data processing in disaggregated data centers (DDCs), a new cloud architecture that we believe will become essential in future data center designs. Researchers have identified unique challenges in DDCs for the OS, network, and architecture design. It is also crucial to understand the implications of DDCs for data management. We describe why a naïve adaptation of DBMSs would lead to performance inefficiencies. Through a set of preliminary microchbenchmarks, we validate the inefficiencies. We then conduct a detailed study of two production DBMSs, PostgreSQL and MonetDB, to understand the performance implications of deploying them on a disaggregated data center. We find that a wide variety of factors come into play, including in-memory vs. out-of-core execution, degree of disaggregation, the choice of join algorithms, and buffer pool sizes. We also find that DDCs can actually be beneficial, in some settings, to DBMSs. Based on these findings, we believe DDCs can be a game changer for data processing systems, advocate rethinking how we should design these systems under the paradigm shift of resource disaggregation, and outline possible solutions to the challenges. In the next chapter, we generalize these proposals to an OS primitive that allows data processing systems to eliminate the performance bottleneck in DDCs to unleash their full potentials.

CHAPTER 4

ACHIEVING OPTIMAL DATA PROCESSING PERFORMANCE IN DDCS

Recent proposals for the disaggregation of compute, memory, storage, and accelerators in data centers promise substantial operational benefits. Unfortunately, for resources like memory, this comes at the cost of performance overhead due to the potential insertion of network latency into every load and store operation. As the results and analysis in Chapter 3 show, this effect is particularly felt by data processing systems due to the size of their working sets, the frequency at which they need to access memory, and the relatively low computation per access. This performance impairment offsets the elasticity benefit of disaggregated memory.

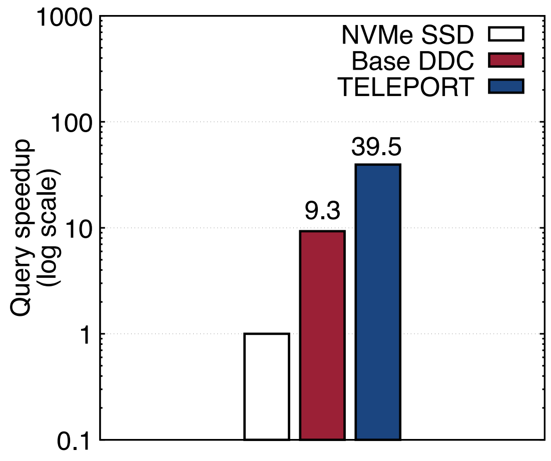
This chapter presents TELEPORT, a *compute pushdown* framework for data processing systems that run on disaggregated architectures; compared to prior work on compute pushdown, TELEPORT is unique in its efficiency and flexibility. We have developed optimization principles for several popular data-intensive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

4.1 Introduction

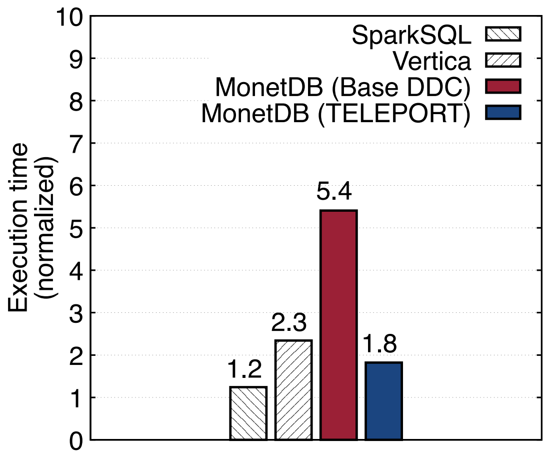
The hardware resources of a disaggregated data center (DDC) are partitioned into physically distinct resource pools all connected via a fast network fabric. This disaggregation of resources promises to fundamentally change the way in which we design and operate cloud infrastructure. This distribution is not only beneficial to the operational and cost efficiency of data centers [259], it also enables more elastic provisioning of resources that expand beyond a single machine. This, in particular, is attractive to data-intensive systems in which the presence of a large memory pool can reduce the amount of data that is spilled to secondary storage, hence improving overall performance. Figure 4.1a demonstrates this benefit empirically (using memory-intensive TPC-H queries): the ability to spill an in-memory query execution to remote memory rather than to a local SSD results in an order of magnitude of performance improvement when memory is constrained (more details in Chapter 3).

There have been a number of recent proposals for resource disaggregation [121, 185, 242]. Some of these propose the complete redesign of applications using novel programming models or custom DBMSs [38, 202, 219, 227, 290]. While these potentially provide good performance in the face of disaggregation, they also typically require radical modifications that block the use of legacy data, applications, and libraries. In contrast, proposals for disaggregated operating systems (OSes) distribute traditional OS responsibilities while emulating the same API/ABI. Applications can, therefore, run with minimal modification. While this, in principle, enables the reuse of existing data-intensive systems like DBMSs and graph processing systems, unfortunately, the performance effects of running these systems unmodified can be significant, offsetting the operation, efficiency, and elasticity benefits of disaggregation.

Chapter 3 presents detailed results and analysis on the overhead of disaggregation. Here, to demonstrate this issue, Figure 4.1b evaluates the *cost of scaling* incurred by DDCs. Specifically, it shows the average execution time of TPC-H queries on several data center configurations compared to a purely local execution that uses same resources (i.e., the same amount of CPU, memory, and disks but all in a single high-end server). For DDCs, we executed MonetDB [19] on two different disaggregated platforms: LegoOS [235], the current



(a) The benefits of DDCs.



(b) The cost of scaling.

Figure 4.1: The benefits and cost of running DBMSs in DDCs.

state-of-the-art disaggregated OS that we investigated in Chapter 3, and TELEPORT, our proposed platform. Both were configured with compute-local memory as 10% of the entire working set. As a reference, we also show the 'cost-of-scaling' for two distributed in-memory DBMSs—SparkSQL [56] and Vertica [6]—running on a more traditional configuration that uses monolithic servers (echoing the study in Section 3.5.3).

The cost of scaling in these systems is a result of the insertion of network communication into execution—in the form of paging to/from remote memory in the case of DDCs, and in the form of message passing in the case of distributed execution. Distributed data processing systems—having been thoroughly optimized over decades—successfully achieve a reasonable 'cost of scaling' (average costs are 1.2× and 2.3× in SparkSQL and Vertica, respectively). The cost of the unmodified execution in a state-of-the-art DDC is, unfortunately, significantly higher: 5.4× on average. As we showed in Section 3.5, this cost can, in the worst case, balloon to two orders of magnitude for some common data analytics tasks. This is despite OS-level optimizations in existing DDC platforms such as caching and prefetching which, on their own, are insufficient.

How can we enable all of the operation, efficiency, and usability benefits of DDCs while ensuring a comparable 'cost-of-scaling' to traditional distributed architectures? This chapter generalizes the proposals in Section 3.8, and our answer is TELEPORT, a novel OS kernel primitive for DDCs that enables—with a single system call, minimal overhead, and no other application changes—data-intensive systems to choose where to execute their application logic. Conceptually, TELEPORT's primitive resembles that of compute pushdown: applications can choose to ship complete function calls to remote memory where the functions can execute using local data. For memory-bound tasks, proximity can improve performance by orders of magnitude. For many such operations, minimal computation is required, maintaining the disaggregation of compute and data in the memory pool. As a preview of TELEPORT's benefits, Figure 4.1b shows that TELEPORT can significantly lower the cost of scaling with DDCs and, as a result, can truly unlock the benefits of DDCs (Figure 4.1a).

TELEPORT differs from prior work on compute pushdown [97, 99, 113, 181, 207, 214, 249] in its focus on the novel environment of memory disaggregation, in which a process's entire address space resides in the remote memory pool, including the text segment, heap, stack, and full page table—compute-local memory is nothing more than a cache. Assuming a consistent instruction set architecture (ISA) across the compute and the memory pools (but not necessarily homogeneous hardware), applying TELEPORT to offload a piece of computation to the memory pool is as straightforward as pointing a process running in the memory pool to the correct program counter, stack, and page table residing in the cache of the compute pool. Not only is this more efficient than traditional pushdown mechanisms, it allows for the use of pointers, complex data structures, and open files—the capabilities of a local function—without additional user effort. TELEPORT's target level of flexibility and ease of use also leads to new challenges unaddressed in prior compute pushdown proposals. For instance, in order to achieve good performance and correctness, updates must be propagated lazily, yet correctly, so as to ensure memory consistency in the presence of distributed execution over a shared process context.

In summary, this chapter makes the following contributions:

- We introduce the design and implementation of TELEPORT, a compute pushdown primitive in the OS kernel designed for optimizing data-intensive systems for resource disaggregation. It presents a uniquely flexible and usable abstraction for mitigating overheads from excessive remote memory accesses.

- To handle parallel threads, we describe a set of specialized synchronization primitives (inspired by prior work on MESI cache coherence [211]) that guarantees memory coherence of a logical process context shared across resource pools and multiple concurrent threads within each pool.

- Finally, we present a set of pushdown-optimized data-intensive systems (DBMS, graph processing, and MapReduce). Applying TELEPORT only involved the selective wrapping of existing function calls with TELEPORT's primitive. These optimized systems are an order of magnitude faster than a state-of-the-art disaggregated OS with minimal code changes, even when the memory pool has limited CPU capacity.

4.2 Background

We provided an extensive background on DDCs in Chapter 3. This section expands the discussion on disaggregated OSes and generalizes the implication of DDCs to more types of data processing systems.

4.2.1 Disaggregated Operating Systems

A disaggregated OS inherits all traditional OS concepts (program contexts, resource allocation, file systems, and isolation) and the original API/ABI. Underneath, the OS implements these functionalities using disaggregated hardware resources. It hides the complexity of infrastructure changes from the data center applications, hence ensuring backward compatibility for big software systems like DBMSs [19, 29, 56] and graph processing systems [114, 79], which have been developed over many years and consist of up to million lines of code [12, 19, 29]. The OS approach is thus more appealing compared to alternatives that either require new programming models [38, 202, 219, 227] or only share subsets of memory [121, 185].

Regardless of the specific architecture, disaggregated OSes allow the complete decoupling of compute and data. Application data in virtual memory spaces resides in the memory pool. The compute pool schedules and executes worker threads/processes with its local memory caching data from the memory pool. This clean separation enables a great benefit—*independent elasticity* with no extra effort, where programs can use an arbitrary number of CPU cores and, independently, allocate arbitrary amounts of memory and storage. For example, DBMSs can create a database of any size in the storage pool, allocate a buffer pool of any size to hold the working set in the memory pool, and spawn any number of query execution workers in the compute pool. Thus, to read a new piece of data from persistent storage, the user-level process in the compute pool will trigger a page fault on its local cache. This page fault is forwarded to the controller in the memory pool, which checks the process's full page table and triggers a recursive page fault that forwards the request to the storage pool. Finally, the requested page will flow back to the CPU node in the reverse direction: the memory controller will page in the data and update the process's page table, and the CPU node will bring that page into its local cache. The whole process is mediated by the disaggregated OS. Traditional OSes would execute these operations in a single machine.

An example of this approach is LegoOS [235], which proposes a *splitkernel* OS. It 'splits' kernel responsibilities across resource-disaggregated nodes, e.g., the piece of the kernel on each compute node manages the process and scheduling of a traditional Linux server, while the piece on each memory node focuses primarily on memory management. While our TELEPORT prototype is implemented on top of LegoOS, its core ideas go beyond a specific OS and can apply to any disaggregated OS that provides complete compute and data decoupling.

4.2.2 System Performance in DDCs

Figure 4.1 shows the benefits of DDCs' large disaggregated memory pools but also their cost of scaling. This cost is particularly pronounced for data processing systems due to frequent memory accesses and the fact that local DRAM accesses are an order of magnitude faster than network communications such as RDMA. Consider the following examples:

Database systems. This part reflects our insights in Chapter 3, especially Section 3.5. DBMSs are designed to execute SQL queries with low latency and high throughput. Data that is

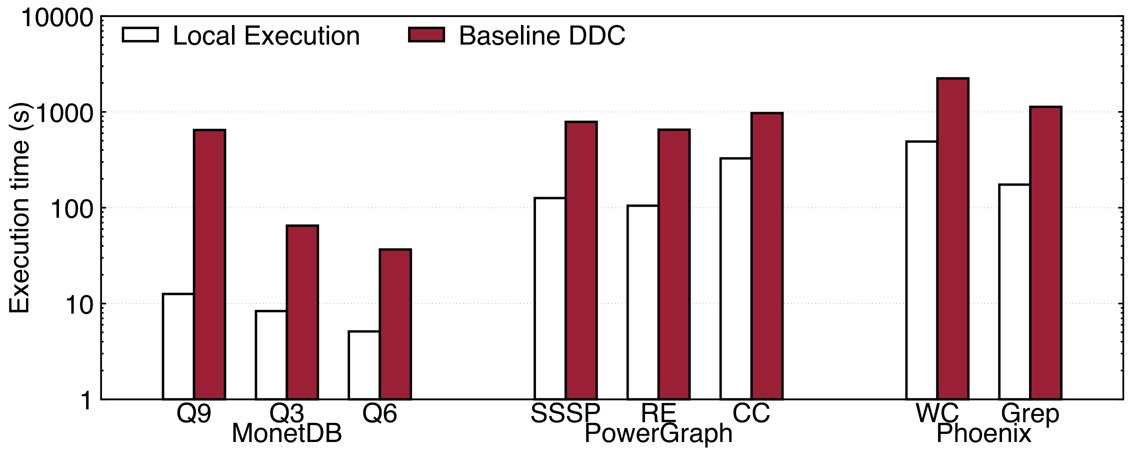
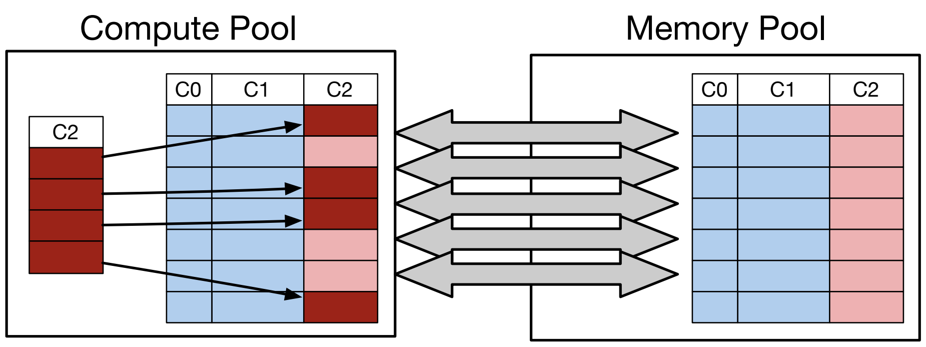


Figure 4.2: DDC performance overhead compared to a monolithic server.

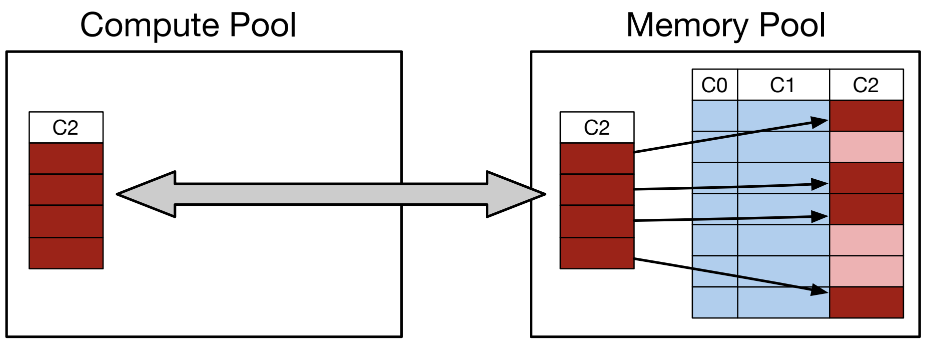
actively used is kept in an in-memory buffer pool to avoid slow disk I/O. In DDCs, however, query execution happens in the compute pool while the buffer pool data lives in remote memory. This arrangement can be expensive. For example, a binary hash join (1) scans the tuples in the outer table, (2) probes the hash index of the inner table, and (3) generates the join results. The random accesses in step (2) can result in substantial cache misses, while step (1) and (3) are a poor fit for typical LRU-based caching strategies [251]. Previous work [73, 235] shows that queries can take up to two orders of magnitude longer to complete (compared to a purely local-memory deployment) for precisely these reasons when the degree of compute-memory disaggregation is high.

Graph processing. Systems like Pregel [183] and PowerGraph [114] process structured pointer-based graph datasets that lead to unpredictable memory access to different parts of the input graphs depending on the query and data characteristics. In PowerGraph, for example, every gather-apply-scatter iteration requires a vertex to communicate with its neighbors to exchange local data for the next round. In a traditional server, this is simply a set of local accesses; in a DDC, each iteration requires expensive remote memory accesses for large graph states.

Data-parallel frameworks. MapReduce-like systems such as Phoenix [148] have interleaved stages of memory-intensive operations. After each processing stage, workers exchange their results with the next set of workers. When co-located on the same server, the communication



(a) Executing a selection in a DDC.



(b) Offloading the selection to the memory pool saves data movement.

Figure 4.3: A data-intensive relational operator example, selection, in DDCs.

is fast; however, in a DDC, intermediate results must all be written just to be fetched back for the next iteration [53].

Summary. In short, these memory-intensive systems have a set of core processing primitives that are computationally lightweight but involve a high degree of memory accesses. Figure 4.2 shows the results of running typical data-intensive queries in a DDC testbed managed by a state-of-the-art disaggregated OS. Slowdowns range from 5× up to 52.4×. Analysis in Chapter 3 shows that remote memory accesses dominate the slowdowns of these systems, and we argue that *the slowdowns are unavoidable with a constrained cache size in the compute pool*. Our position is that, by optimizing the placement of computation, TELEPORT can dramatically improve performance.

4.2.3 Benefits of Computation Pushdown

In this chapter, we focus on alleviating the memory bottleneck by selectively performing computation pushdown from compute to memory pools. To understand the potential benefits, consider the 'selection' operator mentioned in the previous section. Using MonetDB as an example, the implementation of selection takes as input (1) a table, (2) the filter to be applied on the tuples in the table, and (3) an optional candidate list that is the result of previous selections. It performs a full sequential scan of the table and applies the filter. Every tuple that passes the filter is then materialized to a temporary table.

Figure 4.3a depicts how this process would unfold in a DDC: assuming that the working set does not fit in compute-local cache, the selection process needs to bring all tuples in the original table from the buffer pool in the memory pool, resulting in massive data migration, and thus significant execution time increase.

If, instead, we were to migrate this simple, but data-intensive compute operation to the memory pool (Figure 4.3b), the accesses to the original table are all *in situ*, resulting in minimal communication between the compute and memory pools. Even if the computational power of the memory pool is low, most selection filters are computationally inexpensive to run. Hence, the memory-bound nature of the operator would ensure a performance benefit.

4.3 Design of TELEPORT

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

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

4.3.1 The TELEPORT Abstraction

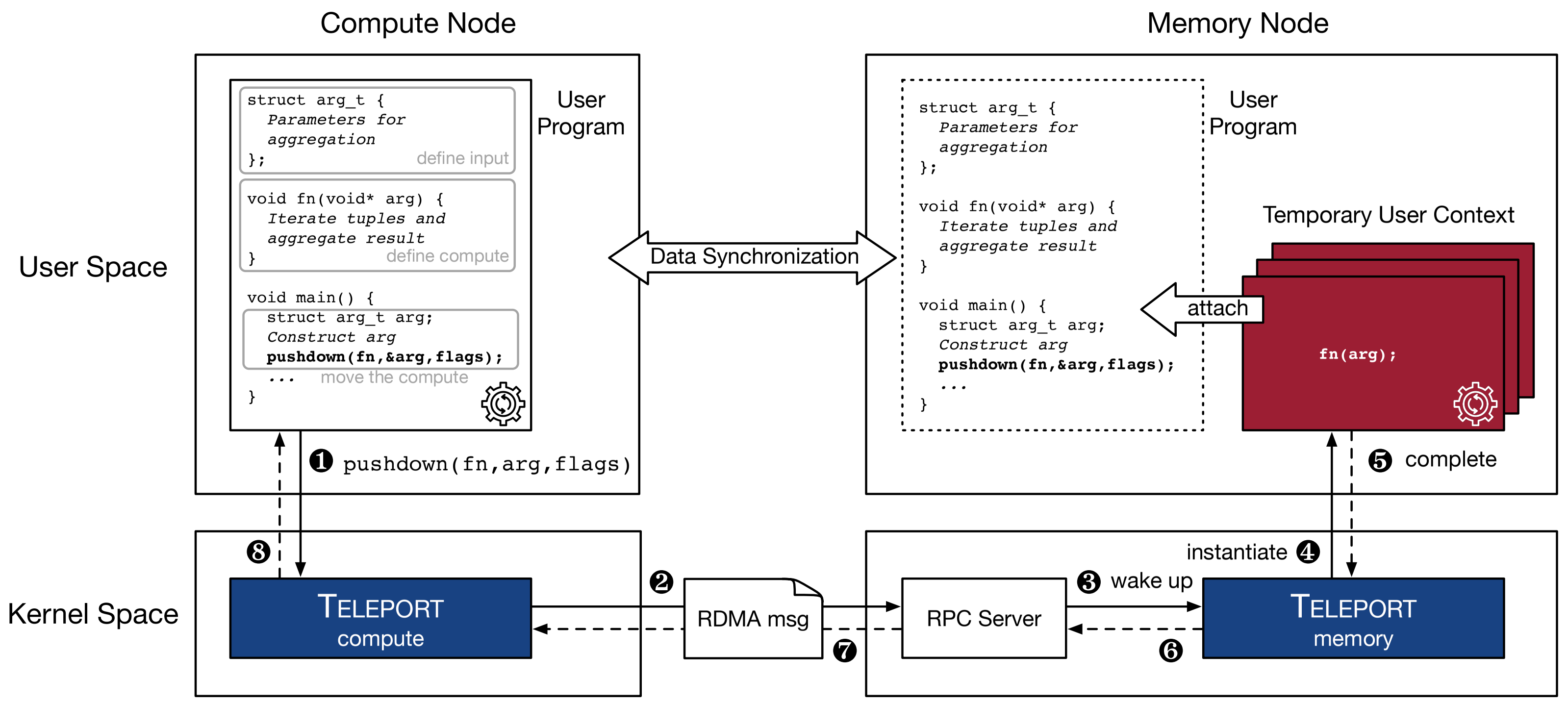
Using TELEPORT, user applications running in the compute pool in a DDC can push arbitrary functions to the memory pool. While prior work has explored, extensively, the concept of compute pushdown in various contexts, TELEPORT is unique in its ability to provide, to pushdown code, unfettered access to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

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

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

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

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



```
struct arg_t {
  Parameters for
  aggregation
};

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

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

Figure 4.4: TELEPORT architecture.

4.3.2 TELEPORTing the Computation

In this subsection, we describe the operation of TELEPORT assuming perfect synchronization of memory stores between the compute pool and memory pool. Later in Section 4.4, we describe how synchronization is implemented in TELEPORT.

Figure 4.4 shows the process of migrating a function. When the application calls the `pushdown` syscall (❶), the application thread stalls and both pointers (`fn` and `arg`) are passed to the compute-pool instance of TELEPORT in the kernel space. The instance then packs the parameters into a pushdown request and sends it to the memory pool's controller using an RDMA write operation that implements a low-latency RPC mechanism (❷).

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

TELEPORT *attaches* the temporary user context to the virtual memory space of the caller application by borrowing the page table of the caller and setting it as the table of the newly created user context. This procedure is akin to the POSIX `vfork` function in that it creates a new process but the virtual address space, file descriptor table, and other parts of the process image are not cloned—rather, the new process shares the resources of the original. Compared to a traditional fork, this procedure is more efficient as the memory pages are neither copied nor set to read-only. Furthermore, memory modifications are supported through the techniques in Section 4.4 and returning from a function simply returns execution to the TELEPORT stub. The end result is that *the temporary context is able to access any code and data of the caller*, specifically `fn`, `arg`, and any data processed by `fn`. Inside the context, `fn` is called with `arg` as the input. Internally, the function dereferences the parameters from the argument pointer and starts the execution.

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

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

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

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

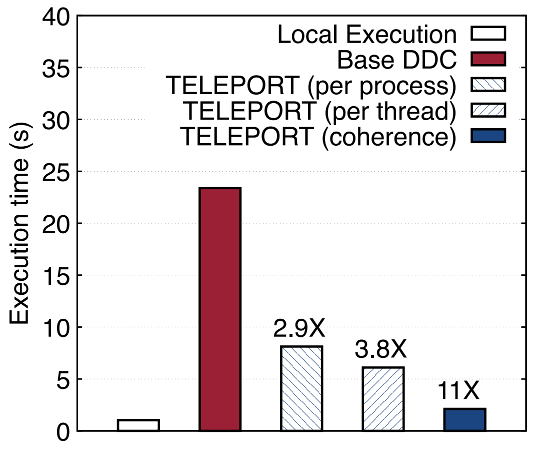


Figure 4.5: Application performance in different systems and with different data sync approaches in TELEPORT.

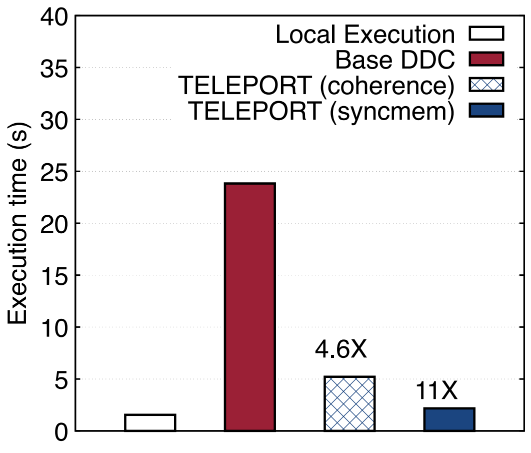


Figure 4.6: The benefit of a manual data sync with `syncmem` when false sharing occurs in the application.

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

4.4 Data Synchronization

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

- *Before pushdown*, where the compute pool may have modifications in its local memory that have not yet been flushed to the memory pool. Changes must be synchronized to ensure that pushdown operates on fresh data.
- *After pushdown*, where the compute pool's cache may be stale. When execution returns

to the compute pool, modified pages should be synchronized back as well.

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

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

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

TELEPORT instead minimizes the amount of data transmitted before, during, and after pushdown. By default, TELEPORT does not transfer any pages when initiating a pushdown. Instead, consistency is kept between the compute and temporary-context page tables with a write-invalidate coherence protocol inspired by MESI [211]. Applications can also instruct TELEPORT to use weaker memory consistency models via optional flag parameters.

To illustrate the importance of TELEPORT's techniques, we consider a microbenchmark involving an application with two threads: a compute-intensive thread performing arithmetic calculations (e.g., expression evaluation in a database query) and a memory-intensive thread randomly accessing a 50 GB memory space (e.g., probing a hash table). The results of the ablation study are in Figure 4.5. When the application runs locally in Linux, each thread finishes in 1s. In the baseline DDC, however, execution slows to 23s because of the memory-intensive thread. Pushdown using the above naïve, full-process approach can speed this up

```
Function Invalidate(pte, write):
    if write then
        pte.present ← False                                # Invalidate this page
    else
        pte.writable ← False                               # Write-protect this page

Function MemorySetup(tmp_context, compute_pgs):
    $t\_mm$ ← Clone of the caller's full page table
    foreach pte in $t\_mm$ do
        c_pte ← compute_pgs[pte.address]                   # Look up this page in the list
        if not c_pte.present then
            Continue                                       # This page is not in the compute pool
        Invalidate(pte, $c\_pte$.writable)                 # Call invalidation on this page
    Set $t\_mm$ as the active page table of tmp_context
```

Figure 4.7: Preparation of the page tables before pushdown execution in the memory pool. $compute\_pgs$ is the transmitted list of pages from the compute pool.

by 2.9×. Separating the two threads and only pushing the memory-intensive thread (and only evicting its memory) does slightly better with a 3.8× speedup over the baseline DDC. With TELEPORT's default coherence, however, the synchronization overhead and the gap between local execution and DDCs are minimized, resulting in a jump up to an 11× speedup.

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

4.4.1 On-demand Memory Synchronization

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

Temporary-context page table construction. When the `pushdown` function is called, the compute pool begins by building a list of memory pages that are either currently in local

```
1  Function ComputeOnPageFault(address, write):
2      ForwardToMemory(address, write)            # Send this request to memory pool

3  Function MemoryOnPageRequest(address, write):
4      $mm$ ← process's full page table
5      $t\_mm$ ← temporary context's page table
6      if not mm[address].present or (write and not mm[address].writable) then
7          ForwardToStorage(address, write)       # Send this request to storage pool
8      pte ← $t\_mm$[address]                      # Get page table entry
9      Invalidate(pte, write)                     # Call invalidation on this page
10     ReturnToCompute(*pte)                      # Send this page back to compute pool
```

Figure 4.8: Handling compute-pool page faults during pushdown.

memory or that have an outstanding page fault request. This list of memory pages and their write permissions are sent to the memory pool as parameters to the pushdown RPC call.

When TELEPORT instantiates the temporary context in the memory pool, it uses the list in the procedure outlined in Figure 4.7. Specifically, when building the context, TELEPORT clones the page table of the caller thread. This cloned page table is identical to the process's page table, except that any writable page in the compute pool is excluded (Figure 4.7, line 3) and any read-only page in the compute pool is also set to read-only locally (Figure 4.7, line 5). In effect, this guarantees that the system begins with the invariant stated at the beginning of this subsection: for each page, (a) the page is writable and only in the compute pool, (b) the page is writable and only in the temporary context in the memory pool, or (c) the page is read-only and can exist in any context.

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

On a compute-pool page fault, the fault is forwarded immediately to the memory pool as normal; however, the corresponding page fault handler in the memory controller changes slightly during pushdown (Figure 4.8, lines 3–10). Specifically, after ensuring that the page is in the temporary context page table, the controller executes an operation similar to Fig-

```
Function MemoryOnPageFault(address, write):
    $mm$ ← process's full page table
    $t\_mm$ ← temporary context's page table
    if not mm[address].present or (write and not mm[address].writable) then
        ForwardToStorage(address, write)        # Send this request to storage pool
    else
        ForwardToCompute(address, write)        # Send this request to compute pool

Function ComputeOnPageRequest(address, write):
    $c\_mm$ ← local page table of the caller
    pte ← $c\_mm$[address]                       # Get page table entry
    if write then
        Evict(pte)              # Invalidate this page and send it back to memory pool
    else
        pte.writable ← False                     # Write-protect this page
        NotifyMemory(address)        # Notify memory pool of this write protection
```

Figure 4.9: Handling memory-pool page faults during pushdown.

ure 4.7, removing the page from the temporary context if the compute pool requested write permissions, or setting it to read-only if it requested only read permissions.

Temporary-context page faults are handled similarly (Figure 4.9), except that we must distinguish between a 'true' page fault, which should be forwarded to the storage pool and a pushdown-related page fault, which invalidates the cached pages in the compute pool. TELEPORT distinguishes this by checking the full page table and the temporary context's page table, both stored locally in the memory pool. Evictions from the memory pool to the storage preserve the correct page table entry (*pte*) dirty bits.

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

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

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

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

Correctness. The correctness of our protocol follows directly from our adherence to the Single-Writer-Multiple-Reader (SWMR) invariant [201]. Just like MESI, writes in TELEPORT are serialized as there is ever only a single writer in the system, and writes are propagated when the other node explicitly invalidates the writer's exclusive 'lock.' Cache coherence makes our system transparently compatible with existing data processing system architectures and their memory consistency models.

4.4.2 Alternative Coherence Mechanisms

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

TELEPORT also provides options (specified via `flags`) for coherence protocols that support weaker memory consistency models. These improve performance, but should be used carefully by programmers to ensure correctness. One simple relaxation is to disable the coherence entirely. This might be useful, for example, if the user wants to manually synchronize pages. An example use is to handle false sharing, which occurs when threads in the compute and memory pool access data (either variables on the stack or allocated memory on the heap) that are not shared but that reside on the same page. Although false sharing is uncommon, Figure 4.6 shows that when it occurs, it can negatively affect the pushdown performance. In this case, users can disable the coherence protocol and manually synchronize the data with `syncmem` at a finer granularity.

Another relaxation follows the default coherence mechanism. Rather than removing pages when the other pool requests write permissions, TELEPORT sets them as read-only. Effectively, this arrangement maintains write serialization for individual memory locations, but relaxes the guarantees of write propagation. Combined with typical processor memory consistency models, this relaxation amounts to an implementation of Partial Store Ordering (PSO) [131]. Again, for applications that can take advantage of this relaxed consistency model (e.g., by converting important reads to RMW instructions or memory fences to explicit synchronization of modified page lists), it may provide better performance. Section 4.7.6 provides a more detailed evaluation of the coherence protocol and benefits of the relaxations when the data contention rate is high.

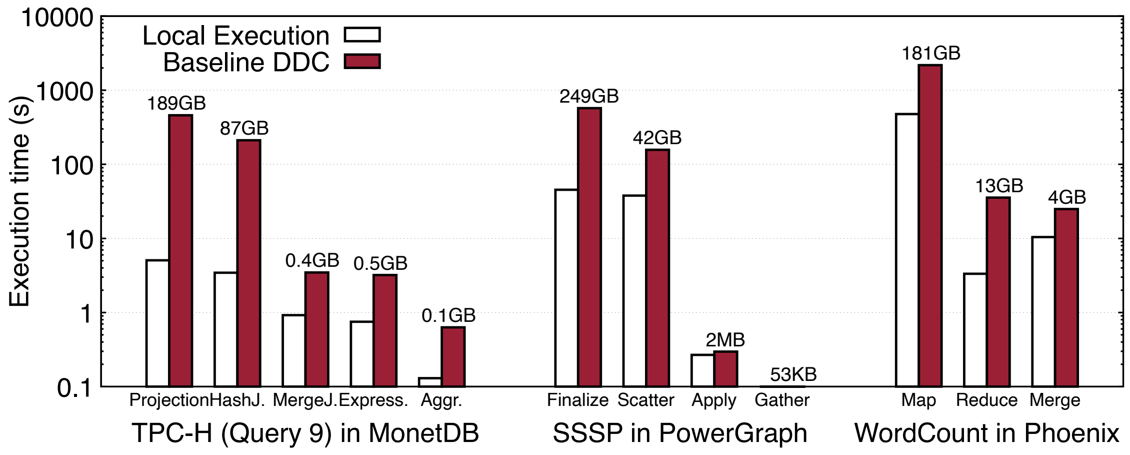


Figure 4.10: Performance breakdown of the query with the greatest cost of scaling in DDCs in each system. For every operator/phase, we show the times in both local and DDC executions and the remote memory accesses in the DDC. A common pattern is that there are one or two arbitrary operators/components dominating the overall query execution time.

4.5 Applying TELEPORT

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

4.5.1 In-memory Database

To evaluate database workloads with TELEPORT, we select MonetDB [19], a columnar in-memory DBMS that provides high performance processing for analytical queries.

Filtering/summarization operators. Several commonly used operators such as *projection*, *aggregation*, and *selection* require simple computation but process a large number of tuples. Further, the main purpose of those operators is to filter/summarize tuples: the result set is typically much smaller than the input (projected column in *projection*, matching tuples in *selection*, and sub-aggregates in *aggregation*). Hence, users should push these operators down,

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

Figure 4.11: The flexibility of TELEPORT enables the pushdown of various memory-intensive operators in existing systems with minimal modification.

particularly when they are highly selective; similar observations were made in the context of disaggregated storage [199, 277]. In DDCs, however, they can be pushed to the memory pool so that the compute pool only receives the summaries for further processing.

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

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

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

Complex queries. A more complex case is Query 9, the most expensive query in the TPC-H

```
void project_int(tuple *res, tuple *ref, tuple *tbl) {
    // actual projection implementation
}

tuple *project_perator(tuple *ref, tuple *table) {
    type t = get_type(table);
    tuple *result = new_column(t, count(ref));
    switch (t) {
        case TYPE_int:
            project_int(result, ref, table);
            break;
        // other types
    }
    return result;
}
```

Figure 4.12: Original projection code in MonetDB.

benchmark in Figure 4.2. Figure 4.10 breaks down its execution time in disaggregated execution (compute-local memory is configured to be 1 GB) with a scale factor of 50 into its constituent operator types. We observe that in addition to the memory-intensive projection operator, hash join also incurs significant remote memory accesses and bottlenecks the overall performance. While hash join tends to have relatively high computational requirements when run on a traditional OS, in a DDC, it becomes severely memory-bound due to random accesses to the hash index. As such, it is a strong candidate for pushdown, as the results in Section 4.7 will later verify. Other operations such as merge join and expressions also experienced degradation in disaggregation, but they are not blockers to end-to-end query performance.

Code modification. Finally, an important criterion for pushdown is the complexity of application changes. Figure 4.11 summarizes the amount of changes required to support each operator pushdown in MonetDB, as well as the size of the specific pushdown function. We observe that modifications across all operators are negligible relative to MonetDB's code base ($\sim$400K LoC), and the amount of code executed in the memory pool is under 100 lines. Figures 4.12 and 4.13 show an example use of TELEPORT—a side-by-side comparison between the original implementation of the projection operator in MonetDB and the pushdown rewrit-

```c
typedef struct {
    tuple *result;
    tuple *ref;
    tuple *table;
} arg_struct;

long fn(void *arg_) {
    arg_struct *arg = (arg_struct *)arg_;
    project_int(arg->result,arg->ref,arg->table);
    return 0l;
}

tuple *project_operator(tuple *ref, tuple *table) {
    type t = get_type(table);
    tuple *result = new_column(t, count(ref));
    switch (t) {
        case TYPE_int:
            arg_struct arg;
            arg.result = result;
            arg.ref = ref;
            arg.table = table;
            pushdown(fn, (void *)&arg, 1 << LAZY_SYNC);
            break;
        // other types
    }
    return result;
}
```

Figure 4.13: TELEPORTing this memory-intensive operator is intuitive.

ten version. Most of the code centers around creating and accessing `arg`.

Automatic query optimization. Automating the porting process is achievable via static analysis and code transformation, given the structured nature of relational operators. An interesting and challenging task is to automatically decide which operators should be pushed down at runtime. There are general trade-offs in applying compute pushdown in DDCs: offloading an operator close to data can reduce the cost of data movement between pools, but it can also incur pushdown overhead, including shipping the operator, potential data synchronization, and degraded computation power. Section 4.7.4 evaluates the impact of these trade-offs and a potential metric for determining the viability of an operator for pushdown. We note, however, that the optimal plan of pushdown is determined by various factors: operator characteristics, workloads, and the DDC configuration.

A potential solution is a DDC-aware query optimizer that captures the resource constraints in different resource pools and finds the optimal plan for operator placement. This chapter focuses on the TELEPORT mechanism and leaves a full investigation of DDC query optimizer design to future work.

4.5.2 Graph Processing

To showcase another challenging data-intensive workload, we look at PowerGraph [12], a high-performance in-memory graph processing system. Similar to prior DDC settings [235, 266], we run PowerGraph in the compute pool and utilize multiple threads as compute workers. The main graph state is in the memory pool.

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

Figure 4.10 shows the time breakdown of this execution on a real-world social network graph [276]. *finalize* and *scatter* account for most of the overhead, although the *gather* phase can also bottleneck other applications, e.g., PageRank. All three components can be TELEPORTed with fewer than 100 lines of code each (see Table 4.11).

4.5.3 MapReduce

Our third use case is Phoenix [148], a native, shared-memory MapReduce system. In Phoenix, there are *map*, *reduce*, and *merge* phases. The *map* phase performs the actual map computation, generates key-value records, and shuffles the records to the reduce workers. We observe that the *map* phase is normally the bottleneck in a DDC because of the shuffle operation, a data-intensive sub-component.

Revisiting Figure 4.10 (the last group), we can examine the performance breakdown of `WordCount` in Phoenix. In this figure, as a point of comparison, we include *reduce* and *merge* execution times as well. We observe that the *map* phase experienced much greater remote memory accesses compared to other phases. The *map* phase, however, is computationally expensive as a whole. To push down only data-intensive operations at a finer granularity, we further divide the *map* phase into *map-compute*, which applies the user-defined map function and generates key-value records, and *map-shuffle*, which shuffles the records among reduce tasks, sub-phases. The *map-shuffle* dominates the running time in DDC execution—95% of *map* time. This suggests moving the *map-shuffle* phase close to the data, which we achieve with minimal code changes (see Figure 4.11); the pushdown function requires only 28 lines of code.

4.6 Implementation

We have implemented a prototype of TELEPORT[1] on top of LegoOS [16]. Similar to LegoOS, TELEPORT uses the Mellanox mlx4 InfiniBand driver for fast network accesses and assumes the x86-64 architecture. It consists of 6,500 lines of C code, split across the kernels for the compute and memory pools, focusing on memory disaggregation.

Our implementation utilizes the RDMA RPC messaging framework atop LITE [258], a two-sided RDMA kernel module implemented by the one-sided *write* verb. The sender directly writes a request with payload to the buffer on the receiver side, where a thread checks for incoming messages. We pre-allocate and register physical memory to the network card as RPC buffers, which are kept separate from the LegoOS buffers to provide a degree of isolation. We describe the details of each kernel as follows.

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

TELEPORT compute kernel. This kernel manages application processes/threads and maintains a local cache of processes' full address space. It supports the `pushdown` system call, sends the request to the memory pool, and handles synchronization. As part of the latter, we needed to add functionality to enable the compute kernel to serve incoming page faults, invalidating the page and flushing the TLB as necessary.

To reduce network cost, our implementation adds an additional optimization to the protocol of Section 4.4. Specifically, it compresses the list of resident pages sent at the beginning of pushdown using *run-length encoding*, which provides 20× reductions in the message size, making it feasible to pack the list of pages and their permissions along with necessary metadata into a single RDMA message.

TELEPORT memory kernel. During pushdown, the memory kernel handles incoming RPC requests and cache coherence. It runs a number of parallel RPC handlers to perform these tasks—each on its own a kernel thread. This number is configurable to reflect the compute power limitation in the memory pool. Upon receiving a pushdown request, the server enqueues it into the workqueue of the memory-pool instance of TELEPORT and eventually processes it in the manner described in Sections 4.3 and 4.4.

4.7 Evaluation

We conduct a comprehensive set of experiments to evaluate the benefits of TELEPORT based on the use cases presented in Section 4.5, the trade-offs in compute pushdown, and the efficiency of TELEPORT designs. We compare TELEPORT with two baselines: (1) a disaggregated baseline, LegoOS, which incurs cost of scaling due to remote memory accesses, and (2) a single-machine baseline, Linux, where the cost is either low when local memory is sufficient for the workload or high when local memory is constrained and data is spilled to secondary storage, e.g., SSDs. In all experiments, the applications, datasets, and number of CPUs used by the applications are consistent across all platforms to ensure a fair comparison.

Experimental setup. The baremetal machines in our testbed have Intel Xeon E5-2630L CPUs

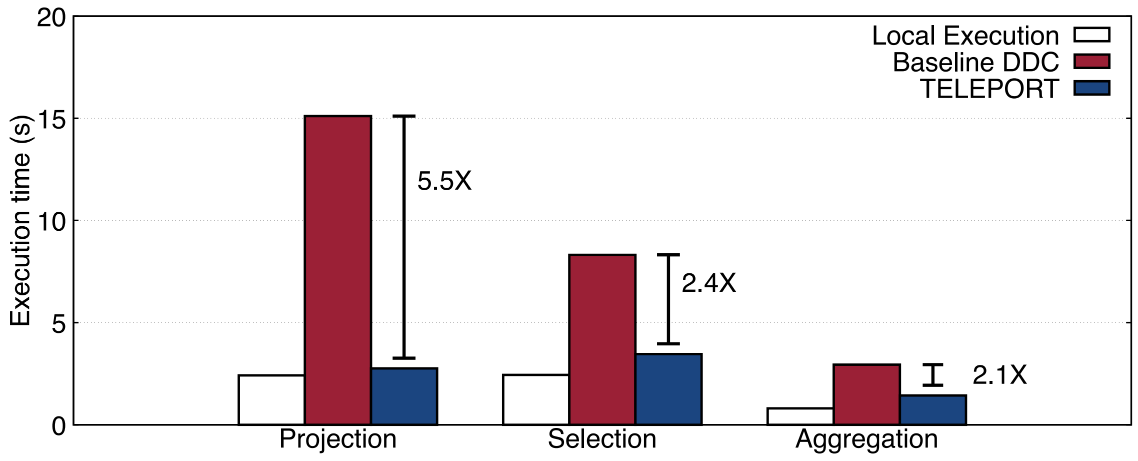


Figure 4.14: The performance improvement of pushing the operators in Q_{filter} to the memory pool with TELEPORT.

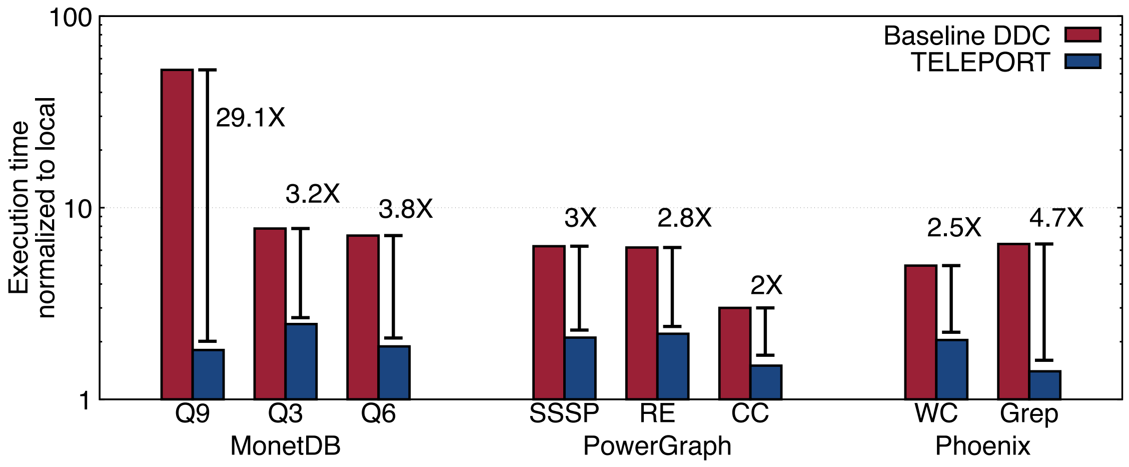


Figure 4.15: TELEPORT improves the performance of a wide range of data processing tasks. By removing expensive data movement, TELEPORT significantly reduces the overhead of memory disaggregation, up to an order of magnitude speedup compared to the baseline DDC.

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

4.7.1 The Effectiveness of TELEPORT

We first quantify the performance improvement achieved by operator pushdown for data-intensive systems over baseline DDC. Hence, this section first focuses on the *cost of disaggregation*. We use a default setup: the CPU cores in the compute and memory pools have the same clock speed, but numbers of cores are different—the memory pool is limited to a single thread; concurrent pushdown requests are serialized. Beyond the cost of disaggregation, Section 4.7.2 showcases the elasticity of DDCs. Section 4.7.3 further investigates the impact of the degree of disaggregation by varying CPU clock speed and the number of threads in the memory pool.

Database microbenchmark. Our first experiment involves the synthetic Q_{filter} query presented in Section 4.5.1. Recall that this query involves a selection operator followed by projections and an aggregation. During setup, we supply as input a `Lineitem` table with 300 million tuples. Figure 4.14 summarizes our main findings for these operators, where the y-axis shows the query execution times in Linux (local execution), LegoOS (baseline DDC), and TELEPORT. We make the following observations. First, compared with the local execution, baseline DDC adds significant overhead, experiencing 3–6$\times$ slowdowns, primarily due to paging data from remote memory. With TELEPORT, the slowdowns are drastically reduced to less than 2$\times$. In fact, TELEPORT is faster than LegoOS by 2.1–5.5$\times$. The improvements are most visible for projection, which would otherwise have to ship many tuples from the remote memory pool just to identify attributes of interest and apply filters.

Database TPC-H benchmark. Figure 4.15 (left figure) compares the performance of MonetDB in Linux, LegoOS, and TELEPORT with the three TPC-H queries (scale factor 50) with the longest execution times, namely Query 9 (Q_9), Query 3 (Q_3), and Query 6 (Q_6). These queries, as described in Section 4.5, consist of a mix of relational operators involving selections, projections, aggregations, hash and merge joins, and expression calculation.

We make the following observations. In all three queries, the significant slowdowns, higher than 50× in the worst case, render the baseline DDC prohibitively costly for database query processing in scaling out the hardware resources. Using TELEPORT, we pushed down a subset of the most bandwidth-intensive operators that bottleneck the DDC performance to the memory pool. The speedup improvements over LegoOS range from 3–29×. TELEPORT, with a compute-local memory that is ∼2% of the database size (1 GB versus 50 GB), is only slightly slower than local execution. The cost of scaling for DBMSs in DDCs with TELEPORT is comparable to the cost in distributed DBMSs as we see in Figure 4.1b.

Graph processing. Our next experiment is on graph processing. Figure 4.15 (center figure) summarizes the results obtained on PowerGraph for three graph queries: SSSP (single-source shortest path), RE (single-source reachability), and CC (connected components). We use as input a real-world social-network graph [276]. Our results show that the cost of scaling in baseline DDC is 5×. In comparison, TELEPORT closes the gap between DDC and local execution quite noticeably, achieving 2–3× speedup over LegoOS. The primary benefits obtained are in pushing down the scatter-gather and finalize stages, as described in Section 4.5.2.

MapReduce. Our final use case is MapReduce using the WC (WordCount) and Grep applications on a real-world NLP dataset consisting of 15 million Reddit comments[2]. Our observations in Figure 4.15 (right figure) are consistent with the other two systems. TELEPORT achieves 2.5× and 4.7× performance improvements over LegoOS, significantly narrowing the gap from local execution in Linux.

The takeaway is that TELEPORT results in up to an order of magnitude performance improvement over the baseline disaggregated OS and minimizes the gap between disaggregated and traditional environments. We note that the goal is not to surpass a local execution where resources are all centralized in a single place, but rather to narrow the performance gap to achieve a low cost of scaling while reaping the benefits of DDCs [109, 235].

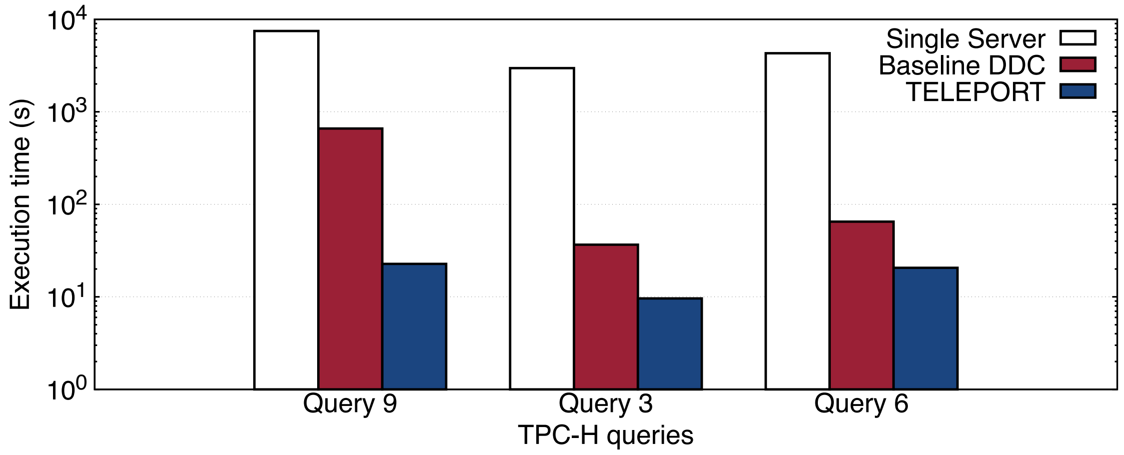


Figure 4.16: Query speedups with large disaggregated memory pools vs. SSDs.

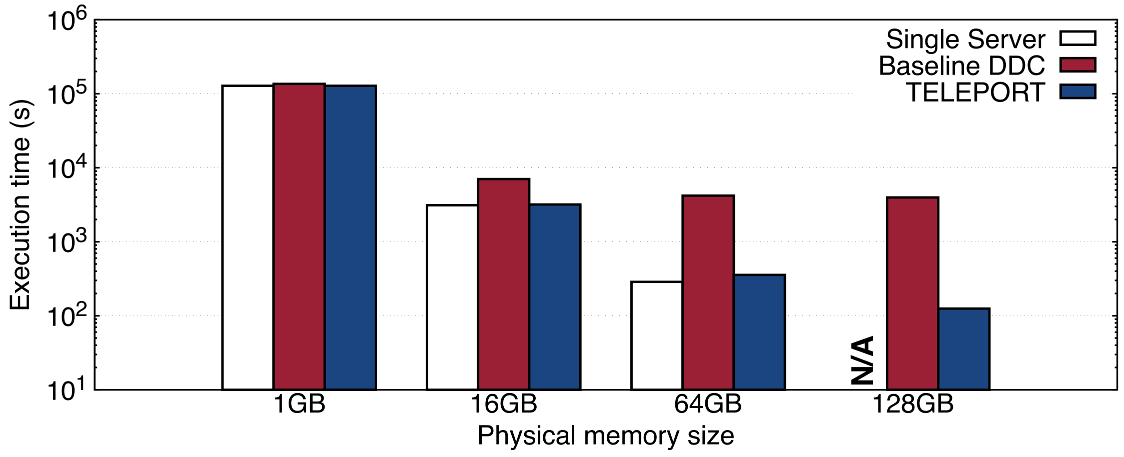


Figure 4.17: The benefits of increasing physical memory for large workloads.

4.7.2 The Benefits of Memory Disaggregation

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

Figure 4.16 shows the results of processing the three most expensive queries (scale factor

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

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

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

Figure 4.17 shows that with 1 GB total memory [3], all platforms perform poorly. In principle, provisioning more total memory will spill less data to disk; however, at 64 GB, the disaggregation cost in LegoOS begins to dominate the execution time, which is significantly longer than the time in Linux. TELEPORT instead effectively minimizes this cost and achieves a similar performance to Linux until 128 GB where Linux cannot match the amount of resources. TELEPORT provides 2.3× higher performance than the best Linux execution, and 31.7× higher performance than LegoOS with the same memory size. These benefits will continue to grow as the workloads become larger.

4.7.3 Varying the Degree of Disaggregation

Our next set of experiments performs a sensitivity analysis on the degree of disaggregation along two dimensions: (1) where the memory pool has lower CPU clock speed compared to CPUs in the compute pool, and (2) degree of parallelism—the number of threads—in the memory pool. We emulate these effects by throttling CPU clock rate in the memory pool and varying the number of threads that are used to process parallel pushdown requests. Our results

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

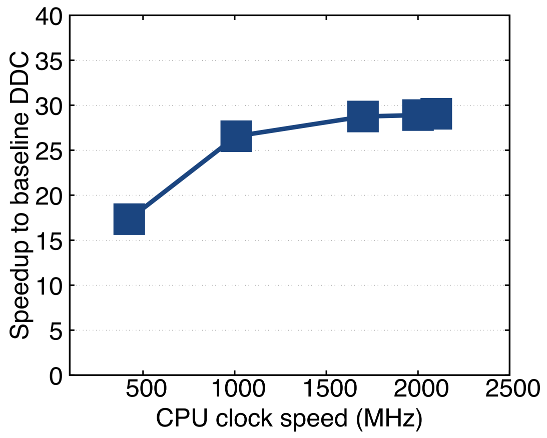


Figure 4.18: Pushdown performance (Q_9) with different computation power settings in the memory pool.

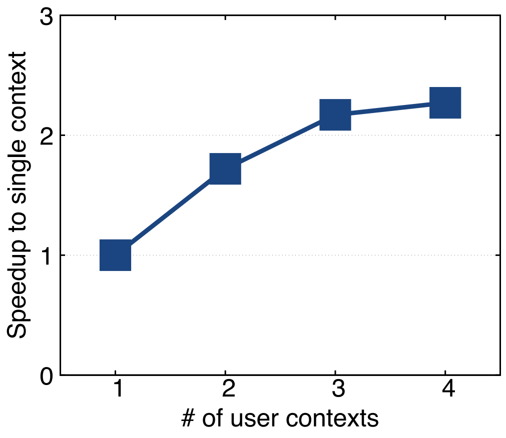
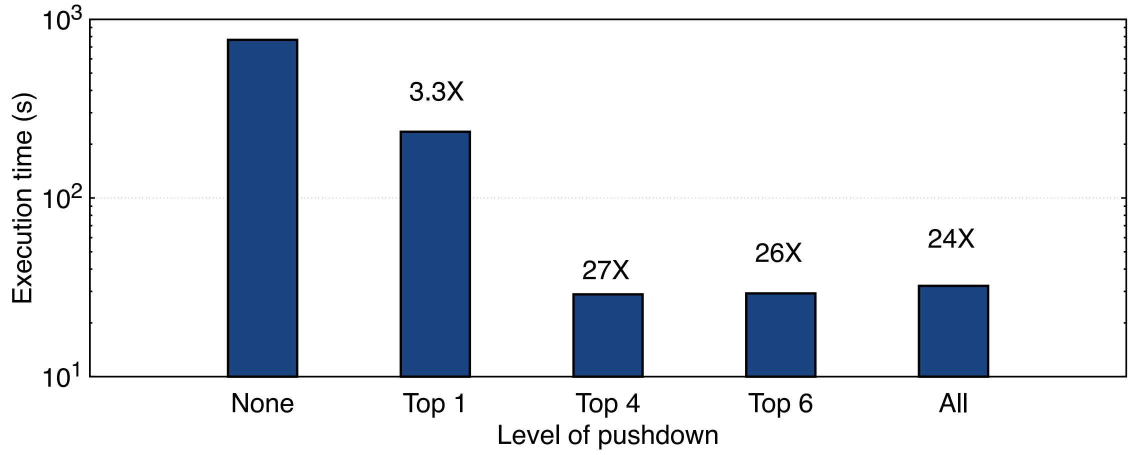


Figure 4.19: The benefits of parallelizing the processing of concurrent pushdown requests in the memory pool.

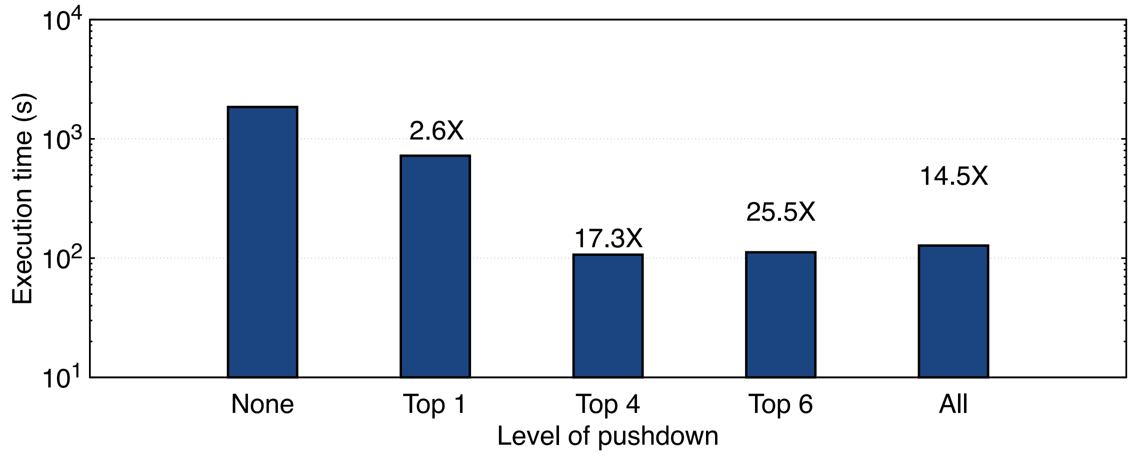
also help future DDC and hardware designers understand sweetspots in cost-performance ratios when determining the relative costs for the disaggregated compute and memory pools.

Figure 4.18 shows the effect that computation power in the memory pool has on pushdown execution time. We control the CPU clock speed with throttling for Q_9 in the MonetDB TPC-H benchmark (scale factor 50). As the CPU capabilities in the memory pool increases from 20% (0.4 GHz) to 100% (2.1 GHz) of the compute pool, query speedup of Q_9 relative to the baseline DDC increases as expected. Our results suggest that even at very modest CPU speeds (0.4 GHz), emulating a memory pool with very limited compute resource and thus a high degree of disaggregation, TELEPORT is still able to achieve a 17 $\times$ speedup over the baseline. Moreover, at clock speeds above 1.7 GHz, the speedups level off at 29$\times$, suggesting there is no need to match the fastest CPU speed to reap the performance benefits of TELEPORT.

Figure 4.19 shows the effect of memory-pool parallelism on the performance of processing concurrent `pushdown` calls. We evaluate a parallel aggregation query on TPC-H `Lineitem` table. We maintain the same CPU speed (2.1 GHz) in both the compute and memory pools. The application uses eight threads in the compute pool, one on each physical core. The memory pool uses two physical cores for the user contexts, emulating a scenario where the disaggregated memory pool does not have significant compute resource dedicated to run the pushdown functions. The y-axis in the figure shows the speedup over a single user context, as



(a) 50% lower CPU clock rate.



(b) 75% lower CPU clock rate.

Figure 4.20: The performance of different levels of pushdown.

we vary the number of parallel user contexts in the memory pool on the x-axis. We find that as the parallelism increases, it takes less time to process the eight concurrent requests as expected. However, we see diminishing returns in speedup, primarily due to context switching overheads when scheduling more threads than there are physical cores.

4.7.4 Varying the Level of Pushdown

Recall from Section 4.5.1 that there are trade-offs in compute pushdown. We now evaluate the impact of these trade-offs by using a metric we call *memory intensity*. To compute memory intensity, we first execute a profiling run in the baseline DDC; memory intensity is then the

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

Table 4.1: The components in executing a pushdown request. Grayed are factors on what TELEPORT has no control.

total remote memory accesses divided by the execution time (i.e., remote memory accesses per second, RM/s). We compute memory intensity for Q_9 in MonetDB and order its eight operators by this metric. For reference, *Projection* has the highest intensity (110K RM/s) and *Group* has the lowest (45K RM/s).

Figure 4.20a shows the performance of different levels of pushdown. We constrain the computation power in the memory pool to be 50% of that in the compute pool. Compared to no pushdown, offloading the most expensive operator to the memory pool brings 3$\times$ performance speedup. The speedup increases to 27$\times$ when we apply TELEPORT to the top four operators. *Being too aggressive, however, backfires*: the speedup decreases to 26$\times$ and 24$\times$ when we push down the top six and all operators, respectively. This is because, for these operators, the benefit of saving network communications does not compensate the overhead of pushdown and a lower CPU clock rate. These effects are magnified when the computation power in the memory pool is more constrained (Figure 4.20b).

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

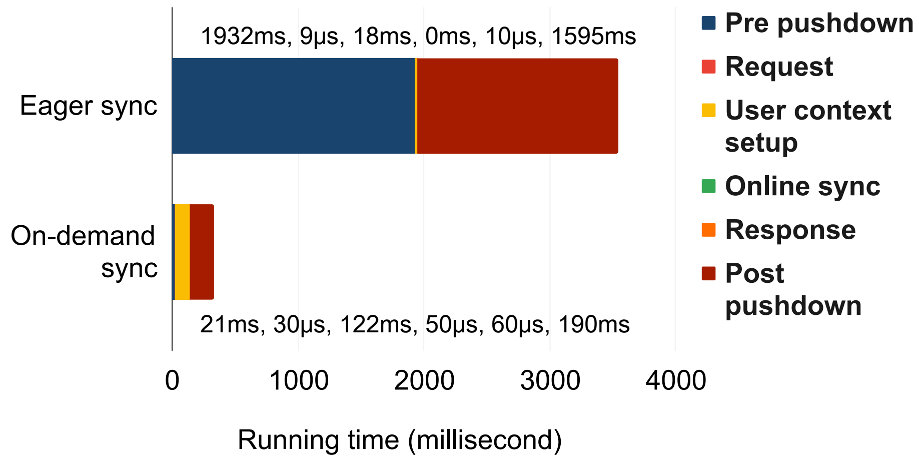


Figure 4.21: TELEPORT performance breakdown with different sync methods.

4.7.5 TELEPORT Execution Breakdown

We next quantify the benefits of on-demand synchronization methods, and provide a comprehensive look at the costs associated with them. Table 4.1 presents a factor analysis on the execution time in TELEPORT for processing a `pushdown` call. This time consists of six parts: (1) pre-pushdown synchronization, (2) pushdown request transfer from the compute pool to the memory pool in the RDMA network, (3) user context setup, (4) pushdown function execution and synchronization during the execution, (5) pushdown response transfer from the memory pool to the compute pool via RDMA, and (6) post-pushdown synchronization. While all parts have factors that are not controlled by TELEPORT, specifically the cache size in the compute pool, the network, and the user function, the data synchronization method in use is important for every part. Message sizes for (2) and (4) also vary across different methods.

Figure 4.21 summarizes our cost breakdown results for a 1 GB local memory in the compute pool (user function time was excluded so that the result can be generalized). In the figure, *eager* memory synchronization is a strawman that synchronizes all pages at the beginning and end of pushdown function execution. The on-demand memory synchronization is our default technique presented in Section 4.4.1. We make the following observations. In both techniques, pre/post pushdown and user context setup are the dominant costs, though at varying degrees. Overall, TELEPORT's on-demand synchronization is significantly faster than eager synchronization (0.3s vs. 3.5s for one `pushdown` call), since data is only fetched on demand as required by the user function. Although synchronizing data on-demand requires extra time in setting up the user context (yellow region in figure) because of page table entry checking described in Section 4.4.1, its substantial savings in parts (1) (blue) and (6) (red) reduce overall execution time by an order of magnitude. Our results suggest that a careful data-synchronization approach does impact performance significantly, compared to sunk costs that are tied to the underlying network speed.

4.7.6 Coherence Protocol Efficiency

Finally, we evaluate the efficiency of TELEPORT's coherence protocol. We do this by extending the microbenchmark in Section 4.4. We add shared memory between the compute-intensive thread and the memory-intensive thread, and vary the contention (where both threads request write permissions) rate from low (0.0001%; one in a million operations) to high (1%; one in a hundred operations).

Figure 4.22 shows the application performance in different systems when the contention between the threads increases. As the contentions in both local execution and base DDC are local to the threads (within the same NUMA node), increasing the contention rate barely affects the performance. In TELEPORT with the default coherence protocol, the contention leads to network communication. At low contention, the application completes in 2.1s. There are observable performance changes when the contention rate reaches 0.1% (2.3s) and 1% (3.7s). Figure 4.23 shows the number of network messages incurred by the protocol. The average messaging latency in our coherence protocol (1.6μs) is close to the raw network latency (1.2μs). In contentions, favoring the memory thread in tiebreaking completes the pushdown faster: 15% improvement at 1% contention rate. Adding more threads increases the contention correspondingly. For example, when we fix the contention rate at 0.1% per thread, increasing the number of compute-intensive threads to four brings the execution time up to 2.9s.

A Weak Ordering [35] relaxation avoids contention between writers. Figures 4.22 and 4.23

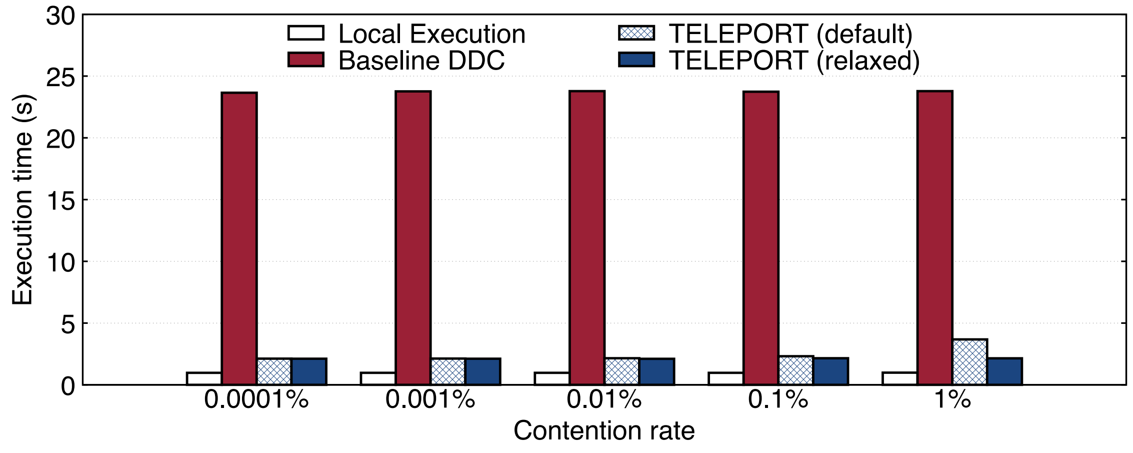


Figure 4.22: Application performance with varying levels of contention.

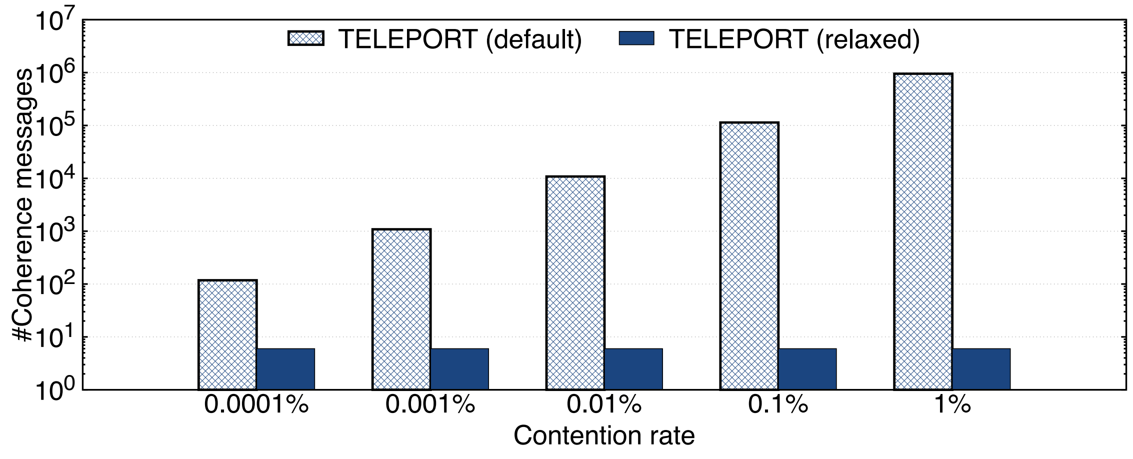


Figure 4.23: The number of coherence messages in TELEPORT.

show that the performance and the number of coherence messages no longer change with the contention rate when the application adopts the relaxation. Other relaxations work similarly for other types of contentions, e.g., the PSO relaxation (Section 4.4.2) for contentions between writers and readers.

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

4.8 Related Work

TELEPORT is related to the classic idea of pushing computation closer to the data [97, 99, 113, 181, 207, 214, 249]. Pushing down selection predicates is also a well-studied technique in databases, including distributed databases [61, 78, 230] and sensor networks [178]. TELEPORT's instantiation of these prior ideas is unique owing to the memory disaggregation setting. TELEPORT has access to a process's entire memory address space, can compute arbitrary functions, can modify the memory at will, and can dereference pointers.

Today's cloud DBMSs already leverage storage disaggregation [1, 8, 55], which decouples processing and data so they can scale independently; however, workers in these systems are still constrained by their local memory—a limitation that memory disaggregation addresses. Operator pushdown has been applied to these storage-disaggregated environments [135, 199, 215, 277]. Unfortunately, they are typically limited by the operations supported by the storage service and, thus, relegated to simple tasks like scanning tuples. Combining TELEPORT and storage pushdown may yield further improvements in a fully disaggregated environment.

Within the DDC and remote memory context, Semeru [266] pushes down part of the JVM garbage collector to remote memory, while the work of Aguilera et al.[38] pushes down pointer chasing, and StRoM [244] pushes down checksum computations to remote SmartNICs. KV-Direct [159] leverages FPGA-based NICs to extend RDMA with native key-value store operation support. TELEPORT is distinct in its generality—it can be used to push down arbitrary functions. This is possible because the stack, heap, and code pages all live remotely as a byproduct of disaggregated OSes.

Improving database systems in DDCs is a timely topic [73, 290]. Redesigned DBMSs [73, 290] can significantly lower the overhead. DDC architectures are continuously evolving. Keeping up with the hardware by redesigning DBMSs requires expensive investment. TELEPORT provides a simpler and more portable alternative for DBMSs to harvest many benefits of DDCs. TELEPORT can also be applied to other data-intensive systems for the same advantage.

4.9 Summary

This chapter proposes TELEPORT, a framework that can flexibly transport a piece of computation to the memory pool for saving expensive data movement and thus improving overall query execution for data processing systems in disaggregated data centers. Our design challenges center around ensuring consistent views on the memory space, synchronization, and temporary context creation in a pushdown call. To use TELEPORT, applications invoke an intuitive system call and customize it for flexibility. By applying TELEPORT to three popular data-intensive systems, (database, graph processing system, and MapReduce), we showcase the significant performance benefit of TELEPORT for DDCs.

CHAPTER 5

REALIZING DDC BENEFITS IN TODAY'S CLOUDS

High-performance data center networking is a key enabler of resource disaggregation. Among all the alternatives, Remote Direct Memory Access (RDMA) is currently the most popular, scalable, and production-ready fast networking technique for memory disaggregation. In this chapter, we present Redy, a cloud service that provides high performance caches using RDMA-accessible remote memory. An application can customize the performance of each cache with a service level objective (SLO) for latency and throughput. By using remote memory, it can leverage stranded memory and spot VM instances to reduce the cost of its caches and improve data center resource utilization, which is a key motivation for DDCs. Redy automatically customizes the resource configuration for the given SLO, handles the dynamics of remote memory regions, and recovers from failures. The experimental evaluation shows that Redy can deliver its promised performance and robustness under remote memory dynamics in the cloud. We augment a production key-value store, FASTER, with a Redy cache. When the working set exceeds local memory, using Redy is significantly faster than spilling to SSDs.

5.1 Introduction

5.1.1 The Case for Remote Memory as Cache

Stateful cloud services store their states on secondary storage, such as server-local SSDs or a cloud storage service. Example storage services are database systems, key-value stores, and JSON stores. Stateful application services embed these data management systems, such as a directory service, document management system, or source code control system. To offer fast response time, these types of services store a subset of their states in memory caches.

When allocating a memory cache, a server need not be limited by its local available memory. It could use physical memory on other servers. Although remote memory has higher access time than the server's local memory due to network latency, there are many reasons why it can be an attractive choice.

First, a server's physical memory capacity is limited. It may have insufficient local memory available for a stateful service, particularly for its peak workloads. In this case, remote memory is the only option. Otherwise, its state has to be spilled to secondary storage, resulting in orders-of-magnitude performance degradation.

Second, some cloud services are satisfied if they can read records in a few microseconds (μs's), which does not require local memory performance. This is currently impossible to achieve with SSDs, but can be supported with fast data center networks [9].

Third, remote memory may be cheaper because it sits on lightly loaded servers. For example, Google, Facebook, and Alibaba report that as much as 50% of server memory in data centers is unutilized [121, 235]. An extreme case is *stranded memory*, which is unusable by its local server because its cores have all been allocated to local VMs. Stranded memory is essentially free. By using this otherwise wasted memory as a cache, a stateful service can run on smaller servers with less server-local cache, thereby reducing cost.

A fourth reason is the trend toward dedicated and disaggregated memory servers whose sole function is to offer memory to remote servers, as Chapters 3 and 4 and recent work [109, 121, 167, 188, 235] present. This approach is becoming more feasible due to fast data center networks, whose point-to-point bandwidth is close to I/O bus bandwidth and is usually underutilized [226, 280]. Cloud service providers already disaggregate compute and storage. By disaggregating memory, they can fully benefit from this expensive resource. Most cloud vendors have not been forthcoming about their internal usage of this capability and do not yet support it for third-party users. However, Google recently reported that it uses disaggregated memory in its BigQuery service [188], and Alibaba is customizing their database systems with memory disaggregation [73, 290].

To be usable as a cache, remote memory must be accessible with very low latency. Remote direct memory access (RDMA) is the natural choice. RDMA is not as fast as local memory, but it is much faster than SSDs and requires little or no CPU involvement.

Today, the typical access time for main memory is 70 nanoseconds (ns) [264]. For RDMA it is a few μs [20, 30, 296]. For SSD it is $\sim$100 μs, but highly variable and often higher, due to garbage collection and concurrent writes. Although RDMA latency is 100x better than SSD, its bandwidth advantage is only 2x–10x (e.g., SSDs are 16-24 Gbit/s and RDMA networks are 48-200 Gbit/s). Still, the difference is significant for applications that need high-throughput data access. Hence, RDMA-accessible remote memory is a natural choice for a cache sitting between these two layers of the memory hierarchy.

5.1.2 Contributions

There are two main challenges in using remote memory as a cache. The first is how to tune RDMA configurations. The choice of optimal configuration depends on the application workload, processor and network characteristics, and service level objective (SLO). Misconfiguration can lead to poor performance. Tuning RDMA is known to be difficult. In a data center, it must be done dynamically, since the choice of processor and network distance between processors can vary. It is therefore important that this tuning be automated.

The second challenge is responding to changes in remote memory availability. A memory region might become unavailable because its server failed or because the memory region allocation was evictable and the system reclaimed it for local VMs. In both cases, the application that was using the cache must be dynamically reconfigured. It must operate without the cache or migrate the cache to another remote memory region and re-populate it.

We propose Redy, a new cloud cache service that efficiently utilizes stranded and unused server memory using RDMA. Unlike prior RDMA stores and caches, it handles failures and reclamations and allows users to customize cache performance. It also requires minimal changes to applications. Our contributions are as follows.

- Stranded memory analysis. We present the results of a study that shows stranded memory is significant and dynamic.
- An RDMA architecture for an SLO-based memory cache service. Unlike previous RDMA systems that optimize for specific performance targets, ours enables the user to customize the target. It automatically finds an RDMA configuration that satisfies the user-provided SLO and minimizes resource cost.
- Dynamic memory management. Redy is elastic. It adds or removes cache regions when client requirements and memory availability changes. It also efficiently migrates cache regions when a remote memory region becomes unavailable.
- Implementation and evaluation with a production key-value store. We deploy Redy with FASTER [190] to improve its performance when the hot set is larger than local memory. We measure its improvement using the YCSB benchmark.

5.2 Motivation

5.2.1 Underutilized Cloud Memory

We take it is as given that stateful applications would benefit from more memory. There is a lot of it in data centers, waiting to be utilized. All major data center operators and cloud providers report that memory is highly underutilized. Studies of traces from Google [116, 257], Microsoft [88, 60], Alibaba [42, 124], and Facebook [121] report memory utilization is under 50% and has strong temporal volatility. We confirm these results for unused memory, and extend them by analyzing the dynamics of stranded memory.

Unallocated memory. We define *unallocated memory* as the fraction of DRAM not allocated to any VM or container. We measured unallocated memory in 100 Azure Compute clusters over 75 days. Compute clusters host mainstream internal and external VM workloads and represent the majority of servers compared to storage or other specialized clusters. We selected clusters with at least 70% of CPU cores in use. Each cluster trace contains time, duration, resource demands, and server-ids for millions of VMs. We find strong diurnal patterns; the typical peak-to-trough ratio is 2. At the median (across clusters and time), 46% of memory is unallocated. The tenth and first percentile are 37% and 28%, respectively.

Stranded memory. A subset of unallocated memory is stranded. At the median, 8% of memory is stranded. This grows as more VMs and containers are allocated with more than 16% stranded at the 90-th percentile and 23% stranded at the 99-th percentile.

We analyze the amount of stranded memory reachable via RDMA by measuring the number of network switches between a server and stranded memory. Figure 5.1 shows the result as a CDF. Half of all servers can reach 1 TB of memory by traversing just one switch, 30 TB by traversing three switches, and 100 TB by traversing five switches. *A small fraction of servers can even reach 1 PB*. Our analysis shows that stranded memory in a public cloud is too significant to ignore.

Stranding Duration. Figure 5.2 shows the distribution of the duration of stranded memory events. A stranding event begins when a server allocates all CPU cores while $\geq$1 GB of memory remains unallocated. It ends when a VM or container on the server terminates, making at least one core available. We find that memory is frequently stranded and unstranded with variable durations of minutes to hours. The median stranding event is 13 minutes, with a 25-th percentile of 6 minutes and a 75-th percentile of 22 minutes. Our analysis shows that the amount and duration of stranded memory are highly dynamic, making it challenging to use it effectively.

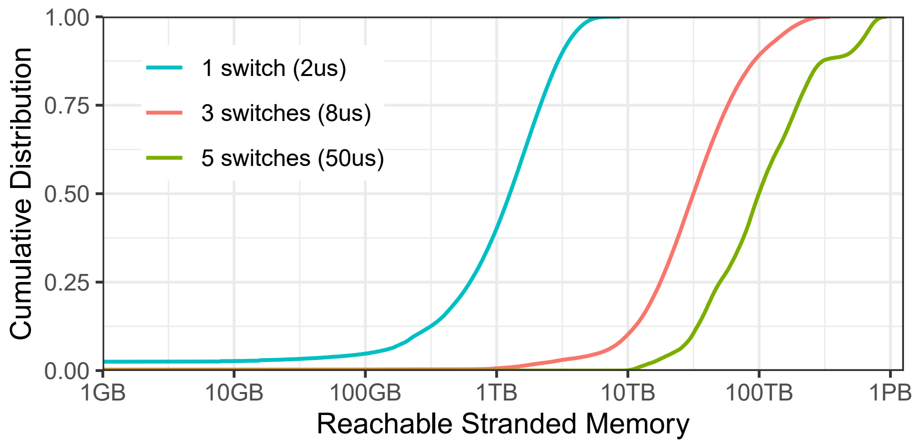


Figure 5.1: The significance of stranded memory.

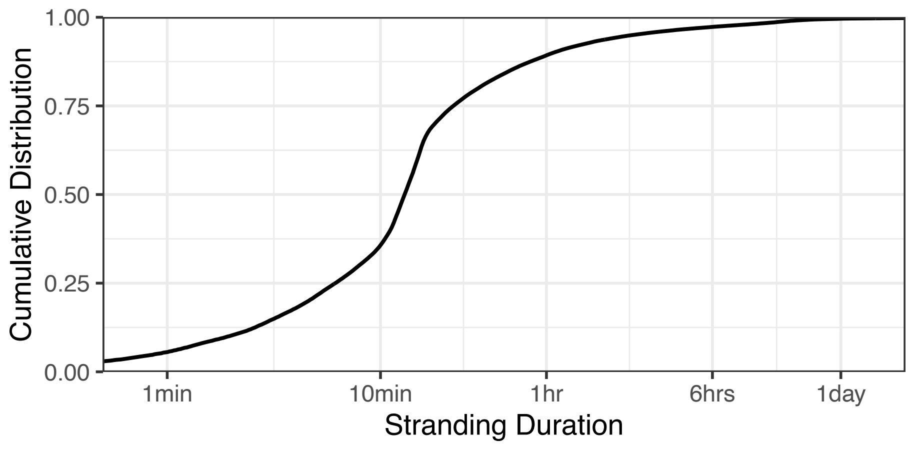


Figure 5.2: The dynamics of stranding events.

5.2.2 Diverse RDMA Configurations

We propose using this unallocated memory for RDMA-accessible remote caches. However, optimizing RDMA's performance is hard. Parallelization, asynchrony, thread contention, batching, one-sided vs. two-sided operations, and CPU bottlenecks all affect RDMA throughput and latency. Performance is also highly sensitive to the underlying hardware. Overall, it is difficult to develop a robust solution for a variety of workloads and configurations.

For example, Figure 5.3 shows the latency and throughput of our caching system, Redy, when writing 8-byte payloads (as in YCSB [87]) to remote memory with three different RDMA configurations. The latency-optimal configuration has 4.1μs latency, which includes 2.9 μs

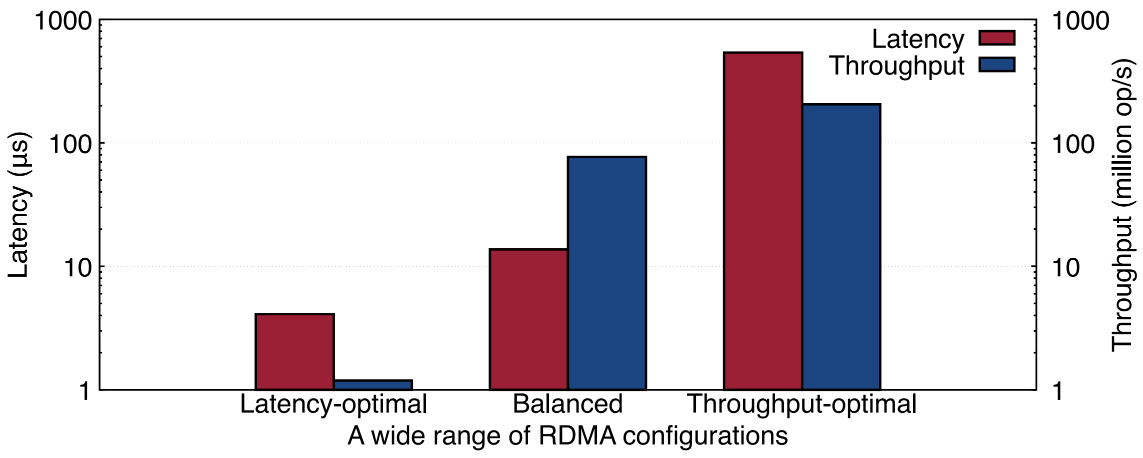


Figure 5.3: The impact of the RDMA configuration in Redy.

network latency, but the throughput is only 1.2 million operations per second (MOPS). The throughput-optimal configuration achieves 205 MOPS, but the latency is 538 μs. The balanced configuration is in between with 14 μs latency and 77 MOPS. We have similar findings for reads and other record sizes.

Many configuration parameters affect throughput, latency, and cost. They often improve one performance metric and degrade another. For example, increasing the number of operations in each RDMA transfer (called the *batch size*) increases throughput but also increases latency per operation. Increasing the number of in-flight transfers improves utilization of an RDMA connection and hence its throughput, but it increases latency. Increasing the number of hardware threads that service RDMA requests on the client and server increases throughput, but also increases cost. These conflicting trade-offs imply the need for optimization.

To solve this optimization problem, we need a software architecture that can dynamically tune these parameters, and an optimization algorithm that finds the optimal point in the parameter space. To address these challenges, we propose that the user guides the choice of configuration by specifying an SLO consisting of the desired throughput and latency of the remote cache. It is the system's job to choose the lowest-cost RDMA configuration that satisfies the SLO and then deploy it. Relating cache performance to application performance is out of scope and a possible topic of future work.

5.3 Redy Architecture

5.3.1 Design Principles

Redy is a cache service that offers underutilized cloud resources to memory-intensive applications. Its design goals are:

1. *Generality and ease of* use. Redy must have a flexible interface that can be easily integrated with a variety of memory-intensive cloud applications.

2. *Customizable performance.* Cloud applications have diverse throughput and latency requirements. Users can customize Redy's performance by providing SLOs for I/O throughput and latency and trade performance for lower cost.

3. *High resource utilization and minimal disruption.* Redy can exploit underutilized resources and stranded memory, thereby improving cloud resource utilization. This utilization improvement should not disrupt existing applications.

4. *Robustness to dynamics.* Resource utilization changes over time. One server may become busy while another becomes underutilized. Redy handles such dynamics, offering robust service as long as resources are accessible somewhere.

5.3.2 Back End

Figure 5.4 shows the architecture of Redy. The front end is implemented by the *Redy client*, which is colocated with its application. It talks to its cluster's back end, which consists of a global cache manager and a set of cache servers that run as VMs. We describe the back end in this subsection and the front end in the next one.

Redy's *cache manager* interacts with the cluster's *VM allocator*. It tracks the available server resources, which it uses to provision VMs. The cache manager offers three operations for allocating a cache: *Allocate*, to allocate one or more VMs for a cache; *Reallocate*, to revise a cache allocation; and *Deallocate*, to drop a cache.

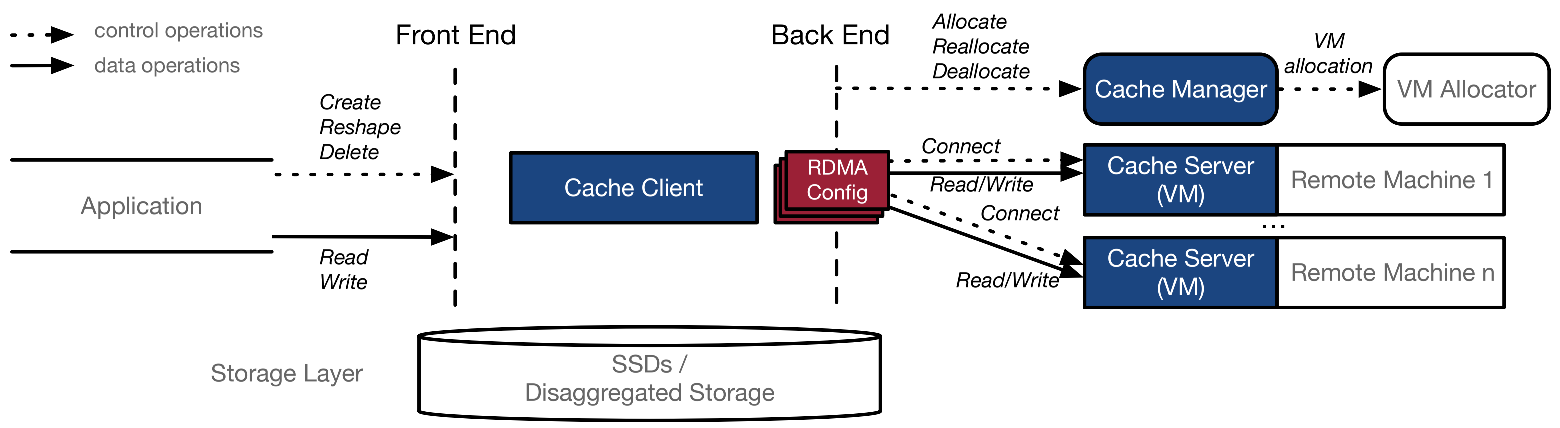


Figure 5.4: Architecture of Redy. An application interacts with the *cache client*. The global *cache manager* asks the data center's VM allocator to reserve VMs to host the cache. *Cache servers* on those VMs coordinate with the client for cache accesses.

The *Allocate* operation takes three parameters: the desired *amount of memory*, an *SLO* that specifies the desired latency and throughput of reads and writes, and a *duration* that specifies the likely lifetime of the cache. The SLO supports the second design goal by enabling the application to customize the cache's performance, for example, by specifying low latency for an interactive application that requires fast response time or high throughput for an analytics application that does data ingestion and query processing. A *duration* of infinity says that the caller is willing to pay full price for a cache that remains active until it is explicitly deallocated or fails. Shorter durations are meant to benefit from spot pricing of excess resources that the cloud vendor is unable to sell at full price [49, 117, 193], thereby improving resource utilization, the third design goal.

To process an *Allocate* request, the cache manager allocates one or more VMs, each of which consists of memory and zero or more cores, and derives an RDMA configuration that will support the requested SLO. It then returns a list of the allocated VMs and the RDMA configuration to use to communicate with them.

If the cache manager cannot satisfy the requested combination of capacity, SLO, and duration, then the *Allocate* request fails. The request has no effect and the cache manager returns an exception to the client. If a VM is a *spot instance*, then the VM allocator is free to *reclaim* the VM's resources, e.g., to sell the resources for a higher price. In this case, the VM allocator alerts the cache manager of the change and gives it time to compensate for the loss of resources. Today's cloud providers give an early warning of 30-120 seconds.

When the cache manager is notified that a VM failed or was reclaimed, it alerts the Redy client, which must be able to cope with the loss. Ideally, it can provision and populate a replacement VM. This reconfiguration activity is a key challenge for Redy. Its solution addresses the fourth design goal. Details are in Section 5.6.

The *Reallocate* operation is used to reconfigure an existing cache. The data in the cache can be truncated or remain unchanged depending on the parameters in the reallocation. The *Deallocate* operation is called to release all VM resources for a cache.

Each VM that hosts a cache runs a *cache server*, which is an agent that processes *Connect*,

API	**Function**
Create(capacity, SLO, duration [, file])	Create a cache with the specified capacity, performance SLO, and duration. Optionally populate the cache with a prefix of the file length 'capacity', and return the ID of the created cache.
Read(ID, dst, addr, size, cb)	Read (async) from a cache with specified address, size, and the callback.
Write(ID, src, addr, size, cb)	Write (async) to a cache with specified address, size, and the callback.
Reshape(ID, capacity, SLO)	Change the configuration of a cache with new capacity and SLO.
Delete(ID)	Delete a cache with specified ID.

Table 5.1: APIs provided by the cache client for applications. The underlined functions are for performing I/Os.

Read, and *Write* operations. These operations depend on RDMA details and are described in Section 5.4.

5.3.3 Front End

A cache client provides a *virtual storage device* abstraction that supports a contiguous byte-addressable address space. The client maps that address space to *memory regions* of the cache's VMs. The size of a memory region is configurable (1 GB by default). The application can perform a read or write operation on the device at an arbitrary address and of arbitrary size (bounded by cache capacity). The client translates the operation into a read or write at the corresponding offset of a memory region. This general device abstraction supports the first design goal.

Table 5.1 lists the client's APIs to create, manage, and access a cache. The *Create* function creates a cache of a given size, performance level SLO, and duration, and optionally initializes its content based on a file. The SLO specifies a maximum average latency and minimum average throughput of reads and of writes. If *Create* can allocate the requested capacity and the cache can satisfy the SLO and duration, then the client receives a list of VMs and the RDMA configuration for the cache and populates it (if the *file* parameter is present); otherwise, it has no effect and returns an exception.

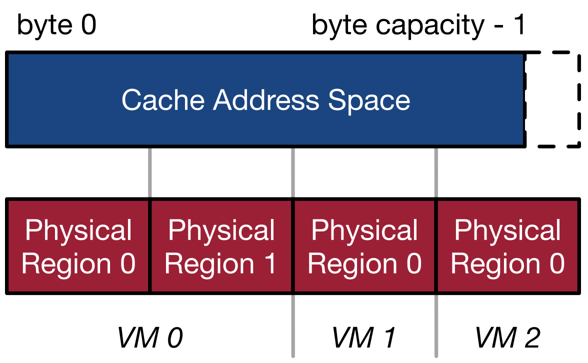


Figure 5.5: A region table maps a cache to VMs.

On receiving the list of VMs, the client constructs a *region table* that maps the cache's address space [0, *capacity*) to memory regions on servers. It divides the address space into *virtual regions*, mapping each one to a *physical region* on a VM (see Figure 5.5). To service a Read or Write for cache address *x*, the client uses the region table to translate *x* into the address on the VM where *x* is stored.

The two data access operations, *Read* and *Write*, are asynchronous, which is important for performance as we explain in Section 5.4. When an I/O operation finishes, its associated callback is invoked.

The *Reshape* function enables an application to change the SLO or capacity of a given cache. There are two cases: the SLO changes or it is unchanged. In the first case, the client calls *Allocate* to find new VMs of the requested size that satisfy the SLO. If it succeeds, the client migrates the old cache to the new one, truncating the end of the cache if it shrank. Then it deallocates the old cache. If *Allocate* fails, the cache is unchanged and the client returns an exception.

In the second case, the client resizes the cache. If the cache shrank, the client truncates it. If that frees up regions, the client calls *Reallocate* to notify the cache manager. If the cache grew, the client extends the address space. If the last region has insufficient unused space, then the client calls *Reallocate* to request more VMs.

If the client succeeds in reshaping the cache, it updates the region table. If it cannot allocate enough memory from the cache manager or the SLO cannot be satisfied based on

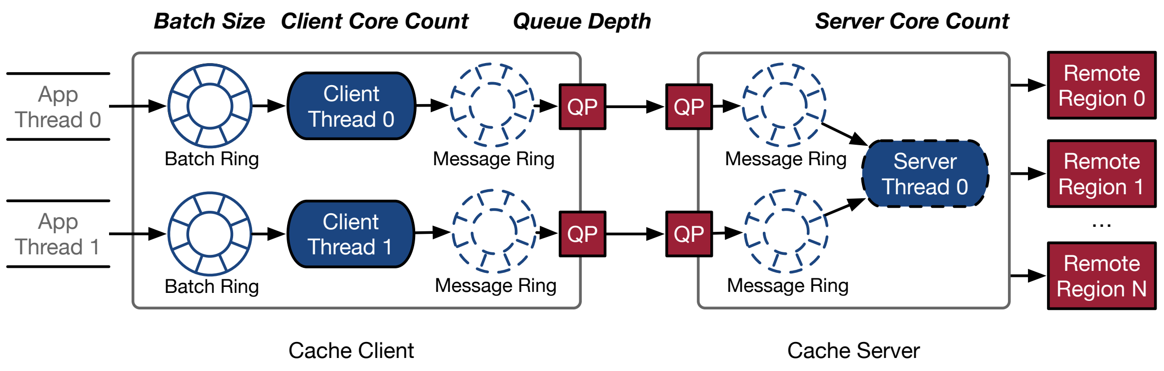


Figure 5.6: RDMA message flow in Redy. Ring buffers enable pipeline parallelism between adjacent threads. Message rings are only used when batch size is greater than one.

current resource availability, then it returns an exception and the cache remains unchanged.

The *Delete* function removes a cache by sending *Deallocate* to the manager. Any later access to the cache will return an exception.

5.4 Remote Cache with RDMA

This section presents the internals of a Redy cache, specifically, how it configures RDMA to access remote memory regions. The next section describes how Redy provides customized cache performance.

5.4.1 RDMA Background

RDMA enables an application on a VM to send requests to its NIC to read or write memory on another VM. It uses *kernel bypass*, which means the application interacts directly with its VM's NIC. The transfers are handled entirely by the NICs, with no OS involvement.

An application talks to its NIC via one or more *queue pairs* (QPs), each of which consists of two workqueues: a *send queue* and a *receive queue* for submitting and receiving requests respectively. Each workqueue has an associated completion queue. Multiple workqueues may share the same completion queue.

Communication can be one-sided or two-sided. With one-sided RDMA, the client application directly accesses the server's memory via *read* and *write* operations. A read operation includes the address and length of the server data to be read and the client location where the data should land. Conversely, a write operation includes the address and length of the client data and the server location where the data should land. The client application polls the completion queue for an event that indicates the operation finished.

Two-sided RDMA offers *send* and *receive* RPC-like operations in which the server CPU processes the client's request. Redy implements two-sided communications, but like previous work [96, 137, 258, 296] does it using one-sided RDMA *writes*, since they are faster.

One-sided RDMA uses session-oriented communication. A QP can only communicate with the QP that it connects to. Messages are delivered in order with no loss or duplicates.

5.4.2 Cache Implementation

Connection Setup. To process *Create* and *Reshape* operations or replace a failed/reclaimed VM, the client asks the cache manager to allocate new VMs. The allocate operation returns a list of VMs and the RDMA configuration to the client. After the client updates the region table, it builds *RDMA connections* by sending a *Connect* message to the cache server on each newly allocated VM. The message includes the number of physical regions the cache uses on the VM and the RDMA configuration. The latter specifies how the client and server communicate: whether communications is one-sided or two-sided, and if two-sided, then how many server CPU cores the cache can use to process RDMA requests. The server allocates the requested number of memory regions, registers them to the NIC, and replies with RDMA access-tokens, one per region, that the client uses to access server memory. When the client receives replies for all *Connect* messages, the cache is ready to use.

Reads and Writes. Redy implements reads and writes on a cache as remote memory accesses (see Figure 5.6). The *Read* and *Write* APIs are asynchronous, so an application can issue requests without waiting for previous ones to finish. The Redy client is multithreaded. Each thread collects read and write requests from an application thread in a *request batch* data structure, which it sends to the server using RDMA. The batch size is configurable from one to hundreds, which we optimize based on performance SLO (details in Section 5.5).

Each *server thread* polls messages from one or more RDMA connections. Upon receiving a request batch, it executes the requests on local memory regions. For a write request, the server thread copies the request's payload to the destination address. For a read request, it copies the requested data from the requested memory address to the response buffer. Finally, it sends a *response batch* that contains the results of all requests to the client through the same RDMA connection on which it received the request batch.

Each client thread polls its RDMA connection to retrieve response batches. For each read response in a batch, the client thread copies the payload to the application buffer specified by the corresponding read request. The client invokes the callback function of each read and write request to complete it.

Redy guarantees that all asynchronous requests are executed in order: requests from an application thread are batched in program order, batches are delivered in order with reliable RDMA connections, and they are processed in order by server threads.

5.4.3 Static Optimizations

Redy's RDMA architecture is optimized with techniques that judiciously exploit RDMA characteristics. Figures 5.7 and 5.8 show the effectiveness of each optimization. Unless otherwise mentioned, latency is the time in μs to process one I/O, which is a Redy read or write call, and throughput is the rate in MOPS. Figure 5.7 shows the median network round trip latency (light red), and the median (dark red) and 99-percentile tail (line with a top) of overall latency. This test uses one application thread, one client thread, and one server thread to read and write 8-byte records in a 1 GB cache with a batch size of one. (Section 5.7 describes the setup and presents more results.) The details are as follows.

Lock-free Communications. To minimize the overhead of exchanging data between threads, we use lock-free ring buffers. Specifically, a client thread accepts I/O requests from an application thread using a *batch ring buffer*, each element of which is a request batch. When a batch becomes full and the RDMA connection is available for another RDMA operation,

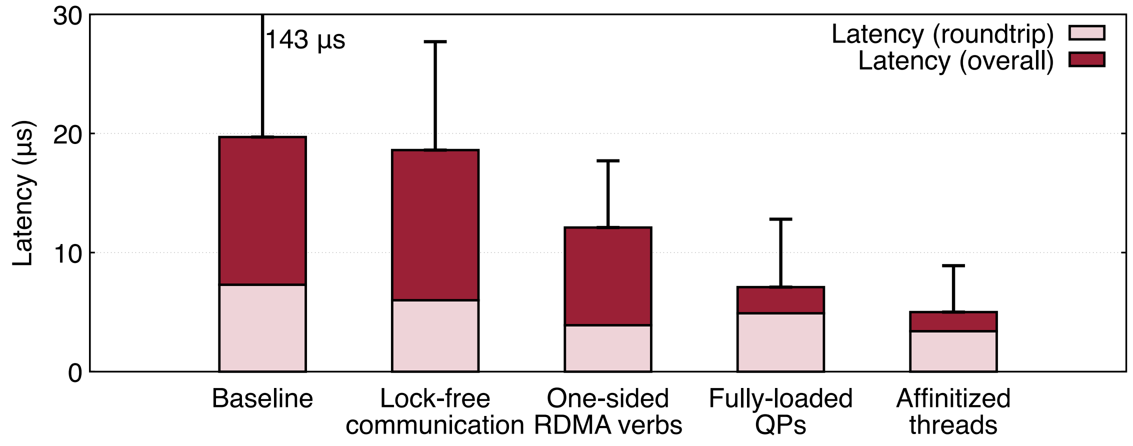


Figure 5.7: Redy optimizations effectively decrease latency.

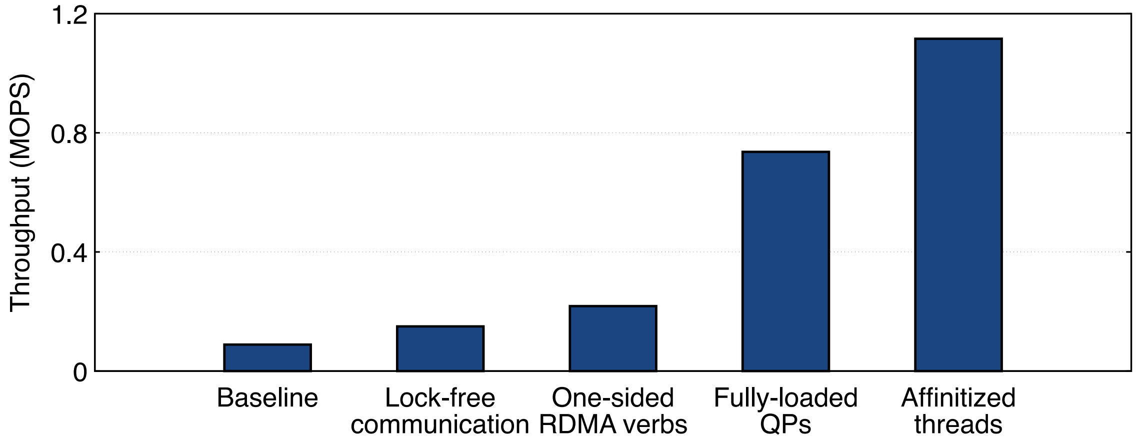


Figure 5.8: Effectiveness on increasing throughput.

the client thread moves the batch to its *message ring buffer*, which is registered to the NIC as RDMA buffers. There is a message ring on the server for every connection. Batch and message rings are based on previous work on lock-free ring buffers using atomic compare-and-swap and fetch-and-add [149] and using ring buffers for RDMA data transfer [96], but are customized for the Redy architecture. These ring buffers allow many requests to be passed and processed efficiently from the client to the server. This optimization eliminates data contention compared to a baseline where application threads use locks to share data with client threads, thereby reducing tail latency by 7× and improving throughput by 68.7%, as shown in Figures 5.7 and 5.8.

One-sided Operations. If a request batch has only one read (or write) request, we translate it to a one-sided *read* (or *write*). Otherwise, we use a *write* to send the request batch to the message ring on the server. This optimization reduces median latency from 19 μs to 12 μs and increases throughput by 45.3%.

Fully-loaded Queue Pairs. The number of in-flight RDMA operations on a connection is called its *queue depth*, which we control by the message ring size. Increasing it reduces waiting time of requests in the batch ring and thus their latency. It also increases network utilization. Compared to one in-flight operation, a queue depth of four reduces latency to 7.1 μs and increases throughput from 0.22 MOPS to 0.74 MOPS, a 3.4× speedup. However, the network latency increases with queue depth due to higher traffic, e.g., comparing the light-blue bars for one-sided RDMA and fully-loaded QPs in Figure 5.7. Although throughput improves when we increase queue depth from four to eight, latency worsens. We measure the performance impact of queue depth, starting from one, and choose the maximum value that improves both latency and throughput.

NUMA-aware Affinitized Threads. OS thread scheduling can negatively affect application performance [156, 253]. To avoid this, we pin Redy threads to physical cores in a NUMA-aware fashion. Each client thread is affinitized to an application thread's NUMA node, which reduces communication overhead between threads and stabilizes communication between client threads and the NIC. This achieves a latency of 5 μs and throughput of 1.1 MOPS, a 30% and 52% improvement respectively over non-affinitized threads.

5.5 SLO-Driven Configuration

5.5.1 Performance Variables

RDMA can transfer messages in just a few microseconds. At that time scale, small changes in the instruction count, synchronization delay, memory contention, or processor cache contention can greatly affect RDMA latency and throughput. These effects can be controlled by the choice of RDMA configuration and how it is used. However, since optimal choices de-

Variable	Description	Lower Bound	Upper Bound
c	the number client threads that process request batches	1	client cores
s	the number of cache server threads	0	c
b	the number requests in a batch	1	$\lceil \frac{4\text{ KB}}{\text{record size}} \rceil$
q	the number of in-flight operations	opt.	NIC spec

Table 5.2: Variables balancing latency and throughput.

pend on the size of cached records and the relative importance of latency and throughput, the choice is necessarily workload dependent.

Based on microbenchmarks and the rich literature on RDMA performance, we have identified four variables that are the primary determinants of Redy cache performance. They are summarized in Table 5.2. Increasing the value of each variable will increase throughput. But it also increases network traffic, which in turn increases the latency of individual requests. Details are as follows.

- *Client core count* (c) - Increasing client threads adds more computation and RDMA connections for more parallelism. This parameter is capped by available CPU cores in the client VM.

- *Server core count* (s) - Increasing threads on the remote server to process batched requests reduces the load on each thread. No server threads are needed if requests are not batched. Each client thread has one RDMA connection, and the server has at most one thread per connection (since the bottleneck of a connection is the network, not server compute), so we cap s at c, i.e., $s \leq c$.

- *Batch size* (b) - Batching small requests improves network bandwidth utilization. In our RDMA tests, bandwidth utilization and throughput stop improving beyond 4 KB data transfers. Therefore, we cap the batch size at $\lceil \frac{4\text{ KB}}{\text{record size}} \rceil$ messages.

- *Queue depth* (q) - Based on the fully-loaded QP optimization, additional in-flight operations improve bandwidth utilization, i.e., increasing throughput but also latency, similarly to b. The upper bound is specified by the NIC, which is 16 in our testbed on Azure HPC clusters [192].

In addition to the trade-off between latency and throughput, there is a trade-off between performance and cost: increasing c and s increases the client and server VM cost.

5.5.2 SLO-based Search

A major challenge of Redy's design is to find an RDMA configuration that satisfies each cache's SLO. Our solution is a two-phase search algorithm: (1) offline modeling and (2) online searching. In offline modeling, we perform measurements to build a function that captures the effect of the configuration parameters (c, s, b, q) on latency and throughput. In online searching, we use the function to search for values of these variables that satisfy the latency and throughput specified by the SLO. Our detailed design is below.

Configuration Space. An *RDMA configuration* is a tuple [c, s b, q] of configuration parameters. Our performance model is a function f that maps each RDMA configuration to I/O latency and throughput (we mix read and write performance in a model by taking the lower-performance operation as they are almost the same in Redy except a few corner cases as we describe in Section 5.7.):

$$f : (c, s, b, q) \rightarrow (\text{latency}, \text{throughput})$$

Given the highest number of client cores C, the largest batch size B defined by the record size, and the NIC-specific queue depth Q, the total number of configurations can be calculated as

$$(\sum_{c=1}^{C}(c+1)) \times B \times (Q - opt.) - C \times (B-1) \times (Q - opt.)$$

where we consider several configuration constraints: (1) the server core count is from zero and to the client core count; (2) if there are no server threads, then batching is disabled so the batch size is one; (3) the minimum queue depth is optimized by the fully-loaded QP technique. Overall the configuration space is $O(C^2 \times B \times Q)$.

In both modeling and searching, we explore the configuration space by incrementally increasing the value of every parameter in a resource-efficient fashion to minimize cost: explore the configurations that do not increase the hardware cost, i.e., increasing b and q, before the configurations that do, i.e., c and s. We increase c before s to minimize use of limited compute resources on remote memory servers, e.g., in a memory-disaggregated environment.

Formally, we define a Redy configuration space as a *five-level tree*. The root represents configuration options for s, the second level for c, the third for b, the fourth for q, and the leaves for latency and throughput. An internal node and the edges below it represents a parameter and its values in increasing order from left to right. A root-to-leaf path represents a configuration. A leaf is the latency and throughput of the path's configuration.

The construction of the tree enforces the aforementioned constraints. For example, all b nodes have only one child ($b = 1$) in the sub-tree of $s = 0$, and the c node under $s = S'$ has $C - S' + 1$ children (from S' to C). To explore the space, we do a pre-order traversal to visit configurations that require fewer server and client threads. In doing so, we are able to reduce overall hardware cost.

Offline Modeling. We use offline measurements to build a performance model (the function f). The model is sensitive to network latency, which varies depending the network distance between the cache client and cache server (cf. Figure 5.1). We build a performance model for each distance in a data-center-scale deployment. A typical data center network has three distances: one switch (intra-rack), three switches (intra-cluster), and five switches (inter-cluster).

The built-in measurement application on the client VM (the largest VM type for the deployment) allocates a server VM with enough cores (also for the largest configuration of interest). It then creates a Redy cache with an arbitrary configuration. The client starts the modeling by telling the manager the number of available cores for the cache, the record size, and the NIC-specific queue depth. The manager builds the tree representing the configuration space, with empty leaves. The manager and the client then repeatedly generate the next configuration to measure (❶) (see Figure 5.9), switch to that configuration, measure its latency and throughput by performing I/O operations on the cache, and report the result to the manager

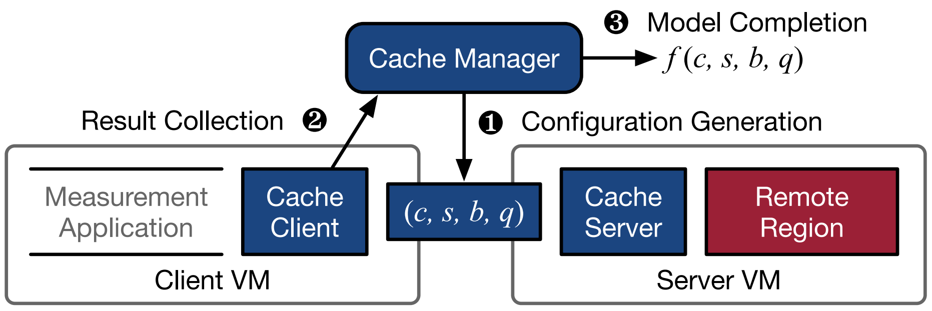


Figure 5.9: Configuration performance modeling.

(❷). When the manager determines that the model is complete (❸), it signals the application to terminate.

The Challenge and Solution. The performance modeling is done offline, when Redy is deployed in a new cloud RDMA environment. Still, the size of the configuration space poses a challenge. In our testbed, a VM has up to 60 cores, of which we assume half are available to a Redy cache, and the NIC-specific queue depth is 16. The model for 8-byte records has ∼3M configurations per network distance. If one measurement takes a minute, including switching to the new configuration, performing I/Os, and reporting the result, then building the model takes over five years to finish! So we cannot measure every configuration.

Our solution applies *interpolation* and *early termination*. With *interpolation*, we only measure configurations where parameter values are powers of 2, and we assume a linear growth of latency and throughput between adjacent measured configurations. For example, $f(1, 1, 1, 3)$ is estimated as the mean of $f(1, 1, 1, 2)$ and $f(1, 1, 1, 4)$. This effectively reduces the number of measurements to $O((\log C)^2 \times \log B \times \log Q)$, which is less than two thousand configurations in the above example.

Early termination removes unnecessary measurements. Ideally, increasing the value of each variable increases the throughput. However, due to factors such as thread and connection contention, increasing a parameter might not improve throughput while increasing latency. When this happens, we stop measuring configurations where only the value of that particular parameter increases. For instance, if the throughput does not improve from $f(4, 2, 2, 2)$ to $f(8, 2, 2, 2)$, there is no point in measuring $f(16, 2, 2, 2)$.

```
1  model ← find the model for the record size
2  config ← empty configuration
3  result ← Traverse(model.root, SLO, config, 1)      # Start the traversal from the root
4  if result = SUCCESS then
5      return config                      # Return the configuration that satisfies the SLO
6  return null

7  Function Traverse(node, SLO, config, level):
8      if level = 5 then
9          if node.latency > SLO.latency then
10             return INVALID                    # SLO can never be satisfied, so terminate
11         if node.throughput ≥ SLO.throughput then
12             return SUCCESS                            # SLO is just satisfied, so terminate
13         return CONTINUE                               # This is the last level, so terminate
14     p ← the parameter at this level
15     node_result ← INVALID
16     foreach child in node's children from left to right do
17         config.p ← edge value to child          # Get the value for the current parameter
18         child_result ← Traverse(child, SLO, config, level+1)     # One level deeper
19         if child_result = SUCCESS then
20             return SUCCESS                            # Configuration is found, so return
21         if child_result = INVALID then
22             return node_result         # Fail to satisfy SLO, so prune remaining children
23         if child_result = CONTINUE then
24             node_result ← CONTINUE                          # Continue the exploration
25     return node_result
```

Figure 5.10: Online SLO-based searching in the manager.

These two optimizations reduce the number of measurements for the above example to 1000, which took only 15 hours. Section 5.7 shows the accuracy of the estimated performance by interpolation. The resulting model will remain accurate if the hardware is stable, i.e., the NICs and switches. When hardware changes, the model should be updated by repeating the modeling, but we speculate that such hardware changes are infrequent, once every few years.

Online Searching. When the cache manager receives an *Allocate* request, it searches for a configuration to satisfy the given SLO. It uses the algorithm sketched in Figure 5.10, which traverses the configuration tree in pre-order with pruning to speed up the process.

Line 1 finds the model for the record size specified in the SLO. Line 2 allocates an empty configuration, which is used as the current configuration during the search. Line 3 invokes the traversal function, starting with the root of the model, s. If the traversal succeeds, then the algorithm returns `config`, *which is guaranteed to have the fewest server threads among all possible configurations and thus incurs minimal cost*; otherwise, it returns *null* (Lines 4-6).

If the current visited node is a leaf (Line 8) and the current configuration violates the latency SLO, then the traversal function returns an "invalid" status (Lines 9-10). If latency and throughput are satisfied, then the search returns "success" (Lines 11-12). Otherwise, the traversal explores internal nodes (Line 13). Line 14 identifies the parameter for the current level, and Line 15 initializes the search result as "invalid". Then Lines 16-25 visit the children of the current node left-to-right. For each child, it updates the current configuration parameter with the edge value and then recursively traverses the subtree rooted at the child (Lines 17-18). If the traversal succeeds, the search stops (Lines 19-20). If it returns "invalid", we can safely prune all the remaining children; since increasing the parameter value can only increase the latency, the latency SLO is violated for all of them (Lines 21-22). Finally, if the current child returns "continue", then the next child is visited (Lines 23-24).

In a test to search 100 random SLOs in a space of three million configurations, pruning reduces the number of explored leaf nodes by 25%. The average search time was only 0.027 seconds. Section 5.7 shows the quality of the returned configurations.

5.6 Remote Memory Management

5.6.1 Resource Allocation

Recall from Section 5.3 that an application invokes *Create* to provision a cache of a given capacity, SLO, and duration. The cache client services the function by issuing an *Allocate* with the same parameters to the cache manager.

First, the cache manager translates the capacity and SLO into an RDMA configuration for each network distance, as described in Section 5.5.2. Then it allocates a VM whose memory and CPU cores are sufficient for the RDMA configuration. Since RDMA configurations vary with network distances, the cache manager has to find the best VM for the configuration associated with each network distance and then choose the least expensive one.

The cache manager must choose VMs from the menu of VM sizes offered by the cloud provider. Each VM size has fixed cores and memory. Today, providers offer relatively few VM sizes with a high ratio of memory to cores and no VMs consisting of stranded memory. A wider range of choices would enable the manager to choose VMs that more closely match the desired RDMA configuration.

Since the set of VM types changes infrequently, the cache manager can maintain a static list of VM types, with each one's memory size and core count, and its price in each cloud region. To service an allocation request, it identifies the VM types in the client's data center with enough memory and cores and chooses one that has lowest cost and is available within the required network distance.

Beyond these static allocation strategies, there are many ways the cache manager can optimize the choice of VMs. They depend on the optimality criteria it uses and on the VM allocation mechanism of the cluster computing platform it runs on. We discuss these criteria and mechanisms below. In some cases, it may be cheaper for the cache manager to select two or more VMs that together satisfy the configuration. Each VM's core-to-memory ratio must be at least that of the configuration, to satisfy the SLO.

Additional cost savings are possible with a spot VM. This is an attractive choice if the cache can be migrated within the 30-120 seconds notice before its VM is reclaimed. This constraint argues for the use of many small VMs instead of a large one, to leave time to migrate each VM cache. We describe migration shortly.

Recent research has shown how to predict the lifetime of spot VMs [50]. This would enable the allocation of VMs that satisfy the requested duration. It could also suggest preemptively migrating a VM's cache, knowing it will likely be reclaimed soon.

At any given time, different VM types might have spot instances available. The cache manager can exploit such cost-saving opportunities by periodically issuing an allocation request for a cheap VM and migrating the cache to it when it becomes available.

The ability of the cache manager to optimize the choice of VMs could be improved by enriching the VM allocator's API. For example, to avoid having the cache manager poll for cheap VMs, the VM allocator could offer an option to alert the cache manager when spot VMs of a certain type become available. It could also offer an option to request the cheaper of one large VM or several smaller VMs based on current spot pricing. VM allocation for spot instances is an active research area. We discuss some recent work in Chapter 5.9.

5.6.2 Dynamic Memory Management

If the VM hosting a cache fails or is reclaimed, then the cache client is notified and must allocate another cache to replace it. For a failure, the cache client can use a copy of the cache to populate the new cache. For a reclamation, the cache client can migrate the cache's content to a new cache. The affected parts of cache are unavailable during recovery and possibly during migration. Afterwards, the entire cache is available and must satisfy its SLO.

The migration period depends in part on the time to provision a new VM. This might exceed the minimum time delay before the spot VM is reclaimed. If this risk is unacceptable or if a VM failure is too disruptive, the cache manager could hold pre-provisioned VMs as targets for migration. Another alternative is replicating the cache. Replica synchronization techniques can be found in [144, 255].

The migration speed also depends on the transfer rate. A tuned RDMA transfer in Redy can fully utilize the network bandwidth.

Migrating a Cache. To migrate the content of an existing cache to a newly allocated VM, the cache client needs to tell the new VM to establish a bandwidth-optimized connection with the existing cache. The new VM uses one-sided reads to copy data from the old VM. During the migration, operations on the migrated regions should be paused until the migration is finished. To minimize this performance impact, we employ two optimizations for reads and writes respectively: *unpaused reads* and *pause-on-migration writes*. In *unpaused reads*, we use the old VM to service read operations, and immediately switch to the new VM when the migration is over.

Unlike reads, writes have to be paused during the migration. But instead of pausing all writes (and dependent reads) to the cache, in *pause-on-migration writes*, we migrate regions one by one and pause writes only to the region being migrated. After a region has been migrated, the cache client updates its region table using the new VM and resumes paused writes. When all regions are migrated, the client signals the old VM to terminate. Section 5.7 evaluates the impact of migration on read and write performance, with and without the optimizations.

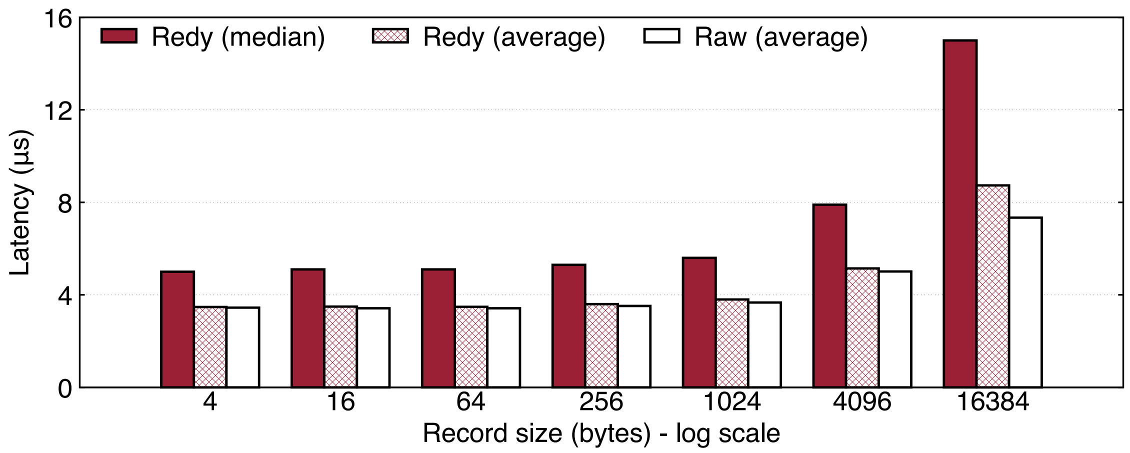
Resizing a Cache. In response to a *Reshape* invocation, the cache client executes operations to grow or shrink the size of the cache. To grow a cache, the client first uses any memory headroom available in the cache's last VM. For additional growth, the client allocates another VM, using the same memory-to-core ratio, batch size, and queue depth as existing VMs. Depending on the price of spot VMs, it could be cheaper (although more disruptive) to allocate a larger VM and migrate the content of the old VM to the new one. After the new VM is allocated, the client updates its region table. The cache client stalls I/O operations while the cache is being resized, although the techniques in cache migration can be applied to maintain some level of performance.

5.7 Evaluation

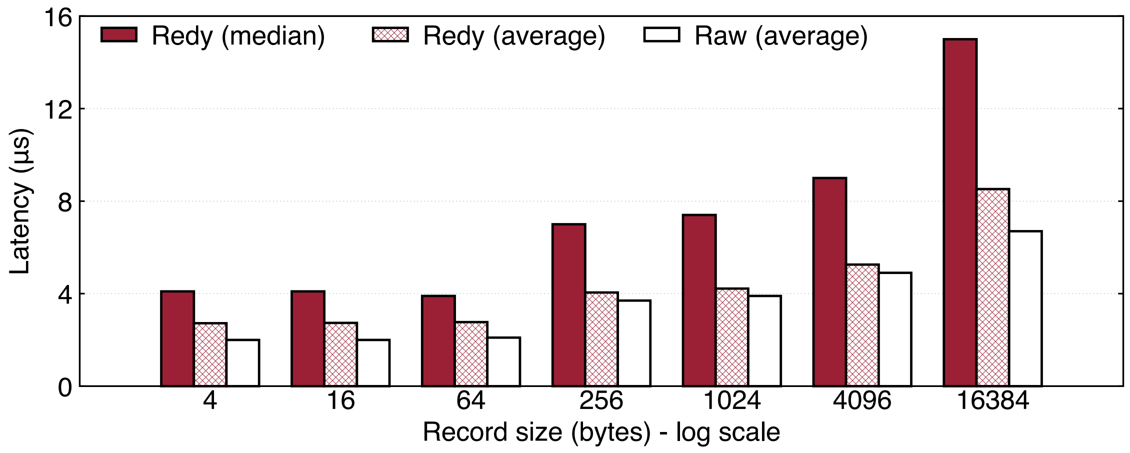
5.7.1 Methodology

Implementation. The implementation of Redy consists of 13700 lines of C++ code. It includes the cache client library for applications, cache manager, cache server (shown in Figure 5.4), and measurement application (in Figure 5.9). The client library has a Common Language Runtime (CLR) wrapper covering all APIs in Table 5.1, to enable access by applications in other languages, such as C#.

RDMA transfer in Redy uses the native RDMA library in Windows, NDSPI [191], which supports all RDMA operations. NDSPI has been used to implement other efficient RDMA-based systems, e.g., FaRM [96]. We implemented an RPC framework based on RDMA for



(a) Read latency with record sizes from 4 B to 16 KB.

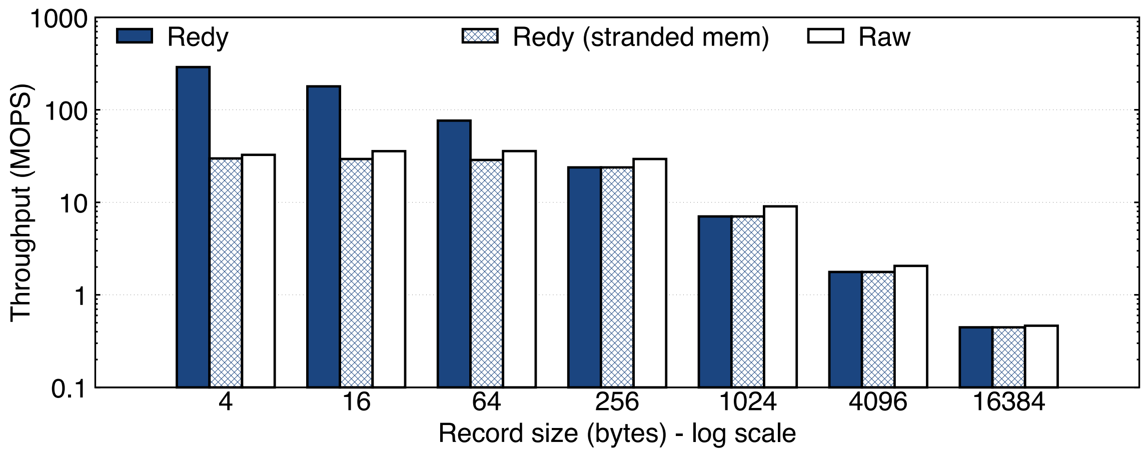


(b) Write latency with record sizes from 4 B to 16 KB.

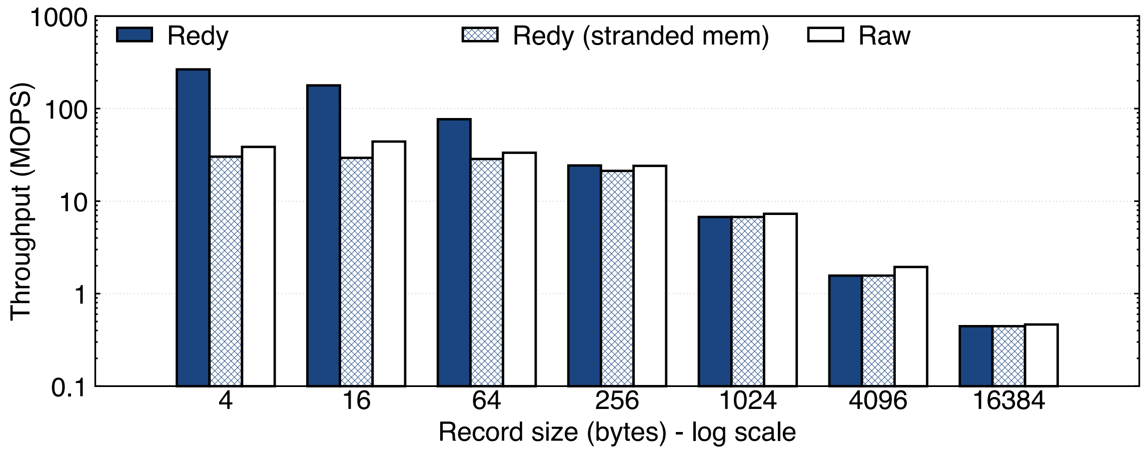
Figure 5.11: The latency of Redy caches with latency-optimal configurations for different record sizes. On average, accessing records up to 4 KB sizes takes less than 5 μs, close to the raw RDMA hardware speed.

efficient operations between clients, servers, and the manager.

Testbed Setup. We evaluate Redy on a Microsoft Azure High Performance Computing cluster [192] using the Standard_HB60rs VMs. Each VM has 60 vCPUs based on two 2.0 GHz AMD EPYC 7551 processors, 228 GB of memory, and a 700 GB Azure premium SSD. We run Windows Server 2019 Datacenter as the OS. Each VM is RDMA-enabled using an NVIDIA Mellanox ConnectX-5 NIC [20].



(a) Read throughput with record sizes from 4 B to 16 KB.



(b) Write throughput with record sizes from 4 B to 16 KB.

Figure 5.12: The throughput of Redy caches with throughput-optimal and stranded-memory configurations. Batching small records improves the throughput by an order of magnitude.

5.7.2 Overall Cache Performance

We first show the overall performance of Redy caches. In this evaluation, we vary data size from very small records (4 bytes) to large blocks (16 KB). For each size, we set up a cache with latency-optimal and throughput-optimal configurations. The purpose of this evaluation is to show Redy's optimal performance for each metric and for different sizes. We compare Redy's cache performance with the raw RDMA network. We measure the latter using the official benchmark tools from Mellanox [11], i.e., `nd_read_lat` and `nd_write_lat` for latency, and `nd_read_bw` and `nd_write_bw` for throughput.

Figures 5.11a and 5.11b show the results of latency benchmarking for reads and writes, respectively. Average latency is close to that of the raw network, 3-4 μs, showing the effectiveness of Redy's latency optimizations described in Section 5.4.3. An interesting finding is that the write latency is significantly lower than the read latency for records smaller than 256 bytes. This is because a small amount of data to be written can be *inlined* as a parameter in the RDMA *write* invocation, thereby avoiding the latency of fetching the data from main memory to the NIC through the PCIe buses. Inlining no longer works when the data exceeds a threshold (172 bytes in our testbed), so the latency increases. In general, the latency is steadily low until 4 KB records and increases significantly after that.

Figure 5.12 shows the results for throughput. Read and write throughput are similar. For example, both reading and writing 16 bytes can achieve about 200 MOPS, an order of magnitude higher than raw network throughput, showing that Redy batching is effective at utilizing the bandwidth. When the record size increases, throughput drops as fewer operations/second are needed to saturate the network. But up to 256 bytes, Redy performs much better than the raw network.

All latency-optimal configurations use one-sided memory access using no server cores, so Redy is particularly cheap for this case. Conversely, for record sizes up to 1KB, high-throughput configurations work best if they have a few cores to support batching.

Between latency-optimal and throughput-optimal configurations, there is a big space of configurations that make trade-offs between latency and throughput. We let the applications customize cache performance using their SLOs.

5.7.3 Performance Customizability

Offline modeling builds interpolated performance models whose accuracy determines whether they can satisfy users' SLOs. The speed of online searching determines how fast we can configure a cache. To evaluate both, we measure the accuracy of the model for the three million configurations in Section 5.5.2 and the time to search configurations for given SLOs using the algorithm in Figure 5.10.

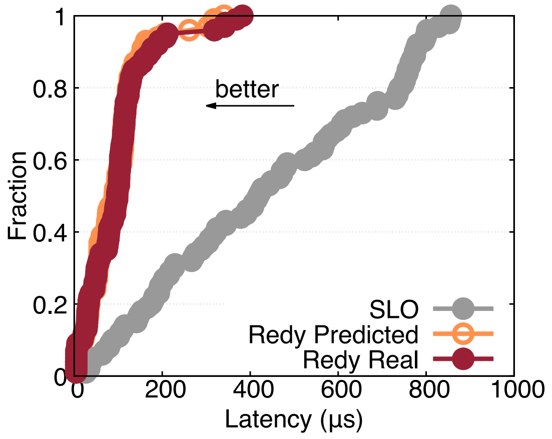


Figure 5.13: Satisfying latency SLOs.

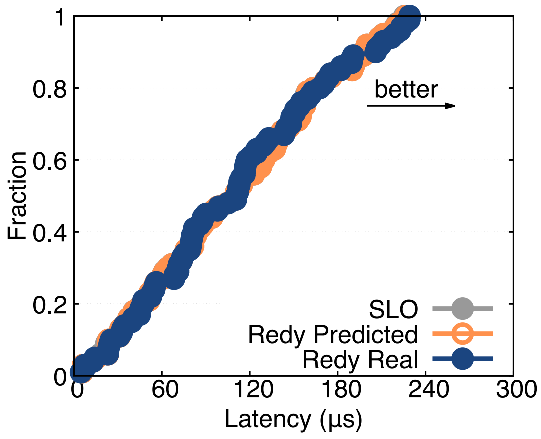


Figure 5.14: Satisfying throughput SLOs.

We draw 100 performance SLOs between the lowest and highest latency and throughput values in the model. An SLO consists of cache latency and throughput, which are drawn independently from a uniform distribution. For each SLO, we search the configuration space for one that Redy predicts will satisfy the SLO. We then configure the cache based on this configuration, measure its latency and throughput, and compare them with the SLO. The accuracy of the model is defined by how close the predicted latency and throughput mimic the real ones. High accuracy means that the real performance will satisfy the SLO.

Figures 5.13 and 5.14 show the results for both latency and throughput. Each figure shows three CDFs—the SLO, predicted, and real performance—of the corresponding metric. Since we draw SLOs randomly between the lowest and highest values, the SLOs in both figures are spread uniformly across their ranges. Figure 5.13 shows that the predicted and real latency are close: 92 μs vs. 98 μs at the median, and 206 μs vs. 212 μs at the 95th percentile. They are all lower than the requested latency—satisfying the SLO. Figure 5.14 shows findings for throughput: the predicted and real throughput values are 110.5 MOPS and 110.7 MOPS respectively at the median, both closely matching the requested throughput of 110.4 MOPS, and are 211.5 MOPS and 219.3 MOPS at the 95th percentile, also close to the requested 211.4 MOPS. The latency of the caches is much lower than the SLOs, while the throughput just reaches the SLOs, because the searching algorithm in Figure 5.10 starts from low-latency low-throughput configurations and gradually moves toward high-latency and high-throughput

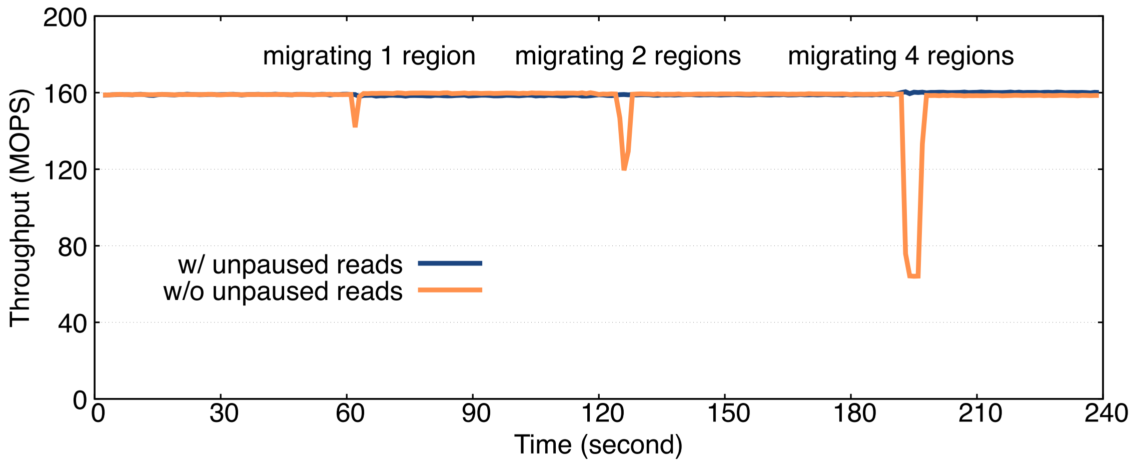


Figure 5.15: The impact of region migration on reads.

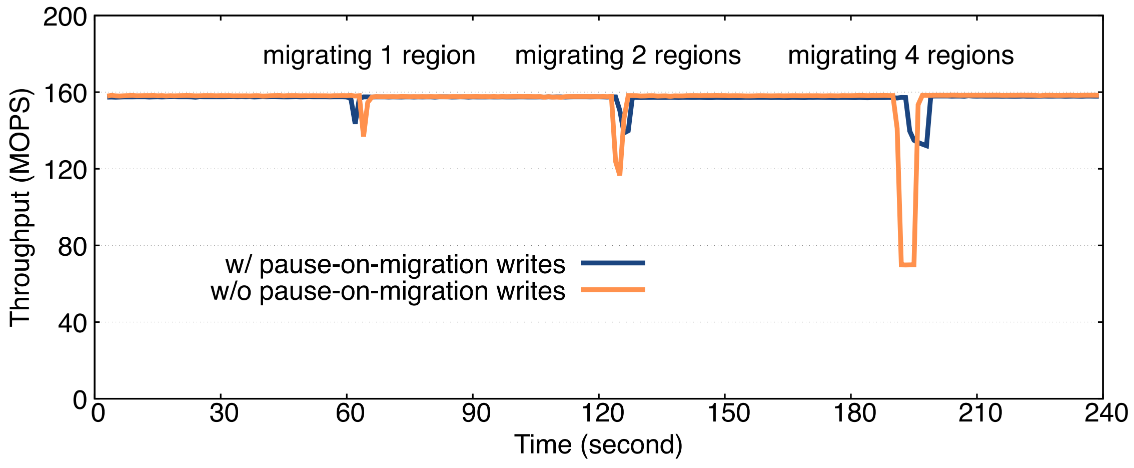


Figure 5.16: The impact of region migration on writes.

ones. This matches our cost-efficient goal: the average client and server core counts of the resulting configurations are 7.3 and 1.5.

Redy is also fast at finding configurations. The time spent on searching for the right configuration for an SLO in the space is 2 μs to 0.12 s with an average of 0.027 s and a median of 0.01 s, achieving interactive speed for cache allocation.

5.7.4 Robustness to Dynamics

When a VM that hosts a part of a cache is to be reclaimed, the cache client requests a new VM (or multiple VMs) from the manager and migrates the affected regions using a throughput-optimized configuration. In our testbed, it takes 1.09 s to online migrate a 1 GB region. This argues for using spot VMs of $\leq$ 27GB, to ensure they can be migrated within 30s. Thus, it is feasible to use spot VMs that are available for only a short time, say a few minutes.

We evaluate the performance impact of region migration using a cache that consists of seven 1 GB regions. Initially, all regions are hosted in one VM. We run this cache with 8-byte records for four minutes and migrate one, two, and four regions at the second, third, and fourth minute respectively to a different VM. We measure the throughput change. Figures 5.15 and 5.16 show that the throughput of both reads and writes drops by around 15%, 25%, and 57% in the migration of one, two, and four regions without optimizations. By contrast, the read throughput with *unpaused reads* is unaffected by the migration, and the write throughput with *pause-on-migration writes* decreases by at most 15%, no matter how many regions are migrated. This demonstrates that Redy minimizes the impact of resource dynamics.

5.8 FASTER with Redy

FASTER is a high-performance open-source key-value store that is used at Microsoft and elsewhere [77, 190]. It is an example of a stateful cloud service that can benefit from using a remote cache, as discussed in Section 5.1.1. We integrate Redy with FASTER to demonstrate its ease of use and practical value.

5.8.1 Data Organization in FASTER

FASTER runs as a multi-threaded library in the address space of an application client. It has a hash index that maps keys to record addresses. The index is stored in the client's memory.

FASTER stores records in a *hybrid log* where the tail of the log is stored in main memory and the remainder is spilled to storage, such as a server-attached SSD or a cloud storage service. The log is organized as a sequence of *segments*. The tail of the main-memory section supports in-place updates. The rest is read-only.

A read operation looks up a record in the index and then retrieves it from memory or

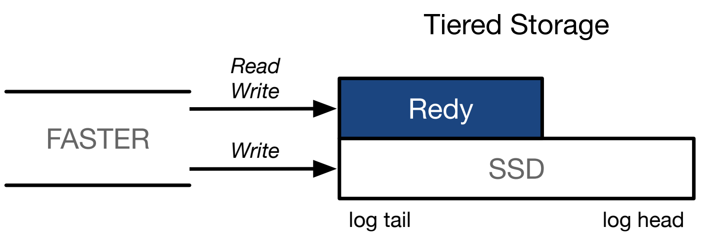


Figure 5.17: FASTER with Redy. New records are appended to both tiers. Reads to records in Redy are only served by Redy.

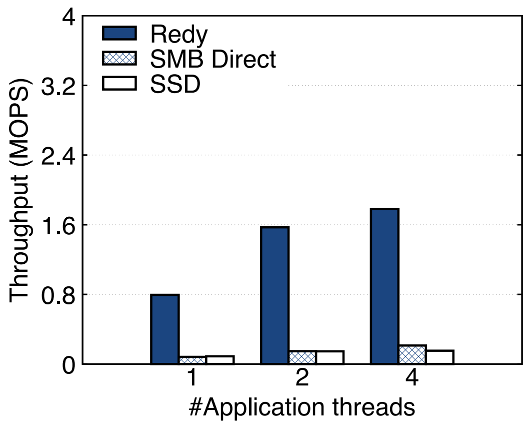
storage. To insert a record, it is appended to the tail and added to the index. To update a record in the read-only portion of the log, it is appended to the tail in main memory and its index entry is updated. To free up main memory, the oldest segment of the read-only main memory section of the log is appended to storage. To free up storage, the oldest segment is read, its reachable records are appended to the log tail, and then it is deallocated.

5.8.2 Integrating Redy

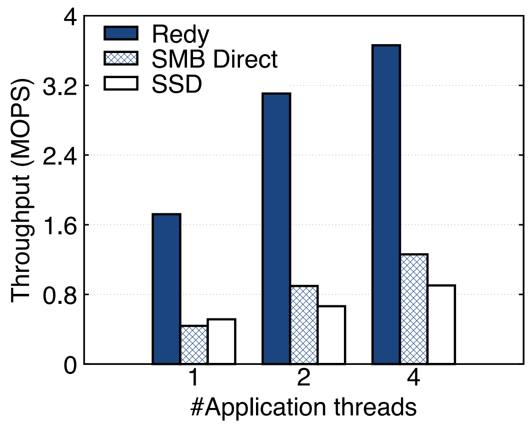
FASTER clients access storage through an interface called `IDevice`, which exposes storage as a byte-addressable sequential address space. FASTER supports *tiered storage*, which is a "meta-device" that wraps a set of `IDevice` implementations, called *tiers*. Each tier is smaller and faster than the next higher tier, and is a replica of a suffix (i.e., tail) of the higher tiers. FASTER services a read operation from the lowest tier that has the data.

To keep the tiers consistent, an append operation is applied to all tiers. It is acknowledged to the client after all tiers have applied the append. A user can alter this semantics via FASTER's *commit point* setting, which is the lowest tier whose commit denotes the completion of an update. This is useful for committing quicker than the highest tier, which may be very slow.

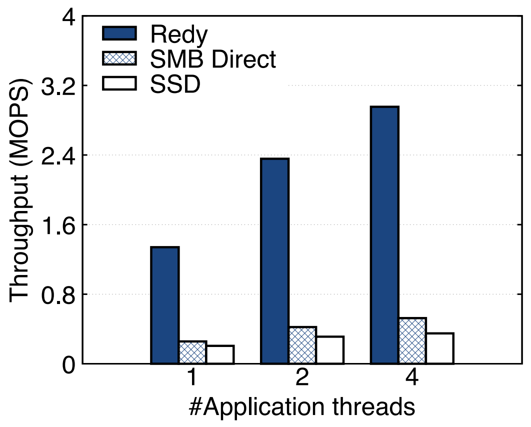
We integrate Redy as an `IDevice` in this tiered storage, as the first tier (see Figure 5.17). An SSD is the second tier, which contains the entire log. Thus, reads are serviced by Redy if the record is stored in the Redy cache. Otherwise, it is serviced by the SSD. Cloud blob storage could be a third tier, as a highly-available backup.



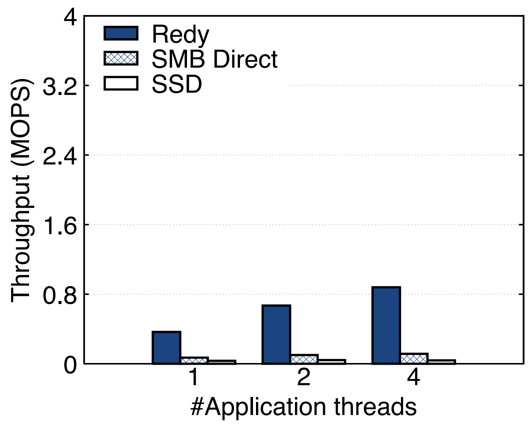
(a) YCSB (uniform), 8B val, 1GB local mem.



(b) YCSB (Zipf), 8B val, 1GB local mem.



(c) YCSB (Zipf), 8B val, 0.1GB local mem.

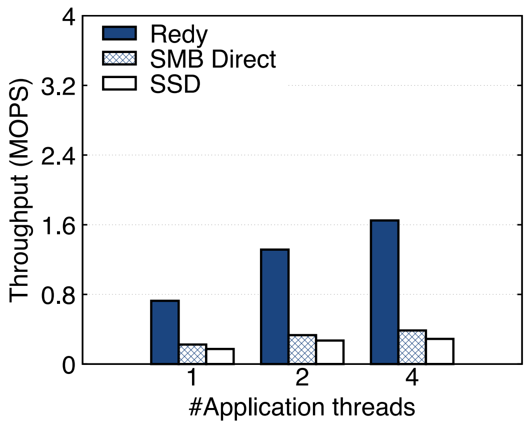


(d) YCSB (uniform), 1KB val, 1GB local mem.

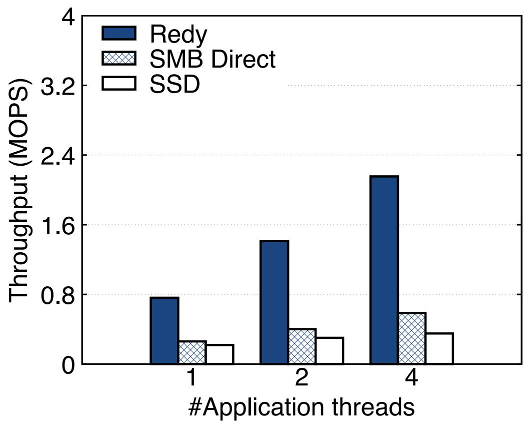
Figure 5.18: FASTER throughput with Redy, SMB Direct, and SSD respectively on the YCSB benchmark when the working set is larger than local memory.

5.8.3 Evaluation

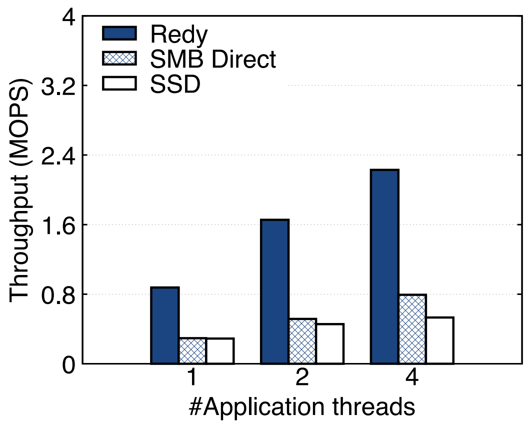
We evaluate the performance of FASTER with Redy using the YCSB benchmark [87] in the same cloud environment as Section 5.7. We compare with two alternatives: a device that only uses local SSD; and a device that accesses remote memory using SMB Direct, an RDMA-enabled file server protocol with higher throughput and lower latency than the regular Windows file server [14]. Throughput is the critical metric for this benchmark, so we configure the Redy cache for high throughput. Our YCSB database contains 250 million key-value records (8-byte key and 8-byte value), ~6 GB in total in FASTER. Every operation is a read governed by either a uniform distribution or a Zipfian distribution ($\theta = 0.99$). Additionally, we use a



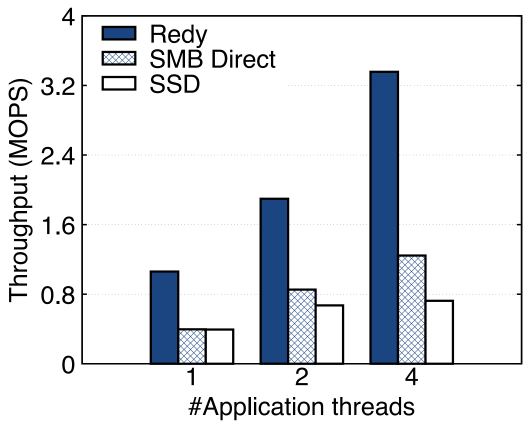
(a) YCSB (Zipf), 1KB val, 10GB local mem.



(b) YCSB (Zipf), 1KB val, 20GB local mem.



(c) YCSB (Zipf), 1KB val, 40GB local mem.



(d) YCSB (Zipf), 1KB val, 80GB local mem.

Figure 5.19: FASTER throughput when varying local memory size.

value size of 1 KB, resulting in a $\sim$260 GB database.

Figure 5.18a shows the throughput of FASTER in MOPS on the uniform workload, with different storage devices. In this experiment, we give FASTER 1 GB of local memory, and the remainder of the log is spilled to the device. In the tiered device, we allocate an 8 GB Redy cache so that all operations are served by Redy. When there is one thread, FASTER achieves 0.8 MOPS with Redy while SMB Direct and SSD are much lower with less than 0.1 MOPS. With two threads, the throughput with Redy increases to 1.6 MOPS. With SMB Direct and SSD it improves to 0.15 MOPS, but that still is 10$\times$ lower than Redy. Adding more threads improves FASTER's performance with all devices, but the gap between Redy and other alternatives remains large. Figure 5.18b shows the results with the Zipf distribution where data

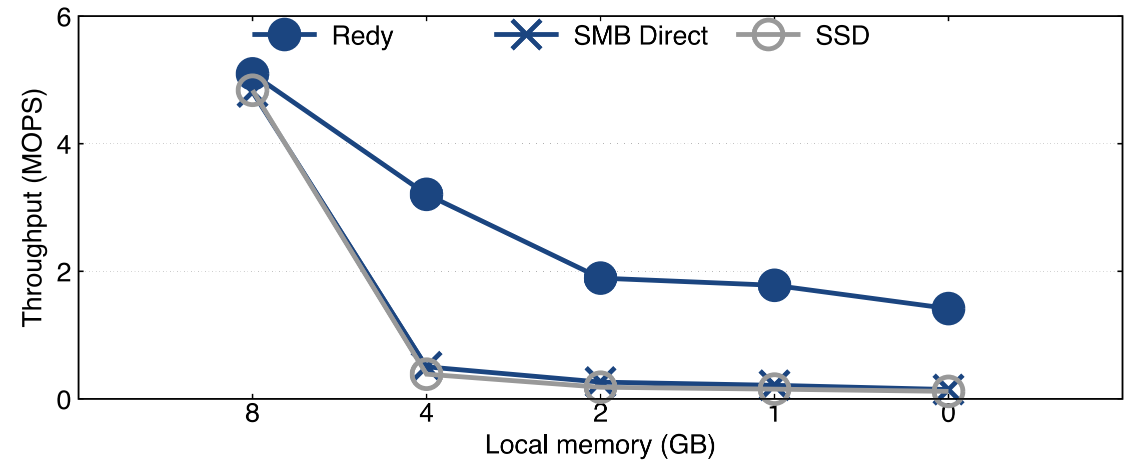


Figure 5.20: FASTER with various local memory sizes on uniform YCSB.

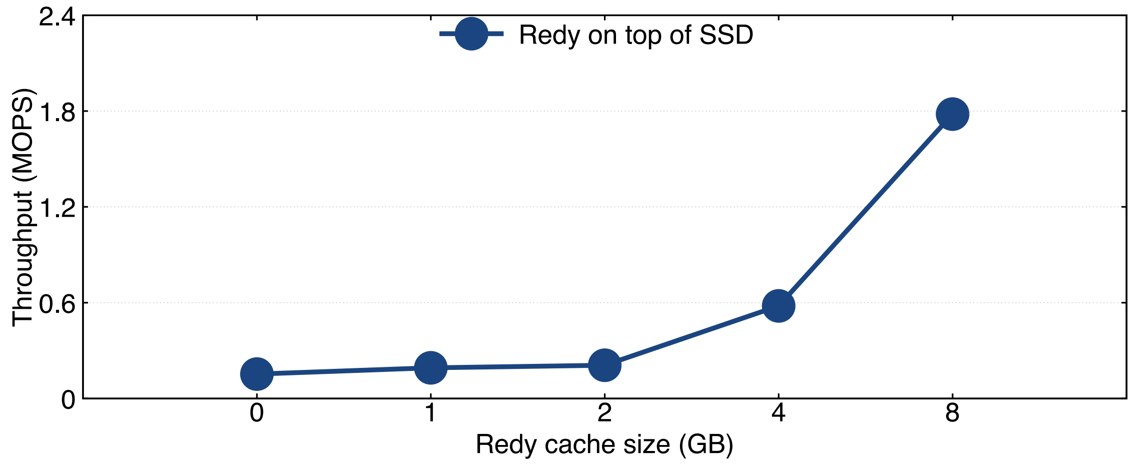


Figure 5.21: Tiered store with various remote cache sizes.

accesses are skewed. FASTER uses local memory to cache frequently-accessed records, which reduces load to the devices. Hence, the throughput is higher than that with the uniform distribution for all devices. However, when we decrease the available local memory for caching in FASTER (similarly when we increase the database size), both the absolute throughput and the relative difference between Redy and other devices become closer to that of the uniform distribution, as shown in Figure 5.18c.

FASTER with Redy achieves higher throughput for large records as well. Figure 5.18d shows that with four threads, the throughput of accessing records with 1 KB values is 0.9 MOPS with Redy, 8× and 20× higher than with SMB Direct and SSD respectively. Figures 5.19a–

5.19d show that even when the client has a local cache as large as 10 GB, 20 GB, 40 GB, and 80 GB respectively, the tail of the Zipfian distribution still bottlenecks the overall performance. Spilling requests to Redy has at least 2× higher throughput than other cloud services, i.e., SMB Direct and SSD storage.

Figure 5.20 varies the size of local memory used by FASTER (with four threads). With 8 GB local memory, FASTER services all (uniform) operations from local memory, achieving high throughput of 5 MOPS. When we spill the entire log to the storage device, FASTER achieves 1.4 MOPS using Redy, vs. 0.15 MOPS and 0.12 MOPS for SMB Direct and SSD. Compared to local memory only, the performance of FASTER with Redy decreases by 72% (vs. 97% with SMB Direct and 98% with SSD); but it saves memory cost by 100%, since it uses stranded memory, which is essentially free.

To show the impact of the cache size in the tiered device we vary the Redy cache size from 0 to 8 GB with 1 GB client local memory (Figure 5.21). As expected, performance increases significantly when more cache is allocated.

In summary, when FASTER's working set exceeds local memory, spilling data to a Redy cache results in better performance than spilling to the RDMA baseline or SSD. We note FASTER using synchronous local-memory outperforms the asynchronous device interface due to I/O code path and context switching overheads. As new high-throughput devices such as Redy become commonplace, we believe this is an important area for future optimization.

5.9 Related Work

Redy is an RDMA-accessible remote dynamic cache targeted for data centers. No systems that we know of offer Redy's SLO-based configuration and dynamic reconfiguration. We summarize related systems and explain the differences as follows.

Cache Servers. A cache server is an in-memory distributed key-value store that supports access by a large number of clients. It is typically used to store content that is accessed over the Internet. Popular cache servers are Memcached [18] and Redis [22]. By contrast, Redy offers inexpensive remote caches in the cloud environment. CompuCache [282] is a cloud service that supports both data caching and compute offloading. However, since it uses RPC, it cannot use stranded memory.

RDMA. RDMA has been a subject for research in the database, systems, and networking communities for many years [105]. Herd [137] and FaRM [96] are RDMA-accessible key-value stores. FaRM also supports multistep transactions, as does [63, 65]. DFI [256] provides a data flow abstraction based on RDMA. Cai et al. [72] propose a distributed shared memory framework with an RDMA-based memory coherence protocol. Liu et al. [169] optimize the bandwidth of RDMA specifically for shuffles. Ziegler et al. [296] report on microbenchmarks of RDMA. Li et al. [161] explore RDMA performance benefits to a DBMS via SMB Direct. Redy is different from these works in its cache design that supports fine-grained data accesses and performance customizability. Kalia et al. [138] provide RDMA developers with guidelines for low-level RDMA optimizations. In comparison, Redy hides RDMA complexities with an easy-to-use cache API. Kalia et al. also explore the benefit of batching, but speculatively and only for unconnected QPs.

VM Scheduling and Migration. Redy's cache manager uses the cluster VM scheduler to allocate VMs for caches. The challenges of allocating VMs for large data centers are discussed in [93, 126, 210, 231]. Redy's allocator is rather unique in requiring a minimum amount of memory that can be partitioned across multiple VMs, each VM satisfying a minimum ratio of cores to memory.

Redy migrates cache when its VM fails or is evicted. This is similar to VM migration, but without the need to freeze program execution to move its state. Some past work on VM migration includes [86, 237, 246, 262]. To mitigate the effect of VM eviction, researchers are exploring dynamic alternatives, where VMs can shrink or grow to offer all unallocated resources on the server where it runs [50, 236]. Extending Redy's ability to exploit dynamic resource allocation is an interesting avenue for future work.

5.10 Summary

This chapter describes Redy, a cloud service that provides high performance caches using RDMA-accessible remote memory. Redy automatically configures resources for a given latency and throughput SLO and automatically recovers from failures and evictions of remote memory regions. We integrated Redy with a production key-value store, FASTER. The experimental evaluation shows that Redy can deliver its promised performance and robustness and hence serve as an efficient solution to data center resource underutilization.

CHAPTER 6

FACILITATING HYPERSCALE NETWORK INNOVATION

Data centers evolve fast, with innovations like resource disaggregation that the previous part of this dissertation focused on. Unfortunately, at-scale evaluation of new data center network innovations is becoming increasingly intractable. This is true for both testbeds, where few can afford a dedicated, full-scale replica of a data center, and simulations, which while originally designed for precisely this purpose, have struggled to cope with the size of today's networks.

This chapter presents an approach for quickly obtaining performance estimates for large data center networks with high accuracy. Our system, MimicNet, provides users with the familiar abstraction of a fine-grained, packet-level simulation for a portion of the network while leveraging redundancy and recent advances in machine learning to quickly and accurately approximate portions of the network that are not directly visible. MimicNet can provide over two orders of magnitude speedup compared to regular simulation for a data center with thousands of servers. Even at this scale, MimicNet estimates of the tail FCT, throughput, and RTT are within 5% of the true results.

6.1 Introduction

Over the years, many novel protocols and systems have been proposed to improve the performance of data center networks [92, 46, 196, 122, 171, 45, 47]. Though innovative in their approaches and promising in their results, these proposals suffer from a consistent challenge:

the difficulty of evaluating systems at scale. Networks, highly interconnected and filled with dependencies, are particularly challenging in that regard—small changes in one part of the network can result in large performance effects in others.

Unfortunately, full-sized data center testbeds that could capture these effects are prohibitively expensive to build and maintain. Instead, most pre-production deployments comprise orders of magnitude fewer devices and fundamentally different network structures. This is true for (1) hardware testbeds [223], which provide total control of the system, but at very high cost; (2) emulated testbeds [208, 263, 275], which model the network but generally at the cost of scale or network effects; and (3) small regions of the production network, which provide *'in vivo'* accuracy but force operators to make a trade-off between scale and safety [294, 239]. The end result is that, often, the only way to ascertain the true performance of the system, at-scale, is to deploy it to the production network.

We note that simulation was originally intended to fill this gap. In principle, simulations provide an approximation of network behavior for arbitrary architectures at an arbitrary scale. In practice, however, modern simulators struggle to provide both simultaneously. As we show in this chapter, even for relatively small networks, packet-level simulation is 3–4 orders of magnitude slower than real-time (5 min of simulated time every $\sim$3.2 days); larger networks can easily take months or longer to simulate. Instead, researchers often either settle for modestly sized simulations and assume that performance translates to larger deployments, or they fall back to approaches that ignore packet-level effects like flow approximation techniques. Both sacrifice substantial accuracy.

In this chapter, we describe MimicNet, a tool for fast *performance estimation* of at-scale data center networks. MimicNet presents to users the abstraction of a packet-level simulator; however, unlike existing simulators, MimicNet only simulates—at a packet level—the traffic to and from a single 'observable' cluster, regardless of the actual size of the data center. Users can then instrument the host and network of the designated cluster to collect arbitrary statistics. For the remaining clusters and traffic that are *not* directly observable, MimicNet approximates their effects with the help of deep learning models and flow approximation techniques.

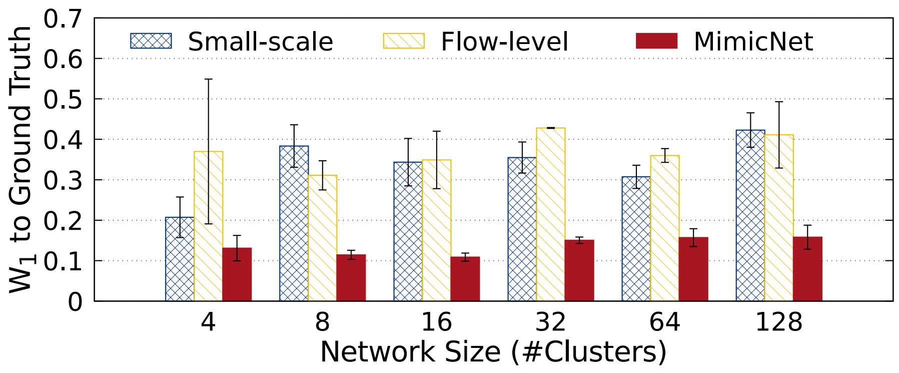


Figure 6.1: Accuracy for MimicNet's predictions of the FCT distribution for a range of data center sizes. Accuracy is quantified via the Wasserstein distance (W_1) to the distribution observed in the original simulation. Lower is better. Also shown are the accuracy of a flow-level simulator (SimGrid) and the accuracy of assuming a small (2-cluster) simulation's results are representative.

As a preview of MimicNet's evaluation results, Figure 6.1 shows the accuracy of its Flow-Completion Time (FCT) predictions for various data center sizes and compares it against two common alternatives: (1) flow-level simulation and (2) running a smaller simulation and assuming that the results are identical for larger deployments. For each approach, we collected the simulated FCTs of all flows with at least one endpoint in the observable cluster. We compared the distribution of each approach's FCTs to that of a full-fidelity packet-level simulation using a W_1 metric. The topology and traffic pattern were kept consistent, except in the case of small-scale simulation where that was not possible (instead, we fixed the average load and packet/flow size). While MimicNet is not and will never be a perfect portrayal of the original simulation, it is 4.1× more accurate than the other methods across network sizes, all while improving the time to results by up to two orders of magnitude.

To achieve these results, MimicNet imposes a few carefully chosen restrictions on the system being modeled: that the data center is built on a classic FatTree topology, that per-host network demand is predictable a priori, that congestion occurs primarily on fan-in, and that a given host's connections are independently managed. These assumptions provide outsized benefits to simulator performance and the scalability of its estimation accuracy, while still permitting application to a broad class of data center networking proposals, both at the end host and in the network.

Concretely, MimicNet operates as follows. First, it runs a simulation of a small subset of the larger data center network. Using the generated data, it trains a Mimic—an approximation of clusters' 'non-observable' internal and cross-cluster behavior. Then, to predict the performance of an N cluster simulation, it carefully composes a single observable cluster with $N-1$ Mimic'ed clusters to form a packet-level generative model of a full-scale data center. Assisting with the automation of this training process is a hyperparameter tuning stage that utilizes arbitrary user-defined metrics (e.g., FCT, RTT, or average throughput) and MimicNet-defined metrics (e.g., scale generalizability) rather than traditional metrics like L1/2 loss, which are a poor fit for a purely generative model.

This entire process—small-scale simulation, model training/tuning, and full-scale approximation—can be orders of magnitude faster than running the full-scale simulation directly, with only a modest loss of accuracy. For example, in a network of a thousand hosts, MimicNet's steps take 1h3m, 7h10m, and 25m, respectively, while full simulation takes over a week for the same network/workload. These results hold across a wide range of network configurations and conditions extracted from the literature. This work contributes:

- Techniques for the modeling of cluster behavior using deep-learning techniques and flow-level approximation. Critical to the design of the Mimic models are techniques to ensure the scalability of their accuracy, i.e., their ability to generalize to larger networks in a zero-shot fashion.

- An architecture for composing Mimics into a generative model of a full-scale data center network. For a set of complex protocols and real-world traffic patterns, MimicNet can match ground-truth results orders of magnitude more quickly than otherwise possible. For large networks, MimicNet even outperforms flow-level simulation in terms of speed (in addition to producing much more accurate results).

- A customizable hyperparameter tuning procedure and loss function design that ensure optimality in both generalization and a set of arbitrary user-defined objectives.

- Implementations and case studies of a wide variety of network protocols that stress MimicNet in different ways.

The framework is available at: `https://github.com/eniac/MimicNet`.

6.2 Motivation

Modern data center networks connect up to hundreds of thousands of machines that, in aggregate, are capable of processing hundreds of billions of packets per second. They achieve this via scale-out network architectures, and in particular, FatTree networks [41, 119, 245]. In the canonical version, the network consists of Top-of-Rack (ToR), Cluster, and Core switches. We refer to the components under a single ToR as a *rack* and the components under and including a group of Cluster switches as a *cluster*. A large data center might have over 100 such clusters.

The size and complexity of these networks make testing and evaluating new ideas and architectures challenging. Researchers have explored many potential directions including verification [51, 103, 146, 147, 180], emulation [254, 263, 275], canaries [239, 294], and runtime monitoring [123, 291]. In reality, all of these approaches have their place in a deployment workflow; however, this chapter focuses on a critical early step: pre-deployment performance estimation using simulation.

6.2.1 Background on Network Simulation

The most popular simulation frameworks include OMNeT++ [177], ns-3 [205], and OPNET [26]. Each of these operates at a packet-level and are built around an event-driven model [261] in which the operations of every component of the network are distilled into a sequence of events that each fire at a designated 'simulated time.' Compared to evaluation techniques such as testbeds and emulation, these simulators provide a number of important advantages:

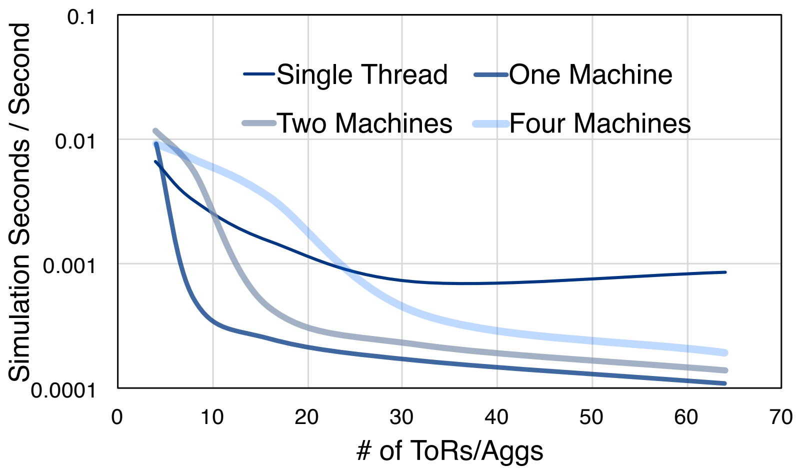


Figure 6.2: OMNeT++ performance on leaf-spine topologies of various size. Even for these small cases, 5 mins of simulation time can take multiple days to process. Results were similar for ns-3 and other frameworks.

- *Arbitrary scale:* Decoupling the system model from both hardware and timing constraints means that, in principle, simulations can encompass any number of devices.
- *Arbitrary extensions:* Similarly, with full control over the simulated behavior, users can model any protocol, topology, design, or configuration.
- *Arbitrary instrumentation:* Finally, simulation allows the collection of arbitrary information at arbitrary granularity without impacting system behavior.

In return for the above benefits, simulators trade off varying levels of accuracy compared to a bare-metal deployment. Even so, prior work has demonstrated their value in approximating real behavior [45, 46, 171, 221, 270].

6.2.2 Scalability of Today's Simulators

While packet-level simulation is easy to reason about and extend, simulating large and complex networks is often prohibitively slow. One reason for this is that discrete-event simulators, in essence, take a massive distributed system and serialize it into a single event queue. Thus, the larger the network, the worse the simulation performs in comparison.

Parallelization. A natural approach to improving simulation speed is parallelization, for instance, with parallel DES (PDES) [107]. In PDES, the simulated network is partitioned into multiple *logical processes*, where each process has its own event queue that is executed in parallel. Eventually, of course, the processes must communicate. In particular, consistency demands that a logical process cannot finish executing events at simulated time t unless it can be sure that no other process will send it additional events at $t_e < t$. In these cases, synchronization may be necessary.

Parallel execution is therefore only efficient when the logical processes can run many events before synchronization is required, which is typically not the case for highly interconnected data center networks. In fact, simulation performance often *decreases* in response to parallelization (see Figure 6.2). Many frameworks instead recommend running several instances with different configurations [100]. This trivially provides a proportional speedup to aggregate simulation throughput, but does not improve the time to results.

Approximation. The other common approach is to leverage various forms of approximation. For instance, flow-level approaches [195] average the behavior of many packets to reduce computation. Closed-form solutions [187] and a vast array of optimized custom simulators [171, 216, 221] also fall in this category. While these approaches often produce good performance; they require deep expertise to craft and limit the metrics that one can draw from the analysis.

6.3 Design Goals

MimicNet is based around the following design goals:

- *Arbitrary scale, extensions, and instrumentation:* Acknowledging the utility of packet-level simulation in enabling flexible and rich evaluations of arbitrary network designs, we seek to provide users with similar flexibility with MimicNet.

- *Orders of magnitude faster results:* Equally important, MimicNet must be able to provide meaningful performance estimates several orders of magnitude faster than existing approaches. Parallelism, on its own, is not enough—we seek to decrease the total amount of work.

- *Tunable and high accuracy:* Despite the focus on speed, MimicNet should produce observations that resemble those of a full packet-level simulation. Further, users should be able to define their own accuracy metrics and to trade this accuracy off with improved time to results.

Explicitly *not* a goal of our framework is full generality to arbitrary data center topologies, routing strategies, and traffic patterns. Instead, MimicNet makes several carefully chosen and domain-specific assumptions (described in Section 6.4.2) that enable it to scale to larger network sizes than feasible in traditional packet-level simulation. We argue that, in spite of these restrictions, MimicNet can provide useful insights into the performance of large data centers, which we validate in Section 6.9.

6.4 MimicNet Overview

MimicNet's approach is as follows. Every MimicNet simulation contains a single 'observable' cluster, regardless of the total number of clusters in the data center. All of the hosts, switches, links, and applications in this cluster as well as all of the remote applications with which it communicates are simulated in full fidelity. All other behavior—the traffic between unobserved clusters, their internals, and anything else not directly observed by the user—is approximated by trained models.

While prior work has also attempted to model systems and networks (e.g., [263, 275]), these systems tend to follow a more traditional script by (1) observing the *entire* system/network and (2) fitting a model to the observations. MimicNet is differentiated by the insight that, by carefully composing models of small pieces of a data center, *we can accurately approximate the full data center network using only observations of small subsets of the network*.

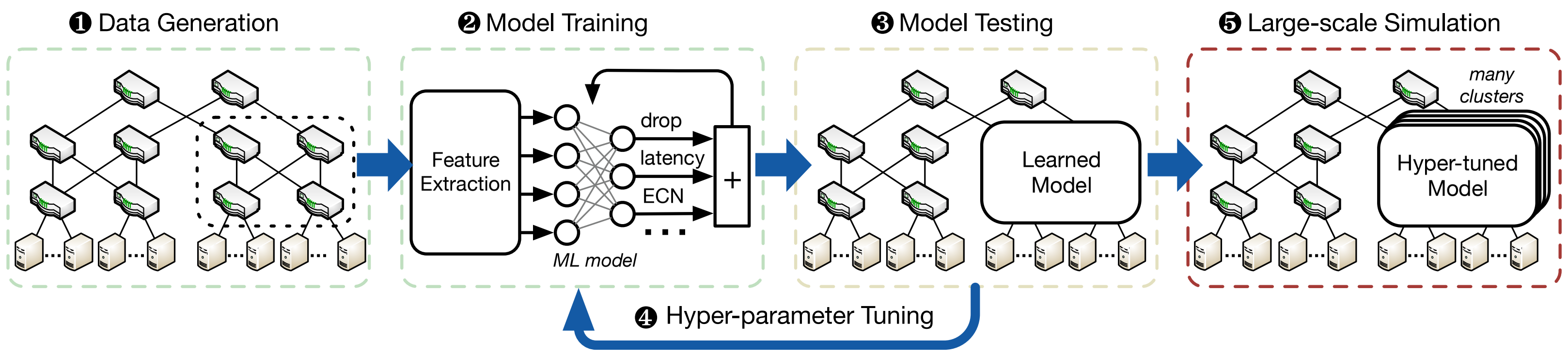


Figure 6.3: The end-to-end, fully automated workflow of MimicNet.

6.4.1 Design

MimicNet constructs and composes models at the granularity of individual data center clusters: Mimics. From the outside, Mimics resemble regular clusters. Their hosts initiate connections and exchange data with the outside world, and their networks drop, delay, and modify that traffic according to the internal queues and logic of the cluster's switches. However, Mimics differ in that they are able to predict the effects of that queuing and protocol manipulation without simulating or interacting with other Mimics—only with the observable cluster.

We note that the goal of MimicNet is not to replicate the effects of any particular large-scale simulation, just to generate results that exhibit their characteristics. It accomplishes the above with the help of two types of models contained within each Mimic: (1) learning-based *internal models* that learn the behavior of switches, links, queues, and intra-cluster cross-traffic; and (2) flow-based *feeder models* that approximate the behavior of inter-cluster cross-traffic. The latter is parameterized by the size of the data center. Together, these models take a sequence of observable packets and their arrival times and output the cluster's predicted effects:

- Whether the packets are dropped as a result of the queue management policy.
- When the packets egress the Mimic, given no drop.
- Where the packets egress, based on the routing table.
- The contents of the packets after traversing the Mimic, including modifications such as TTL and ECN.

Workflow. The usage of MimicNet is depicted in Figure 6.3. It begins with a small subset of the full simulation: just two user-defined clusters communicating with one another (❶). This full-fidelity, small-scale simulation is used to generate datasets for training (❷) and testing (❸) with supervised learning of the models described above. Augmenting this training phase is a configurable hyper-parameter tuning stage in which MimicNet explores various options for modeling with the goal of maximizing both (a) user-defined, end-to-end accuracy metrics

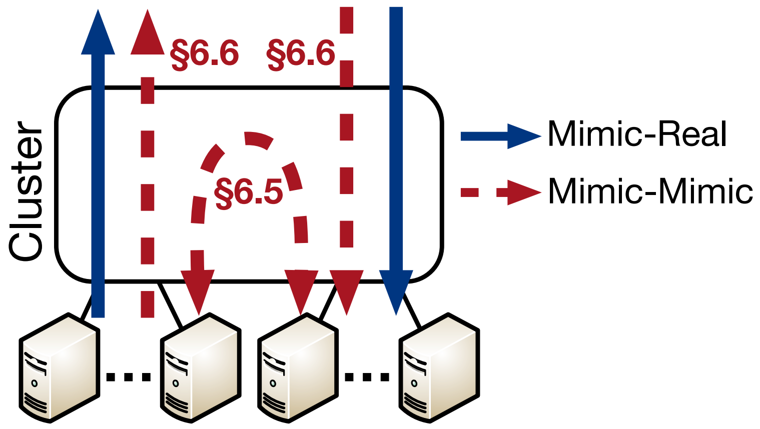


Figure 6.4: Breakdown of traffic in a to-be-approximated cluster. MimicNet approximates all traffic that does not interact with the observable cluster (dotted-red lines) using the models in the referenced sections.

like throughput and FCT, and (b) generalizability to larger configurations and different traffic matrices (❹).

Using the trained models, MimicNet assembles a full-scale simulation in which all of the clusters in the network (save one) are replaced with Mimics (❺). For both data generation and large-scale simulation, MimicNet uses OMNeT++ as a simulation substrate. A key feature of MimicNet is that the traditionally slow steps of ❶, ❷, ❸, and ❹ are all done at small scale and are, therefore, fast as well.

Performance analysis. To understand MimicNet's performance gains, consider the Mimic in Figure 6.4 and the types of packets that flow through it. At a high level, there are two such types: (1) traffic that interacts with the observable cluster (Mimic-Real), and (2) traffic that does not (Mimic-Mimic).

As a back-of-the-envelope computation, assume that we simulate N clusters, $N \gg 2$. Also assume that T is the total number of packets sent in the full simulation of the data center and that p is the ratio of traffic that leaves a cluster vs. that stays within it (inter-cluster-to-intra-cluster), $0 \leq p \leq 1$. The number of packets that leave a single cluster in the full simulation is then approximately $\frac{Tp}{N}$.

Because Mimics only communicate with the single observable cluster and not each other, the number of packets that leave a Mimic in an approximate simulation is instead:

$$\frac{Tp}{N(N-1)}$$

Thus, the total number of packets generated in a MimicNet simulation (the combination of all traffic generated at the observable cluster and $N-1$ Mimics) is:

$$\frac{T}{N} + \frac{(N-1)Tp}{N(N-1)} = \frac{T+Tp}{N}$$

The total decrease in packets generated is, therefore, a factor between $\frac{N}{2}$ and N with a bias toward N when traffic exhibits cluster-level locality. Fewer packets and connections generated mean less processing time and a smaller memory footprint. It also means a decrease in inter-cluster communication, which makes the composed simulation more amenable to parallelism than the full version.

6.4.2 Restrictions

MimicNet makes several domain-specific assumptions that aid in the scalability and accuracy of the MimicNet approach.

- *Failure-free FatTrees:* MimicNet assumes a FatTree topology, where the structure of the network is recursively defined and packets follow a strict up-down routing. This allows it to assume symmetric bisection bandwidth and to break cluster-level modeling into simpler subtasks.

- *Traffic patterns that scale proportionally:* To ensure that models trained from two clusters scale up, MimicNet requires a per-host synthetic model of flow arrival, flow size, packet size, and cluster-level locality that is independent of the size of the network. In other words (at least at the host level), users should ensure that the size and frequency of packets in the first step resemble those of the last step. We note that popular datasets used in recent literature already adhere to this [46, 64, 171, 197].

- *Fan-in bottlenecks:* Following prior work, MimicNet assumes that the majority of congestion occurs on fan-in toward the destination [134, 245]. This allows us to focus accuracy efforts on only the most likely bottlenecks.

- *Intra-host isolation:* To enable the complete removal of Mimic-Mimic connections at end hosts, MimicNet requires that connections be logically isolated from one another inside the host—MimicNet models network effects but does not model CPU interactions or out-of-band cooperation between connections.

MimicNet, as a first step toward large-scale network prediction is, thus, not suited for evaluating every data center architecture or configuration. Still, we argue that MimicNet can provide useful performance estimates of a broad class of proposals.

We also note that not all of the restrictions are necessarily fundamental. We briefly speculate on possible techniques to relax the restrictions.

Topology and routing. In principle, deep learning models could learn the behavior of arbitrary network topologies, and even incorporate the effects of failures and more exotic routing policies, e.g., those used in optical circuit-switched networks. This would require a unified model instead of the ingress/egress/routing models that we currently use, which may slow down the training and execution of the system. The only piece that would be difficult to relax is the implicit requirement that the network be decomposed in a way that small-scale results are representative of a subset of the larger scale simulation. Random networks, would therefore be challenging for the MimicNet approach; however, heterogeneous but structured networks may be possible, as described below.

Traffic patterns. The expectations of compatible traffic generators in MimicNet are carefully selected, and thus, would be difficult to separate from the MimicNet approach. Certainly, MimicNet could be used on packet traces rather than the synthetic patterns used in this work (by characterizing the trace using a distribution). We also note that it may be possible to relax the symmetry assumption by training distinct models for different types of clusters, e.g., frontend clusters, Hadoop clusters, and storage clusters. More baked-in are the requirements that per-cluster traffic adhere to a consistent distribution regardless of the size of the simulation; however, given that clusters maintain the same capacity, it is reasonable to expect that they maintain similar demand.

Bottleneck locations. The assumption that the most common bottlenecks exist in the downward-facing direction of a packet's path allows MimicNet to elide the modeling of effects like oversubscription coming out of the hosts and core-level congestion from inter-Mimic traffic. These could easily be added back in via similar mechanisms to inter-Mimic modelling, but at additional performance costs.

Host-internal isolation. MimicNet's removal of connections from the host is a large source of improved performance as those implementations tend to be more complicated and require more state than even switch queues. Hosts and connections also outnumber, significantly other components in the simulation. Their removal from the network is replaced by Mimic-Net's constituent models, but the hosts in Mimics actually have fewer connections. The effects of CPU contention could likely be modelled accurately. The effects of out-of-band cooperation between connections, e.g., an RCP-like mechanism running on each hosts, could also potentially be modelled with sufficient domain-expertise. Both would add to the execution time, though the training time could be parallelized.

6.5 Internal Models

As mentioned, Mimics are composed of two types of models. The first type models internal cluster behavior. Its goal is twofold:

- For *external traffic* (both Mimic-Real and Mimic-Mimic), to be able to predict how the network of the cluster will affect the packet: whether it drops, its latency, its next hop, and any packet modifications.
- For *internal traffic* (between hosts in the same Mimic), to remove it and bake its effects into the above predictions. In other words, during inference, the model should account for the observable effects of internal traffic without explicitly seeing it.

Note that not all observable effects need to be learned, especially if the result can be computed using a simple, deterministic function, e.g., TTLs or ECMP. However, for others—drops, latency, ECN marking, NDP truncation, and so on—the need for the models to scale to unobserved configurations presents a unique challenge for generalizable learning. To address the challenge, MimicNet carefully curates training data, feature sets, and models with an explicit emphasis on ensuring that generated models are *scale-independent*.

6.5.1 Small-scale Observations

MimicNet begins by running a full-fidelity, but small-scale simulation to gather training data.

Simulation and instrumentation. Data generation mirrors the depiction in Figure 6.3. Users first provide their host and switch implementations in a format that can be plugged into the C++-based OMNeT++ simulation framework.

Using these implementations, MimicNet runs a full-fidelity simulation of two clusters connected via a set of Core switches. Among these two clusters, we designate one as the cluster to be modeled and dump a trace of all packets entering and leaving the cluster. In a FatTree network, this amounts to instrumenting the interfaces facing the Core switches and the Hosts. Between these two junctures are the mechanics of the queues and routers—these are what is learned and approximated by the Mimic internal model.

Pre-processing. MimicNet takes the packet dumps and matches the packets entering and leaving the network using identifiers from the packets (e.g., sequence numbers). Examining the matches helps to determine the length of time it spent in the cluster and any changes to the packet. There are two instances where a 1-to-1 matching may not be possible: loss and multicast. Loss can be detected as a packet entering the cluster but never leaving. Multicast must be tracked by the framework. Both can be modeled.

6.5.2 Modeling Objectives

MimicNet models the clusters' effects as machine learning tasks. More formally, for each packet of external traffic, i:

Latency regression. We model the time that i spends in the cluster's network as a bounded continuous random variable and set the objective to minimize the Mean Absolute Error (MAE) between the real latency and the prediction:

$$\text{min} \sum |y_i^l - \hat{y}_i^l|,$$

where y_i^l is $(L_{\text{max}} + \epsilon)$ if the packet is dropped and $(\textit{lat} \in [L_{\text{min}}, L_{\text{max}}])$ otherwise. $\hat{y}_i^l$ is the predicted latency. To improve the accuracy of this task, MimicNet uses *discretization* in training latency models. Specifically, MimicNet quantizes the values using a linear strategy:

$$f(y^l) = \left\lfloor \frac{y^l - L_{\text{min}}}{L_{\text{max}} - L_{\text{min}}} \times D \right\rfloor$$

where D is the hyperparameter that controls the degree of discretization. By varying D, we can trade off the ease of modeling and the recovery precision from discretization.

Drops and packet modification classification. For most other tasks, classification is a better fit. For example, the prediction of a packet drop has two possible outcomes, and the objective is to minimize Binary Cross Entropy (BCE):

$$\text{min} \sum -y_i^d \log \hat{y}_i^d - (1 - y_i^d) \log(1 - \hat{y}_i^d)$$

where y_i^d is 1 if i is dropped and 0 otherwise, and $\hat{y}_i^d \in [0, 1]$ is the predicted probability that i is dropped. Packet modifications like ECN-bit prediction share a similar objective.

Both regression and classification tasks are modeled together with a unified loss function, which we describe in Section 6.5.4.

6.5.3 Scalable Feature Selection

With the above formulations, MimicNet must next select features that map well to the target predictions. While this is a critical step in any ML problem, MimicNet introduces an additional constraint—that the features be *scalable*.

Feature	Count
Local rack	*# Racks per cluster*
Local server	*# Servers per rack*
Local cluster switch	*# Cluster switches per cluster*
Core switch traversed	*# Core switches*
Packet size	*single integer value*
Time since last packet	*single real value (discretized)*
EWMA of the above feature	*single real value (discretized)*

Table 6.1: Basic set of scalable features.

A scalable feature is one that remains meaningful regardless of the number of clusters in the simulation. Consider a packet that enters the Mimic cluster from a Core switch and is destined for a host within the cluster. The local index of the destination rack ([0, R) for a cluster of R racks) would be a scalable feature as adding more clusters does not affect the value, range, or semantics of the feature. In contrast, the IP of the source server would NOT be a scalable feature. This is because, with just two clusters, it uniquely identifies the origin of the packet, but as clusters are added to the simulation, never-before-seen IPs are added to the data.

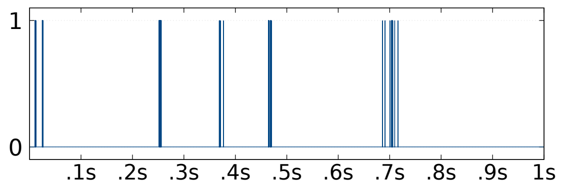
Table 6.1 lists the scalable features in a typical data center network with ECMP and TCP, applicable to both ingress and egress packets. Other scalable features that are not listed include priority bits, packet types, and ECN markings.

MimicNet performs two transformations on the captured features: one-hot encoding the first four features to remove any implicit ordering of devices and discretizing the two time-related features as in Section 6.5.2. Crucially, all of these features can quickly be determined using only packets' headers, switch routing tables, and the simulator itself.

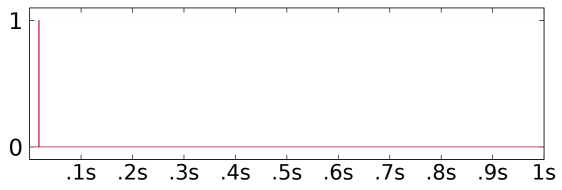
6.5.4 DCN-friendly Loss Functions

The next task is to select an appropriate training loss function. Several characteristics of this domain make it difficult to apply the objective functions of Section 6.5.2 directly.

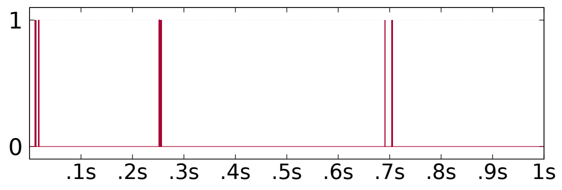
Class imbalances. Even in heavily loaded networks, adverse events like packet drops and



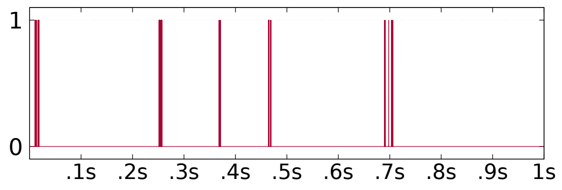
(a) Ground truth



(b) Pred w/ BCE



(c) Pred w/ 0.6 WBCE



(d) Pred w/ 0.9 WBCE

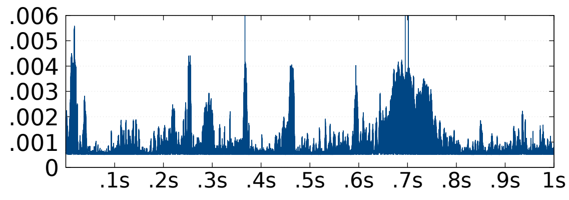
Figure 6.5: Ground truth and LSTM-predicted drops for a one-second test set using different loss functions. The y-axis is 1 for dropped, 0 for not. Ground truth has 0.3% drop rate and BCE loss has 0.01%. WBCE results in more realistic drop rates depending on the weight (w=0.6: 0.14%; w=0.9: 0.49%).

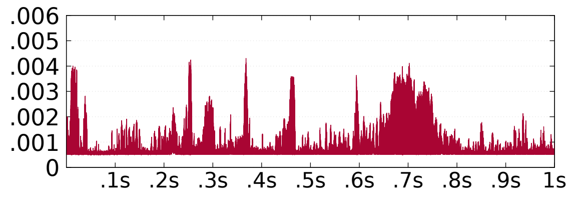
ECN tagging are relatively rare occurrences. For example, Figure 6.5a shows an example trace of drops over a one-second period in a simulation of two clusters. 99.7% of training examples in the trace are delivered successfully, implying that a model of loss could achieve high accuracy even if it always predicts 'no drop.' Figure 6.5b exemplifies this effect using an LSTM trained using BCE loss on the same trace as above. It predicts a drop rate of almost an order of magnitude lower than the true rate.

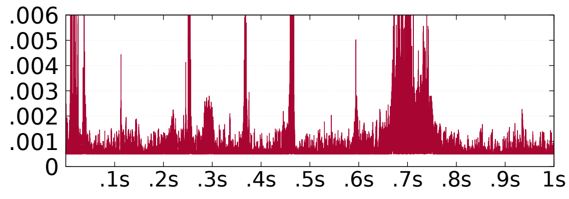
To address this instance of class imbalance, MimicNet takes a cost-sensitive learning approach [98] by adopting a Weighted-BCE (WBCE) loss:

$$\ell^d = -(1-w)\sum y_i^d \log \hat{y}_i^d - w\sum(1-y_i^d)\log(1-\hat{y}_i^d)$$

where w is the hyperparameter that controls the weight on the drop class. Figure 6.5c and 6.5d show that weighting drops can significantly improve the prediction accuracy. We note, however, that setting w too high can also produce false positives. From our experience, 0.6∼0.8 is a reasonable range, and we rely on tuning techniques in Section 6.7.2 to find the best w for a given network configuration and target metric.


(a) Ground truth


(b) Pred w/ MAE (1.4×10^{-4})


(c) Pred w/ MSE (3.3×10^{-4})

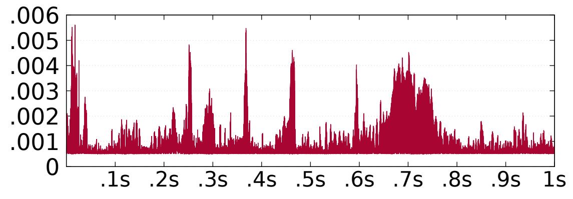

(d) Pred w/ Huber (1.1×10^{-4})

Figure 6.6: Ground truth and LSTM-predicted latency (in seconds) for a one-second test set using different loss functions. With each, we report the output of the objective, MAE (listed in parentheses). Unfortunately, using MAE directly as the loss function fails to capture outliers. Instead, Huber produces more realistic results and a better eventual MAE score.

Outliers in latencies. In latency, an equivalent challenge is accurately learning tail behavior. For example, consider the latencies from the previous trace, shown in Figure 6.6a. While most values are low, a few packets incur very large latencies during periods of congestion; these outliers are important for accurately modeling the network.

Unfortunately, MAE as a loss function fails to capture the importance of these values, as shown in the latency predictions of an MAE-based model (Figure 6.6b), which avoids predicting high latencies. We note that the other common regression loss function, Mean Squared Error (MSE), has the opposite problem—it squares the loss for each sample and produces models that tend to overvalue outliers (Figure 6.6c).

MimicNet strikes a balance with the Huber loss [130]:

$$\ell^l = \sum H_\delta(y_i^l, \hat{y}_i^l)$$

$$H_\delta(y^l, \hat{y}^l) = \begin{cases} \frac{1}{2}(y^l - \hat{y}^l)^2, & \text{if } |y^l - \hat{y}^l| \leq \delta, \\ \delta|y^l - \hat{y}^l| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

where $\delta \in \mathbb{R}^+$ is a hyperparameter. Essentially, the Huber loss assumes a heavy-tailed error distribution and uses the squared loss and the absolute loss under different situations. Figure 6.6d shows results for a model trained with the Huber loss ($\delta = 1$). In this particular case, it reduces inaccuracy (measured in MAE) of the 99-pct latency from 13.2% to only 2.6%.

Combining loss functions. To combine the above loss functions during model training, MimicNet normalizes all values and weights them using hyperparameters. Generally speaking, a weight that favors latency over other metrics is preferable as regression is a harder task than classification.

6.5.5 Generalizable Model Selection

Finally, with both features and loss functions, MimicNet can begin to model users' clusters. The model should be able to learn to approximate the mechanics of the queues and interfaces as well as cluster-local traffic and its reactions to network conditions (e.g., as a result of congestion control).

Many models exist and the optimal choice for both speed and accuracy will depend heavily on the target network. To that end, MimicNet can support any ML model. Given our desire for generality, however, it currently leverages one particularly promising class of models: LSTMs. LSTMs have gained recent attention for their ability to learn complex underlying relationships in sequences of data without explicit feature engineering [129].

Ingress/egress decomposition. To simplify the required models and improve training efficiency, MimicNet models ingress and egress traffic separately. This approach is partially enabled by MimicNet's requirement of strict up-down routing, the intrinsic modeling of cluster-local traffic, and the assumption of fan-in congestion. While there are still some inaccuracies that arise from this decision (e.g., the effect of shared buffers), we found that this choice was another good speed/accuracy tradeoff for all architectures we tested. For each direction of traffic, the LSTMs consist of an input layer and a stack of flattened, one-dimensional hidden layers. The hidden size is *#features* $\times$ *#packets* where *#packets* is the number of packets in a sample, and *#features* is post one-hotting.

Congestion state augmentation. While in principle, LSTMs can retain 'memory' between predictions to learn long-term patterns, in practice, they are typically limited to memory on the order of 10s or 100s of samples. In contrast, the traffic seen by a Mimic may exhibit self-similarity on the order of hundreds of thousands of packets. Our problem, thus, exhibits properties of multiscale models [83].

Because of this, we augment the LSTM model with a piece of network domain knowledge: an estimation of the presence of congestion in each cluster's network. Specifically, four distinct states are considered: (1) little to no congestion, (2) increasing congestion as queues fill, (3) high congestion, and (4) decreasing congestion as queues drain. These states are estimated by looking at the latency and drop rate of recently processed packets in the cluster. By breaking the network up into these four coarse states, the LSTM is able to efficiently learn patterns over these regimes, each with distinct behaviors. This feature is added to the others in Table 6.1.

6.6 Feeder Models

While the above (internal) models can model the behavior of the queues, routers, and internal traffic of a cluster, the complete trace of external traffic is still required to generate accurate results. In the terminology of Figure 6.4, internal models bake in the effects of the intra-cluster traffic, but the LSTMs are trained on *all* external traffic, not just Mimic-Real.

To replace the remaining non-observable traffic, the internal models are augmented with a *feeder* whose role is to estimate the arrival rate of inter-Mimic traffic and inject them into the internal model. Creating a feeder model is challenging compared to internal cluster models as inter-Mimic traffic is not present in the small-scale simulation and varies as the simulation scales. MimicNet addresses this by creating a parameterized and fully generative model that uses flow-level approximation techniques to predict the packet arrival rate of Mimic-Mimic traffic in different network sizes.

The feeder model is trained in parallel to the internal models. MimicNet first derives from the small-scale simulation characteristic packet interarrival distributions for all external flows, separated by their direction (ingress/egress). In our tests, we observed, as others have in the past [64, 158] that simple log-normal or Pareto distributions produced reasonable approximations of these interarrival times. Nevertheless, more sophisticated feeders can be trained and parameterized in MimicNet. During the full simulation, the feeders will take the hosts' inter-cluster demand as a parameter, compute a time-series of active flow-level demand, and draw packets randomly from that demand using the derived distributions.

Crucially, when feeding packets, the feeders generate 'packets' independently, pass their raw feature vectors to the internal models, and immediately discard any output. This means that internal models' hidden state is updated as if the packets were routed without actually incurring the costs of creating, sending, or routing them. While this approach shares the weaknesses of other flow-level approximations, like the removal of intra-cluster traffic, these packets are never directly measured and, thus, an approximation of their effect is sufficient. Further, while the traffic is never placed in the surrounding queues, i.e., queues of the Core switch or the egress queues on the Hosts; as prior work has noted, the majority of drops and congestion are found elsewhere in the network [245].

6.7 Tuning and Final Simulation

MimicNet composes Mimics into a parallelized large-scale data center simulation. In addition to designing the internal and feeder models with scale-independence in mind, it ensures the models survive scaling with a hyper-parameter tuning phase.

6.7.1 Composing Mimics

An N-cluster MimicNet simulation consists of a single real cluster, $N-1$ Mimic clusters, and a proportional number of Core switches. The real cluster continues to use the user implementation of Section 6.5.1, but users can add arbitrary instrumentation, e.g., by dumping `pcaps` or queue depths.

The Mimic clusters are constructed by taking the ingress/egress internal models and feeders developed in the previous sections and wrapping them with a thin shim layer. The layer intercepts packets arriving at the borders of the cluster, periodically takes packets from the feeders, and queries the internal models with both to predict the network's effects. The output of the shim is, thus, either a packet, its egress time, and its egress location; or its absence. Adjacent hosts and Core switches are wired directly to the Mimic, but are otherwise unaware of any change.

Aside from the number of clusters, all other parameters are kept constant from the small-scale to the final simulation. That includes the feeder models and traffic patterns, which take a size parameter but fix other parameters (e.g., network load and flow size).

6.7.2 Optional Hyper-parameter Tuning

Mimic models contain at least a few hyper-parameters that users can optionally choose to tune: WBCE weight, Huber loss δ, LSTM layers, hidden size, epochs, and learning rate among others. MimicNet provides a principled method of setting these by allowing users to define their own optimization function. This optimization function is distinct from the model objectives or the loss functions. Instead, they can evaluate end-to-end accuracy over arbitrary behavior in the simulation (for instance, tuning for accuracy of FCTs). Users can add hyper-parameters or end-to-end optimization functions depending on their use cases.

For every tested parameter set, MimicNet trains a set of models and runs validation tests to evaluate the resulting accuracy and its scale-independence. Specifically, MimicNet runs an approximated and full-fidelity simulation on a held-out validation workload in three configurations: 2, 4, and 8 clusters. It then compares the two versions using the user's metric.

The full-fidelity comparison results are only gathered once, and the MimicNet results are evaluated for every parameter set, but the sizes are small enough that the additional time is nominal. Based on the user-defined metric, MimicNet uses Bayesian Optimization (BO) to pick the next parameter set of the highest 'prediction uncertainty' via an *acquisition function* of EI (expected improvement). BO can quickly converge on the optimal configuration.

MimicNet supports two classes of metrics natively.

MSE-based metrics. For 1-to-1 metrics, MimicNet provides a framework for computing MSE.

For example, when comparing the FCT of the same flow in both simulations:

$$\text{MSE} = \frac{1}{|Flows|} \sum_{f \in Flows} (\text{realFCT}_f - \text{mimicFCT}_f)^2$$

A challenge in using this class of metrics is that the set of completed flows in the full-fidelity network and MimicNet are not necessarily identical—over a finite running timespan, flow completions that are slightly early/late can change the set of observed FCTs. To account for this, we only compute MSE over the intersection, i.e.,

$$Flows = \{f \mid (\exists\ \text{realFCT}_f \wedge (\exists\ \text{mimicFCT}_f)\}$$

By default, MimicNet ignores models with overlap $< 80\%$.

Wasserstein-based metrics. Unfortunately, not all metrics can be framed as above. Consider per-packet latencies. While in training we assume that we can calculate a per-packet loss and back-propagate, in reality when a drop is mistakenly predicted, the next prediction should reflect the fact that there is one fewer packet in the network, rather than adhering to the original packet trace. In some protocols like TCP, the loss may even cause packets to appear in the original but not in any MimicNet version or vice versa.

MimicNet's hyper-parameter tuning phase, therefore, allows users to test distributions, e.g., of RTTs, FCTs, or throughput, via the Wasserstein metric. Also known as the Earth Mover's Distance, the metric quantifies the minimum cost of transforming one distribution to the other [106]. Specifically, for a one-dimensional CDF, the metric (W_1) is:

$$W_1 = \int_{-\infty}^{+\infty} |CDF_{\text{real}}(x) - CDF_{\text{mimic}}(x)|$$

W_1 values are scale-dependent, with lower numbers indicating greater similarity.

6.8 Prototype Implementation

We have implemented a prototype of the full MimicNet workflow in C++ and Python on top of PyTorch/ATen and the OMNeT++ [177] simulation suite. Given an OMNeT++ router and host implementation, our prototype will generate training data, train/hypertune a set of MimicNet models, and compose the resulting models into an optimized, full-scale simulation. This functionality totals to an additional 25,000 lines of code.

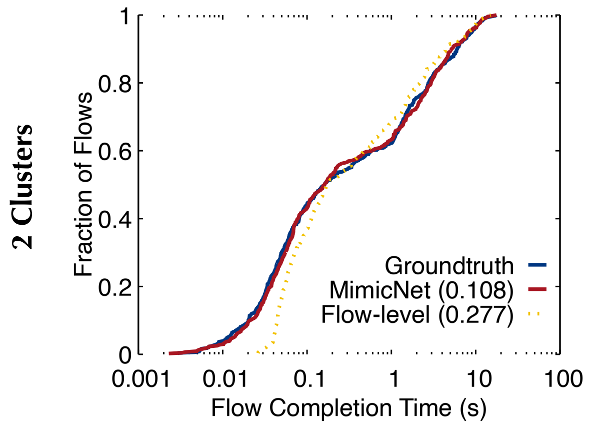
Simulation framework. MimicNet is built on OMNeT++ v4.5 and INET v2.4 with custom C++ modules to incorporate our machine learning models into the framework. To ensure that the experiments are repeatable, all randomness, including the seeds for generating the traffic are configurable. They were kept consistent between variants and changed across training, testing, and cross validation.

Parallel execution. A side benefit MimicNet is that it significantly reduces the need for synchronization in a parallel execution. In order to take advantage of this property, we parallelize each cluster of the final simulation using an open-source PDES implementation of INET [248].

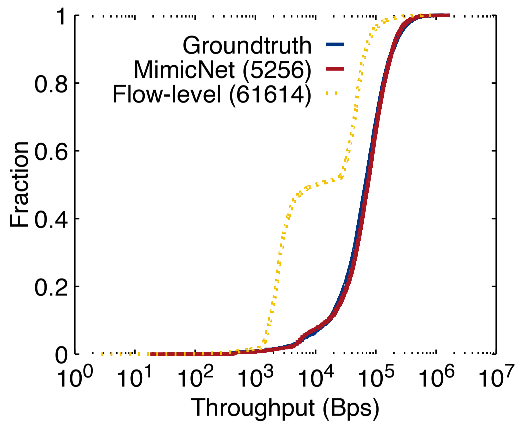
Machine learning framework. Our LSTM models are trained using PyTorch 0.4.1 and CUDA 9.2 [204, 212]. Hyperparameter tuning was done with the assistance of `hyperopt` [27]. At runtime, Mimic cluster modules accept OMNeT++ packets, extract their features, perform a forward step of the LSTMs, and forward the packet via ECMP based on the result. For speed, our embedded LSTMs were custom-built inference engines that leverage low-level C++ and CUDA functions from the Torch, cuDNN, and ATen libraries.

6.9 Evaluation

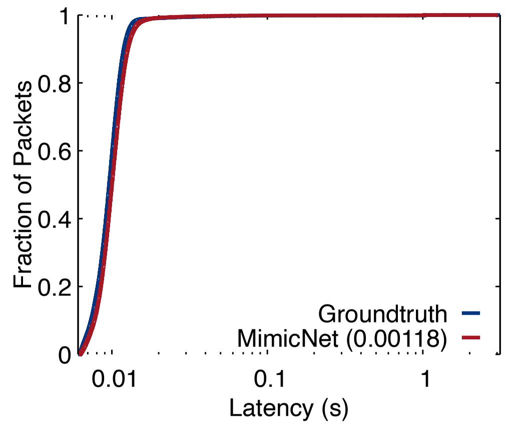
Our evaluation focuses on several important properties of MimicNet including: (1) its accuracy of approximating the performance of data center networks, (2) the scalability of its accuracy to large networks, (3) the speed of its approximated simulations, and (4) its utility for comparing configurations.



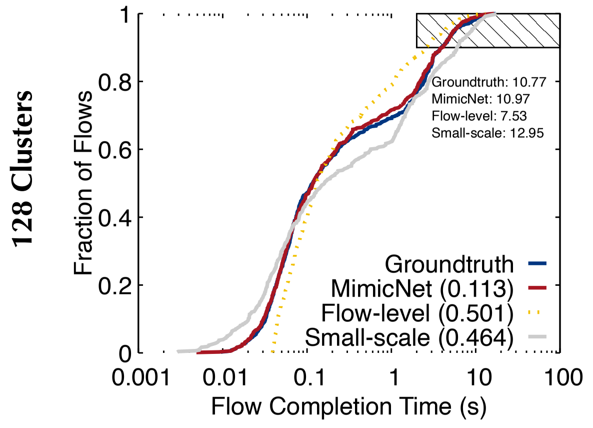
(a) FCT distribution



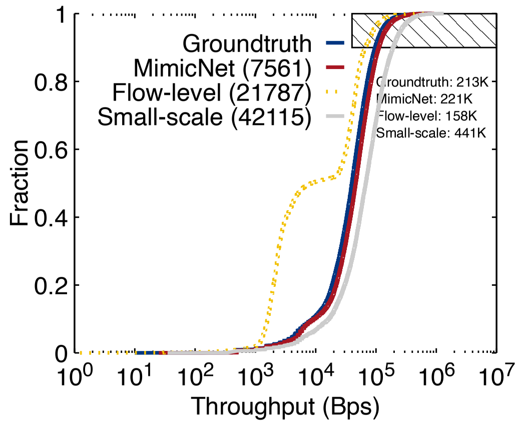
(b) Throughput distribution



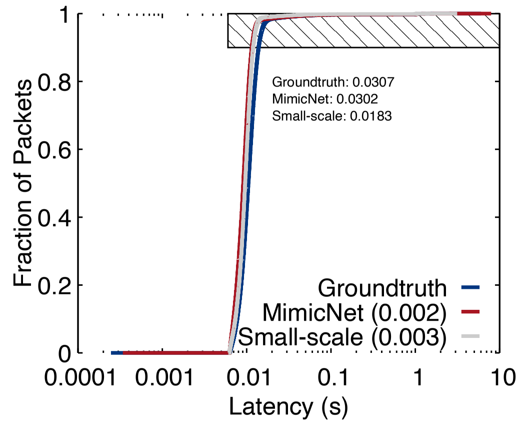
(c) Packet RTT distribution



(d) FCT distribution



(e) Throughput distribution



(f) Packet RTT distribution

Figure 6.7: The accuracy of MimicNet in the baseline configuration for 2 clusters and 128 clusters. Also shown are results from SimGrid and the assumption that small-scale results are representative. W_1 to ground truth is shown in parentheses. We annotate the 99-pct value of each metric for every approach at the tail in 128 clusters.

Methodology. Our simulations all assume a FatTree topology, as described in Section 6.2. We configured the link speed to be 100 Mbps with a latency of 500 μs. To scale up and down the data center, we adjusted the number of racks/switches in each cluster as well as the number of clusters in the data center. We note that higher speeds and larger networks were not feasible due to the limitation of needing to evaluate MimicNet against a full-fidelity simulation, which would have taken multiple years to produce even a single equivalent execution.

The base case uses TCP New Reno, Drop Tail queues, and ECMP. To test MimicNet's robustness to different network architectures, we use a set of protocols: DCTCP [46], Homa [197], TCP Vegas [69], and TCP Westwood [186] that stress different aspects of MimicNet. Our workload uses traces from a well-known distribution also used by many recent data center proposals [46, 197]. By default, the traffic utilizes 70% of the bisection bandwidth and the mean flow size is 1.6 MB. All experiments were run on CloudLab [223] using machines with two Intel Xeon Silver 4114 CPUs and an NVIDIA P100 GPU. When evaluating flow-level simulation, we use the SimGrid [76] v3.25 and its built-in `FatTreeZone` configured with the same topology and traffic demands as full/MimicNet simulation.

Evaluation metrics. As mentioned in Section 6.7.2, traditional per-prediction metrics like training loss are not useful in our context. Instead, we leverage three end-to-end metrics: (1) FCT, (2) per-server Throughput binned into 100 ms intervals, (3) and RTT. In the flow-level simulation, FCT is computed using flow start/end times, Throughput is computed with a custom load-tracking plugin, and RTT is not possible to compute. In MimicNet and full simulation, all three are computed by instrumenting the hosts in the observable cluster to track packets sends and ACK receipts. Where applicable, we compare CDFs using W_1.

6.9.1 MimicNet Models Clusters Accurately

We begin by evaluating MimicNet's accuracy when replacing a single cluster with a Mimic before examining larger configurations in the next section. Note that in this configuration, there is no need for feeder models. Rather, this experiment directly evaluates the effect of replacing a cluster's queues, routers, and cluster-local traffic with LSTMs. For this test, we use

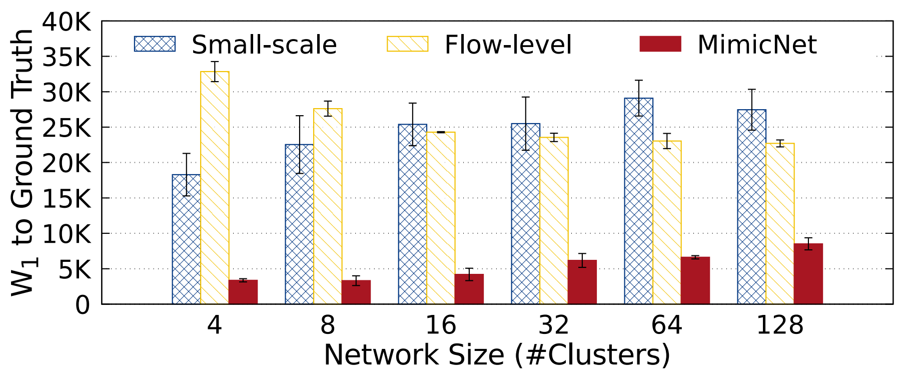


Figure 6.8: Throughput Scalability.

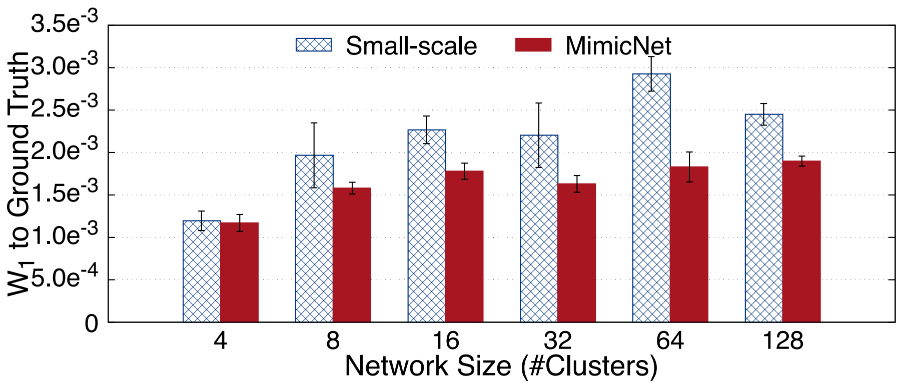


Figure 6.9: RTT Scalability. Flow-level simulation is too coarse-grained to provide this metric.

the baseline set of protocols described above. The final results use traffic patterns that are not found in the training or hyper-parameter validation sets.

Figure 6.7a–Figure 6.7c show CDFs of our three metrics for this test. As the graphs show, MimicNet achieves very high accuracy on all metrics. The LSTMs are able to learn the requisite long-term patterns (FCT and throughput) as well as packet RTTs. Across the entire range, MimicNet's CDFs adhere closely to the ground truth, i.e., the full-fidelity, packet-level simulation; just as crucial, the shape of the curve is maintained. Flow-level simulation behaves much worse.

6.9.2 MimicNet's Accuracy Scales

A key question is whether the accuracy translates to larger compositions where traffic interactions become more complex and feeders are added. We answer that question using a simulation composed of 128 clusters (full-fidelity simulation did not complete for larger sizes). In MimicNet, 127 clusters are replaced with the same Mimics as the previous subsection. Figure 6.7d–Figure 6.7f show the resulting accuracy. There are a couple of interesting observations.

First, while the accuracy of MimicNet estimation does decrease, the decrease is nominal. More concretely, for FCT, throughput, and RTT, we find W_1 metrics of 0.113, 7561, and 0.00158 compared to the ground truth, respectively. For reference, we also plot SimGrid and the original 2-cluster simulation's results. The W_1 error between 2-cluster simulation and 128-cluster groundtruth are 311%, 457%, and 70% higher than MimicNet's values; the W_1s of FCT and throughput between SimGrid and the groundtruth are similarly high. The results indicate that our composition methods are successfully approximating the scaling effects. Critically, MimicNet also predicts tails well: the p99 of MimicNet's FCT, throughput, and RTT distributions are within 1.8%, 3.3%, and 2% of the true result.

We evaluate MimicNet's scalability of accuracy more explicitly in Figures 6.1, 6.8, and 6.9. Here, we plot the W_1 metric of all three approaches for several data center sizes ranging from 4 to 128. Recall that the 2-cluster results essentially hypothesize that FCT, throughput, and RTT do not change as the network scales. An upward trend on their W_1 metric in all three graphs suggests that the opposite is true. Compared to that baseline, MimicNet on average achieves a 43% lower RTT W_1 error, 78% lower throughput error, and 63% lower FCT error. In all cases, MimicNet also shows much lower variance across workloads, demonstrating better predictability at approximating large-scale networks.

6.9.3 MimicNet Simulates Large DCs Quickly

Equally important, MimicNet can estimate performance very quickly. The multiple phases of MimicNet—small-scale simulation, model training and hyper-parameter tuning, and large-

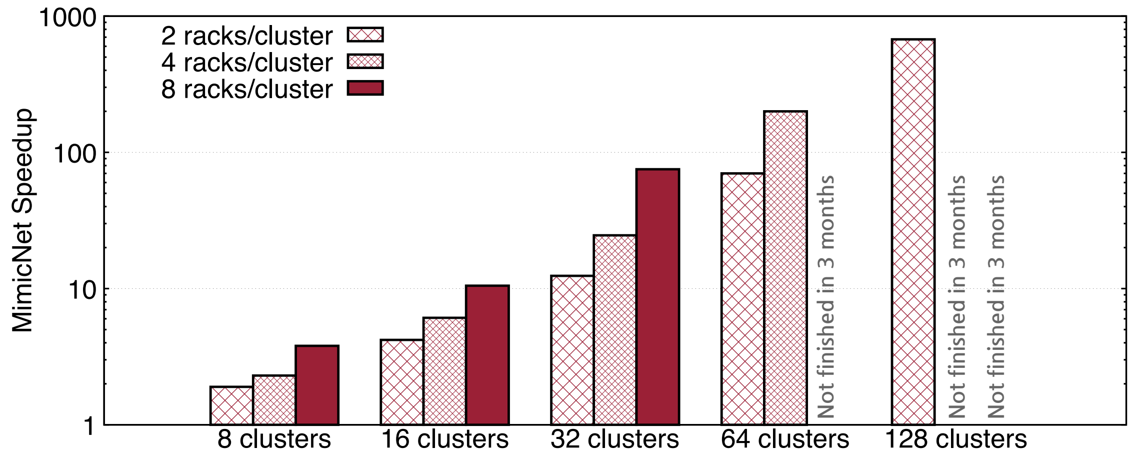


Figure 6.10: Simulation running time speedup brought by MimicNet in different sizes of data centers. In a network of 128 clusters (256 racks), MimicNet reduces the simulation time from 12 days to under 30 minutes, achieving more than two orders of magnitude speedup. The speedups are consistent and stable across different workloads.

scale composition—each require time, but combined, they are still faster than running the full-fidelity simulation directly. By paying the fixed costs of the first two phases, the actual simulation can be run while omitting the majority of the traffic and network connections.

Execution time breakdown. Table 6.2 shows a breakdown of the running time of both the full simulation and MimicNet, factored out into its three phases for the 128 cluster, 1024 host simulation in Figure 6.1. For 20 seconds of simulated time, the full-fidelity simulator required almost 1w 5d. In contrast, MimicNet, in aggregate, only required 8h 38m, where just 25m was used for final simulation—a 34× speedup. Longer simulation periods or multiple runs for different workload seeds would have led to much larger speedups.

Simulation time speedup. We focus on the non-fixed-cost component of the execution time in order to better understand the benefits of MimicNet. Figure 6.10 shows the speedup of MimicNet after the initial, fixed cost of training a cluster model. For each network configuration, we run both MimicNet and a full simulation over the exact same sets of generated workloads. We then report the average speedup and the standard error across those workloads.

In both systems, simulation time consists of both setup time (constructing the network, allocating resources, and scheduling the traffic) as well as packet processing time. MimicNet

	Factor	Time
MimicNet	Small-scale simulation	1h 3m
	Training and hyper-param tuning	7h 10m
	Large-scale simulation	**25m**
Full	Simulation	**1w 4d 22h 25m**

Table 6.2: Running time comparison for 20 s of simulated time of a 128 cluster, 1024 host data center. Benefits of MimicNet increase with simulated time and the size of the network as the first two values for MimicNet are constant.

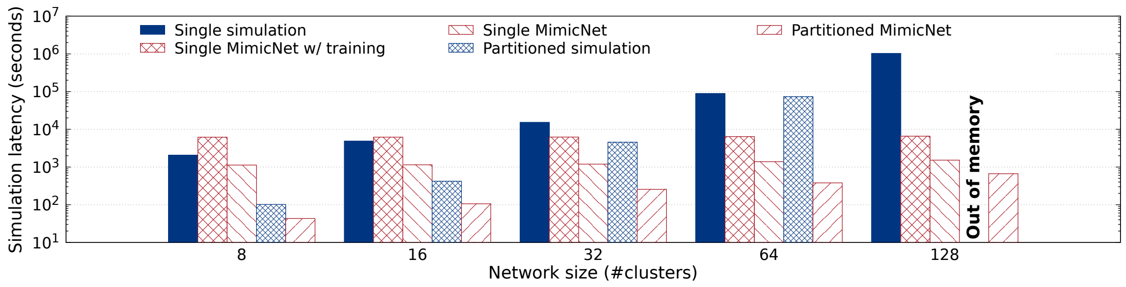


Figure 6.11: Simulation latency with different network sizes.

substantially speeds up both phases.

MimicNet can provide consistent speedups up to 675× for the largest network that full-fidelity simulation was able to handle. Above that size, full-fidelity could not finish within three months, while MimicNet can finish in under an hour. Somewhat surprisingly, *MimicNet is also 7× faster than flow-level approximation at this scale* as SimGrid must still track all of the Mimic-Mimic connections.

Groups of simulations. We also acknowledge that simulations are frequently run in groups, for instance, to test different configuration or workload parameters. To evaluate this, we compare several different approaches to running groups of simulations and evaluate them using two metrics: (1) *simulation latency*, i.e., the total time it takes to obtain the full set of results, and (2) *simulation throughput*, i.e., the average number of aggregate simulation seconds that can be processed per second. We first focus on the effect of network size.

Simulation latency: For latency, N cores in a machine, and S simulation seconds, we consider five different approaches: (1) single simulation, i.e., one full simulation that runs

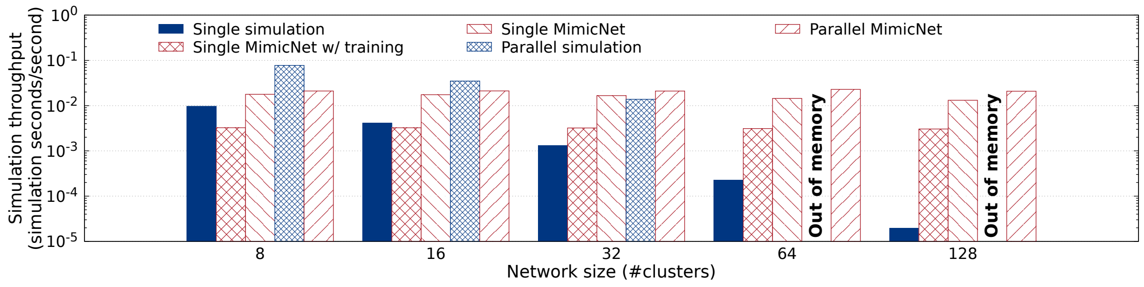


Figure 6.12: Simulation throughput with different network sizes.

on a single core and simulates S seconds; (2) single MimicNet w/ training, i.e., one end-to-end MimicNet instance, running from scratch; (3) single MimicNet, i.e., one MimicNet instance that reuses an existing model; (4) partitioned simulation, i.e., N full simulations, each simulating S/N seconds; and (5) partitioned MimicNet, i.e., N MimicNet instances, each simulating S/N seconds. N=20 as our machines have 20 cores.

Figure 6.11 shows the results for network sizes ranging from 8 to 128 clusters. We make the following observations. First, when the network is relatively small, the model training overhead in MimicNet is significant, so 'single MimicNet w/ training' takes longer than 'single simulation' to finish. When the network size reaches 64 clusters, even when training time is included, MimicNet runs faster than any full simulation approach. When the network is as large as 128 clusters, MimicNet is 2-3 orders of magnitude faster than full simulations. The results hold when partitioning, with MimicNet gaining an additional advantage in larger simulations where the removal of the majority of packets/connections introduces substantial gains to the memory footprint of the simulation group.

Simulation throughput: For throughput, we consider a similar set of five approaches. Specifically, the first three are identical to (1)–(3) above, while the last two run for S seconds to maximize throughput: (4) parallel simulation, i.e., N full simulations, each simulating S seconds and (5) parallel MimicNet, i.e., N MimicNet instances, each simulating S seconds.

Figure 6.12 shows the throughput results for the range of network sizes. Overall, MimicNet maintains high throughput regardless of the network size because the amount of observable traffic is roughly constant. Single simulation, on the other hand, slows down substantially

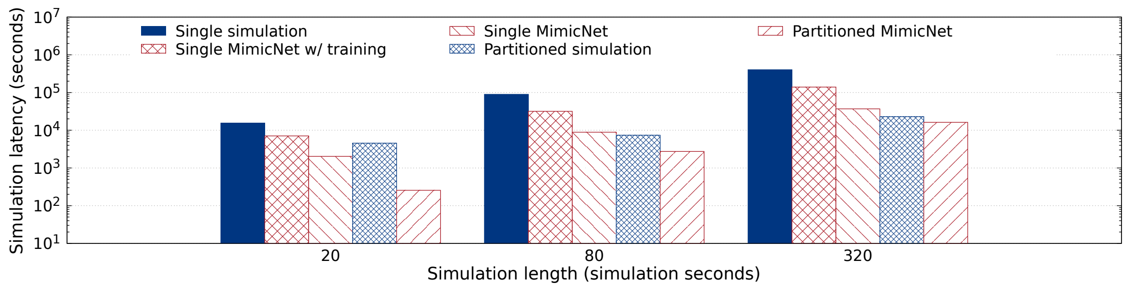


Figure 6.13: Simulation latency with different simulation lengths.

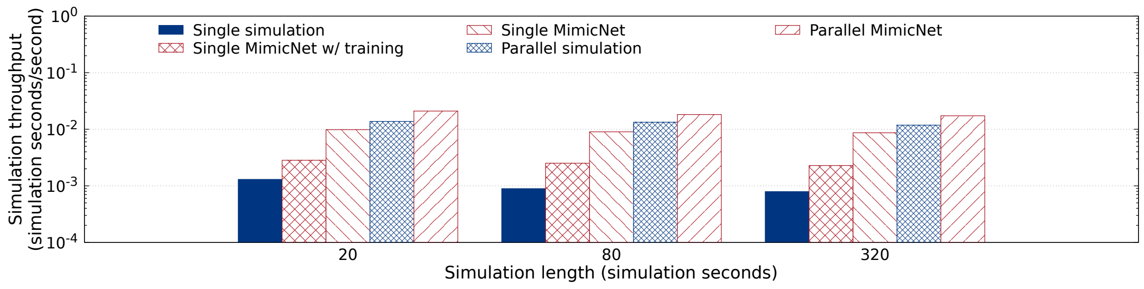


Figure 6.14: Simulation throughput with different simulation lengths.

as the size of the network grows, and at 128 clusters, full simulation is almost *five orders of magnitude slower than the real-time*. As mentioned in Section 6.2, a remedy prescribed by many simulation frameworks is to run multiple instances of the simulation. Our results indeed show that the throughput of parallel simulation compared to single simulation improves by up to a factor of N. When contrasted to the scale-independent throughput of MimicNet, however, a single instance of MimicNet overtakes even parallel simulation at 32 clusters. Larger parallelized instances begin to suffer from the memory issues described above, but even with unlimited memory, MimicNet would still likely outperform parallel simulation by 2–3 orders of magnitude at 128 clusters.

We also evaluate the effect of simulation length. For these experiments, we fix the network size as 32 clusters and vary the simulation length from 20 simulation seconds to 320 simulation seconds. Figures 6.13 and 6.14 show the simulation latency and throughput results, respectively, for different simulation approaches.

The results are somewhat expected: the relative simulation speeds of different approaches barely change with the simulation length. When simulation length increases, the latency of

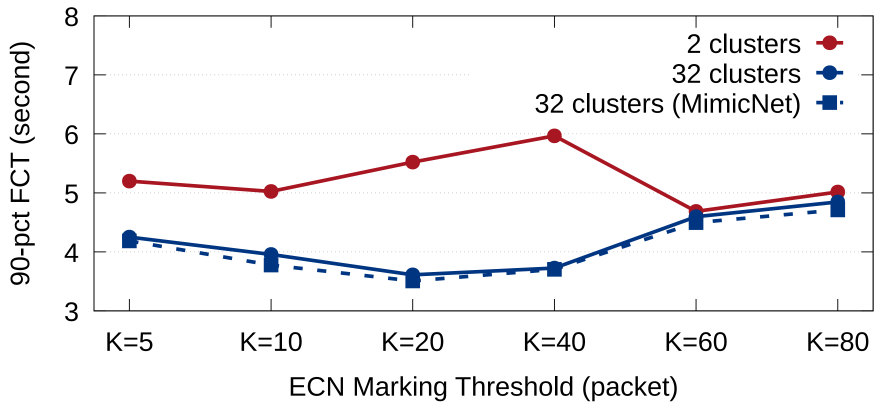


Figure 6.15: Tuning the marking threshold K in DCTCP: the configuration that achieves the lowest 90-pct FCT is different between 2 clusters ($K = 60$) and 32 clusters ($K = 20$). MimicNet provides the same answer as the full simulation for 32 clusters, but it is $12\times$ faster.

each approach increases correspondingly. The latency of full simulations increases slightly slower than that of MimicNet because the constant simulation setup overhead in full simulations is significantly higher than MimicNet. The relative latency eventually stabilizes—the latency of single MimicNet is lower than that of single simulation, even when the model training time is included in MimicNet, and partitioned MimicNet is better than partitioned simulation. For all approaches, the simulation throughput does not change at all with the simulation length. Similarly, single MimicNet outperforms single full simulations, and parallel MimicNet outperforms parallel full simulations. The speedup of MimicNet further grows when the simulation scales to larger networks.

6.9.4 Use Cases

MimicNet can approximate a wide range of protocols and provide actionable insights for each. This section presents two potential use cases: (1) a method of tuning configurations of DCTCP and (2) a performance comparison of several data center network protocols.

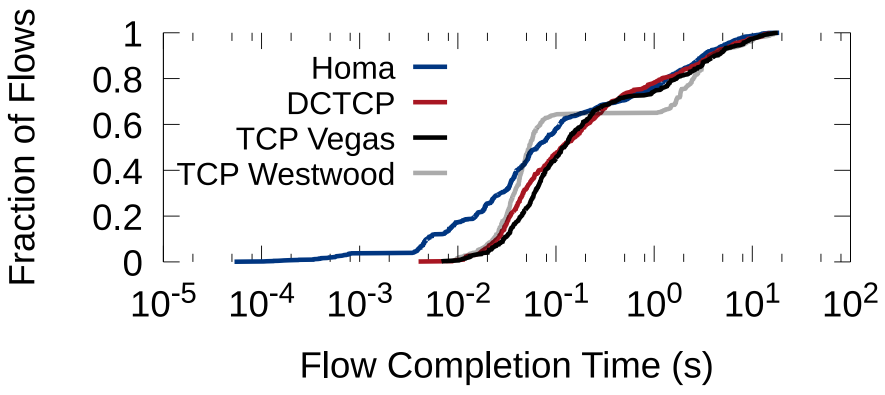
Configuration tuning. DCTCP leverages ECN feedback from the network to adjust congestion windows. An important configuration parameter mentioned in the original paper is the ECN marking threshold, K, which influences both the latency and throughput of the protocol.

Essentially, a lower K signals congestion more aggressively ensuring lower latency; however, a K that is too low may underutilize network bandwidth, thus limiting throughput. FCTs are affected by both: short flows benefit from lower latency while long flows favor higher throughput. The optimal K, thus, depends on both the network and workload. Further, a simulation's prescription for K has implications for its feasibility, its latency/throughput comparisons to other protocols, and the range of parameters that an operator might try when deploying to production.

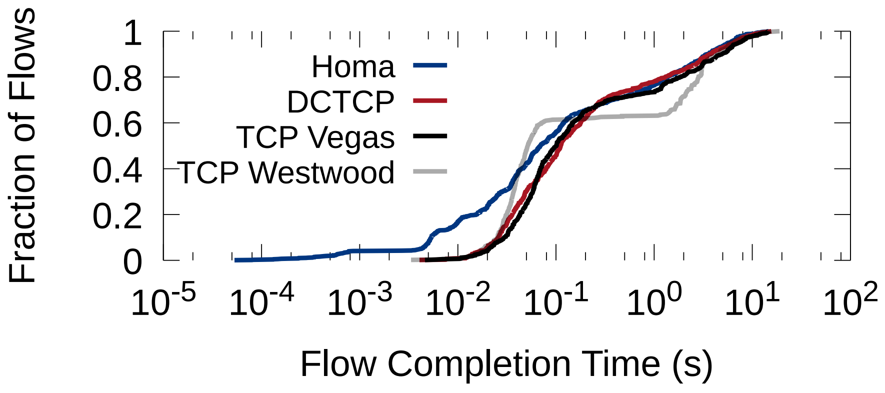
Figure 6.15 compares the 90-pct FCT for different Ks. Looking only at the small-scale simulation, one may be led to believe that the optimal setting for our workload is $K = 60$. Looking at the larger 32-cluster simulation tells a very different story—one where $K = 60$ is among the worst of configurations tested and $K = 20$ is instead optimal. MimicNet successfully arrives at the correct conclusion.

Comparing protocols. Finally, MimicNet is accurate enough to be used to compare different transport protocols. We implement an additional four such protocols that each stress MimicNet's modeling in different ways. Homa is a low-latency data center networking protocol that utilizes priority queues—a challenging extra feature for MimicNet as packets can be reordered. TCP Vegas is a delay-based transport protocol that serves as a stand-in for the recent trend of protocols that are very sensitive to small changes in latency [196, 150]. TCP Westwood is a sender-optimized TCP that measures the end-to-end connection rate to maximize throughput and avoid congestion. DCTCP ($K = 20$) uses ECN bits, which add an extra feature and prediction output compared to the other protocols. We run the full MimicNet pipeline for each of the protocols, training separate models. We then compare their performance over the same workload, and we evaluate the accuracy and speed of MimicNet for this comparison. The FCT results are in Figure 6.16.

As in the base configuration, for all protocols, MimicNet can match the FCT of the full-fidelity simulation closely. In fact, on average, the approximated 90-pct and 99-pct tails by MimicNet are within 5% of the ground truth. Because of this accuracy, MimicNet performance estimates can be used to gauge the rough relative performance of these protocols. For example, the full simulation shows that the best and the worst protocol for 90-pct of FCT is



(a) Ground truth of the comparison

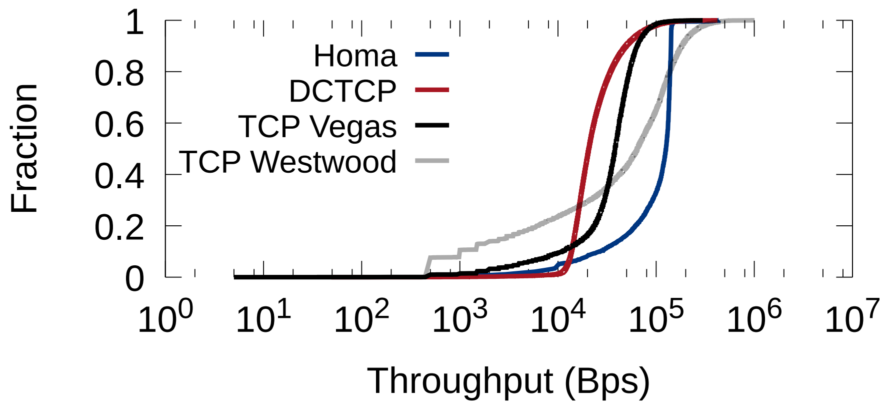


(b) The approximation of MimicNet.

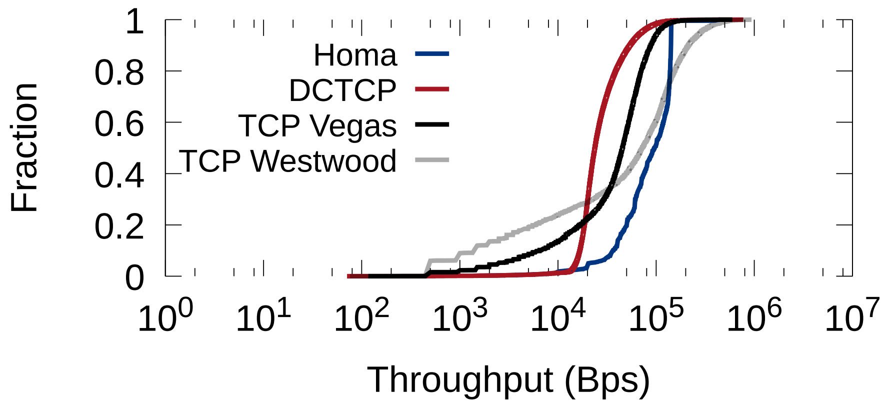
Figure 6.16: FCT distributions of Homa, DCTCP, TCP Vegas, and TCP Westwood for a 32-cluster data center.

Homa (3.1 s) and TCP Vegas (4.5 s); MimicNet predicts the correct order with similar values: Homa with 3.3 s and TCP Vegas with 4.6 s. While the exact values may not be identical, MimicNet can predict trends and ballpark comparisons much more accurately than the small-scale baseline. It can arrive at these estimates in a fraction of the time—12× faster.

Figures 6.17 and 6.18 show throughput and packet RTT comparisons respectively. Similar to FCT, MimicNet can closely match the throughput and RTT of a real simulation for all protocols. We can use the estimation of MimicNet to compare these protocols—not only their general trends of throughput and RTT distributions, but also their ranking at specific points. For example, TCP Westwood achieves the best 90 percentile throughput performance due to its optimizations on utilizing network bandwidth; in comparison, DCTCP has the lowest



(a) The ground truth of the comparison



(b) The approximation of MimicNet.

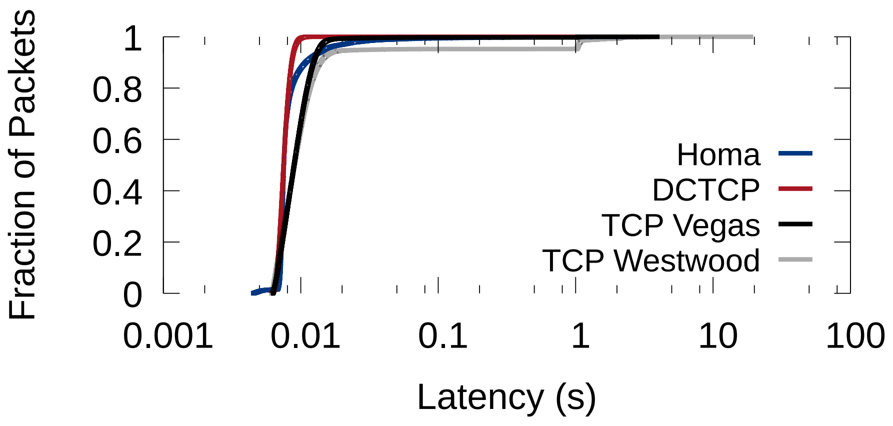
Figure 6.17: Comparison of throughput distributions.

throughput at this particular point. MimicNet successfully predicts the order. The situation in RTT, however, is the opposite: TCP Westwood now has the highest 90 percentile latency, while DCTCP performs the best among these four protocols. This comparison is also correctly predicted by MimicNet.

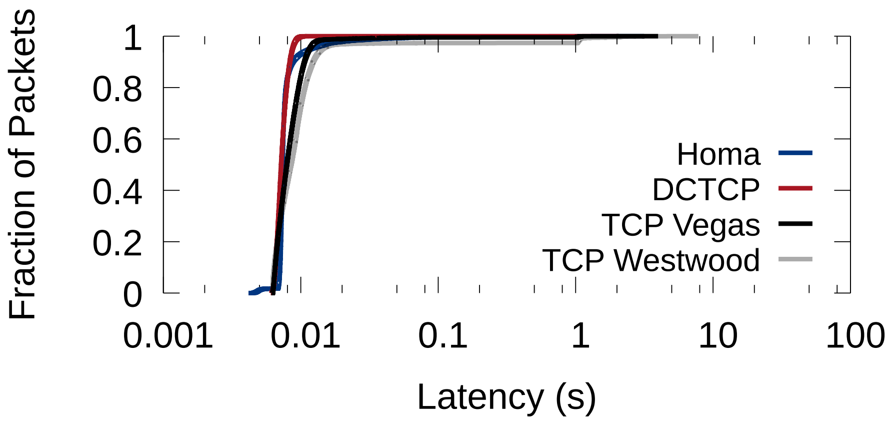
6.9.5 Compute Consumption

A potential concern in using MimicNet is its compute resource consumption: it uses GPU resources for model training and runtime inference while the full simulations only use CPUs. We now evaluate this aspect.

Specifically, we calculate the total number of floating-point operations (FLOPs) in both



(a) The ground truth of the comparison



(b) The approximation of MimicNet.

Figure 6.18: Comparison of packet RTT distributions.

CPUs (for both full simulations and MimicNet) and GPUs (for MimicNet only) of the simulation approaches in Section 6.9.3 as their compute resource consumption. Figure 6.19 shows the result for the evaluation of latency (similar findings in the evaluation of throughput). Indeed, MimicNet shows significant computational load, primarily because of the use of GPUs for training and inference. This makes its compute consumption higher than full simulations when the network to be simulated is small, especially when the training overhead is counted. However, in large networks, e.g., 128 clusters, the use of deep learning models in MimicNet pays off by much lower simulation latency, and *its total compute consumption is lower than full simulations even with the computational overhead in training models*. We leave the further optimization on MimicNet compute consumption to future work.

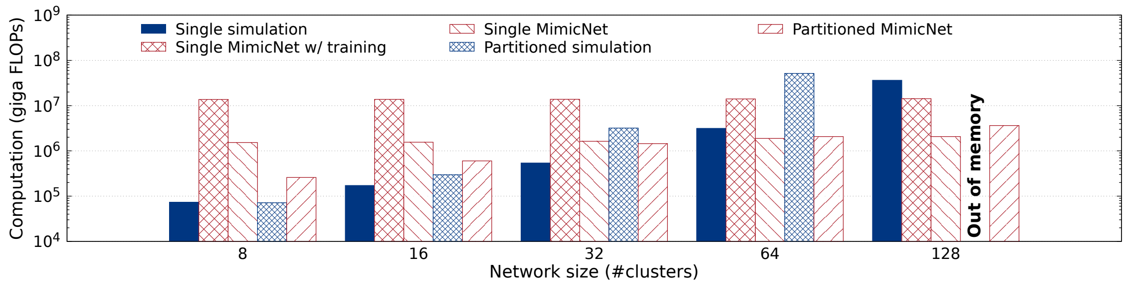


Figure 6.19: Compute consumption in different simulation approaches.

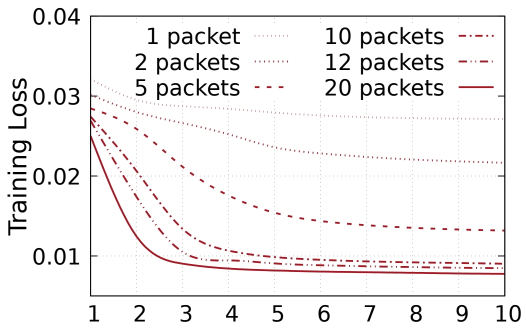
6.9.6 The Impact of Model Complexity

We note that in MimicNet, a significant, domain-specific factor in model complexity is the size of the training window. The window is a number of packets (their features) that we input to the model. This size decides (1) the amount of data that the model learns from one sample, and (2) the hidden size of our LSTM model. Having a larger window helps learning and potentially improves the prediction accuracy, but at the cost of training and inference speed.

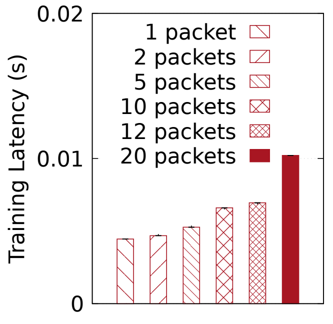
Figure 6.20 shows both of these effects on the training of an ingress model. From Figure 6.20a, we can see that a window size of only 1 packet performs very poorly, even after several epochs. The training accuracy is quickly improved with additional packets in the window, but this comes with diminishing returns after the window size reaches the BDP of the network (around 12 packets). Figure 6.20b shows a reverse trend for training time. This suggests that the BDP of the network strikes a good balance between accuracy and speed for the LSTM model. We also evaluated the impact of the window size on the validation accuracy and the inference speed and made similar observations. Figure 6.21 shows the result.

6.10 Related Work

Packet-level simulation. As critical tools for networking, simulators have existed for decades [155]. Popular choices include ns-3 [205, 128], OMNeT++ [177], and Mininet [154]. When simulating large networks, existing systems tend to sacrifice one of scalability or granularity. BigHouse, for instance, models data center behavior using traffic drawn from em-

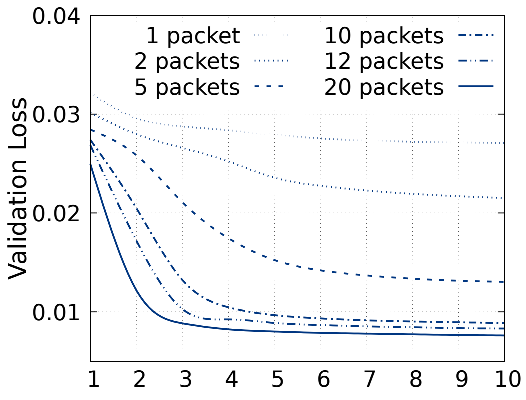


(a) Training loss descent

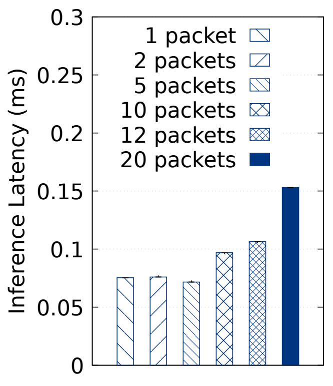


(b) Training latency

Figure 6.20: The impact of the window size on modeling accuracy and speed. The BDP of the network is around 12 packets. More packets in the window help loss descent (through epochs), but can make the training slower (training latency is per batch in Python).



(a) Validation loss descent



(b) Inference latency

Figure 6.21: The impact of the window size on modeling (validation) accuracy and inference latency per packet in C++.

pirically generated distributions and a model of how traffic distributions translate to a set of performance metrics [187]. Our system, in contrast, begins with a faithful reproduction of the target system, providing a more realistic simulation.

Emulators. Another class of tools attempts to build around real components to maintain an additional level of realism [40, 263, 170]. Flexplane [208], for example, passes real, production traffic through models of resource management schemes. Pantheon [275] runs real congestion control algorithms on models of Internet paths. Unfortunately, emulation's dependency on real components often limits the achievable scale. Scalability limitations even impact systems like DIABLO [254], which leverages FPGAs to emulate devices with low cost, but may still require $\sim$\$1 million to replicate a large-scale deployment.

Phased deployment. Also related are proposals such as [240, 295] reserve slices of a production network for A/B testing. While showing true at-scale performance, they are infeasible for most researchers.

Preliminary version. Finally, we note that a published preliminary version of this work explored the feasibility of approximating packet-level simulations using deep learning [142]. This chapter represents a substantial evolution of that work. Critical advancements include the notion of scale-independent features, end-to-end hyper-parameter tuning methods/metrics that promote scalability of accuracy, the addition of feeder models, improved loss function design, and other machine learning optimizations such as discretization. These are in addition to significant improvements to the MimicNet implementation and a substantially deeper exploration of the design/evaluation of MimicNet.

6.11 Summary

This chapter presents MimicNet, which enables fast performance estimates of large data center networks. MimicNet exploits the symmetric, hierarchical structure of data centers to break their simulation down into a composition of several clusters. Through judicious use of machine learning and other modeling techniques, MimicNet exhibits super-linear scaling compared to full simulation while retaining high accuracy in replicating observable traffic. While we acknowledge that there is still work to be done in making the process simpler and even more accurate, the design presented here provides a proof of concept for the use of machine learning and problem decomposition for the approximation of large networks.

CHAPTER 7

CONCLUSION

How do today's systems perform at processing massive data on hyperscale data center infrastructure? Our extensive performance studies in this dissertation have provided a clear answer: they suffer from many inefficiencies. A key observation that we make is that network communications are becoming the primary source of these inefficiencies. This is true in current data centers, where a hierarchical and oversubscribed network connects a large number of servers. Existing data processing systems send messages across distributed processes without considering the characteristics of the network, while it is increasingly challenging for the network to provide a simple abstraction for *many* distributed servers. It is also true in future data centers, where hardware resources are physically disaggregated. Existing data processing systems are not aware of the novel architectural difference and end up moving excessive amounts of data through the network. Hence, this dissertation advocates that centering the designs of data processing systems around network communication bottlenecks is a fundamental approach to efficient hyperscale data processing. This approach will only become more critical as the degree of distribution continues to increase because of fast-growing data volumes and the loose coupling between hardware components.

We summarize the contributions of this dissertation as follows.

- **A hyperscale graph processing system.** GraphRex proposes a set of network-aware relational operators, i.e., *Global Operators*, that encompass a collection of optimizations customized for graph processing at data center scale. These optimizations exploit the semantics of relational operators and the underlying data center network topology to reduce data transfer over expensive network links. They demonstrated order-of-magnitude communication efficiency in current data centers.

- **An in-depth investigation on data processing with disaggregated memory.** We conducted the first in-depth study of data management systems in disaggregated data centers. Through extensive evaluation and analysis on production database systems, we confirmed the performance overhead of disaggregation incurred by frequently moving data through the network. We also found that the elasticity of disaggregated memory can in fact benefit memory-intensive workloads. Our results provide important insights on deploying and developing data processing systems in future data centers.

- **A compute pushdown primitive for disaggregated data centers.** TELEPORT enables compute pushdown for disaggregated memory. It adopts a lazy data synchronization approach to ensure pushdown efficiency. As part of the operating system, TELEPORT requires minimal code modification on its applications. With TELEPORT, data processing systems can determine how to best move computation and data in disaggregated data centers to eliminate the high overhead of disaggregation.

- **A high-performance cache with remote memory.** Redy is an RDMA-accessible cache service using remote memory. It automatically configures RDMA to satisfy user-provided performance SLO and minimize resource cost. Redy caches are robust to remote memory dynamics. Memory-intensive applications can use Redy to improve their performance on large workloads, as well as data center memory utilization.

- **A hyperscale network evaluation framework.** MimicNet adopts deep learning to approximate data center network behavior. As a packet-level simulator, it learns at small scale and uses the learned models to compose simulations at large scale. MimicNet arrives at accurate evaluation results orders of magnitude faster than existing approaches. MimicNet can significantly speed up the data center innovation cycle.

7.1 Other Work

We have worked on several projects that we did not include in this dissertation. These works focused on other aspects of data processing such as applications, cost efficiency, and fault tolerance. We briefly describe them here.

Data science application with machine learning. Crowdfunding is a new mechanism for connecting entrepreneurs with thousands of online investors. However, no previous studies have investigated its effectiveness and what strategies startups should adopt to maximize their chances. With a 7-month data collection, we tracked the activities of over 4000 startups on AngelList, a popular crowdfunding website, Twitter, and Facebook. With the dataset, we predicted whether startups succeed or fail in crowdfunding based on novel machine learning techniques and features that describe startup social engagement [289].

Performance and cost analysis of graph processing. Many graph processing systems have adopted distributed, shared-nothing architectures to run in a cluster of machines. Others argued that a single machine is often enough. There has been no consensus on which approach is better. From a user perspective, it is difficult to select the best system given a workload. We performed the first study of the performance and cost of state-of-the-art graph processing systems [286]. Our analysis revealed that the systems that achieve the highest performance are often different than those with the lowest cost. Optimality depends on the input graph, query, and targeted metric. Our detailed analysis provides useful insights for the selection and development of graph systems, including GraphRex.

Faster and cheaper remote caching. Remote caching systems like Redy, Redis, and memcached let applications offload large states to remote servers. However, even with fast caches, performance degradation is still significant due to the difference between local and remote memory access latency. We proposed CompuCache [282], a new service that supports offloading *both data and computation over the data* to remote caches. CompuCache achieves higher performance by single-round-trip offloading with *server-side pointer-chasing*. It also uses spot VMs as cache servers for lowering the cost. Since spot VMs are unreliable, CompuCache reacts quickly to failures. Our experiments showed that the throughput of CompuCache is 200× higher than that of Redis, achieving 126 million offloading invocations per second with a single server.

7.2 Future Work

The network-centric systems we have built sketch out the broad picture of hyperscale data processing and show the difficulties of achieving good performance. Many challenges remain as both applications and infrastructure evolve. In this section, we discuss three directions where we can extend the work presented in this dissertation by applying principled approaches to process data in more diverse forms and in next-generation networks.

Diverse workloads as data processing. This dissertation scopes in the processing of a few representative cloud workloads (relational databases, key-values, and graphs). Other types of workloads are also increasingly important and can just be processed as data. One example is processing streams from IoT (Internet of Things) devices, machine-generated logs, and cameras, where the data is dynamic and states in the processing are unbounded.

Another popular example is machine learning (ML) applications, which are now powered by massive models, datasets, and computing infrastructure. Distributed ML can experience expensive network overheads in some of its critical components such as aggregating and broadcasting parameters between workers and parameter servers in data parallel training. ML models are becoming extremely large—up to hundreds of billions of parameters. Training and executing such massive models as special hyperscale data processing tasks are interesting future directions. Exploring the space for these workloads would need to involve both network and application-specific expertise.

Emerging cloud computing paradigms like blockchains and serverless can also suffer from network bottlenecks in data centers, e.g., transaction ordering in permissioned blockchains and data shuffling between serverless functions. Although specific optimizations have been proposed recently for these platforms, it is appealing to apply network-centric designs to address the bottlenecks in systematic and scalable ways.

New cloud trends. An important trend in data center design is that cloud providers are augmenting their data centers with more domain and application-specific accelerators such as SmartNICs, FPGAs and ASICs to meet the rapidly increasing computation demand. Another trend is that data center networks are becoming more programmable. An example is programmable network switches, which can do more than just forwarding packets. Users can execute a wide range of operations over packets in network at line rate. *Automatically* optimizing data processing by *universally* leveraging these accelerators and programmable switches, e.g., offloading components that are costly for end hosts, is a direction we plan to investigate. Techniques that we developed in TELEPORT and CompuCache will likely be useful.

Next-generation Internet. Like data center networks, the Internet is becoming ultra fast. In the coming 5G networks, edge devices can communicate with cloud servers with multi-Gbps bandwidth. 6G is expected to be orders of magnitude faster. High-speed Internet may have two implications. First, more data will likely be uploaded to clouds for processing with stricter timing requirements. This would add higher pressure to both cloud infrastructure and data processing systems. Investigating the new challenges introduced by massive load increase from the Internet is an important direction. The other implication is that data processing jobs can scale beyond data centers—operations can be efficiently placed on and moved between edge devices and cloud servers. This will likely create new computing paradigms that are much larger than today's distributed processing. The whole Internet would work like a giant data center, in which we can utilize global resources for data processing at any scale. Many techniques that this dissertation has introduced may still be useful, e.g., network awareness, caching, and automatic compute offloading, but there will be new challenges. In particular, wide-area networks can experience great performance variation due to the mobility of edge devices. Proposing new architectures to improve network robustness and codesigning data processing systems to provide end-to-end guarantees are also promising directions.

BIBLIOGRAPHY

[1] Amazon aurora db clusters. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Overview.html.

[2] Amazon migrates 50 pb of analytics data from oracle to aws. https://aws.amazon.com/solutions/case-studies/amazon-migration-analytics/. Accessed in July, 2022.

[3] Apache giraph. http://giraph.apache.org/.

[4] Apache hive. https://hive.apache.org/.

[5] Apache spark - unified analytics engine for big data. https://spark.apache.org.

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

[7] Bigdatalog. https://github.com/ashkapsky/BigDatalog.

[8] Bigquery for data warehouse practitioners. https://cloud.google.com/solutions/bigquery-data-warehouse.

[9] Connectx-6 single/dual-port adapter supporting 200Gb/s with VPI. https://www.nvidia.com/en-us/networking/infiniband-adapters/connectx-6 (visited on 12/24/2021).

[10] Databricks sql - customer stories. https://databricks.com/product/databricks-sql. Accessed in July, 2022.

[11] Fabric performance tools. https://docs.mellanox.com/display/winof2/Fabric+Performance+Utilities (visited on 12/24/2021).

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

[13] How google search organizes information. https://www.google.com/search/howsearchworks/how-search-works/organizing-information/. Accessed in July, 2022.

[14] Improve Performance of a File Server with SMB Direct. https://microsoft.com/en-us/library/jj134210.aspx.

[15] Introducing data center fabric, the next-generation facebook data center network. https://bit.ly/3rckiai.

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

[17] Lz4 - extremely fast compression. http://lz4.github.io/lz4/.

[18] memcached - a distributed memory object caching system. https://memcached.org/.

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

[20] Nvidia mellanox connectx-5 single/dual-port adapter supporting 100Gb/s with VPI. https://www.nvidia.com/en-us/networking/infiniband-adapters/connectx-5/ (visited on 12/24/2021).

[21] The parallelism operator (aka exchange). https://blogs.msdn.microsoft.com/craigfr/2006/10/25/the-parallelism-operator-aka-exchange/.

[22] Redis. https://redis.io/.

[23] The future of analytics in the cloud. https://s3.amazonaws.com/bizzabo.file.upload/J2AUieXBSpWr2YAUqZHK_GrazianoThe%20Future%20of%20Analytics%20-%20RMOUG.pdf. Accessed in July, 2022.

[24] TPC-H SF100 non-parallel plans, SQL Server 2008. http://www.qdpma.com/tpch/TPCH100_Query_plans.html.

[25] Will the latest ai kill coding? https://towardsdatascience.com/will-gpt-3-kill-coding-630e4518c04d. Accessed in July, 2022.

[26] Opnet network simulator, 2015. https://opnetprojects.com/opnet-network-simulator/.

[27] Hyperopt, 2018. http://hyperopt.github.io/hyperopt/.

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

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

[30] Connectx®-3 pro en single/dual-port adapters 10/40/56gbe adapters w/ pci express 3.0. https://www.mellanox.com/products/ethernet-adapters/connectx-3-pro, 2021.

[31] Daniel J. Abadi, Samuel R. Madden, and Miguel C. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proc. SIGMOD*, 2006.

[32] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proc. SIGMOD*, 2008.

[33] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. Materialization strategies in a column-oriented DBMS. In *Proc. ICDE*, 2007.

[34] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 431–446, 2016.

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

[36] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. 2018.

[37] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. 2017.

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

[39] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 1–13, Philadelphia, PA, 2014. USENIX Association.

[40] M. Al-Fares, R. Kapoor, G. Porter, S. Das, H. Weatherspoon, B. Prabhakar, and A. Vahdat. Netbump: User-extensible active queue management with bumps on the wire. In *2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 61–72, Oct 2012.

[41] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, August 17-22, 2008*, pages 63–74. ACM, 2008.

[42] Alibaba. Alibaba cluster trace program. `https://github.com/alibaba/clusterdata`.

[43] Alibaba. ApsaraDB for POLARDB: A next-generation relational database - Alibaba cloud. `https://www.alibabacloud.com/products/apsaradb-for-polardb`, 2019.

[44] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 469–482, Berkeley, CA, USA, 2017. USENIX Association.

[45] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: distributed congestion-aware load balancing for datacenters. In Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy, editors, *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 503–514. ACM, 2014.

[46] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In Shivkumar Kalyanaraman, Venkata N. Padmanabhan, K. K. Ramakrishnan, Rajeev Shorey, and Geoffrey M. Voelker, editors, *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, New Delhi, India, August 30 -September 3, 2010*, pages 63–74. ACM, 2010.

[47] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, San Jose, CA, 2012. USENIX.

[48] Amazon-Aurora. Amazon aurora - Relational database built for the cloud - AWS. `https://aws.amazon.com/rds/aurora/`, 2019.

[49] Amazon Web Services. Amazon ec2 spot instances. `https://aws.amazon.com/aws-cost-management/aws-cost-optimization/spot-instances/` (visited on 12/24/2021).

[50] Pradeep Ambati, Iñigo Goiri, Felipe Vieira Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing slos for resource-harvesting vms in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI* , pages 735–751. USENIX Association, 2020.

[51] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM.

[52] T. E. Anderson, D. E. Culler, and D. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, Feb 1995.

[53] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. 2020.

[54] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. July 2020.

[55] 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*, pages 1743–1756. ACM, 2019.

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

[57] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1753–1758. ACM, 2017.

[58] Krste Asanović. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. 2014.

[59] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. 2000.

[60] Azure. Azure public dataset. https://github.com/Azure/AzurePublicDataset.

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

[62] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-scale in-memory join processing using RDMA. 2015.

[63] 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. *Proc. VLDB Endow.*, 12(13):2325–2338, 2019.

[64] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, 2010.

[65] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, 2016.

[66] Spyros Blanas, Paraschos Koutris, and Anastasios Sidiropoulos. Topology-aware parallel data processing: Models, algorithms and systems at scale. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[67] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact representations of separable graphs. In *Proc. SODA*, 2003.

[68] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. An experimental analysis of a compact graph representation. In *Proc. ALENEX*, 2004.

[69] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. *SIGCOMM Comput. Commun. Rev.*, 24(4):24–35, October 1994.

[70] Yingyi Bu, Vinayak Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 8(2), 2014.

[71] 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. *PVLDB*, 11(11):1604–1617, 2018.

[72] 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. *Proc. VLDB Endow.*, 11(11):1604–1617, 2018.

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

[74] Xiang Cao, Kewal Keshaorao Panchputre, and David Hung-Chang Du. Accelerating data shuffling in mapreduce framework with a scale-up NUMA computing architecture. In *HPC '16*.

[75] Amanda Carbonari and Ivan Beschastnikh. Tolerating Faults in Disaggregated Datacenters. 2017.

[76] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.

[77] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. FASTER: A concurrent key-value store with in-place updates. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD 2018*, pages 275–290. ACM, 2018.

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

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

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

[81] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 98–109, New York, NY, USA, 2011. ACM.

[82] Xu Chu, Ihab F. Ilyas, and Paraschos Koutris. Distributed data deduplication. *Proc. VLDB Endow.*, 9(11):864–875, July 2016.

[83] Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*, 2016.

[84] Cisco Systems. *Data Center Design Summary*, August 2014. https://www.cisco.com/c/dam/en/us/td/docs/solutions/CVD/Aug2014/DataCenterDesignSummary-AUG14.pdf.

[85] Citus-Data. Citus data: Worry-free postgres. built to scale out. https://www.citusdata.com/, 2019.

[86] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In Amin Vahdat and David Wetherall, editors, *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005)*. USENIX, 2005.

[87] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[88] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th*

Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017, pages 153–167. ACM, 2017.

[89] Paolo Costa, Hitesh Ballani, and Dushyanth Narayanan. Rethinking the network stack for rack-scale computers. 2014.

[90] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O&#039;Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 29–42, San Jose, CA, 2012. USENIX.

[91] Enrique Dans. Netflix: Big data and playing a long game is proving a winning strategy. `https://bit.ly/3ONAbjS`, 2020.

[92] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004.

[93] Christina Delimitrou, Daniel Sánchez, and Christos Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In Shahram Ghandeharizadeh, Sumita Barahmand, Magdalena Balazinska, and Michael J. Freedman, editors, *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015*, pages 97–110. ACM, 2015.

[94] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. Adaptive query processing. *Found. Trends Databases*, 1(1):1–140, 2007.

[95] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, March 1990.

[96] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. 2014.

[97] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. Woo Kang, I. Kim, and G. Daglikoca. The architecture of the DIVA processing-in-memory chip. 2002.

[98] Charles Elkan. The foundations of cost-sensitive learning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 973–978, 2001.

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

[100] G. Ewing, Krzysztof Pawlikowski, and Donald Mcnickle. Akaroa-2: Exploiting network computing by distributing stochastic simulation. *Proceedings of 13th European Simulation Multiconference, ESM'99*, pages 175–181, 6 1999.

[101] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. Scaling-up in-memory datalog processing: Observations and techniques. *Proc. VLDB Endow.*, 12(6):695–708, 2019.

[102] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In Sujata Banerjee and Srinivasan Seshan, editors, *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 51–66. USENIX Association, 2018.

[103] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 469–483, Berkeley, CA, USA, 2015. USENIX Association.

[104] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global memory management in client-server database architectures. In Li-Yan Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, pages 596–609. Morgan Kaufmann, 1992.

[105] Philip Werner Frey and Gustavo Alonso. Minimizing the hidden cost of RDMA. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*, pages 553–560. IEEE Computer Society, 2009.

[106] Andrew Frohmader and Hans Volkmer. 1-wasserstein distance on the standard simplex. *CoRR*, abs/1912.04945, 2019.

[107] Richard M. Fujimoto. Parallel discrete event simulation. In *Proceedings of the 21st Winter Simulation Conference, Washington, DC, USA, December 4-6, 1989*, pages 19–28, 1989.

[108] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. Extrema predicates in deductive databases. *J. Comput. Syst. Sci.*, 51(2):244–259, 1995.

[109] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. 2016.

[110] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. 2016.

[111] Jana Giceva, Gerd Zellweger, Gustavo Alonso, and Timothy Roscoe. Customized OS support for data-processing. In *Proceedings of the 12th International Workshop on*

Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016, pages 2:1–2:6. ACM, 2016.

[112] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 350–361, 2011.

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

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

[115] Joseph E. Gonzalez, Reynold Xin, Ankur Dave, Dan Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[116] Google. Google cluster data. `https://github.com/google/cluster-data`.

[117] Google Cloud. Preemptible virtual machines. `https://cloud.google.com/preemptible-vms/` (visited on 12/24/2021).

[118] Goetz Graefe and et al. Extensible query optimization and parallel execution in volcano. In *Query Processing for Advanced Database Systems, June 1991*.

[119] Albert G. Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In Pablo Rodriguez, Ernst W. Biersack, Konstantina Papagiannaki, and Luigi Rizzo, editors, *Proceedings of the ACM SIGCOMM 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Barcelona, Spain, August 16-21, 2009*, pages 51–62. ACM, 2009.

[120] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jeong-Uk Kang Jonghyun Yoon, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. 2016.

[121] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with INFINISWAP. 2017.

[122] 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 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 202–215, New York, NY, USA, 2016. ACM.

[123] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 139–152, New York, NY, USA, 2015. ACM.

[124] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: an analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service, IWQoS 2019, Phoenix, AZ, USA, June 24-25, 2019*, pages 39:1–39:10. ACM, 2019.

[125] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 287–298. ACM Press, 1999.

[126] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E. Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 845–861. USENIX Association, 2020.

[127] Khalid Hasanov and Alexey L. Lastovetsky. Hierarchical optimization of MPI reduce algorithms. In *Parallel Computing Technologies - 13th International Conference, PaCT 2015, Petrozavodsk, Russia, August 31 - September 4, 2015, Proceedings*, pages 21–34, 2015.

[128] Thomas R. Henderson, Mathieu Lacage, and George F. Riley. Network simulations with the ns-3 simulator. In *In Sigcomm (Demo*, 2008.

[129] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[130] Peter J. Huber. Robust estimation of a location parameter. In *The Annals of Mathematical Statistics*, pages 73–101, 1964.

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

[132] Intel. Optimize data structures and memory access patterns to improve data locality. `https://goo.gl/xQ3ZGT`, 2012. Intel.

[133] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 407–420, New York, NY, USA, 2015. ACM.

[134] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, and Changhoon Kim. Eyeq: Practical network performance isolation for the multi-tenant

cloud. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, Boston, MA, June 2012. USENIX Association.

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

[136] Maja Kabiljo, Dionysis Logothetis, Sergey Edunov, and Avery Ching. A comparison of state-of-the-art graph processing systems. https://code.facebook.com/posts/319004238457019/a-comparison-of-state-of-the-art-graph-processing-systems/.

[137] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy, editors, *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 295–306. ACM, 2014.

[138] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In Ajay Gulati and Hakim Weatherspoon, editors, *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 437–450. USENIX Association, 2016.

[139] U Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. GBASE: An efficient analysis platform for large graphs. In *Proc. VLDB*, 2012.

[140] Kostas Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharonik, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klosx, and T. Berends. Rack-scale Disaggregated Cloud Data Centers: The dReDBox Project Vision. 2016.

[141] Naga Praveen Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: scalable load balancing using programmable data planes. In Brighten Godfrey and Martín Casado, editors, *Proceedings of the Symposium on SDN Research, SOSR 2016, Santa Clara, CA, USA, March 14 - 15, 2016*, page 10. ACM, 2016.

[142] Charles W. Kazer, João Sedoc, Kelvin K. W. Ng, Vincent Liu, and Lyle H. Ungar. Fast network simulation through approximation or: How blind men can describe elephants. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks, HotNets 2018, Redmond, WA, USA, November 15-16, 2018*, pages 141–147. ACM, 2018.

[143] Kimberly Keeton. The Machine: An Architecture for Memory-centric Computing. 2015.

[144] Bettina Kemme, Ricardo Jiménez-Peris, and Marta Patiño-Martínez. *Database Replication*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[145] Jessica Kent. Big data analytics show covid-19 spread, outcomes by region. https://healthitanalytics.com/news/big-data-analytics-show-covid-19-spread-outcomes-by-region, 2020.

[146] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. *SIGCOMM Comput. Commun. Rev.*, 42(4):467–472, September 2012.

[147] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 59–72, Berkeley, CA, USA, 2015. USENIX Association.

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

[149] Alexander Krizhanovsky. Lock-free multi-producer multi-consumer queue on ring buffer. https://www.linuxjournal.com/content/lock-free-multi-producer-multi-consumer-queue-ring-buffer (visited on 12/24/2021).

[150] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.

[151] Jay Kyathsandra and Eric Dahlen. Intel Rack Scale Architecture Overview. http://presentations.interop.com/events/las-vegas/2013/free-sessions---keynote-presentations/download/463, 2013.

[152] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 31–46, 2012.

[153] Monica S. Lam, Stephen Guo, and Jiwon Seo. Socialite: Datalog extensions for efficient social network analysis. In *Proceedings of the 2013 IEEE International Conference on Data Engineering*, 2013.

[154] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[155] LBNL. network simulator man page. https://ee.lbl.gov/ns/man.html.

[156] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 743–754. ACM, 2014.

[157] Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34:892–901, October 1985.

[158] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, Feb 1994.

[159] 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. 2017.

[160] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA. 2016.

[161] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016*, pages 355–370. ACM, 2016.

[162] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. Numa-aware algorithms: the case of data shuffling. In *CIDR' 13*.

[163] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. 2009.

[164] Kevin Lim, Yoshio Turnet, Jichuan Chang, Jose Renato Santos, and Parthasarathy Ranganathan. Disaggregated Memory Benefits for Server Consolidation. Technical Report HPL-2011-31, HP Laboratories, 2011.

[165] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. 2009.

[166] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. 2012.

[167] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 189–200. IEEE Computer Society, 2012.

[168] Yongsub Lim, U Kang, and Christos Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge & Data Engineering*, 26(12):3077–3089, 2014.

[169] Feilong Liu, Lingyan Yin, and Spyros Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. *ACM Trans. Database Syst.*, 44(4):17:1–17:45, 2019.

[170] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.

[171] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network, 2013.

[172] Vincent Liu, Danyang Zhuo, Simon Peter, Arvind Krishnamurthy, and Thomas Anderson. Subways: A case for redundant, inexpensive data center edge links. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 27:1–27:13, New York, NY, USA, 2015. ACM.

[173] Xin Liu, Ashraf Aboulnaga, Kenneth Salem, and Xuhui Li. CLIC: client-informed caching for storage servers. In Margo I. Seltzer and Richard Wheeler, editors, *7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA. Proceedings*, pages 297–310. USENIX, 2009.

[174] Savia Lobo. Google ai introduces snap, a microkernel approach to 'host networking'. `https://bit.ly/3QhzVu3`, 2019.

[175] Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 97–108, 2006.

[176] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, pages 340–349, Arlington, Virginia, United States, 2010. AUAI Press.

[177] OpenSim Ltd. Omnet++, 2018. `http://omnetpp.org`.

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

[179] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. Dynamic query re-planning using QOOP. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 253–267. USENIX Association, 2018.

[180] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 290–301, New York, NY, USA, 2011. ACM.

[181] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. 2000.

[182] Zoltan Majo and Thomas R. Gross. Matching memory access patterns and data placement for NUMA systems. In *Proc. CGO*, 2012.

[183] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010.

[184] Stefan Manegold, Peter A. Boncz, and Niels Nes. Cache-conscious radix-decluster projections. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 684–695. Morgan Kaufmann, 2004.

[185] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. July 2020.

[186] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang. TCP westwood: Bandwidth estimation for enhanced transport over wireless links. In Christopher Rose, editor, *MOBICOM 2001, Proceedings of the seventh annual international conference on Mobile computing and networking, Rome, Italy, July 16-21, 2001*, pages 287–297. ACM, 2001.

[187] David Meisner, Junjie Wu, and Thomas F. Wenisch. Bighouse: A simulation infrastructure for data center systems. In *IEEE International Symposium on Performance Analysis of Systems & Software*, 2012.

[188] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. Dremel: A decade of interactive SQL analysis at web scale. *Proc. VLDB Endow.*, 13(12):3461–3472, 2020.

[189] Cade Metz. The facebook special: How intel builds custom chips for giants of the web. `https://www.wired.com/2013/05/facebook-and-intel/`, 2013.

[190] Microsoft. Faster: Fast persistent recoverable log and key-value store + cache, in c# and c++. `https://microsoft.github.io/FASTER/` (visited on 12/24/2021).

[191] Microsoft. Ndspi interfaces. `https://docs.microsoft.com/en-us/previous-versions/windows/desktop/cc904391(v=vs.85)`.

[192] Microsoft Azure. Azure high-performance computing. `https://azure.microsoft.com/en-us/solutions/high-performance-computing/` (visited on 12/24/2021).

[193] Microsoft Azure. Azure spot virtual machines. https://azure.microsoft.com/en-us/pricing/spot/ (visited on 12/24/2021).

[194] Microsoft-SQL-Database. Sql database – cloud database as a service | Microsoft Azure. https://azure.microsoft.com/en-us/services/sql-database/, 2019.

[195] Vishal Misra, Wei-Bo Gong, and Don Towsley. Fluid-based analysis of a network of aqm routers supporting tcp flows with an application to red. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 151–160, New York, NY, USA, 2000. ACM.

[196] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 537–550, New York, NY, USA, 2015. ACM.

[197] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[198] Walaa Eldin Moustafa, Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. Datalography: Scaling datalog graph analytics on graph processing systems. In *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, pages 56–65, 2016.

[199] Ingo Müller, Renato Marroquin, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. 2020.

[200] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-efficient aggregation: Hashing is sorting. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1123–1136. ACM, 2015.

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

[202] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. July 2015.

[203] Hao Ni, Xin Dong, Jinsong Zheng, and Guangxi Yu. *An Introduction to Machine Learning in Quantitative Finance*. WORLD SCIENTIFIC (EUROPE), 2021.

[204] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[205] nsnam. ns-3, 2017. http://nsnam.org.

[206] Open Compute Project. *Server/SpecsAndDesigns*, June 2018. http://www.opencompute.org/wiki/Server/SpecsAndDesigns.

[207] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: Acomputation model for intelligent memory. 1998.

[208] Amy Ousterhout, Jonathan Perry, Hari Balakrishnan, and Petr Lapukhov. Flexplane: An experimentation platform for resource management in datacenters. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 438–451, Boston, MA, 2017. USENIX Association.

[209] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. 2015.

[210] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13*, pages 69–84. ACM, 2013.

[211] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. 1984.

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

[213] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. Quickstep: A data platform based on the scaling-up approach. *PVLDB*, 11(6):663–676, 2018.

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

[215] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. 2020.

[216] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, 2015.

[217] Postgres-XL. Postgres-xl: Open source scalable sql database cluster. https://www.postgres-xl.org/, 2019.

[218] PostgreSQL. PostgreSQL: The world's most advanced open source relational database. https://www.postgresql.org/, 2022.

[219] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. USA, 2010.

[220] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In Jay R. Lorch and Minlan Yu, editors, *NSDI' 19*, pages 193–206. USENIX Association, 2019.

[221] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 266–277, New York, NY, USA, 2011. ACM.

[222] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 353–364. IEEE Computer Society, 2003.

[223] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[224] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 410–424, 2015.

[225] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM '15*, 2015.

[226] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. *Comput. Commun. Rev.*, 45(5):123–137, 2015.

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

[228] Semih Salihoglu and Jennifer Widom. GPS: a graph processing system. In *Conference on Scientific and Statistical Database Management, SSDBM '13, Baltimore, MD, USA, July 29 - 31, 2013*, pages 22:1–22:12, 2013.

[229] Allison Dulin Salisbury. Impacts of moocs on higher education. https://www.insidehighered.com/blogs/higher-ed-gamma/impacts-moocs-higher-education?v2, 2014.

[230] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz

Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shivakumar Venkataraman. F1 query: Declarative querying at scale. *Proc. VLDB Endow.*, 11(12):1835–1848, 2018.

[231] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13*, pages 351–364. ACM, 2013.

[232] Jonathan J.J.M. Seddon and Wendy L. Currie. A model for unpacking big data analytics in high-frequency trading. *Journal of Business Research*, 70:300–307, 2017.

[233] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 6(14):1906–1917, 2013.

[234] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 433–448. ACM, 2019.

[235] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. 2018.

[236] Prateek Sharma, Ahmed Ali-Eldin, and Prashant J. Shenoy. Resource deflation: A new approach for transient resource reclamation. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 33:1–33:17. ACM, 2019.

[237] Prateek Sharma, Stephen Lee, Tian Guo, David E. Irwin, and Prashant J. Shenoy. Spotcheck: designing a derivative iaas cloud on the spot market. In Laurent Réveillère, Tim Harris, and Maurice Herlihy, editors, *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015*, pages 16:1–16:15. ACM, 2015.

[238] Robert Sheldon. Remote direct memory access (rdma). `https://www.techtarget.com/searchstorage/definition/Remote-Direct-Memory-Access`, 2021.

[239] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 365–378, Berkeley, CA, USA, 2010. USENIX Association.

[240] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 365–378, Berkeley, CA, USA, 2010. USENIX Association.

[241] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1135–1149, 2016.

[242] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A Network Architecture for Disaggregated Racks. 2019.

[243] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *Proc. Data Compression Conference*, 2015.

[244] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: Smart remote memory. 2020.

[245] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 183–197. ACM, 2015.

[246] Rahul Singh, Prateek Sharma, David E. Irwin, Prashant J. Shenoy, and K. K. Ramakrishnan. Here today, gone tomorrow: Exploiting transient servers in datacenters. *IEEE Internet Comput.*, 18(4):22–29, 2014.

[247] Panagiotis Sioulas and Anastasia Ailamaki. Scalable multi-query execution using reinforcement learning. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1651–1663. ACM, 2021.

[248] Mirko Stoffers, Ralf Bettermann, James Gross, and Klaus Wehrle. Enabling distributed simulation of omnet++ inet models. In *Proceedings of the 1st OMNeT++ Community Summit*, 2014.

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

[250] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7), June 1981.

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

[252] S. Sudarshan and Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings.*, pages 501–511, 1991.

[253] Kian-Lee Tan, Qingchao Cai, Beng Chin Ooi, Weng-Fai Wong, Chang Yao, and Hao Zhang. In-memory databases: Challenges and opportunities from software and hardware perspectives. *SIGMOD Rec.*, 44(2):35–40, 2015.

[254] Zhangxi Tan, Zhenghao Qian, Xi Chen, Krste Asanovic, and David Patterson. Diablo: A warehouse-scale computer network simulator using fpgas. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 207–221, New York, NY, USA, 2015. ACM.

[255] Douglas B. Terry. *Replicated Data Management for Mobile Computing*. Synthesis Lectures on Mobile and Pervasive Computing. Morgan & Claypool Publishers, 2008.

[256] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. DFI: the data flow interface for high-speed networks. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1825–1837. ACM, 2021.

[257] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 30:1–30:14. ACM, 2020.

[258] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. 2017.

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

[260] 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 Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1541–1555. ACM, 2018.

[261] Andras Varga. The omnet++ discrete event simulation system. *Proc. ESM'2001*, 9, 01 2001.

[262] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In Laurent Réveillère, Tim Harris, and Maurice Herlihy, editors, *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015*, pages 18:1–18:17. ACM, 2015.

[263] K. V. Vishwanath, D. Gupta, A. Vahdat, and K. Yocum. Modelnet: Towards a datacenter emulation environment. In *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*, pages 81–82, Sep. 2009.

[264] Krishnaswamy Viswanathan. Intel memory latency checker. https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html (visited on 12/24/2021).

[265] Werner Vogels. https://twitter.com/werner/status/25137574680.

[266] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. 2020.

[267] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.

[268] Todd R. Weiss. Google launches tpu v4 ai chips. https://www.hpcwire.com/2021/05/20/google-launches-tpu-v4-ai-chips/, 2021.

[269] Janet Wiener and Nathan Bronson. Facebook's top open data problems. https://research.facebook.com/blog/2014/10/facebook-s-top-open-data-problems/, 2014.

[270] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *NSDI*, 2011.

[271] Emile Witteman. Google to make its own custom server chips as well. https://www.techzine.eu/news/infrastructure/57391/google-to-make-its-own-custom-server-chips-as-well/, 2021.

[272] Jiacheng Wu, Jin Wang, and Carlo Zaniolo. Optimizing parallel recursive datalog evaluation on multicore machines. In Zachary Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1433–1446. ACM, 2022.

[273] Sam Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. Beyond the wall: Near-data processing for databases. In *Proceedings of the International Workshop on Data Management on New Hardware*, 2015.

[274] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Management of multilevel, multiclient cache hierarchies with application hints. *ACM Trans. Comput. Syst.*, 29(2):5:1–5:51, 2011.

[275] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, Boston, MA, 2018. USENIX Association.

[276] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. 2012.

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

[278] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 247–260, New York, NY, USA, 2009. ACM.

[279] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. Analyticdb: Real-time OLAP database system at alibaba cloud. *Proc. VLDB Endow.*, 12(12):2059–2070, 2019.

[280] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In Steve Uhlig and Olaf Maennel, editors, *Proceedings of the 2017 Internet Measurement Conference, IMC 2017, London, United Kingdom, November 1-3, 2017*, pages 78–85. ACM, 2017.

[281] Qizhen Zhang, Akash Acharya, Hongzhi Chen, Simran Arora, Ang Chen, Vincent Liu, and Boon Thau Loo. Optimizing declarative graph queries at large scale. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1411–1428. ACM, 2019.

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

[283] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. Redy: Remote dynamic memory cache. *Proc. VLDB Endow.*, 15(4):766–779, 2022.

[284] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. Rethinking data management systems for disaggregated data centers. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[285] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Understanding the effect of data center resource disaggregation on production dbmss. *Proceedings of the VLDB Endowment*, 13(9):1568–1581, May 2020.

[286] Qizhen Zhang, Hongzhi Chen, Da Yan, James Cheng, Boon Thau Loo, and Purushotham V. Bangalore. Architectural implications on the performance and cost of graph analytics systems. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, pages 40–51. ACM, 2017.

[287] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Optimizing data-intensive systems in disaggregated data centers with TELEPORT. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1345–1359. ACM, 2022.

[288] Qizhen Zhang, Kelvin K. W. Ng, Charles W. Kazer, Shen Yan, João Sedoc, and Vincent Liu. Mimicnet: fast performance estimates for data center networks with machine learning. In *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 287–304. ACM, 2021.

[289] Qizhen Zhang, Tengyuan Ye, Meryem Essaidi, Shivani Agarwal, Vincent Liu, and Boon Thau Loo. Predicting startup crowdfunding success through longitudinal social engagement analysis. In Ee-Peng Lim, Marianne Winslett, Mark Sanderson, Ada Wai-Chee Fu, Jimeng Sun, J. Shane Culpepper, Eric Lo, Joyce C. Ho, Debora Donato, Rakesh Agrawal, Yu Zheng, Carlos Castillo, Aixin Sun, Vincent S. Tseng, and Chenliang Li, editors, *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, pages 1937–1946. ACM, 2017.

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

[291] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 479–491, New York, NY, USA, 2015. ACM.

[292] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas E. Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 362–375, 2017.

[293] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Xuan Kelvin Zou, Hang Guan, Arvind Krishnamurthy, and Thomas Anderson. RAIL: A case for redundant arrays of inexpensive links in data center networks. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 561–576, Boston, MA, 2017. USENIX Association.

[294] Danyang Zhuo, Qiao Zhang, Xin Yang, and Vincent Liu. Canaries in the network. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 2016.

[295] Danyang Zhuo, Qiao Zhang, Xin Yang, and Vincent Liu. Canaries in the network. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 2016.

[296] Tobias Ziegler, Viktor Leis, and Carsten Binnig. RDMA communciation patterns. *Datenbank-Spektrum*, 20(3):199–210, 2020.