Scaling Up Concurrent Analytical Workloads on Multi-Core Servers

THÈSE N^{O} 7222 (2016)

PRÉSENTÉE LE 24 NOVEMBRE 2016

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES ET APPLICATIONS DE TRAITEMENT DE DONNÉES MASSIVES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Iraklis PSAROUDAKIS

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2016

The nonexistent is whatever
we have not sufficiently desired.
— Nikos Kazantzakis, *Report to Greco*

To my parents, to my family, to my friends…

Acknowledgements

In this small section, I would like to express my immense gratitude to all the people who helped me reach “the light at the end of the tunnel” and offered me great joy and happiness.

First and foremost, I would like to thank my advisor *Anastasia Ailamaki*. With her constant encouragement and guidance, she helped me find my true calling and materialize my potential. I also thank Natassa for being the force behind my internship at the SAP HANA team, which later turned into a fruitful collaboration for my PhD thesis.

I would also like to thank the jury members of my PhD thesis committee for their participation and their comments: *James Larus*, *Thomas Neumann*, and *Ravi Rajwar*.

Next, I would like to thank my collaborators and colleagues at the DIAS lab of EPFL. *Manos Athanassoulis* helped me with the first steps in the research world, and mentored me during my first PhD project. I was anxiously bickering at him but he always found a way to calm me down. Then there is the super duo of *Danica Porobic* and *Pinar Tözün*, whose advice, experience, and warm attitude were second to none. I would also like to thank *Ioannis Alagiannis* for soothing us with cool frappés, *Manos Karpathiotakis* for being at the forefront of all entertainment activities inside and outside the lab, *Matt Olma* for his awesomeness and for directing all the high-quality movies of the lab, *Mirjana Pavlovic* for her beautiful presence and attitude, *Renata Borovica* for giving us hopeful glimpses of family life, *Darius Sidlauskas* for being a great company and capturing me in some of my most awkward poses during our trips, *Adrian Popescu* for capturing us in astonishing pictures, *Raja Appuswamy* for constantly motivating us to go out after work, *Utku Sirin* for worthily continuing the legacy of the aforementioned super duo, *Eleni Tzirita Zacharatou* for mesmerizing us with countless trip pictures, *Georgios Psaropoulos* for effortlessly picking up the baton at SAP, *Odysseas Papapetrou* for being an awesome officemate, *Angelos Anadiotis* for our short breaks of fresh air, *Thomas Heinis* for always having the most unexpected and hilarious comments, *Radu Stoica* for switching our attention to the important details, *Erietta Liarou* for our wonderful cooperation, *Gaidioz Benjamin Cyrille Damien* and *Lionel Sambuc* for helping me and translating my abstract in French, *Farhan Tauheed* for even more amazing photographs, *Satya Valluri* for our interesting research discussions, *Giannakopoulou Styliani Asimina* for being an amazing junior student, *Cesar Matos* for all the coffees I drank, *Tahir Azim* for his help during my dry-runs, and of course *Dimitra Tsaoussis Melissargos* and *Erika Raetz* for helping us in all administrative tasks.

I feel extremely grateful for having more awesome friends in Lausanne. I would like to thank *Vasileios Trigonakis* for being an invaluable friend who stood by me through all the years of my PhD, *Konstantina Christakou* and *Javier Picorel Obando* for being awesome friends and a great company, *Christina Vlachou* and *Iris Safaka* for helping define an unforgettable era, *Michalis Zervos*, *Alexandros Daglis* and *Onur Yürüten* for being amazing flatmates, and all other friends in Lausanne including, but not limited to, *Eleni Smyrnaiou*, *Katerina Liapi*, *Stella Sarri*, *Anastasia Mavridou*, *George Chatzopoulos*, *Stefanos Skalistis*, *Bojana Paunovic*, *Sofia Karygianni*, *Dorina Thanou*, *David Kozhaya*, *Tudor David*, *Stanko Novakovic*, *Milos Vasic*, *Mohammed El Seidy*, and *Emre Atsan*.

Due to our collaboration with the SAP HANA team, I was also lucky enough to spend a great deal of my PhD time at another wonderful place: in Heidelberg in Germany. I would like to thank all the colleagues in the SAP HANA team. I would like to thank *Norman May* for being a great mentor and helping me with all our publications and my thesis, *Tobias Scheuer* for our effortless collaboration on the task scheduler, *Abdelkader Sellami* for helping transfer our research to the real world, *Wolfgang Lehner* and *Kai-Uwe Sattler* for our collaboration, *Arne Schwarz* for shaping an amazing campus team, *Ingo Mueller* and *Ismail Oukid* for giving me strength and for participating together in the SIGMOD programming contest, *Florian Wolf* for being a great junior student to advise and work with, *David Kernert* and *Max Wildemann* for our outings, *Marcus Paradies* and *Michael Rudolf* for our discussions and their advice, *Robert Brunel* for the amazing photographs, *Elena Vasilyeva* for her warm attitude and russian treats, *Frank Tetzel* and *Lucas Lersch* for being great new members of the team, *Matthias Hauck* and *Thomas Bach* for their support, and *Hannes Rauhe* for the hilarious discussions. I would also like to thank all other members, collaborators, and friends from SAP including, but not limited to, *Alexander Boehm*, *Roman Dementiev*, *Thomas Willhalm*, *Daniel Booss*, *Konstantinos Stergios*, *Konstantinos Drachtidis*, *Ioannis Kostis*, *Panagiotis Germanakos*, *Waltraud Germanakos*, and *Panagiotis Drouvelis*. Of course I had the chance to make more great friends in Heidelberg as well. I would especially like to thank *Christina Papadimitraki*, *Anthi Papasavva*, *Costas Demetriades*, *Helena Kotsis*, *Ioanna Lampropoulou*, *Nikos Doumpas*, *Gloria Llopart*, and *Dimitris Chloridis* for their company and support.

Additionally, I would like to thank my friends during my time in Irakleio and Athens, who all supported me and contributed to my journey to reach this point. I would like to thank *Vaggelis Polychronopoulos*, *Anna Schizaki*, *Maria Petraki*, and *Elina Vasilikogiannaki* for being invaluable friends since childhood. Moreover, I would like to thank my friends during my time in Athens: *Georgios Santikos*, *Lina Stavridou*, *Renata Raidou*, *Minos Kontos*, *Alexandros Sigaras*, *Dimitris Rizadis*, *Theodora Chaspari*, *Victoria Pismiri*, and *Bill Stavroulakis*.

Last but not least, I would like to say many thanks to my family, without whom this journey would not have been possible: my mother *Evangelia Psaroudaki*, my father *Evangelos Psaroudakis*, and my brother *Ioannis Psaroudakis*.

Finally, I would like to acknowledge the funding that supported my PhD research. My work and research on sharing across concurrent queries was supported by the European Commission Collaborative Project "BigFoot", Grant Agreement Number 317858. My work and research on NUMA-aware data placement and task scheduling was supported by SAP S.E.

Abstract

Today, an ever-increasing number of researchers, businesses, and data scientists collect and analyze massive amounts of data in database systems. The database system needs to process the resulting highly concurrent analytical workloads by exploiting modern multi-socket multi-core processor systems with non-uniform memory access (NUMA) architectures and increasing memory sizes. Conventional execution engines, however, are not designed for many cores, and neither scale nor perform efficiently on modern multi-core NUMA architectures. Firstly, their query-centric approach, where each query is optimized and evaluated independently, can result in unnecessary contention for hardware resources due to redundant work found across queries in highly concurrent workloads. Secondly, they are unaware of the non-uniform memory access costs and the underlying hardware topology, incurring unnecessarily expensive memory accesses and bandwidth saturation. In this thesis, we show how these scalability and performance impediments can be solved by exploiting sharing among concurrent queries and incorporating NUMA-aware adaptive task scheduling and data placement strategies in the execution engine.

Regarding sharing, we identify and categorize state-of-the-art techniques for sharing data and work across concurrent queries at run-time into two categories: reactive sharing, which shares intermediate results across common query sub-plans, and proactive sharing, which builds a global query plan with shared operators to evaluate queries. We integrate the original research prototypes that introduce reactive and proactive sharing, perform a sensitivity analysis, and show how and when each technique benefits performance. Our most significant finding is that reactive and proactive sharing can be combined to exploit the advantages of both sharing techniques for highly concurrent analytical workloads.

Regarding NUMA-awareness, we identify, implement, and compare various combinations of task scheduling and data placement strategies under a diverse set of highly concurrent analytical workloads. We develop a prototype based on a commercial main-memory column-store database system. Our most significant finding is that there is no single strategy for task scheduling and data placement that is best for all workloads. In specific, inter-socket stealing of memory-intensive tasks can hurt overall performance, and unnecessary partitioning of data across sockets involves an overhead. For this reason, we implement algorithms that adapt task scheduling and data placement to the workload at run-time.

Our experiments show that both sharing and NUMA-awareness can significantly improve the performance and scalability of highly concurrent analytical workloads on modern multi-core servers. Thus, we argue that sharing and NUMA-awareness are key factors for supporting faster processing of big data analytical applications, fully exploiting the hardware resources of modern multi-core servers, and for more responsive user experience.

Keywords: Database management systems, Analytical processing systems, Multi-socket multi-core servers, Non-uniform hardware topologies, Sharing data and work, Task scheduling, Data placement, NUMA-awareness

Résumé

Aujourd'hui, un nombre toujours croissant de chercheurs, d'entreprises et de scientifiques recueillent et analysent des quantités massives de données à l'aide de systèmes de gestion de base de données. Le système de base de données doit traiter les tâches d'analyses en parallèle en exploitant les systèmes de processeurs multi-coeur multi-socket modernes avec des architectures d'accès mémoire non uniforme (NUMA), ceci pour des quantités de mémoire croissantes. Les moteurs d'exécution classiques ne sont cependant pas conçus pour les processeurs multi-coeur et leur performance n'augmente pas de manière importante sur les architectures modernes multi-coeur NUMA. Tout d'abord, leur approche centrée sur la requête, où chaque requête est optimisée et évaluée de façon indépendante, peut avoir pour effet une contention inutile autour des ressources matérielles en raison de calculs redondants qu'on retrouve dupliqués dans des requêtes concurrentes. Deuxièmement, ils ne prennent pas en compte les coûts d'accès mémoire non uniformes et la topologie non uniforme du matériel, occasionnant des accès mémoire inutilement coûteux et la saturation de la bande passante. Dans cette thèse, nous montrons comment ces obstacles d'évolutivité et de performance peuvent être dépassés en exploitant le partage entre les requêtes concurrentes ainsi que l'intégration dans le moteur d'exécution de stratégies adaptatives de planification de tâches et de placement de données tenant compte de la NUMA.

En ce qui concerne le partage, nous identifions les techniques connues de partage de données et de calculs à travers des requêtes concurrentes en cours d'exécution et nous les classons en deux catégories : le partage réactif, qui réutilise les résultats intermédiaires de sous-plans communs à plusieurs requêtes, et le partage proactif, qui, pour exécuter l'ensemble des requêtes, établit un plan d'exécution global basé sur des opérateurs partagés. Nous intégrons à notre étude les prototypes de recherche originaux qui introduisent le partage réactif et proactif, effectuons une analyse de sensibilité, et montrons comment et quand chaque technique améliore la performance. Notre conclusion la plus importante est que les partages réactif et proactif peuvent être combinés afin d'exploiter leurs avantages spécifiques pour traiter un ensemble de requêtes analytiques hautement concurrentes.

En ce qui concerne la prise en compte des architectures NUMA, nous identifions, mettons en œuvre, et comparons des différentes combinaisons de stratégies de planification de tâches et de placement de données dans un vaste ensemble de charges de travail analytiques hautement concurrents. Nous développons un prototype basé sur un système commercial de base de données en mémoire. Notre constatation la plus importante est qu'il n'y a pas de stratégie optimale unique pour la planification des tâches et le placement des données. En particulier, le vol des tâches gourmandes en mémoire entre les sockets peut nuire à la performance, et le partitionnement inutile des données sur les sockets implique une surcharge. Pour cette raison, nous mettons en œuvre des algorithmes qui adaptent la planification des tâches et le placement des données à la charge de travail en cours d'exécution.

Nos expériences montrent que le partage et la prise en compte des architectures NUMA peuvent améliorer les performances et l'évolutivité des charges de travail analytiques hautement concurrentes sur des serveurs multi-coeur modernes de manière significative. Ainsi, nous défendons l'idée que le partage et l'adaptation à la NUMA sont des facteurs clés pour offrir un traitement plus efficace de l'analyse de données en exploitant pleinement les ressources matérielles des serveurs multi-coeur modernes, et offrir une expérience utilisateur plus réactive.

Mots clefs : Systèmes de gestion de base de données, Systèmes de traitement analytique, Serveurs multi-socket multi-coeurs, Topologies matérielles non uniformes, Partage des données et du travail, Planification des tâches, Placement de données, Adaptation à la NUMA

Contents

Acknowledgements **i**

Abstract (English/Français) **v**

Table of Contents **ix**

List of Figures **xiii**

List of Tables **xvii**

1 Introduction **1**

1.1 Modern Multi-Core Processor Servers 2

1.2 Why Conventional DBMS Do Not Scale Up Efficiently 3

1.3 How to Scale Up Efficiently 4

1.3.1 Dimension #1: Sharing Data and Work Across Concurrent Queries 4

1.3.2 Dimension #2: NUMA-Aware Task Scheduling and Data Placement . . . 5

1.4 Thesis Statement, Contributions, and Methodology 7

2 Background and Related Work **11**

2.1 Disk-Based Row-Stores vs. Main-Memory Column-Stores 11

2.2 Sharing Techniques 12

2.2.1 Reactive Sharing 15

2.2.2 The QPipe Execution Engine 16

2.2.3 Proactive Sharing 17

2.2.4 The CJOIN Operator 19

2.3 Task Scheduling 20

2.4 Non-Uniform Memory Access (NUMA) 22

2.4.1 Multi-Socket Servers and NUMA-Awareness 22

2.4.2 Related Work in NUMA-Aware Task Scheduling and Data Placement . . 26

2.5 Overview of SAP HANA 29

3 Sharing Data and Work Across Concurrent Queries **33**

3.1 Introduction 33

3.1.1 Methodologies for Sharing Data and Work 33

3.1.2 Integrating Reactive and Proactive Sharing 34

3.1.3 Optimizing Reactive Sharing 34
3.1.4 Reactive vs. Proactive Sharing 35
3.1.5 Contributions 36
3.2 Integrating Reactive and Proactive Sharing 36
3.2.1 Benefits of Applying Reactive Sharing to Proactive Sharing 37
3.2.2 CJOIN as a QPipe Stage 38
3.3 Optimizing Reactive Sharing 39
3.3.1 Design of a Shared Pages List 42
3.4 Experimental Evaluation 43
3.4.1 Experimental Methodology 43
3.4.2 Impact of Concurrency 45
3.4.3 Impact of Data Size 46
3.4.4 Impact of Similarity 49
3.4.5 SSB Query Mix 51
3.5 Conclusions 52

4 Task Scheduling for Highly Concurrent Main-Memory Workloads 53
4.1 Integrating Task Scheduling in a DBMS 53
4.1.1 Integration in SAP HANA 55
4.1.2 Task Scheduler Architecture 56
4.1.3 Dynamic Adjustment of Concurrency Level 58
4.1.4 Dynamic Adjustment of Task Granularity 61
4.1.5 Experimental Evaluation 62
4.2 Scaling Up Mixed Transactional and Analytical Workloads 65
4.2.1 Mixed Workloads in SAP HANA 67
4.2.2 Mixed Workloads in HyPer 68
4.2.3 Setting Up the CH-benCHmark 70
4.2.4 Experimental Evaluation 71
4.3 Summary and Conclusions 77

5 NUMA-Aware Task Scheduling and Data Placement Strategies 79
5.1 Introduction 79
5.2 Data Placement of a Main-Memory Column 80
5.2.1 Tracking Memory 82
5.3 NUMA-Aware Task Scheduling 84
5.3.1 NUMA-Aware Scheduling of Scans 86
5.4 Experimental Evaluation of Concurrent Scans 88
5.4.1 Uniformly Distributed Workload 89
5.4.2 Skewed Workload 93
5.5 NUMA-Aware Scheduling of Aggregations and Equi-Joins 96
5.5.1 The Overhead of Partitioning and Stealing 98
5.5.2 TPC-H and SAP BW-EML Benchmarks 101
5.6 Summary and Conclusions 102

6 Adaptive NUMA-Aware Task Scheduling and Data Placement 103
6.1 Introduction 103
6.2 Tracking Resource Utilization 105
6.3 Adaptive Data Placement 108
6.3.1 Abstract Workflow 108
6.3.2 Information and Eligibility of RUH 110
6.3.3 Reducing the Utilization Imbalance 112
6.3.4 Data Placer 118
6.4 Adaptive Task Stealing 119
6.5 Experimental Evaluation 121
6.5.1 Adaptive Data Placement 123
6.5.2 Adaptive Task Stealing 124
6.5.3 Partitioning Overhead 125
6.5.4 Changing Workload 126
6.5.5 Workload Mix 127
6.5.6 TPC-H benchmark 127
6.6 Conclusions 128

7 Conclusions and Future Directions 131
7.1 Thesis Summary 131
7.2 Future Directions 132
7.3 Impact 134

Bibliography 153

Curriculum Vitae 155

List of Figures

1.1 4-socket server. 2

1.2 The execution engine in a typical DBMS. 3

1.3 Thesis methodology 8

2.1 Evaluation of three concurrent queries using (a) a query-centric model, (b) shared scans, (c) reactive sharing, and (d) proactive sharing. 15

2.2 (a) Example of reactive sharing with two queries having a common sub-plan below the join operator. (b) A step and a linear window of opportunity [91]. (c) Conceptual design of QPipe, with the packets of an example of a query plan dispatched to the relevant stages. 16

2.3 Example of shared selection and hash-join operators. 17

2.4 The CJOIN operator uses proactive sharing by evaluating a GQP for star queries. 19

2.5 An 8-socket server with Intel Xeon E7-8870 processors. 23

2.6 A 4-socket server with Intel Xeon E7-4880 v2 processors. 24

2.7 The general database architecture of SAP HANA. 30

2.8 Example of a dictionary-encoded column. 30

2.9 (a) The data structures of the main and the delta parts of a column. (b) The delta part of a column is periodically merged into the main part. 32

3.1 Integration scheme of CJOIN as a QPipe stage. 38

3.2 Sharing identical results during SP with: (a) push-only model and (b) a SPL. . . 40

3.3 Evaluating multiple identical TPC-H Q1 queries with a push-based model during reactive sharing (leftmost graphs), and with a pull-based model during reactive sharing (middle graphs), using either 1 H/W thread (top graphs) or all available H/W threads (bottom graphs) of the server. The rightmost graphs show the corresponding speedups of the two methods of reactive sharing over not sharing. 41

3.4 Design of a shared pages list. 42

3.5 Size of SPL. 43

3.6 The SSB Q3.2 SQL template and the query plan. 45

3.7 Experiment with memory-resident (left) and disk-resident (middle) database of SF 1. The table (right) includes measurements for the case of 256 concurrent queries. 45

3.8 8 queries with a memory-resident database of SF 10. The table includes measurements for 30% selectivity. 47
3.9 Memory-resident database of SF 10 and 30% selectivity. The table includes measurements for 256 queries. 48
3.10 8 concurrent queries with disk-resident databases. The table includes measurements for the case of SF 100. 49
3.11 Disk-resident database of SF 1 and 16 possible plans. The table includes measurements for 256 queries. 50
3.12 Evaluating 512 concurrent queries with a varying similarity factor, for a SF 100. The table includes the sharing opportunities of reactive sharing (average of all iterations). 50
3.13 Disk-resident database of SF 30. Response time (left) and throughput experiment (middle), varying the number of concurrent queries and clients respectively. The table (right) includes measurements for both experiments. 51

4.1 Our task scheduler integrates two main thread pools of SAP HANA. 56
4.2 The data structures used by the task scheduler. 57
4.3 The task scheduler's types of worker threads. 59
4.4 Experiment with TPC-H queries. Measurements on the right-hand side are for the case of 1024 TPC-H concurrent queries. 63
4.5 Timelines for *Fixed* (left) and *Hints* (right), for the case of 1024 TPC-H queries. . 64
4.6 Conceptual figures of how we expect the performance of mixed workloads to be affected by (a) data freshness, (b) flexibility, and (c) scheduling. 66
4.7 (a) HyPer design using virtual memory facilities to create snapshots for analytics [101]. (b) Restricting flexibility to support further optimizations of query plans [117]. (c) Conceptual figure of scheduling for pipeline R of the query plan of (b) [117]. 69
4.8 Schema of the CH-benCHmark. 71
4.9 Performance of the default configuration of SAP HANA. 73
4.10 Performance of SAP HANA when intra-query parallelism is disabled. 74
4.11 Performance of HyPer with the lowest level of analytical data freshness. 75
4.12 Performance of HyPer with an intermediate level of analytical data freshness. . 76

5.1 (a) Impact of NUMA on the performance of concurrent clients issuing scan-heavy queries. (b) Memory throughput of the sockets for the case of 1024 clients. 80
5.2 Different data placements of a dictionary-encoded column on four sockets. . . 81
5.3 Example of a PSM after adding the virtual memory ranges to track (bold lines). 83
5.4 The design of our NUMA-aware task scheduler. 85
5.5 Task scheduling (a) for a scan or index lookups to find qualifying matches, and (b) for the output materialization, for an IVP-placed column. 86
5.6 Evaluating *OS*, *Target*, and *Bound* scheduling strategies, with *RR*-placed columns. 90
5.7 As Figure 5.6, on the 8-socket Westmere-EX server. 90
5.8 The effect of the *RR*, *IVP*, and *PP* data placements. 91

5.9 Violin plots of the latency distributions. 91
5.10 Evaluating 1120 clients with different data placement and scheduling strategies, on a 32-socket server. 92
5.11 Evaluating different selectivities. 93
5.12 Evaluating the *OS*, *Target*, and *Bound* strategies, with *RR*-placed columns. . . . 94
5.13 Evaluating the *RR*, *IVP*, and *PP* data placements, with the *Bound* strategy. . . . 94
5.14 As Figure 5.13, with a high selectivity. 95
5.15 As Figure 5.14, with the *Target* strategy. 95
5.16 NUMA-aware aggregation with a table partitioned across two sockets. 97
5.17 256 clients issuing aggregation or join queries on the 4-socket server, under different cases of selectivity, available tables, stealing, and partitioning. Each result of LLC misses belongs to the corresponding case of the left-hand side graphs. 99
5.18 TPC-H and SAP BW-EML with different *PP* granularities and scheduling strategies.101

6.1 A conceptual example of our adaptive data placement. 104
6.2 Tracking the resource utilization of tasks, TBP and TGP, and sockets. The values are taken from instances of our experiments on our 4-socket server. 106
6.3 Each table part (TBP), table group part (TGP), and socket is associated with a resource utilization history (RUH). 107
6.4 Abstract workflow of the Data Placer (DP). 108
6.5 Conceptual examples of calculating the partitioning eligibility of (a) an ineligible RUH, and (b) an eligible RUH. 112
6.6 Calculating the utilization imbalance of a pair of sockets before and after (a) moving, and (b) partitioning. 114
6.7 Calculating the imbalance considering (a) that the utilization of RUHs' owner does not increase, and (b) that it actually increases after moving a RUH's owner. 115
6.8 Calculating the imbalance considering (a) that the utilization of RUHs' owners does not increase, and (b) that it actually increases after partitioning a RUH's owner. 116
6.9 Calibration experiment for three NUMA servers. 120
6.10 Adaptive data placement of three active tables (4-socket server). 123
6.11 Experiment showing how adaptive task stealing disallows stealing of memory-intensive classes (4-socket server). 124
6.12 The overhead of partitioning for aggregations (8-socket server). 125
6.13 The impact of the number of groups (8-socket server) 126
6.14 Three different workload phases (8-socket server). 126
6.15 Balancing the utilization of a stable workload mix (8-socket server). 127
6.16 TPC-H throughput run (32-socket server). 128

List of Tables

2.1 Sharing methodologies employed by a query-centric model and the research prototypes we examine. 13
2.2 Performance characteristics of the multi-socket servers that we use. 24

3.1 Rules of thumb for sharing data and work across typical concurrent queries. . . 36

5.1 Workload properties best fitted for each data placement, and key characteristics. 82

6.1 Configurable parameters used in our algorithms. 109
6.2 Further characteristics of the NUMA servers of Table 2.2 that we use in this chapter's experimental evaluation. 121

1 Introduction

Traditionally, *online transaction processing (OLTP)* workloads have been the motivating force behind the early relational *database management systems (DBMS)* of the 1970's, and many modern DBMS following their trail [84]. OLTP workloads are composed of short-lived transactions that read or modify operational data, and are typically standardized, submitted through application layers such as an Enterprise Resource Planning (ERP) software or an online shop. During the 1990's, however, a different form of data analysis emerged. The increasing importance of business intelligence led to another class of long-running, scan-heavy, ad-hoc queries, namely *online analytical processing (OLAP)* workloads [51]. Due to these substantial differences from OLTP workloads, OLAP workloads are supported by specialized database systems that are typically used in data warehouses [102]. Operational data is periodically extracted from OLTP systems, transformed, and loaded into data warehouses for analytics.

Nowadays, the importance of analytical workloads is even more prominent, as the collection and analysis of massive data is a key factor for the competitiveness of numerous businesses, and the insight required in many scientific endeavors [181]. Terms such as "data deluge" or "big data" have been coined to characterize the prominence of complex analytics [5]. Examples include the Large Hadron Collider (LHC) that produces around 30 petabytes of data annually [4], or the 1000 Genomes Project that has generated about 200 terabytes of data [5]. The main characteristics of big data, all of which together comprise the five "V's" of big data [95, 198], are: (a) volume, due to the large size of the data, (b) variety, such as unstructured or audiovisual data, (c) veracity, referring to the trustworthiness of the data, (d) velocity, meaning the frequency of incoming data and analytical requests, and finally (e) value, referring to the collective benefits of big data analytics.

In this thesis, we focus more on the velocity of incoming analytical requests, and specifically on the increasing concurrency of analytical workloads. According to a study of the data warehouses market [172], the majority of businesses using data warehouses serviced up to 50 concurrent users in 2012. The study projects that in the near future, data warehouses will need to service up to 1000 concurrent users. Another example from the industry is found in the sizing guide of SAP for business intelligence infrastructure [7], mentioning a number of 500 active concurrent users for business analytics.

DBMS are being called to scale up on modern hardware in order to efficiently process highly concurrent analytical workloads. DBMS need to improve their performance by efficiently exploiting the increase in processing power offered by a modern multi-core processor server.

1.1 Modern Multi-Core Processor Servers

The underlying hardware of database systems has been constantly evolving over the past decades. The hardware provides better performance and more parallelism which can be used by the database system to scale up and handle highly concurrent workloads. The way that parallelism is offered by the hardware, however, is non-uniform. Parallelism is enabled at different levels of a system, ranging from single-core capabilities to multi-socket multi-core capabilities [27, 28].

Since the first processors of the 1970's, processing power has been steadily improving according to Moore's Law [136], which states that the number of transistors in microprocessors doubles approximately every two years. Initially, the processing power was being increased by improving the performance of a single-core processor. Parallelism features at the level of a single core appeared with simultaneous multi-threading (SMT) [187]. Since the last decade, it has become increasingly difficult to cram smaller and smaller transistors into the same central processing unit (CPU). Heat dissipation has become a major limitation for further improving the performance of a single core [89]. As a way out of these constraints, processor vendors are providing more cores in a single CPU.

Most systems with just a few cores use a symmetric multiprocessing (SMP) architecture, where all cores use a single bus to access shared I/O devices and main memory. SMP simplifies both hardware and software, since all cores and shared resources are treated equally and uniformly. The common bus, however, creates a significant bottleneck for further scaling up the number of cores. In addition, main memory sizes have been increasing rapidly. Processor vendors are solving this issue by de-centralizing main memory, forming a *non-uniform memory access (NUMA)* architecture [41, 106, 174]. Typically, a multi-core processor is attached to one socket. The cores share a local memory bus, controller, and last-level cache. Then, multiple sockets are interconnected with a communication network in order to enable a processor core to access remote memory of another socket. Figure 1.1 shows

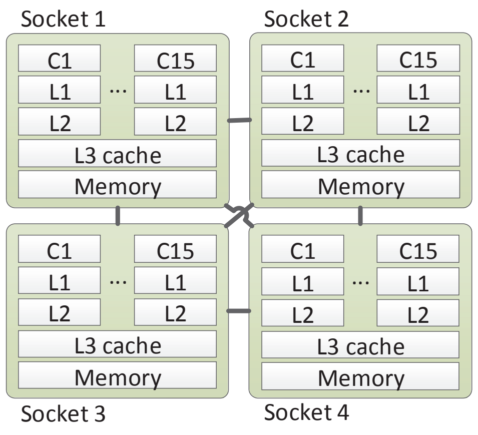


Figure 1.1 – 4-socket server.

a conceptual example of a 4-socket server (see Section 2.4 for more details). The term *multi-socket server* refers to a shared-memory server with multiple sockets of multi-core processors and a cache-coherent NUMA architecture. This means that the typical memory hierarchy of SMT architectures is extended by one more level after the last-level cache, which includes the interconnect network. Communication costs across the memory hierarchy can vary greatly, even by an order of magnitude. Additionally, the bandwidth of an interconnect link is an additional bottleneck to be considered [41, 156].

In summary, a modern server offers high parallelism, with a high number of cores, but at the expense of uniformity. In order to efficiently scale up, DBMS need to both efficiently exploit the available high parallelism, and leverage the knowledge about the non-uniformity of the underlying NUMA architecture.

1.2 Why Conventional DBMS Do Not Scale Up Efficiently

Typical DBMS optimize and execute each query independently [166], following a *query-centric* approach [91]. Figure 1.2 shows the execution engine in a typical DBMS, executing three query plans after the relevant SQL queries have been parsed and optimized [166]. Simply servicing highly concurrent OLAP workloads with a typical execution engine can result in contention for hardware resources, and inefficient use of the performance and parallelism offered by modern hardware, due to two orthogonal issues: lack of (a) sharing, and (b) NUMA-awareness.

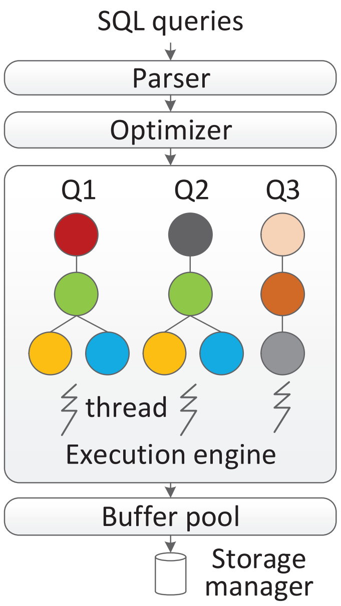


Figure 1.2 – The execution engine in a typical DBMS.

Sharing. A DBMS is expected to handle a high number of concurrent queries with a limited number of computing resources. Although high concurrency itself is a challenge in query execution, the total amount of work can be reduced significantly through synergy among queries. Workloads with increased concurrency can have several queries with common parts of data and work, creating sharing opportunities. For example, queries Q1 and Q2 in Figure 1.2 share a common sub-plan, visualized with operators having the same color, which is executed redundantly two times. Nevertheless, the query-centric model misses these sharing opportunities. Sharing can avoid redundant computations across concurrent queries, decrease unnecessary contention for I/O, CPU and memory resources, and significantly improve overall performance [47, 91, 98, 157].

NUMA-awareness. Typical execution engines are *NUMA-agnostic*: they assume uniformity of memory accesses in the underlying hardware. Figure 1.2 shows a typical case without intra-query parallelism, where each query plan is executed with a logical thread. Thread scheduling, and any in-memory data placement, across the sockets of a multi-socket server are left to the operating system (OS). This can lead to numerous performance problems. For example, issuing too few or too many threads can result in the underutilization or overcommitment of CPU resources [158]. More importantly, the OS cannot readily decide on which sockets to place data and threads. The reason is that the OS lacks application knowledge, and it cannot easily predict the access patterns of database workloads, which are generally more demanding and unpredictable than other typical applications [41]. Thus, query execution often results in uncoordinated and inefficient scheduling and data placement on multi-socket servers. The potential performance problems include slow remote memory accesses and unnecessary bandwidth bottlenecks in a socket's memory controller or an interconnect link [41, 64, 106].

1.3 How to Scale Up Efficiently

To efficiently scale up analytical workloads on modern multi-core processor servers, DBMS need to consider both aforementioned orthogonal issues. Sharing presents an opportunity to be exploited to avoid unnecessary contention of resources due to redundant work and improve performance, especially for highly concurrent workloads that involve sharing opportunities. NUMA-awareness is necessary to scale up on multi-socket servers, by preferring fast local memory accesses and avoiding unnecessary bandwidth bottlenecks. In this section, we summarize the most prominent related work and our contributions in both dimensions of sharing and NUMA-awareness. Chapter 2 includes a detailed discussion of related work.

1.3.1 Dimension #1: Sharing Data and Work Across Concurrent Queries

A variety of ideas have been proposed to exploit sharing. Query-centric DBMS include components that promote sharing, such as buffer pool management techniques [173], materialized views [169], and shared scans [211]. These techniques, however, do not explicitly share across concurrent queries inside the execution engine. More recently, there have been research prototypes focusing on explicit sharing of data and work across concurrent queries at run-time [33, 47, 77, 91].

Our contributions. In this thesis, we include a short survey of the state-of-the-art run-time sharing techniques. We categorize them into two sharing techniques. The first one, *reactive sharing*, reuses common intermediate results across concurrent queries. Reactive sharing is implemented inside an operator-centric execution engine, QPipe [91]. The second run-time sharing technique, *proactive sharing*, builds a global query plan with shared operators to evaluate the whole mix of concurrent queries. The CJOIN operator employs proactive sharing across equi-joins of concurrent star queries [47]. The technique is later advanced to additional operators and general schemas [33, 77].

Furthermore in this thesis, we analyze these novel run-time sharing techniques in order to fully understand how they work, how they compare them to each other, and how they can be used efficiently in practice. We integrate reactive and proactive sharing in the same system, by integrating their original research prototypes, QPipe and CJOIN. We first show that the original implementation of reactive sharing, using a push-based model, where shared intermediate results are pushed to further operators in query plans, involves a serialization point which constitutes a bottleneck on multi-core servers. We propose a pull-based model for reactive sharing, which does not require a serialization point. Additionally, we perform an extensive comparative sensitivity analysis and show how and when it is beneficial to use each one of the sharing techniques. More importantly, we show that reactive and proactive sharing are orthogonal techniques and can be combined to exploit the advantages of both [157, 159].

1.3.2 Dimension #2: NUMA-Aware Task Scheduling and Data Placement

As mentioned in Section 1.2, if the DBMS is NUMA-agnostic and leaves scheduling and in-memory data placement to the OS, there can be numerous performance problems. In this thesis, we show how a NUMA-aware DBMS can avoid these problems by assuming full control of scheduling and in-memory data placement. Our NUMA-aware analysis and implementation of scheduling and data placement is based on a prototype of SAP HANA, a commercial main-memory DBMS [72]. The choice of a main-memory DBMS is due to the fact that NUMA-awareness is more crucial for main-memory DBMS than conventional disk-based DBMS, since disk I/O is the primary bottleneck for disk-based DBMS and NUMA-awareness is a secondary bottleneck (see Section 2.1).

Related work in task scheduling. As a first step towards taking control of scheduling, we propose that the DBMS decouples its scheduling from the OS, by employing *task scheduling* [12, 43, 68, 125, 158]. Task scheduling uses a number of worker threads to process all operations for the entire application lifetime. Tasks, which encapsulate operations, are stored in task pools, and worker threads are employed to process the tasks. A worker thread continuously takes tasks from a task pool and executes the tasks. If a task pool becomes empty, then work or *task stealing* is typically used, i.e., a worker thread attempts to steal tasks from other task pools. The OS is only aware of the worker threads, and the task scheduler is in full control of how the tasks are assigned or scheduled to the worker threads.

Our contributions to task scheduling. In this thesis, we show how task scheduling can be employed for highly concurrent main-memory workloads. We show how we handle blocking tasks by issuing additional worker threads. Also we show that task scheduling can effortlessly support intra-query task parallelism, while avoiding performance problems related to the over-commitment of CPU resources. However, we show that excessive intra-query task parallelism incurs an unnecessary overhead on performance, which can be avoided by using a *concurrency hint*, reflecting the system's recent CPU utilization, to adapt task granularity [158]. Although this thesis is mainly focused on OLAP workloads, we show that for mixed OLTP and OLAP workloads scheduling is one of the significant factors, among data freshness and flexibility, that affect how mixed workloads utilize resources. Specifically for scheduling, we show that

OLAP workloads tend to dominate over the concurrent OLTP workloads, pinpointing the need for workload management and task prioritization [162].

Related work in NUMA-aware task scheduling and data placement. Since our initial evaluation of task scheduling for main-memory DBMS, task scheduling has been prominent in related work about NUMA-awareness in main-memory DBMS. The research prototypes HyPer [117] and ERIS [103] show that task scheduling can easily help model the execution engine after the topology of the underlying hardware on multi-socket servers. Task pools and worker threads can be bound to the sockets of the server, and tasks can have an affinity for a socket. In regard to data placement, the research prototypes typically *partition data*, e.g., tables, across multiple sockets. In regard to scheduling, queries are parallelized with multiple tasks, and each task is issued to the task pool of the socket that contains the data the task targets, in order to prefer fast local memory accesses over slower remote memory accesses. If a socket does not have local tasks, then inter-socket task stealing is used, i.e., a worker thread attempts to steal tasks from task pools of remote sockets.

Our contributions to NUMA-aware task scheduling and data placement. The research prototypes in related work follow a static approach by always partitioning data and using inter-socket task stealing [103, 117]. An analysis of different data placement and task scheduling strategies is missing, which we conduct in this thesis. We extend our initial task scheduler for NUMA-aware task scheduling on multi-socket servers, and we provide a comprehensive analysis of the performance of various data placement and task scheduling strategies under different kinds of workloads, and identify any involved trade-offs [161]. Our analysis identifies two major trade-offs.

First, inter-socket task stealing can help saturate CPU resources in case of skewed workloads, but may hurt overall performance if stolen tasks are memory-intensive, saturating and overwhelming the bandwidth of remote memory controllers and the interconnect links. Second, partitioning data across multiple sockets can also help balance utilization across sockets in case of skewed workloads, but involves an overhead in processing and parallelizing queries. This overhead can be avoided in case the utilization can be balanced without partitioning.

The two trade-offs mentioned above depend on the workload which can change at run-time. To exploit these trade-offs, we propose a design that adapts the task scheduling and data placement strategy to the workload at run-time [163]. We track the history of CPU and memory bandwidth utilization of tables, tasks, and sockets. Regarding task scheduling, we dynamically disallow inter-socket task stealing for memory-intensive tasks. Regarding data placement, our execution engine periodically balances the utilization across sockets by either moving or repartitioning tables, while avoiding the overhead of unnecessary partitioning.

1.4 Thesis Statement, Contributions, and Methodology

Thesis Statement

Analytical workloads in conventional database systems do not scale and perform efficiently as the degree of parallelism offered by modern multi-socket multi-core processor servers increases. Sharing data and work across concurrent queries, and NUMA-aware adaptive task scheduling and data placement, can significantly improve the scalability and performance of concurrent analytical workloads on modern servers.

Thesis Contributions

In this thesis, we analyze the impact of (a) sharing data and work across concurrent queries, and (b) adaptive NUMA-aware task scheduling and data placement on the performance of highly concurrent analytical workloads on multi-core servers. Our main activities are:

- We conduct a short survey of state-of-the-art run-time sharing techniques and categorize them into reactive and proactive sharing techniques.
- We integrate reactive and proactive sharing in the same system, by integrating the original research prototypes that introduce them: QPipe for reactive sharing, and the CJOIN operator for proactive sharing. We perform an experimental analysis to show how and when reactive and proactive sharing can improve performance.
- We integrate task scheduling in a prototype based on a commercial main-memory DBMS, namely SAP HANA. We perform an experimental evaluation of task scheduling for highly concurrent analytical workloads. As an extension, we also evaluate mixed workloads, and show that task scheduling, data freshness, and flexibility are significant factors that affect how mixed workloads perform and utilize resources.
- We implement and experimentally evaluate NUMA-aware task scheduling and data placement strategies under diverse workload characteristics.

We make the following findings:

- Reactive and proactive sharing techniques are orthogonal and can be combined to exploit the advantages of both.
- Inter-socket stealing of memory-intensive tasks can hurt overall performance.
- Unnecessary data partitioning across sockets can incur a performance overhead.

Our technical contributions are the following:

- We propose a pull-based model for reactive sharing that does not require a serialization point when sharing common intermediate results.
- We show how the task scheduler can avoid CPU underutilization by issuing more worker threads when tasks block, and how to avoid the overhead of excessive intra-query task parallelism by adapting task granularity using a concurrency hint that reflects the system's recent CPU utilization.
- We develop algorithms to adapt task scheduling and data placement to the workload at run-time, in order to balance utilization across sockets while avoiding the overhead of stealing memory-intensive tasks and unnecessary data partitioning.

Thesis Methodology

The methodology we follow is visualized in Figure 1.3, and is composed of the following steps:

1. *Survey run-time sharing techniques*
   Survey techniques for sharing data and work across concurrent queries at run-time, detailing the commonalities and differences [157] (see Chapter 2).
2. *Evaluate run-time sharing techniques*
   Identify when and how to use reactive and proactive sharing to eliminate redundant work and improve overall performance of the query mix [157, 159] (see Chapter 3).
3. *Employ and evaluate task scheduling*
   Employ and evaluate task scheduling for highly concurrent workloads in a main-memory DBMS, detailing the involved benefits and challenges [26, 158, 162] (see Chapter 4).

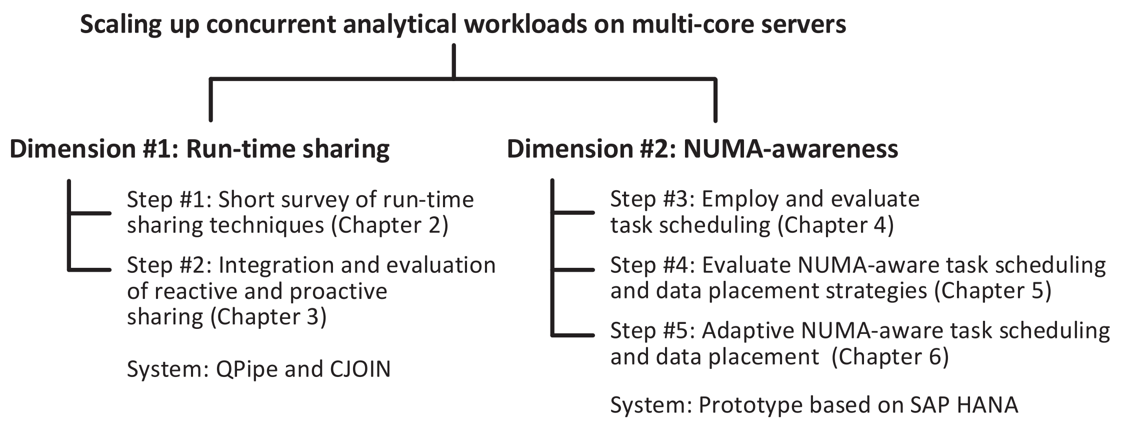

Scaling up concurrent analytical workloads on multi-core servers
Dimension #1: Run-time sharing
Step #1: Short survey of run-time sharing techniques (Chapter 2)
Step #2: Integration and evaluation of reactive and proactive sharing (Chapter 3)
System: QPipe and CJOIN
Dimension #2: NUMA-awareness
Step #3: Employ and evaluate task scheduling (Chapter 4)
Step #4: Evaluate NUMA-aware task scheduling and data placement strategies (Chapter 5)
Step #5: Adaptive NUMA-aware task scheduling and data placement (Chapter 6)
System: Prototype based on SAP HANA

Figure 1.3 – Thesis methodology

4. *Evaluate NUMA-aware task scheduling and data placement strategies*
   Analyze and evaluate the performance of concurrent analytics in a main-memory DBMS with various coordinated task scheduling and data placement strategies on multi-socket servers [161] (see Chapter 5).

5. *Adaptive NUMA-aware task scheduling and data placement*
   Adapt the data placement and task scheduling strategy in a main-memory DBMS to the workload at run-time in order to avoid unnecessary remote memory accesses and bandwidth bottlenecks [163] (see Chapter 6).

Outline

The rest of this thesis is organized as follows: Chapter 2 gives a necessary background on the topics discussed in this thesis, and mentions related work. Chapter 3 presents how and when to share data and work across concurrent queries at run-time. Chapter 4 presents how task scheduling can be used in the execution engine of a main-memory DBMS and explores its benefits and challenges. Chapter 5 identifies and evaluates NUMA-aware task scheduling and data placement strategies for main-memory analytical workloads. Chapter 6 presents how to adapt the task scheduling and data placement strategy to the workload at run-time. Finally, Chapter 7 concludes by summarizing this work, and presents further opportunities for extending this work.

2 Background and Related Work

In this thesis, we show how an execution engine can efficiently scale up analytics on modern multi-core servers by (a) sharing data and work across concurrent queries, and by (b) employing adaptive NUMA-aware data placement and task scheduling at run-time. In this chapter, we present related work in the literature, related to both run-time sharing and NUMA-aware analytical processing. Additionally, we give an overview of the necessary background for the topics discussed in this thesis.

We begin by explaining the main differences, advantages, and disadvantages of disk-based row-store DBMS versus main-memory column-store DBMS (see Section 2.1). We continue with a short survey of sharing techniques, focusing on run-time sharing techniques (see Section 2.2). Afterwards, we give an overview of task scheduling and related work (see Section 2.3). Finally, we discuss NUMA and related work about NUMA-awareness (see Section 2.4).

2.1 Disk-Based Row-Stores vs. Main-Memory Column-Stores

Conventional relational DBMS are based on the principles introduced by System R [35]. Data is stored on hard disk in row format: a record's attributes are placed contiguously on storage. The benefit of a *row-store*, i.e., a DBMS which stores data in row format, is that it can easily achieve high performance for record writes, and thus for OLTP workloads [181]. Examples include numerous popular DBMS following this architecture such as Microsoft SQL Server, IBM DB2, Oracle, MySQL, and PostgreSQL.

For OLAP workloads, disk-based DBMS are not ideal. Disk accesses are slow and can become a major performance bottleneck for analytics [205]. This is one of the reasons why sharing techniques, such as buffer pool management techniques, can have a large impact on performance (see Section 2.2). In our evaluation of run-time sharing techniques, we use a disk-based row-store, namely the QPipe execution engine [91] on top of the Shore-MT storage manager [99], and we are able to show the impact of sharing on the performance of disk accesses and of further query operators in the execution engine (see Chapter 3).

Nowadays, we witness a shift to main-memory DBMS. Memory storage capacities and bandwidth have been doubling roughly every three years, its price has been dropping by a factor of 10 every five years, and its latency is shorter than hard disks by several orders of magnitude [205]. Modern high-end NUMA servers can have terabytes of main memory. All these factors have made it possible to store large databases entirely in main memory. Furthermore when main memory is not sufficient, compression techniques can be used to fit more data [22, 72, 118, 205], or fit only the actively used hot part of the database in main memory [66, 74, 180].

Additionally, for the efficient evaluation of OLAP workloads, there is a shift towards column-stores. In a *column-store*, the values of all records for a single attribute are stored contiguously on storage. There are several advantages for analytical workloads. Queries typically reference only a few attributes, thus the DBMS needs to read only the required attributes for processing a given query [23, 181]. Also, vectorization techniques can be used during query processing. Vectorization involves fast processing of blocks of values of a column [195, 210], instead of the Volcano style with per-tuple iterators [23, 82]. Moreover, compression techniques, apart from saving memory space, can also decrease CPU overhead [22, 118, 197].

Examples of main-memory column-stores include SAP HANA [72], Oracle [63], IBM DB2 BLU [167], Microsoft SQL Server [111], HyPer [101]. For our NUMA-awareness analysis, we use a prototype based on SAP HANA, a commercial main-memory column-store. This allows us to show the significant impact of NUMA-aware data placement and task scheduling on the performance of highly concurrent analytics on modern main-memory column-store DBMS (see Chapter 4, Chapter 5, Chapter 6).

2.2 Sharing Techniques

In this section, we present a short survey on sharing techniques, focusing on run-time sharing techniques. We first mention a variety of ideas that have been proposed to exploit sharing in conventional query-centric systems, including buffer pool management techniques, materialized views, caching and multi-query optimization. Then, we proceed to more recent research prototypes that introduce techniques for run-time sharing across concurrent queries. In Table 2.1, we summarize the sharing methodologies used by conventional query-centric systems, and the research prototypes we examine.

Sharing in the I/O layer. By sharing data, we refer to techniques that coordinate and share the accesses of queries in the I/O layer. The typical query-centric DBMS incorporates a buffer pool and employs eviction policies [55, 100, 132, 151]. Queries, however, communicate with the buffer pool manager on a per-page basis, thus it is difficult to analyze their access patterns. Additionally, if multiple queries start scanning the same table at different times, scanned pages may not be reused.

Table 2.1 – Sharing methodologies employed by a query-centric model and the research prototypes we examine.

System	Conventional query-centric model	QPipe	CJOIN	DataPath	SharedDB
Sharing in the execution engine	Query Caching, Materialized Views, MQO	Reactive sharing	Proactive sharing (for joins of star queries)	Proactive sharing	Proactive sharing (with batched execution)
Sharing in the I/O layer	Buffer pool management techniques	Circular scan of each table	Circular scan of the fact table	Asynchronous linear scan of each disk	Circular scan of in-memory table partitions
Depends on a specific storage manager?	No	No	No	Special I/O subsystem (read-only requests)	Crescando (read and update requests)

For this reason, shared scans have been proposed. Circular scans [58, 60, 91] are a form of shared scans. They can handle numerous concurrent scan-heavy analytical queries as they reduce buffer pool contention, and they avoid unnecessary I/O accesses to the underlying storage devices. Furthermore, more elaborate shared scans can be developed for main-memory scans [164], and for servicing different fragments of the same table or different groups of queries depending on their speed [107, 211].

Shared scans can be used to handle concurrent updates as well. The Crescando [188] storage manager performs a circular scan over memory-resident table partitions, interleaving the reads and the updates of a batch of queries along the way. The scan first executes the update requests of the batch for a scanned tuple in their arrival order, and then the read requests.

Shared scans, however, are not immediately translated to a fast linear scan of a disk, e.g. if multiple scans are performed on the disk. The DataPath system [33], which uses a disk array as secondary storage, stores relations column-by-column by hashing pages to the disks at random. During execution, it reads pages from the disks asynchronously but sequentially, thus aggregating the throughput of a sequential scan on every disk of the array. The process is asynchronous from the execution engine. If the execution engine is busy, scanned pages are dropped and reproduced later when the disk's linear scan wraps around to the same point.

Sharing in the execution engine. By sharing work among queries, we refer to techniques that avoid redundant computations inside the execution engine. A conventional query-centric DBMS typically uses query caching and materialized views. Caching query results or intermediate results is used to reduce the response time of repeating queries [176]. Materialized views [34, 54, 169, 170] are pre-computed and cache common intermediate results that can be used by new queries. The selection of materialized views is done in an off-line manner by examining common sub-expressions in workloads that are known in advance. Both, however, do not exploit sharing opportunities among in-progress queries.

Multi-Query Optimization (MQO) techniques [171, 175] are an important step towards more sophisticated sharing methodologies. MQO detects and reuses common sub-expressions among queries. A global plan is built from these sub-expressions for all queries in the batch. The sub-expressions are evaluated only once during the execution phase. MQO makes the optimization phase more complex, but is suitable for long-running analytical queries, as the savings might more than offset this overhead. We note that the global plan of MQO is different from the global plan with shared operators in proactive sharing that we describe later. The former is used to evaluate common sub-expressions once, while the latter is used to enable shared operators for the whole query mix. There are two main disadvantages of classic MQO: (a) it operates on batches of queries only during the optimization phase, and (b) it depends on materializing shared intermediate results, at the expense of memory. This cost can be alleviated by using pipelining [62], which exploits the parallelization provided by multi-core processors. The query plan is divided into sub-plans and operators are evaluated in parallel.

Reactive and proactive run-time sharing techniques. Recently, more advanced forms of sharing have appeared for sharing data and work across concurrent queries at run-time. We categorize these state-of-the-art run-time sharing techniques into: (a) reactive sharing [91], and (b) proactive sharing [33, 47, 48, 77]. Both leverage forms of pipelined execution and sharing methodologies which bear some superficial similarities with MQO. These techniques, however, provide deeper and more dynamic forms of sharing at run-time.

Reactive sharing is based on *simultaneous pipelining (SP)*, a technique introduced in QPipe [91], an operator-centric execution engine, where each relational operator is encapsulated into a self-contained module called a stage. Each stage detects common sub-plans among concurrent queries, evaluates only one and pipelines the results to the rest when possible. Simultaneous pipelining is considered as reactive sharing, since it acts by using the sharing opportunities that inherently exist in the workload, e.g., common sub-plans.

Proactive sharing uses a *global query plan (GQP) with shared operators*, and is introduced in the CJOIN operator [47, 48]. A single shared operator is able to evaluate multiple concurrent queries. CJOIN uses a GQP, consisting of shared hash-join operators that evaluate the joins of multiple concurrent queries simultaneously. More recent research prototypes extend the logic to additional operators and to more general cases [33, 77]. A GQP can be considered as proactive sharing, since it analyzes the workload beforehand and composes shared operators synergistically in a global query plan.

Figure 2.1 illustrates how a query-centric model, shared scans, reactive sharing, and proactive sharing operate through a simple example of three concurrent queries which perform natural joins without selection predicates and are submitted at the same time. The last two queries have a common plan, which subsumes the plan of the first. Figure 2.1a shows how a traditional query-centric model evaluates the queries independently, missing any sharing opportunities. Figure 2.1b depicts how shared scans can share the scan of a table, reducing contention for CPU resources, the buffer pool and the underlying I/O device. Figure 2.1c shows how reactive sharing identifies common sub-plans, evaluates only $Q2$, reuses intermediate results for $Q1$ and final results for $Q3$. Figure 2.1d illustrates how proactive sharing builds a GQP with shared

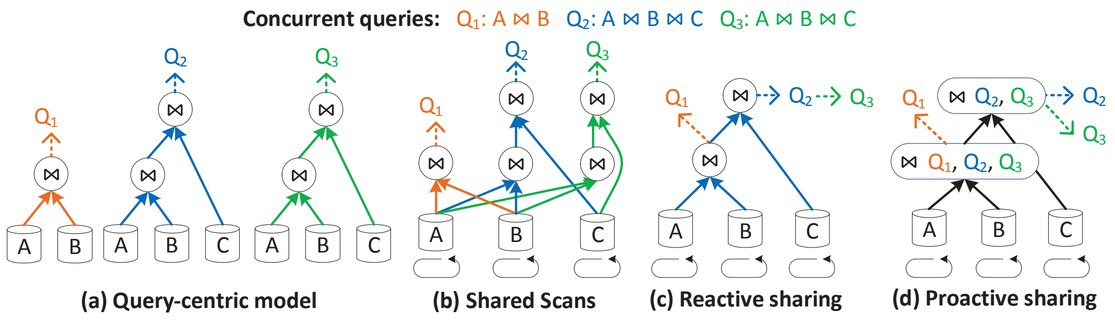


Figure 2.1 – Evaluation of three concurrent queries using (a) a query-centric model, (b) shared scans, (c) reactive sharing, and (d) proactive sharing.

join operators that evaluate all three queries. We note that shared scans are used with both reactive and proactive run-time sharing techniques.

In the next sections, we delve into the details of reactive sharing first and then of proactive sharing. We also give an overview of the research prototypes that introduce them. We use these prototypes to integrate and evaluate reactive and proactive sharing in Chapter 3.

2.2.1 Reactive Sharing

Reactive sharing identifies identical sub-plans among concurrent queries at run-time, evaluates only one and pipelines the results to the rest simultaneously [91]. Figure 2.2a depicts an example of two queries that share a common sub-plan below the join operator (along with any selection and join predicates), but have a different aggregation operator above the join. With reactive sharing, only one of them is evaluated, and the results are pipelined to the other aggregation operator.

Fully sharing common sub-plans is possible if the queries arrive at the same time. Otherwise, sharing opportunities may be restricted. The amount of results that a newly submitted $Q2$ can reuse from the *pivot operator*, i.e., the top operator of the common sub-plan, of the in-progress $Q1$, depends on the type of the pivot operator and on the arrival time of $Q2$ during $Q1$'s execution. This relation is expressed as a *window of opportunity* (WoP) for each relational operator [91]. Figure 2.2b depicts two common WoP, a step and a linear WoP.

A step WoP expresses that $Q2$ can reuse the full results of $Q1$ if it arrives before the first output tuple of the pivot operator. If $Q2$ arrives afterwards, it cannot share the pivot operator and needs to re-issue the computation independently. Joins and aggregations have a step WoP. The point where the step WoP closes depends on when the first result is computed by the operator. For example, a scalar aggregation has a "full" WoP, as the result is produced at the end of its execution [91]. A linear WoP signifies that $Q2$ can reuse the results of $Q1$ from the moment it arrives up until the pivot operator finishes. Then, $Q2$ needs to re-issue the operation in order to compute the results that it missed before it arrived. Sorts and table scans have a linear WoP.

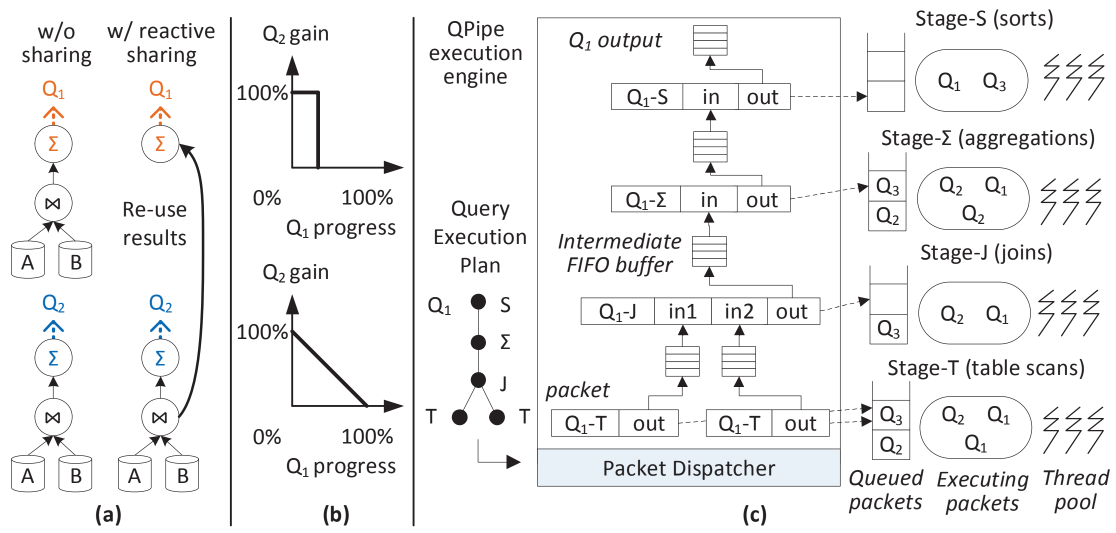


Figure 2.2 – (a) Example of reactive sharing with two queries having a common sub-plan below the join operator. (b) A step and a linear window of opportunity [91]. (c) Conceptual design of QPipe, with the packets of an example of a query plan dispatched to the relevant stages.

In fact, the linear WoP of the table scan operator is translated into a circular scan of each table: when $Q2$ arrives during $Q1$'s scan of a table, $Q2$ reuses the scanned pages from $Q1$. When $Q1$ finishes, $Q2$ starts scanning from the beginning of the table to scan the pages that it missed.

We note that caching query results or intermediate results [176] is a technique similar to reactive sharing which shares common intermediate results at run-time. Actually they are orthogonal and complementary. Reactive sharing can improve a query result cache by avoiding identical sub-plans at run-time, with no previous entries in the result cache [91].

2.2.2 The QPipe Execution Engine

QPipe [91] is a relational execution engine that supports reactive sharing. QPipe is based on the paradigms of staged databases [90]. The conceptual design is depicted in Figure 2.2c. Each relational operator is encapsulated into a self-contained module called a *stage*. Each stage has a queue for work requests and employs a thread pool for processing the requests.

An incoming query execution plan is converted into a series of inter-dependent *packets*. The packet dispatcher then dispatches each packet to the relevant stage for evaluation. Data flow between packets is implemented through FIFO (first-in, first-out) buffers and page-based exchange, following a push-only model with pipelined execution. The buffers also regulate differently-paced actors: a parent packet may need to wait for incoming pages of a child and, conversely, a child packet may wait for a parent packet to consume its pages.

This design allows each stage to monitor only its packets for detecting sharing opportunities efficiently. If it finds an identical packet, and their inter-arrival delay is inside the WoP of the pivot operator, it attaches the new packet (*satellite* packet) to it (*host* packet). While it evaluates the host packet, QPipe copies the results of the host packet to the output FIFO buffer of the satellite packet.

2.2.3 Proactive Sharing

Reactive sharing is limited to common sub-plans. If two queries have similar sub-plans but with different selection predicates for the involved tables, reactive sharing is not able to share them. Nevertheless, the two queries still share a similar plan that exposes sharing opportunities. It is possible to employ *shared operators*, where a single shared operator can evaluate both queries simultaneously. The basic technique for enabling them is sharing tuples among queries and correlating each tuple to the queries, e.g., by annotating tuples with a *bitmap* whose bits signify the queries that the tuple is relevant to.

This technique facilitates overall sharing. Tuples can be shared among active queries, as interested queries read the same tuples and skip those that are not relevant to them. Also, it allows for shared operators that can share work among queries with similar sub-plans but possibly different predicates. A shared operator receives input tuples with their bitmaps already modified accordingly by preceding shared operators. Then, it examines a tuple's relevant queries and evaluates its unit of work considering their possibly different predicates.

The simplest shared operator is a shared selection, that can evaluate multiple queries that select tuples from the same relation. For each received tuple, the shared selection toggles the bits of its attached bitmap according to the selection predicates of the queries. A hash-join can also be easily shared by queries that share the same equi-join predicate (more relaxed requirements are possible [33]). Figure 2.3 shows a conceptual example of how a single shared hash-join is able to evaluate two queries. It starts with the build phase by receiving tuples from the shared selection operator of the inner relation. Then, the probe phase begins by receiving tuples from the shared selection operator of the outer relation. The hash-join proceeds normally, by additionally performing a bitwise AND operation between the bitmaps of the joined tuples in order to preserve the relevance of the output tuples to the active queries.

The most significant advantage is that a single shared operator can evaluate many similar queries. For example, a shared hash-join can evaluate numerous queries having the same equi-join predicate, and possibly different selection predicates. In the worst case, the union of

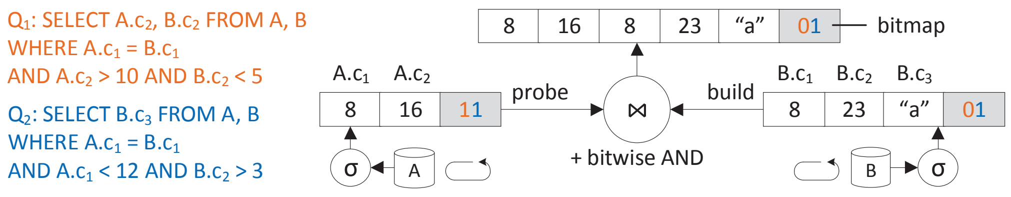

Q1: SELECT A.c2, B.c2 FROM A, B
WHERE A.c1 = B.c1
AND A.c2 > 10 AND B.c2 < 5
Q2: SELECT B.c3 FROM A, B
WHERE A.c1 = B.c1
AND A.c1 < 12 AND B.c2 > 3
8 | 16 | 8 | 23 | "a" | 01
bitmap
A.c1 A.c2
8 | 16 | 11
probe
⋈
+ bitwise AND
build
B.c1 B.c2 B.c3
8 | 23 | "a" | 01
σ
A
B
σ

Figure 2.3 – Example of shared selection and hash-join operators.

the selection predicates may force it to join the whole two relations. The disadvantage of a shared operator in comparison to a query-centric one is that it entails increased bookkeeping. For example, a shared hash-join maintains a hash table for the union of the tuples of the inner relation selected by all queries, and performs bitwise operations between the bitmaps of the joined tuples. For low concurrency, as shown by our experiments in Chapter 3, query-centric operators outperform shared operators. A similar trade-off is found for the specific case of shared aggregations on multi-core processors [56].

By using shared scans and shared operators, a global query plan (GQP) can be built for evaluating all concurrent queries. GQP are introduced by CJOIN [47, 48], an operator based on shared selections and shared hash-joins for evaluating the joins of star queries [102]. GQP are advanced by the DataPath system [33] for more general schemas, by tackling the issues of routing and optimizing the GQP for a newly incoming query. DataPath also adds support for a shared aggregate operator, that calculates a running sum for each group and query.

Both CJOIN and DataPath handle new queries immediately when they arrive. This is feasible due to the nature of the supported shared operators: selections, hash-joins and aggregates. Some operators, however, cannot be easily shared. For example, a sort operator cannot easily handle new queries that select more tuples than the ones being sorted [33]. To overcome this limitation, SharedDB [77] batches queries for every shared operator. Batched execution is also in line with Crescando, the underlying storage manager, that also uses batching for performing shared scans. Batching allows standard algorithms to be easily extended to support shared operators, as they work on a fixed set of tuples and queries. SharedDB supports shared sorts and various shared join algorithms, not being restricted only to equi-joins. Later work from the same team focuses on the special case of shared hash-joins with batched execution, advances the parallelism features of shared hash-joins, and explores additional benefits of batched execution, such as sharing the build phase of a hash-join and avoiding overhead in repeatedly updating hash tables for incoming new queries [128]. Nevertheless, batched execution has the following drawbacks, which can be disadvantageous for OLAP workloads: a new query may suffer increased latency, and the latency of a batch is dominated by the longest-running query. One other significant reason for batched execution is that SharedDB aims to support OLTP workloads in addition to OLAP workloads. This is enabled by Crescando which interleaves update and read requests along a main-memory shared scan, and by pre-computing the GQP. SharedDB assumes that the types of possible queries are given in advance in the form of prepared statements in order to pre-compute the GQP.

In the realm of optimization of proactive sharing, each research prototype suggests alternative optimization approaches that fit their implementation. CJOIN proposes an online approach for re-ordering the shared hash-joins of its GQP according to the selectivities of the query mix [47, 48]. DataPath proposes an A*-style search to integrate a new query into the ongoing GQP [33]. SharedDB authors, due to batched execution, propose a heuristic algorithm to generate a GQP by considering the whole query mix and not just integrating a new query [78].

2.2.4 The CJOIN Operator

For our analysis, we use the CJOIN operator [47, 48] which introduced GQP for the case of star schemas. Without loss of generality, we restrict our evaluation in Chapter 3 to star schemas, and correlate our observations to more general schemas (used, e.g., by DataPath [33] or SharedDB [77]).

Star schemas are common for organizing data in relational data warehouses. They allow for numerous performance enhancements [102]. A star schema consists of a large *fact table*, that stores the measured information, and is linked through foreign-key constraints to smaller *dimension tables*. A *star query* is an analytical query over a star schema. It typically joins the fact table with several dimension tables and performs operations such as aggregations or sorts.

CJOIN evaluates the joins of all concurrent star queries, using a GQP with shared scans, selections and hash-joins. Figure 2.4 shows the GQP that CJOIN evaluates for two star queries. CJOIN adapts the GQP with every new star query. If a new query references already existing dimension tables, the existing GQP can evaluate it. If a new query joins the fact table with a new dimension table, the GQP is extended with a new shared selection and hash-join. Due to the semantics of star schemas, the directed acyclic graph of the GQP takes the form of a chain.

CJOIN exploits this form to facilitate the evaluation of the GQP. It materializes the small dimension tables and stores in-memory the selected dimension tuples in the hash tables of the corresponding shared hash-joins. Practically, for each dimension table, it groups the shared scan, selection and hash-join operators into an entity called *filter*. When a new star query is admitted, CJOIN pauses, adds newly referenced filters, updates already existing filters, augments the bitmaps of dimension tuples according to the selection predicates of the new star query, and then continues. Parts of the admission phase, such as the scan of the involved dimension tables, can be done asynchronously while CJOIN is running [48].

Consequently, CJOIN can evaluate the GQP using a single pipeline: the *preprocessor* uses a circular scan of the fact table, and flows fact tuples through the pipeline. The data flow in the pipeline is regulated by intermediate buffers, similar to QPipe. The filters in-between are

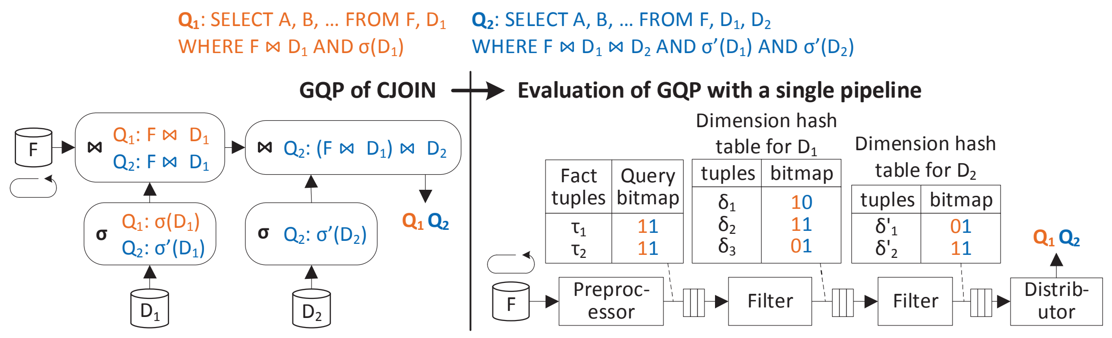


Figure 2.4 – The CJOIN operator uses proactive sharing by evaluating a GQP for star queries.

actually the shared hash-joins that join the fact tuples with the corresponding dimension tuples and additionally perform a bitwise AND between their bitmaps. At the end of the pipeline, the *distributor* examines the bitmaps of the joined tuples and forwards them to the relevant queries. For every new query, the preprocessor admits it, marks its point of entry on the circular scan of the fact table and signifies its completion when it wraps around to its point of entry on the circular scan.

This concludes our short survey on run-time sharing techniques. Chapter 3 continues the topic of run-time sharing techniques, by integrating and evaluating them. Here, we continue with related work on task scheduling, NUMA, and NUMA-awareness.

2.3 Task Scheduling

Task scheduling uses worker threads to process all operations for the entire application lifetime. Tasks, encapsulating operations, are stored in task pools, and a number of worker threads, typically a worker thread per hardware thread, are employed by the task scheduler to process the tasks. A worker thread continuously takes tasks from a task pool and executes them. Next, we present related work about task scheduling. In later chapters, we develop and employ a task scheduler for main-memory workloads (see Chapter 4), and then focus on NUMA-aware task scheduling (see Chapter 5) and adaptive inter-socket task stealing (see Chapter 6).

Task scheduling for parallel programs and parallel systems is a broad field of research. Early related work focuses on static scheduling [83, 104], which is typically done at compile time and assumes that basic information about tasks (such as processing times, dependencies, synchronization, and communication costs) and the target machine environment are known in advance. Given perfect information, a static scheduling algorithm attempts to produce the optimal assignment of tasks to processors, that ideally balances their loads and minimizes scheduling overheads and memory referencing delays [88]. Perfect information, however, is hard to obtain for modern shared-memory multi-core servers and modern applications where tasks may be generated dynamically and at a fast pace. For example, queries in a DBMS arrive dynamically, and information about them can only be estimated, often with high relative errors. More recent related work focuses on dynamic scheduling, which is done on-the-fly at run-time [104, 153, 207]. Dynamic task scheduling does not require information about the submitted tasks a priori and provides automatic load-balancing and improved portability between different hardware architectures [131]. Dynamic task scheduling may also use run-time measurement to re-adapt scheduling decisions [137, 145, 207] for better data locality [207] or NUMA-awareness [41, 149].

The common design of recent dynamic task schedulers involves task pools where tasks are submitted dynamically at run-time, and the scheduler employs a set of threads to work on the tasks [94]. There are two main categories of task schedulers [68]: breadth-first schedulers [140] and work-first schedulers [43] with various task-stealing techniques [24, 134, 142, 149] for load-balancing and improved predictability in real-time applications [131]. In breadth-first schedulers, when a task is created, it is placed in the task pools and the parent task continues execution. In work-first schedulers, the thread of the parent task switches to execute child tasks, for potentially better data locality [43]. The task scheduler that we design and employ in this thesis uses dynamic task scheduling, and has a mixture of both approaches: when a task generates a group of new tasks, the parent thread takes upon one of the new tasks, following the work-first approach, while the rest of the tasks are dispatched to the task pools, following the breadth-first approach, for load-balancing (see Chapter 4).

Hoffmann et al. [94] provide a survey on different task pool implementations. Distributed task pools with stealing achieve best performance, as they minimize synchronization overheads and stealing amends load-balancing issues. We follow a similar approach for our task scheduler. Johnson et al. [97] decouple contention management from scheduling and load management, to combine the advantages of spin and blocking locks. Our task scheduler is not concerned with how locks are used, but uses the information about blocked tasks to dynamically adjust its concurrency level (see Chapter 4).

In highly concurrent analytical workloads with a large number of partitionable operations issuing tasks, granularity of tasks plays an important role in communication, synchronization, and scheduling costs [3, 52, 57, 122, 134]. Our experimental results corroborate these observations for main-memory DBMS. A recent analysis models and evaluates task schedulers, and shows that task granularity also affects cache locality [204]. Recent task scheduling frameworks such as OpenMP [11, 37] and Intel Thread Building Blocks [12] regulate task granularity by requiring the developer to express parallelism in a higher-level abstract manner and use this information. For our task scheduler, we assume that there is no central mechanism for data parallelism, and that partitionable operations define their task granularity independently. This is the common case for DBMS, where optimizing the granularity of a single partitionable operation alone involves considerable research effort (see [42], for example, for partitioning in hash-joins). Our solution it to give a concurrency hint to task creators, reflecting recent CPU availability, which they can use as a limit when adjusting the task granularity of their partitionable operations (see Chapter 4).

In this thesis, we also evaluate task scheduling for mixed OLTP and OLAP main-memory workloads as well (see Chapter 4). For mixed workloads, we demonstrate the need for task prioritization of the transactional workload over the dominating concurrent analytical workload. Further related work identifies that, apart from task prioritization, the task granularity of analytical queries also plays a significant role in this effect [201].

In regard to multi-socket servers, task scheduling has been prominent in NUMA-related work. The reason is that task scheduling can effortlessly reflect the underlying topology of a multi-socket server. Next, we discuss related work to NUMA, and NUMA-aware task scheduling and data placement. In Chapter 5, we modify our initial task scheduler of Chapter 4 for multi-socket servers, and present an analysis of NUMA-aware task scheduling and data placement strategies. In Chapter 6, we proceed to adapt task scheduling and data placement to the workload on multi-socket servers.

2.4 Non-Uniform Memory Access (NUMA)

In this section we give more details about NUMA and related work in NUMA-awareness. Section 2.4.1 gives details and examples of multi-socket servers with a NUMA architecture. It also gives an overview of the available facilities of the OS that the DBMS can use to achieve NUMA-aware task scheduling and data placement. Section 2.4.2 discusses related work for NUMA-aware task scheduling and data placement.

2.4.1 Multi-Socket Servers and NUMA-Awareness

As introduced in Section 1.1, processor vendors are scaling up hardware by interconnecting sockets of multi-core processors, each with its own memory [27]. Memory is decentralized, forming a non-uniform memory access (NUMA) architecture. Typically, multiple processor cores are attached to one socket, and share a local memory bus, controller, and last-level cache. Then, multiple sockets are interconnected with a communication network in order to enable a processor core to access remote memory of another socket. Furthermore, for the typical case of shared-memory systems, a cache-coherency protocol is used to keep multiple references to the same memory address consistent. Both the inter-processor communication protocol and the cache-coherence protocol are specific to each system and vendor. In this thesis, we use the term *multi-socket server* to refer to a shared-memory server with multiple sockets of multi-core processors and a cache-coherent NUMA (ccNUMA) architecture [112, 123].

Note on terminology. There is a difference between what is referred to as a socket and as a NUMA node [106]. A *socket* in a server is a slot that allows for placing and replacing a central processing unit (CPU). A *NUMA node*, according to the ACPI specification [61], is a collection of hardware resources that may contain processors, memory, etc. At a minimum, a NUMA node must have a an interface to the interconnect between nodes [61]. With respect to performance, this means that accesses to memory within a NUMA node have roughly the same latency and do not require a hop through an interconnect [10]. Thus, a socket does not necessarily correspond to a NUMA node. For example, the IBM MAX5 memory expansion unit [192] consists of a NUMA node without processors. Another example is the Cluster-on-Die mode, supported by the Intel Xeon E5-2600 v3 processor family (code named Haswell-EP), which logically divides a processor into two NUMA nodes [135].

In this thesis, we handle the typical case of multi-socket servers where each socket has dedicated memory and corresponds to a NUMA node [10, 106]. For this reason, we use the terms socket and NUMA node interchangeably. Next, we visualize two examples of multi-socket

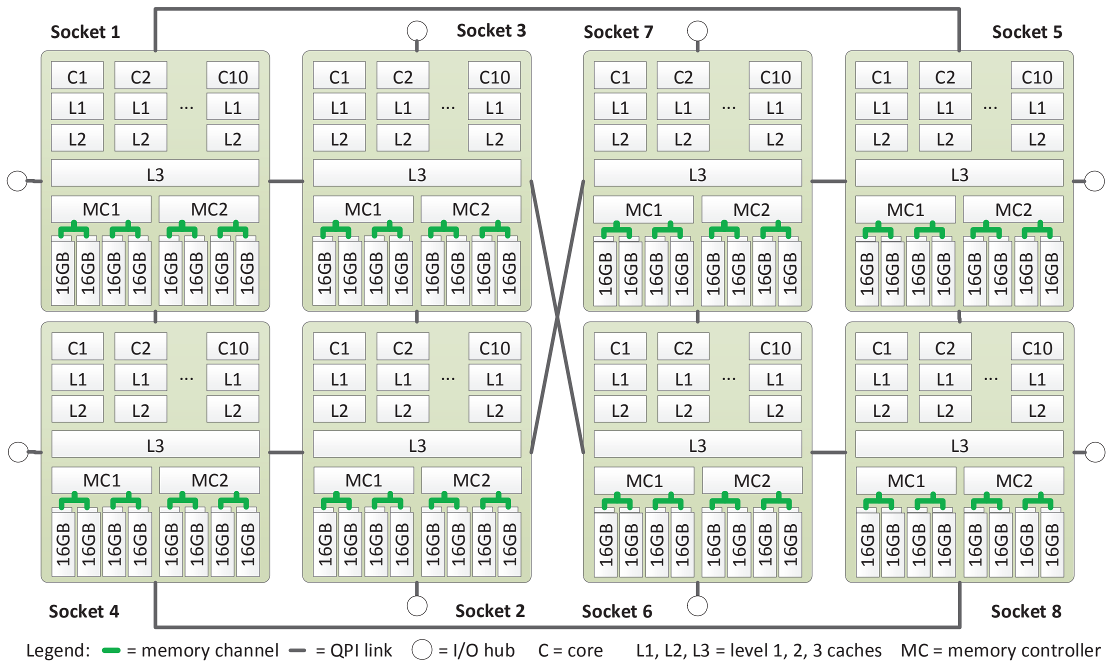

Socket 1
Socket 3
Socket 7
Socket 5
Socket 4
Socket 2
Socket 6
Socket 8
C1
C2
C10
L1
L2
L3
MC1
MC2
16GB
Legend: = memory channel = QPI link = I/O hub C = core L1, L2, L3 = level 1, 2, 3 caches MC = memory controller

Figure 2.5 – An 8-socket server with Intel Xeon E7-8870 processors.

servers, of different processor generations and NUMA topologies, that we use in our experiments (see Chapters 5-6). Our purpose is to show how multi-socket servers are designed, and pinpoint the similarities and differences across their performance characteristics.

Example with Intel Westmere-EX processors. Figure 2.5 depicts an 8-socket multi-core server (consisting of two IBM x3950 X5 boxes [192]). Each socket is comprised of a ten-core Intel Xeon E7-8870 processor (code named Westmere-EX) at 2.40 GHz, with hyper-threading possible (2 SMT threads per core). Each core has a 32 KB L1-I/D caches and a 256 KB L2 cache. Each processor has a 30 MB L3 cache, shared by all its cores. Each socket has two memory controllers (MC). Each MC supports two Intel SMI channels [2]. Each SMI channel supports two memory channels, with up to 2 DDR3 DIMM attached on each channel. Our configuration has one 16 GB DIMM per memory channel. Each MC operates in "lockstep" mode [2], thus each SMI channel is visualized as one memory channel in Figure 2.5. The server has a total of 160 H/W threads, and 1 TB RAM. The inter-processor network is comprised of Intel QPI (QuickPath Interconnect) links [1].

Example with Intel Ivybridge-EX processors. Figure 2.6 depicts a 4-socket multi-core server. Each socket is comprised of a 15-core Intel Xeon E7-4880 v2 processor (code named Ivybridge-EX) at 2.50 GHz, with hyper-threading possible (2 SMT threads per core). Each core has a 32 KB L1-I/D caches and a 256 KB L2 cache. Each processor has a 37.5 MB L3 cache, shared by all its cores. Each socket has two memory controllers (MC). Each MC supports two Intel SMI2 channels [6]. Each SMI2 supports two memory channels, with up to 3 DDR3 DIMM attached on each channel. Our configuration has one 16 GB DIMM per memory channel. Each

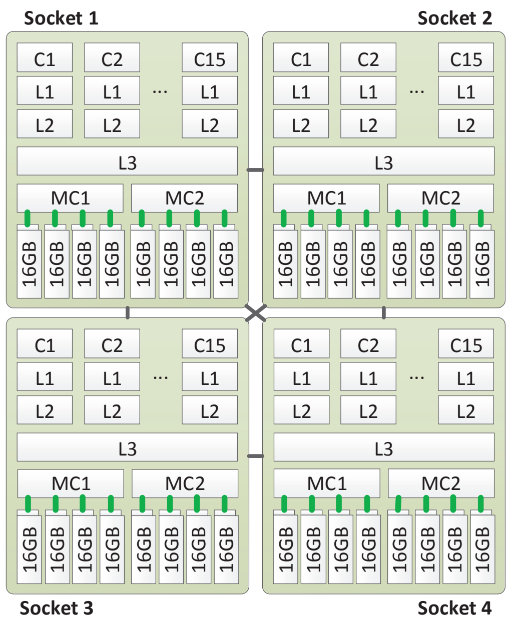


Figure 2.6 – A 4-socket server with Intel Xeon E7-4880 v2 processors.

MC operates in "independent" mode [6, 194], thus each SMI2 channel is visualized as two memory channels in Figure 2.6. The server has a total of 120 H/W threads, and 512 GB RAM. The inter-processor network is comprised of Intel QPI links [1]. The Ivybridge-EX processor generation is more recent than the Westmere-EX processor generation.

Performance characteristics of multi-socket servers. The two examples above show that multi-sockets servers can vary by multiple factors, such as the number of sockets, the topology, the maximum number of hops from one socket to another, the processor generation, etc. These factors affect the performance characteristics of each server. In Table 2.2, we show the local and inter-socket latencies and peak memory bandwidths for the multi-socket servers that we have access to and use throughout our NUMA analysis in this thesis. The measurements are taken with Intel Memory Latency Checker [190].

Table 2.2 – Performance characteristics of the multi-socket servers that we use.

Server / Statistic	(1) 8x10-core Intel Xeon E7-8870 (Westmere-EX) at 2.40GHz	(2) 4x15-core Intel Xeon E7-4880v2 (Ivybridge-EX) at 2.50GHz	(3) 8x15-core Intel Xeon E7-8880v2 (Ivybridge-EX) at 2.50GHz	(4) 32x15-core Intel Xeon E7-8890v2 (Ivybridge-EX) at 2.80GHz	(5) 32x18-core Intel Xeon E7-8890v3 (Haswell-EX) at 2.50GHz
Memory per socket	128 GB	128 GB	128 GB	768 GB	512 GB
Local latency	163 ns	108 ns	110 ns	112 ns	120 ns
1 hop latency	197 ns	170 ns	320 ns	193 ns	320 ns
Max hops latency	245 ns	170 ns	390 ns	500 ns	590 ns
Local B/W	20.6 GB/s	70 GB/s	70 GB/s	47.5 GB/s	45.5 GB/s
1 hop B/W	11 GB/s	12.5 GB/s	10.5 GB/s	11.8 GB/s	15 GB/s
Max hops B/W	4.9 GB/s	12.5 GB/s	9.5 GB/s	9.8 GB/s	7.3 GB/s
Total local B/W	103 GB/s	280 GB/s	560 GB/s	1520 GB/s	1363 GB/s

The first and second servers correspond to the previous examples of Figure 2.5 and Figure 2.6. The third server is an 8-socket server, with a topology similar to the first server. The fourth server is a rack-scale SGI UV 300 system, and the fifth server is a rack-scale SGI UV 300H system, both having 32 sockets [21]. The fifth server has a more recent processor generation. The SGI servers have specialized interconnects to scale up to 32 sockets [21, 112].

The performance characteristics of the different multi-socket servers show several interesting facts about NUMA architectures. First, accesses to remote memory are slower than local memory. On the 32-socket servers, the max hop latency is around 5x slower than the local latency. Second, interconnects typically have a much lower bandwidth than that of a socket. On the third server, the bandwidth of an interconnect is up to around 7x lower than the bandwidth of a socket. Finally, the cache-coherence protocol is also important. On the older Westmere-EX processor generation of the first server, the broadcast-based snooping cache-coherence protocol utilizes the interconnects even for local memory accesses [1]. This is why the total local bandwidth of the first server is not the aggregated local bandwidth of each socket. The more recent processor generations use directory-based cache coherence.

NUMA-awareness. As is obvious by the performance characteristics, NUMA introduces new considerations for software performance in comparison to uniform memory access architectures. These are pinpointed by Blagodurov et al. [41]: (a) accesses to remote memory are slower than local memory, (b) the bandwidth of a socket can be separately saturated, and (c) the bandwidth of an interconnect can be separately saturated. Due to the lack of knowledge about inter-socket routing or the cache coherence, a NUMA-aware application attempts to solve the above challenges in a simple way: optimizing for local memory accesses instead of remote accesses, and avoiding unnecessary centralized bandwidth bottlenecks of either sockets or interconnects.

Several of these considerations have been revisited in further related work. David et al. [65] explore the challenges that non-uniform hardware entails for synchronization scalability, and conclude that in order to be able to scale, synchronization should better be confined to a single uniform socket. Albutiu et al. [30] explore similar considerations for NUMA-aware DBMS, and offer a new guideline: the processor prefetcher can hide the slower latency of sequential remote accesses. This guideline, however, should not be abused. As we show in this thesis, multiple concurrent memory-intensive tasks, e.g., scans, should not be stolen across sockets, as they can hurt overall performance (see Chapter 5). We propose an adaptive technique that allows stealing based on the memory intensity of the tasks (see Chapter 6).

To sum up, a DBMS needs to become NUMA-aware and handle both the scheduling of its operations and queries onto sockets, and the placement of its data structures across sockets. Next, we describe the facilities available to the DBMS by the OS to achieve NUMA-aware scheduling and data placement.

NUMA-aware task scheduling. A NUMA-aware task scheduler uses a pool of worker threads per socket. Each pool's worker threads are *bound*, e.g., by using Linux's `sched_setaffinity` system call, to the cores of the corresponding socket [103, 117]. This guarantees worker threads always locally access data placed on the socket. A task is queued to a task pool associated with the socket where the data processed by the task is placed. We use a similar NUMA-aware task scheduler in our NUMA analysis (see Chapter 5).

NUMA-aware data placement. The OS organizes physical memory into fixed-sized (typically 4 KB) pages [106]. When an application requests memory, the OS allocates new virtual memory pages. Application libraries are responsible for organizing smaller allocations. Typically, virtual memory is not immediately backed by physical memory. On the first page fault, the OS actually allocates physical memory for that page. In Linux, the default placement strategy for the page is the *first-touch policy* [106]: on the first page fault, the OS allocates physical memory from the local socket (unless it is exhausted). Another moderate placement strategy is to use *interleaving* [106], distributing pages in a round-robin fashion across all sockets. This avoids worst-case performance scenarios with unnecessary centralized bandwidth bottlenecks, and averages memory latencies. It involves, however, a lot of remote memory accesses. For stronger control, additional facilities are provided by the OS. In Linux, e.g., `move_pages` can be used to query and move already touched virtual memory. For best performance, the DBMS should use these OS facilities to fully control and track the physical location of its virtual memory. We use these facilities in our NUMA analysis (see Chapter 5).

By coordinating scheduling and data placement, the DBMS can prefer local memory accesses, avoid unnecessary bandwidth bottlenecks, and become NUMA-aware. Next, we look at related work in NUMA-aware task scheduling and data placement. In Chapter 5, we evaluate various strategies for NUMA-aware task scheduling and data placement. In Chapter 6, we adapt task scheduling and data placement to the workload at run-time.

2.4.2 Related Work in NUMA-Aware Task Scheduling and Data Placement

In the literature, there has been a recent wave of NUMA-aware related work. Next, we categorize related work in the database community into NUMA-aware static solutions, adaptive solutions, black-box solutions, and standalone operators. Afterwards, we mention prominent related work in the fields of co-scheduling, operating systems, and distributed systems. In all cases, we explain the main differences and similarities to this thesis.

Static solutions. With respect to data placement, most DBMS not mentioning advanced NUMA optimizations indirectly rely on the static first-touch policy for data placement, e.g., Vectorwise [209], Microsoft SQL Server's column-store [111], or IBM DB2 BLU [167]. Oracle's distributed manager decides the NUMA location of columnar data when the topology changes [138], but not when the workload changes. HyPer [117] chunks all data, and statically distributes them uniformly over the sockets. In this thesis, we show that unnecessary partitioning involves an overhead for performance (see Chapter 5). We show that the DBMS should adapt the data placement to the workload at run-time (see Chapter 6).

With respect to task scheduling, most DBMS allow inter-socket task stealing. IBM DB2 BLU processes data in chunks so that they fit in the CPU cache [167]. Each worker thread processes one chunk at a time and can steal chunks. In HyPer [117], each worker thread processes local data chunks through a whole pipeline of operators, and inter-socket stealing is enabled. In a recent thesis describing how to parallelize query plans in Vectorwise with task scheduling [86], inter-socket stealing is allowed based on task priorities and the contention of sockets. In this thesis, we show that inter-socket stealing of memory-intensive tasks can hurt overall performance (see Chapter 5). We show that the DBMS should adapt inter-socket stealing to the workload at run-time, and dynamically disallow the stealing of memory-intensive tasks (see Chapter 6).

Adaptive solutions. Two state-of-the-art research prototypes use an adaptive NUMA-aware solution for data placement: ERIS [103] and ATraPos [155]. ERIS is a storage manager that employs adaptive repartitioning. ERIS uses a worker per core, which processes tasks targeting a particular data partition. While ERIS targets storage operations, in this thesis we target analytical workloads consisting of multiple operators (see Chapter 5). We show, in addition, how partitioning involves an overhead, and can be avoided altogether depending on the workload. ATraPos uses dynamic repartitioning for OLTP workloads, to avoid transactions crossing partitions and avoid inter-partition synchronization [155]. ATraPos's data-oriented execution model uses a worker per partition, similar to ERIS. While ATraPos optimizes for the latency of transactions, in this thesis we focus on optimizing the throughput of analytical workloads by balancing socket utilization at run-time (see Chapter 6). Furthermore, we note that in contrast to HyPer, ERIS, and ATraPos, we analyze data placement strategies for dictionary-encoded columns (see Section 2.5).

Black-box solutions. In the systems community, the significance of data placement and scheduling on servers with a NUMA architecture has been studied since the early 90's. Typically, black-box solutions are proposed that track applications' accesses to memory pages to predict applications' behaviors and dynamically use page migration, replication or caching to improve the local memory accesses of applications. Black-box solutions have been suggested that can exploit the hardware advantages of ccNUMA and COMA (Cache-Only Memory Architecture) designs [70, 147, 179, 206], or that can track memory accesses by instrumenting the OS kernel or the applications [44, 45, 109, 110] or by using hardware counters [129, 130, 184, 185, 189], to support page migration and/or replication. Certain black-box solutions consider some application knowledge by leveraging parallel and iterative frameworks such as OpenMP [127, 146, 149, 183]. More recent examples of black-box solutions focus on multi-core multi-socket servers, handle thread migration and consider bandwidth bottlenecks as well, such as DINO [41] and Carrefour [64]. Black-box solutions, however, do not use application knowledge and attempt to predict the behavior of applications by tracking memory accesses. Without application knowledge, results for a complex applications such as DBMS, whose behavior can be changing, are sometimes sub-optimal [41, 199]. Giceva et al. [79] use application knowledge and employ a DBMS-focused black-box static approach to characterize and group the shared operators of a predefined global query plan, and place them on a NUMA server with the main aim of improving overall energy efficiency. In this thesis, we also employ a DBMS-focused black-box approach, but geared towards adapting data placement and task scheduling at run-time (see Chapter 6).

Standalone operators. There is also related work on NUMA-aware standalone operators. Albutiu et al. [30] show that prefetching can hide the latency of sequential remote accesses, constructing a competitive sort-merge join. Hash-joins, however, are shown to be superior [38, 108]. Yinan et al. [120] optimize data shuffling on a fully-interconnected NUMA topology. Most related work, however, optimize for low concurrency with a static data placement using all sockets of the server. In this thesis, we do not focus on optimizing single operators on multi-socket servers, but on improving the performance of highly concurrent workloads with multiple NUMA-aware analytical operators by adapting task scheduling and data placement to the workload at run-time (see Chapter 6).

Co-scheduling. Co-scheduling techniques attempt to optimize performance by considering multiple resources, such as CPU utilization, cache efficiency, memory buffers, disk accesses, etc [75, 114]. NUMA-awareness is a recent dimension that has not been extensively considered in co-scheduling techniques. In this thesis, we focus on NUMA-aware scheduling, by preferring local memory accesses and avoiding unnecessary bandwidth bottlenecks on sockets or interconnects. Our contributions can help to integrate NUMA-awareness as a significant additional dimension in further co-scheduling techniques.

Operating systems. In the OS community, there are proposals for supporting the performance requirements of high performance applications, such as DBMS, through the notion of multikernel design, i.e., running customized light-weight kernels instead or alongside full-weight kernels such as Linux [39, 76]. Giceva et al. suggest that light-weight kernels for DBMS should support, among other functionality, NUMA-aware task scheduling and memory management [80]. Most mainstream OS, however, such as Linux that we use in this thesis, do not follow the multikernel design yet.

Distributed systems. It is long known that the data placement problem in distributed systems is NP-complete [69, 208]. The input is a characterization of the data and a workload. We refer to [152] for a discussion of solution methods. The most advanced method we are aware of relies on a reduction to a graph partitioning problem which is passed to a heuristic solver that may need minutes to run [81].

Similar solvers are employed in several distributed DBMS tools. Examples include the SAP Data Distribution Optimizer (DDO) [19], the physical database design advisor in DB2 [168], the database tuning advisor in MS SQL Server [25], and Oracle's distribution manager [138]. The produced data placement is static but the solver can be triggered again manually by the administrator or automatically after a change in the network topology. The data placement, however, does not adapt automatically to new workload characteristics. In our experience workloads are rarely completely predictable, and there is a tendency towards highly concurrent workloads generated by several applications. Our aim is to adapt the data placement across NUMA nodes to the workload at run-time. The aforementioned solvers cannot quickly adapt to a changing workload and cannot be immediately applied to our dynamic setting. Our heuristic algorithm, however, considers only a few alternative placements, and can quickly adapt to the workload (see Chapter 6).

Partitioning specifications and table groups. Our adaptive data placement uses two notions found in automated distributed setups (see Chapter 6) [8, 19]. First, only tables for which the administrator has defined a *partitioning specification* (e.g., hash partitioning on a column) are automatically repartitioned. Second, *table groups (TG)* can be defined to recognize associated tables used by multiple-input operators. We track utilization at the level of TG. When our adaptive data placement decides to move or repartition a TG across sockets, it does so for all the tables of the TG. Equi-joins on tables of a TG that are partitioned over the joined columns (copartitioned tables), can be executed mostly locally at the sockets with the collocated table parts [8]. TG and partitioning specifications can be defined manually by the administrator, suggested by one of the aforementioned solvers for a workload, or given for popular workloads, e.g., SAP BW [8].

2.5 Overview of SAP HANA

In this thesis, we use a prototype based on SAP HANA, a commercial main-memory DBMS, to integrate and evaluate task scheduling (see Chapter 4), identify and evaluate NUMA-aware task scheduling and data placement strategies (see Chapter 5), and adapt task scheduling and data placement to the workload at run-time (see Chapter 6). In this section, we give an overview of SAP HANA.

SAP HANA aims to efficiently support OLTP and OLAP workloads [72]. It supports a multitude of data formats and provides facilities for an extensive spectrum of enterprise applications, under a flexible and componentized architecture [71]. The general database architecture is depicted in Figure 2.7. We refer readers to [72, 177] for a more extensive overview. SAP HANA incorporates four main-memory storage engines to support various workloads: (a) a column-store, that efficiently supports OLAP-dominated and mixed workloads, (b) a row-store, that is suited for OLTP workloads, (c) a text engine, and (d) a graph engine [72]. Further components provide extensions for enterprise applications, and various application interfaces such as SQLScript and calculation models.

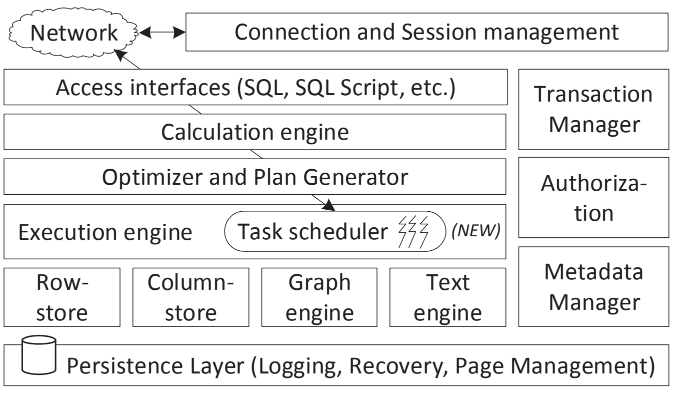

Network
Connection and Session management
Access interfaces (SQL, SQL Script, etc.)
Calculation engine
Optimizer and Plan Generator
Execution engine
Task scheduler
(NEW)
Transaction Manager
Authoriza-tion
Metadata Manager
Row-store
Column-store
Graph engine
Text engine
Persistence Layer (Logging, Recovery, Page Management)

Figure 2.7 – The general database architecture of SAP HANA.

In this architecture realizing parallelism is the key to good scalability. For scaling out, SAP HANA supports distributed query execution plans. Scaling up on every node is achieved through multi-threaded algorithms that exploit data parallelism. These algorithms are implemented with a special focus on hardware-conscious design and high scalability on modern multi-core processors. Our task scheduler improves the scalability of these parallel algorithms on a single shared-memory multi-core server (see Chapter 4). Furthermore, quick response times for short-running queries are provided by an efficient session management and client interface [113].

Main-Memory Column-Store

In this thesis, we use the main-memory column-store storage engine of SAP HANA for OLAP workloads. Next, we describe the basic data structures in the main-memory column-store. Similar data structures appear in other main-memory column-stores as well, although naming of the data structures can be different [105, 111, 117].

The data of an uncompressed column can be stored sequentially in a vector in main-memory. Compression techniques are typically employed to reduce the amount of consumed memory, and potentially speed up processing. The simplest and most common compression is dictionary encoding [118, 96, 105, 111]. In Figure 2.8, we show the data structures that compose a column in a generic column-store, along with an optional index.

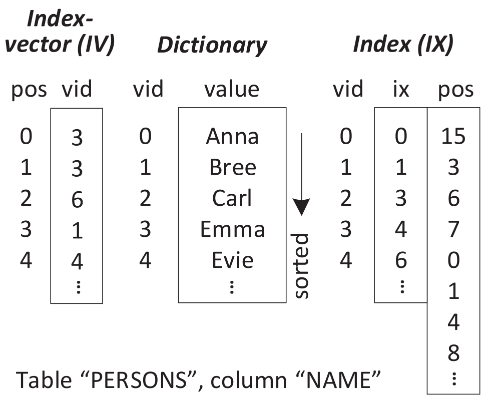


Figure 2.8 – Example of a dictionary-encoded column.

The *indexvector (IV)* is an integer vector of dictionary-encoded values, called *value identifiers (vid)*. The *position (pos)* relates to the row in the relation/table. In SAP HANA, the *dictionary* stores the sorted real values of *vid*. Fixed-width values are stored inline, while variable-length ones may require, e.g., using pointers or prefix-compressed storage for strings. A value lookup in the dictionary can be done with binary search, but one can also implement predicates like less-or-equal directly on the *vid*. The IV can be further compressed using bit-compression, i.e., using the least number of bits (called *bitcase*) to store the *vid*. Vectorization enables the efficient implementation of scans, including their predicates, using SIMD (Single instruction, multiple data) instructions. In this thesis, we use scans implemented with SSE instructions, or AVX2 instructions if supported by the processors, on bit-compressed IV [195, 197].

An optional inverted *index (IX)* can speed up low-selectivity lookups without scanning the IV. It stores the positions where a *vid* appears in the IV. The simplest IX consists of two vectors. Each position of the first correlates to a *vid*, and holds an index towards the second. The second vector holds the, possibly multiple, positions of a *vid* in the IV.

In this thesis, we identify and evaluate different data placement strategies for placing the above data structures across the sockets of a multi-socket server (see Chapter 5). We also show how analytical operations can be parallelized in a NUMA-aware fashion by considering the data placement (see Chapter 5). Furthermore, we adapt the data placement on a multi-socket server to the workload at run-time (see Chapter 6).

Mixed OLAP and OLTP workloads

SAP HANA supports fully interactive ACID transactions [113]. With respect to transactional isolation, the transaction manager of SAP HANA uses multi-version concurrency control (MVCC) to ensure consistent read operations [20, 193]. The default isolation level is "read committed", which provides statement level snapshot isolation, and two modes of transaction level snapshot isolation, "repeatable read" and "serializable", are supported as well [20]. During a transaction when rows are inserted, updated, or deleted, the system sets exclusive locks on the affected rows for the duration of the transaction [20].

For our analysis of mixed OLTP and OLAP main-memory workloads in Chapter 4, we use the main-memory column-store storage engine of SAP HANA. We note that a hybrid data layout can improve the performance of mixed workloads [29, 85], but, for our analysis, we focus on assessing the scalability of concurrency rather than different data layout approaches.

To support fast transaction processing in SAP HANA's main-memory column-store, each column is composed of two parts: the *main*, and the *delta*, as shown in Figure 2.9a. Data in both parts are dictionary-encoded, using an IV and a dictionary, as shown in Figure 2.8. Data in the main is dictionary encoded using a sorted dictionary. The dictionary-encoded data is static, bit-compressed, and further compressed for fast scanning. The delta supports transactional operations, and includes recently added, updated, and deleted data. The delta's dictionary is unsorted, and a cache-sensitive B+-tree is employed for fast lookups. To respect transactional semantics, read operations query both the main and the delta.

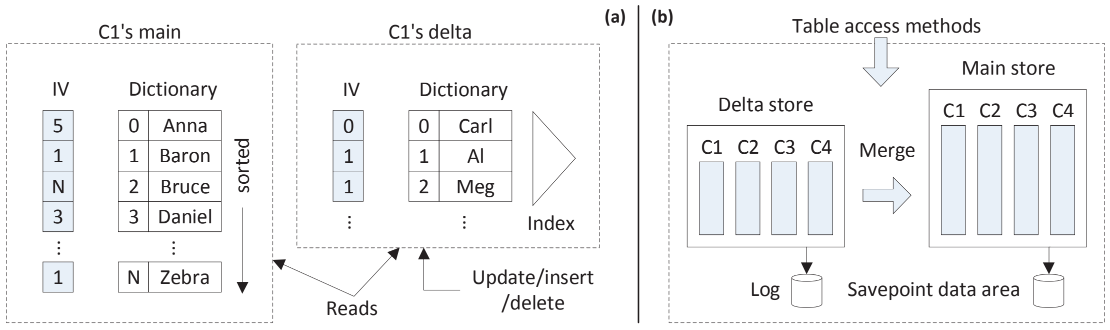


Figure 2.9 – (a) The data structures of the main and the delta parts of a column. (b) The delta part of a column is periodically merged into the main part.

Allowing the delta part to grow incessantly compromises performance of both analytical and transactional operations due to the increasing bookkeeping overhead of the delta's dictionary and index. Thus, the delta is periodically merged into the main part, as shown in Figure 2.9b, reconstructing the static data structures of the main part, and preparing an empty delta for new data. For recovery, the merge operation may store a savepoint in persistent memory, and further transactional operations (in the delta) are typically logged.

We continue the discussion of mixed workloads in Chapter 4. Through an experimental evaluation, we pinpoint the main issues affecting the scalability of mixed workloads: data freshness, flexibility, and task scheduling.

3 Sharing Data and Work Across Concurrent Queries

In this chapter, we tackle the first issue of why conventional DBMS do not scale up efficiently on modern multi-core servers (see Chapter 1). We present how and when the execution engine of a DBMS should employ reactive and proactive run-time sharing techniques across concurrent queries to improve the overall performance of the query mix.

Publications. Parts of this chapter have been published in [157, 159].

3.1 Introduction

Today, in the era of data deluge, the concurrency requirements for analytical applications can reach up to 1,000 concurrent users (see Chapter 1). General-purpose DBMS, however, cannot easily handle OLAP workloads with such concurrency [47]. A limiting factor is their query-centric model: DBMS optimize and execute each query independently (see Section 1.2). Concurrent queries, however, often exhibit overlapping data accesses or computations. The query-centric model misses the opportunities of sharing work and data, and results in performance degradation due to the contention of concurrent queries for I/O, CPU and RAM.

3.1.1 Methodologies for Sharing Data and Work

A variety of ideas have been proposed to exploit sharing, including buffer pool management techniques, materialized views, caching and multi-query optimization (see Section 2.2). More recently, data is being shared at the I/O layer using *shared scans* (with variants also known as circular scans, cooperative scans or clock scan) [58, 188, 211]. In this chapter, we evaluate run-time sharing techniques at the level of the execution engine. We distinguish two predominant methodologies: (a) reactive sharing [91], and (b) proactive sharing [33, 47, 48, 77].

Reactive sharing, alternatively known as *Simultaneous Pipelining (SP),* is introduced in QPipe [91], an operator-centric execution engine, where each relational operator is encapsulated into a self-contained module called a *stage.* Each stage detects common sub-plans among concurrent queries, evaluates only one and pipelines the results to the rest when possible. *Proactive sharing* is introduced in the CJOIN operator [47, 48]. Proactive sharing analyzes the workload and builds synergistically *shared operators* in a *global query plan (GQP)*. A single shared operator is able to evaluate multiple concurrent queries. CJOIN uses a GQP, consisting of shared hash-join operators that evaluate the joins of multiple concurrent star queries simultaneously. More recent research prototypes extend the logic to additional operators and to more general cases [33, 77].

Before continuing, please see Section 2.2 for a short survey of run-time sharing techniques, and of QPipe and CJOIN. We use terminology introduced in that section.

3.1.2 Integrating Reactive and Proactive Sharing

In order to perform our analysis and experimental evaluation of reactive vs. proactive sharing, we integrate the original research prototypes that introduced them into one system: we integrate the CJOIN operator as an additional stage of the QPipe execution engine (see Section 3.2) on top of the Shore-MT storage manager [99]. Thus, we can dynamically decide whether to evaluate multiple concurrent queries with the standard query-centric relational operators of QPipe, with or without reactive sharing, or the proactive sharing offered by CJOIN.

Furthermore, this integration allows us to combine the two sharing techniques, showing that they are in fact orthogonal. As shown in Figure 2.1d of Section 2.2, proactive sharing misses the opportunity of sharing common sub-plans, and redundantly evaluates both $Q2$ and $Q3$. Reactive sharing can be applied to shared operators to complement proactive sharing with the additional capability of sharing common sub-plans (see Section 3.2).

3.1.3 Optimizing Reactive Sharing

For the specific case of reactive sharing, it is shown in the literature [98, 164] that if there is a serialization point, enforcing aggressive sharing does not always improve performance. In cases of low concurrency and sufficient available resources, it is shown that the system should first parallelize with a query-centric model before sharing.

To calculate the turning point where sharing becomes beneficial, a prediction model is proposed [98] for determining at run-time whether reactive sharing is beneficial. In this chapter, however, we show that the serialization point is due to the push-based communication originally employed by reactive sharing [91, 98]. We show that pull-based communication can drastically minimize the impact of the serialization point, and is better suited for sharing common results on servers with multi-core processors.

We introduce *Shared Pages Lists* (SPL), a pull-based sharing approach that eliminates the serialization point caused by push-based sharing during reactive sharing. SPL are data structures that store the intermediate results of relational operators, and allow for a single producer and multiple consumers. SPL make reactive sharing always beneficial and reduce response times by up to 1.6x in cases of high concurrency, compared to not sharing. The original push-based reactive sharing design and implementation [91, 98], on the other hand, increases response times by up to 5.7x, compared to not sharing (see Section 3.3).

3.1.4 Reactive vs. Proactive Sharing

Having optimized reactive sharing, and having integrated the CJOIN operator in the QPipe execution engine, we proceed to perform an extensive analysis and experimental evaluation of reactive vs. proactive sharing (see Section 3.4). Our work answers two fundamental questions: *when* and *how* an execution engine should share to improve performance.

Sharing in the execution engine. We identify a performance trade-off between using a query-centric model and sharing. For a high number of concurrent queries, the execution engine should share, as it reduces contention for resources and improves performance in comparison to a query-centric model. For low concurrency, however, sharing is not always beneficial.

With respect to reactive sharing, we corroborate previous related work [98, 164], that if reactive sharing entails a serialization point, then enforcing aggressive sharing does not always improve performance. Our newly optimized reactive sharing with SPL, however, eliminates the serialization point, making reactive sharing always beneficial.

With respect to proactive sharing, we corroborate previous work [33, 47, 48, 77] that shared operators are efficient in reducing contention for resources and in improving performance for high concurrency (see Section 3.4.2). The design of a shared operator, however, inherently increases bookkeeping in comparison to the typical operators of a query-centric model. Thus, for low concurrency, we show that shared operators result in worse performance than the traditional query-centric operators (see Section 3.4.2).

Moreover, we show that reactive sharing can be applied to proactive sharing, in order to get the best out of the two worlds. Reactive sharing can reduce the response time of proactive sharing by 20%-50% for workloads with common sub-plans (see Section 3.4.4).

Sharing in the I/O layer. Our experimental results with reactive sharing also corroborate previous work relating to shared scans. The pull-based reactive sharing for the table scan stage is basically a circular scan per table. Our experiments show that a circular scan can either retain or improve the performance of typical analytical workloads both in cases of low and high concurrency, compared to independent scans (see Section 3.3).

Rules of thumb. Putting all our observations together, we deduce a few rules of thumb for sharing, presented in Table 3.1. Our rules of thumb apply for the case of typical OLAP workloads involving ad-hoc, long running, scan-heavy queries over relatively static data.

When	*How* to share in the	
	Execution Engine	I/O Layer
Low concurrency	Query-centric operators + reactive sharing	Reactive sharing (shared scans)
High concurrency	Proactive sharing + reactive sharing	

Table 3.1 – Rules of thumb for sharing data and work across typical concurrent queries.

3.1.5 Contributions

In this chapter, we perform an experimental analysis of two run-time work sharing methodologies, (a) reactive sharing, and (b) proactive sharing, based on the original research prototypes that introduce them. Our analysis answers two fundamental questions: *when* and *how* an execution engine should employ run-time sharing in order to improve the performance of typical analytical workloads. Our work makes the following main contributions:

- **Integration of reactive and proactive sharing:** We show that reactive and proactive sharing are orthogonal, and can be combined to take the best of the two worlds (Section 3.2). We experimentally show that reactive sharing can further improve the performance of proactive sharing by 20%-50% for workloads that expose common sub-plans.
- **Pull-based reactive sharing:** We introduce Shared Pages Lists (SPL), a pull-based approach for reactive sharing that eliminates the sharing overhead of push-based reactive sharing. Pull-based reactive sharing is better suited for multi-core servers than push-based reactive sharing, and can reduce response times by up to 1.6x compared to not sharing (see Section 3.3).
- **Evaluation of reactive vs. proactive sharing:** We analyze the trade-offs of reactive and proactive sharing, and their combination. We detail through an extensive sensitivity analysis when each one is beneficial (Section 3.4). We show that query-centric operators combined with reactive sharing result in better performance for cases of low concurrency, while the shared operators of proactive sharing, enhanced by reactive sharing, are better suited for cases of high concurrency.

Outline. The rest of this chapter is organized as follows. Section 3.2 describes our implementation for integrating reactive and proactive sharing. Section 3.3 presents Shared Pages Lists, our pull-based solution for sharing common results during reactive sharing. Section 3.4 includes our experimental evaluation. We present our conclusions in Section 3.5.

3.2 Integrating Reactive and Proactive Sharing

By integrating reactive and proactive sharing, we can exploit the advantages of both forms of sharing. In Section 3.2.1, we describe how reactive sharing can conceptually improve the performance of the shared operators of proactive sharing in the presence of common sub-plans, using several examples. These observations apply to general GQP, and are applicable to the research prototypes we mention in Section 2.2. We continue in Section 3.2.2 to describe our implementation based on CJOIN and QPipe.

3.2.1 Benefits of Applying Reactive Sharing to Proactive Sharing

Identical queries. If a new query is completely identical with an ongoing query, reactive sharing takes care to reuse the final results of the ongoing query for the new query. If we assume that the top-most operators in a query plan have a full step WoP (e.g. when final results are buffered and given wholly to the client instead of being pipelined), the new query does not need to participate at all in the GQP, independent of its time of arrival during the ongoing query's evaluation. This is the case where the integration of reactive and proactive sharing offers the maximum performance benefit. Additionally, admission costs are completely avoided, the tuples' bitmaps do not need to be extended to accommodate the new query (translating to fewer bitwise operations), and the latency of the new query is decreased to the latency of the remaining part of the ongoing query.

Shared selections. If a new query has the same selection predicate as an ongoing query, reactive sharing allows to avoid the redundant evaluation of the same selection predicate from the moment the new query arrives until the end of evaluation of the ongoing query (a selection operator has a linear WoP). For each tuple, reactive sharing copies the resulting bit of the shared selection operator for the ongoing query, to the position in the tuple's bitmap that corresponds to the new query.

Shared joins. If a new query has a common sub-plan with an ongoing query under a shared join operator, and arrives within the step WoP, reactive sharing can avoid extending tuples' bitmaps with one more bit for the new query for the sub-plan. The join still needs to be evaluated, but the number of bitwise operations can be reduced.

Shared aggregations. If a new query has a common sub-plan with an ongoing query under a shared aggregation operator, and arrives within the step WoP, reactive sharing avoids calculating a redundant sum. It copies the final result from the ongoing query.

Admission costs. For every new query submitted to the GQP of proactive sharing, an admission phase is required that possibly re-adjusts the GQP to accommodate it. In case of common sub-plans, reactive sharing can avoid part of the admission costs. The cost depends on the implementation.

For CJOIN [47, 48], the admission cost of a new query includes (a) scanning all involved dimension tables, (b) evaluating its selection predicates, (c) extending the bitmaps attached to tuples, (d) increasing the size of hash tables of the shared hash-joins to accommodate newly selected dimension tuples (if needed), and (e) stalling the pipeline to re-adjust filters [47, 48]. For identical queries, reactive sharing can avoid these costs completely. For queries with common sub-plans, reactive sharing can avoid parts of these costs, such as avoiding scanning dimension tables for which selection predicates are identical.

For DataPath [33], reactive sharing can decrease the optimization time of the GQP if it assumes that the common sub-plan of a new query can use the same part of the current GQP as the ongoing query. For SharedDB [77], reactive sharing can help to start a new query before the next batch at any operator if it has a common sub-plan with an ongoing query and has arrived within the corresponding WoP of the operator.

3.2.2 CJOIN as a QPipe Stage

We integrate the original CJOIN operator into the QPipe execution engine as a new stage, using Shore-MT [99] as the underlying storage manager. In Figure 3.1, we depict the new stage that encapsulates the CJOIN pipeline.

The CJOIN stage accepts incoming QPipe packets that contain the necessary information to formulate a star query: (a) the projections for the fact table and the dimension tables to be joined, and (b) the selection predicates. The CJOIN operator does not support selection predicates for the fact table [47], as these would slow the preprocessor significantly. Fact table predicates are evaluated on the output tuples of CJOIN.

To improve admission costs we use batching, following the original CJOIN proposal [48]. In one pause of the pipeline, the admission phase adapts the filters for all queries in the batch. During the execution of each batch, additional new queries form a new batch to be subsequently admitted.

With respect to threads, there is a number of threads assigned to filters (we assume the horizontal configuration of CJOIN [47, 48]), each one taking a fact tuple from the preprocessor, passing it through the filters up to the distributor. The original CJOIN uses a single-threaded distributor which slows the pipeline significantly. To address this bottleneck, we augment the distributor with several *distributor parts*. Every distributor part takes a tuple from the

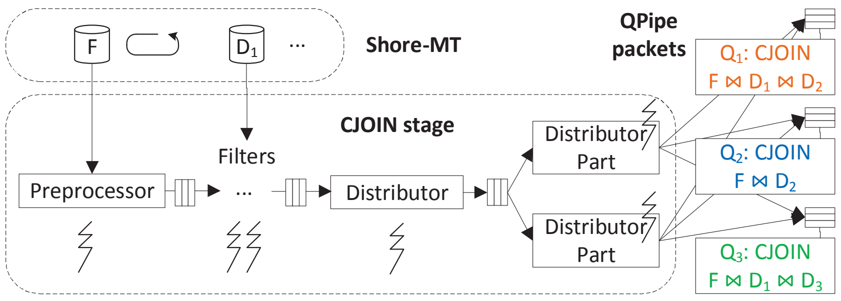


Figure 3.1 – Integration scheme of CJOIN as a QPipe stage.

distributor, examines its bitmap and determines relevant CJOIN packets. For each relevant packet, it performs the projection of the star query and forwards the tuple to the output buffer of the packet.

CJOIN supports only shared hash-joins. Subsequent operators in a query plan, e.g., aggregations or sorts, are query-centric. Nevertheless, our evaluation gives us insight on the general behavior of shared operators in a GQP with proactive sharing, as joins typically comprise the most expensive part of a star query.

Reactive sharing for the CJOIN stage. We enable reactive sharing for the CJOIN stage with a step WoP. Evaluating the identical queries Q_2 and Q_3 of Figure 2.1d of Section 2.2 employing reactive sharing, requires only one packet entering the CJOIN stage. The second satellite packet reuses the results.

CJOIN is itself an operator and we integrate it as a new stage in QPipe. As with any other QPipe stage, reactive sharing is applied on the overall CJOIN stage. Conceptually, our implementation applies reactive sharing for the whole series of shared hash-joins in the GQP. Our analysis, however, gives insight on the benefits of applying reactive sharing to fine-grained shared hash-joins as well. This is due to the fact that a redundant CJOIN packet involves all redundant costs we mentioned in Section 3.2.1 for admission, shared selections operators and shared hash-joins. Our experiments show that the cost of a redundant CJOIN packet is significant and reactive sharing decreases it considerably.

3.3 Optimizing Reactive Sharing

In this section, we present design and implementation issues of reactive sharing, and how to address them. Contrary to intuition, it is shown in the literature that reactive sharing is not always beneficial: if there is a serialization point during reactive sharing, then sharing common results aggressively can lead to worse performance, compared to a query-centric model that implicitly exploits parallelism [98, 164]. When the producer (host packet) forwards results to consumers (satellite packets), it is in the critical path of the evaluation of the remaining nodes of the query plans of all involved queries. Forwarding results can cause a significant serialization point. In this case, the DBMS should first attempt to exploit available resources and parallelize as much as possible with a query-centric model, before sharing. A prediction model is proposed [98] for determining at run-time whether sharing is beneficial. In this section, however, we show that reactive sharing is possible without a serialization point, thus rendering reactive sharing always beneficial.

The serialization point is caused by strictly employing push-based communication. Pipelined execution typically uses FIFO buffers to exchange results between operators [47, 48, 91, 98]. This allows to decouple query plans and have a distinct separation between queries, similar to a query-centric design. During reactive sharing, simultaneous pipelining forces the single thread

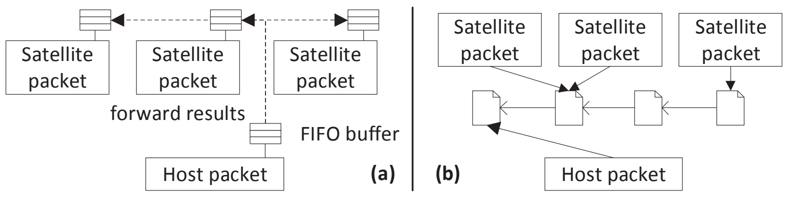


Figure 3.2 – Sharing identical results during SP with: (a) push-only model and (b) a SPL.

of the pivot operator of the host packet to forward results to all satellite packets sequentially (see Figure 3.2a), which creates a serialization point.

This serialization point is reflected in the prediction model [98], where the total work of the pivot operator includes a cost for forwarding results to all satellite packets. By using copying to forward results [98], the serialization point becomes significant and delays subsequent operators in the plans of the host and satellite packets. This creates a trade-off between sharing and parallelism, where in the latter case a query-centric model without sharing is used.

Sharing vs. Parallelism. We demonstrate this trade-off with the following experiment, similar to the experiment of [98], which evaluates reactive sharing for the table scan stage with a memory-resident database. Though the trade-off applies for disk-resident databases and other stages as well, it is more pronounced in this case. Our experimental configuration can be found in Section 3.4. We evaluate two configurations of the QPipe execution engine: (a) *No Sharing (FIFO)*, which evaluates query plans independently without any sharing, and (b) *CS (FIFO)*, with reactive sharing enabled only for the table scan stage, thus supporting circular scans (CS). FIFO buffers are used for pipelined execution and copying is used to forward pages during reactive sharing, following the original push-only design [91, 98]. We evaluate identical TPC-H [15] Q1 queries, submitted at the same time, with a database of scale factor 1. We use either 1 H/W thread on the server, to simulate a uniprocessor, or all 24 H/W threads available on the server. The leftmost graphs of Figure 3.3 show the response times of the configurations, while varying the number of concurrent queries.

For the case of 1 H/W thread, *CS (FIFO)* has always equal or better performance than *No Sharing (FIFO)*. For the case of 24 H/W threads, however, *CS (FIFO)* has always equal or worse performance than *No Sharing (FIFO)*. Push-based reactive sharing suffers from low utilization of CPU resources, due to the aforementioned serialization point of SP. The critical path increases with the number of concurrent queries. For 128 concurrent queries, it increases the response time by up to 5.7x, compared to the query-centric model, which evaluates queries independently. In terms of CPU load, it uses on average 2.4 H/W threads, while the query-centric model uses on average 23 H/W threads. To sum up, push-based reactive sharing can be used if only 1 H/W thread is available to the DBMS, but should not be used if all H/W threads of the multi-core server are available to the DBMS. This trade-off corroborates the prediction model for uniprocessor and multi-core servers [98].

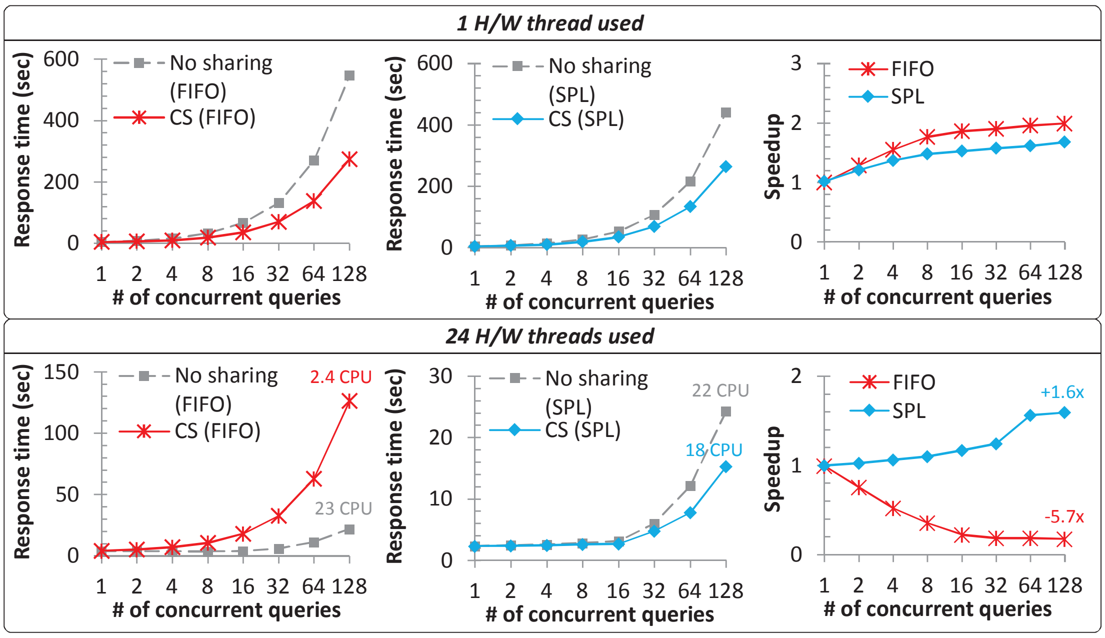


Figure 3.3 – Evaluating multiple identical TPC-H Q1 queries with a push-based model during reactive sharing (leftmost graphs), and with a pull-based model during reactive sharing (middle graphs), using either 1 H/W thread (top graphs) or all available H/W threads (bottom graphs) of the server. The rightmost graphs show the corresponding speedups of the two methods of reactive sharing over not sharing.

Nevertheless, the impact of the serialization point of reactive sharing can be minimized. Simply copying tuples in a multi-threaded way would not solve the problem, due to synchronization overhead and increased required CPU resources. A solution would be to forward tuples via pointers, a possibility not considered by the original system. We can, however, avoid unnecessary pointer chasing; by employing pull-based communication, we can share the results and eliminate forwarding altogether. In essence, we transfer the responsibility of sharing the results from the producer to the consumers. Thus, the total work of the producer does not include any forwarding cost. Our pull-based communication model is suited for reactive sharing for any stage with a step or linear WoP. Our pull-based model eliminates the serialization point of the push-based model for reactive sharing, leading to better scalability on servers with modern multi-core processors with virtually no sharing overhead.

To achieve this, we create an intermediate data structure, the *Shared Pages Lists* (SPL). SPL have the same usage as the FIFO buffers of the push-only model. A SPL, however, allows a single producer and multiple consumers. A SPL is a linked list of pages, depicted in Figure 3.2b. The producer adds pages at the head, and the consumers read the list from the tail up to the head independently.

To show the benefits of SPL, we run the same experiment as before, by employing SPL instead of FIFO buffers. The middle graphs of Figure 3.3 show the response times of the configurations, while varying the number of concurrent queries. When reactive sharing does not take place, a SPL has the same role as a FIFO buffer, used by one producer and one consumer. Thus, the *No Sharing (SPL)* line has a similar trend with the *No Sharing (FIFO)* line. During reactive sharing, however, a single SPL is used to share the results of one producer with all consumers.

With SPL, reactive sharing has the same or better performance than not sharing, for both cases of 1 and 24 H/W threads, for all cases of concurrency. We avoid using a prediction model altogether, for deciding whether to share or not. Parallelism is achieved due to the minimization of the serialization point. For the case of 24 H/W threads, for high concurrency, *CS (SPL)* uses more CPU resources than *CS (FIFO)*, and reduces response times by 1.6x in comparison to *No Sharing (SPL)*. Our results corroborate the employment of shared scans for DBMS that handle scan-heavy analytical workloads (see Section 2.2).

3.3.1 Design of a Shared Pages List

Figure 3.4 depicts a SPL. It points to the head and tail of the linked list. The host packet adds pages at the head. Satellite packets read pages from the SPL independently. Due to different concurrent actors accessing the SPL, we associate a lock with it. Contention for locking is minimal in all our experiments, mainly due to the granularity of pages we use (32 KB). A lock-free linked list, however, can also be used to address any scalability problems.

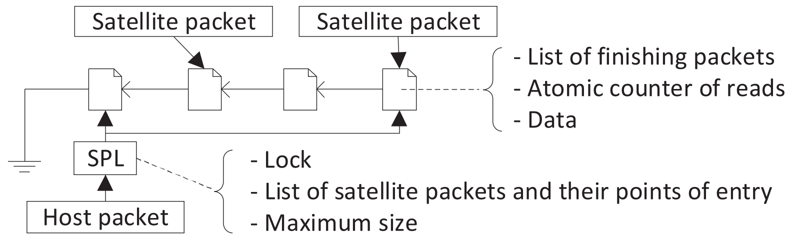

Satellite packet
Satellite packet
- List of finishing packets
- Atomic counter of reads
- Data
SPL
- Lock
- List of satellite packets and their points of entry
- Maximum size
Host packet

Figure 3.4 – Design of a shared pages list.

If we allow the SPL to be unbounded, we can achieve the maximum parallelism possible, even if the producer and the consumers move at different speeds. There are practical reasons, however, why the SPL should not be unbounded, similar to the reasons why a FIFO buffer should not be unbounded, including: saving RAM and regulating differently paced actors.

To investigate the effect of the maximum size, we ran the experiment of Figure 3.3, for the case of 8 concurrent queries, varying the maximum size of SPL up to 512 MB. Figure 3.5 shows the results. We observe that changing the maximum size of the SPL does not significantly affect the response time. Hence, we chose a maximum size of 256 KB for our experiments to minimize the memory footprint of SPL.

In order to decrease the size of the SPL, the last consumer is responsible for deleting the last page. Each page has an atomic counter with the number of consumers that will read this page. When a consumer finishes processing a page, he decrements its counter and deletes the page if he sees a zero counter. In order to know how many consumers will read a page, the SPL stores a list of active satellite packets. The producer assigns their number as the initial value of the atomic counter of each emitted page.

Linear window of opportunity. To handle a linear WoP, such as circular scans, the SPL stores the point of entry of every consumer. When the host packet finishes processing, the SPL is passed to the next host packet that handles the processing for re-producing missed results.

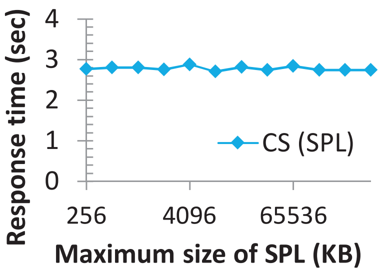


Figure 3.5 – Size of SPL.

When the host packet emits a page, it checks for consumers whose point of entry is this page, and will need to finish when they reach it. The emitted page has attached to it a list of these finishing packets, which are removed from the active packets of the SPL (they do not participate in the atomic counter of subsequently emitted pages). When a consumer (packet) reads a page, it checks whether it is a finishing packet, in which case, it exits the SPL.

Shared scans and SPL. Pull-based models, similar to SPL, have been proposed for shared scans that are specialized for efficient buffer pool management and are based on the fact that all data is available for accessing (see Section 2.2). SPL differ because they are generic and can be used during reactive sharing at any operator which may be producing results at run-time. It is possible, as well, to use shared scans for table scans, and use SPL during reactive sharing for other operators.

3.4 Experimental Evaluation

This section includes our experimental sensitivity analysis of reactive vs. proactive sharing techniques. We also show how proactive sharing can be enhanced by reactive sharing to exploit the advantages of both for cases of high concurrency.

3.4.1 Experimental Methodology

We compare five configurations of the QPipe execution engine:

- *CS*, supporting reactive only for the table scan stage, i.e., circular scans. Subsequent operators are query-centric, evaluated separately with pipelining, without sharing. This serves as our baseline.
- *RS*, supporting reactive sharing additionally for the join stage. It improves performance over *CS*, in cases of high similarity, i.e. common sub-plans. In cases of low similarity, it behaves similar to *CS*.

- *PS*, proactive sharing, which is the result of our integration of CJOIN into QPipe, hence the joins in star queries are evaluated with a GQP of shared hash-joins. We remind that CJOIN only supports shared hash-joins, thus subsequent operators are query-centric. Nevertheless, this configuration allows us to compare shared hash-joins with the query-centric ones used by the previous configurations, giving us insight on the performance characteristics of general shared operators.

- *PS+RS*, which additionally supports reactive sharing for the CJOIN stage (see Section 3.2.2). We use this configuration to evaluate the benefits of combining proactive and reactive sharing. It behaves similar to *PS* in cases of low similarity in the query mix.

In all our experiments, reactive sharing for the aggregation and sorting stages is off. This is done on purpose to isolate the benefits of reactive sharing for joins only, so as to better compare *RS* and *PS+RS*.

We use the Star Schema Benchmark (SSB) [141], read-only, and Shore-MT [99] as the storage manager. SSB is a simplified version of TPC-H [15] where the tables lineitem and order have been merged into lineorder and there are four dimension tables: date, supplier, customer and part. Shore-MT is an open-source multi-threaded storage manager developed to achieve scalability on multi-core platforms.

We use a Sun Fire X4470 server with four hexa-core processors Intel Xeon E7530 at 1.86 GHz, with hyper-threading disabled and 64 GB of RAM. Each core has a 32 KB L1 instructions cache, a 32 KB L1 data cache, and a 256 KB L2 cache. Each processor has a 12 MB L3 cache, shared by all its cores. For storage, we use two 146 GB 10 kRPM SAS 2.5” disks, configured as a RAID-0. The O/S is a 64-bit SMP Linux (Red Hat), with a 2.6.32 kernel.

We clear the file system caches before every measurement. All configurations use a large buffer pool that fits datasets of scale factors up to 30 (scanning all tables reads 21 GB of data from disk). SPL are used for exchanging results among packets. We use 32 KB pages and a maximum size of 256 KB for a SPL (see Section 3.3).

Unless stated otherwise, every data point is the average of multiple iterations with standard deviation less or equal to 10%. Furthermore, we mention the average CPU usage and I/O throughput of representative iterations (averaged only over their activity period), to gain insight on the performance of the configurations.

Our sensitivity analysis is presented in the following Sections 3.4.2-3.4.4. We vary (a) the number of concurrent queries, (b) whether the database is memory-resident or disk-resident, (c) the selectivity of fact tuples, (e) the scale factor, and (d) the query similarity which is modeled in our experiments by the number of possible different submitted query plans. We measure performance by evaluating multiple concurrent instances of SSB Q3.2. It is a typical star query that joins three of the four dimension tables with the fact table. The SQL template and the execution plan are shown in Figure 3.6. We select a single query template for our

```
SELECT   c_city, s_city, d_year,
         SUM(lo_revenue) as revenue
FROM     customer, lineorder, supplier, date
WHERE    lo_custkey = c_custkey
         AND lo_suppkey = s_suppkey
         AND lo_orderdate = d_datekey
         AND c_nation = [NationCustomer]
         AND s_nation = [NationSupplier]
         AND d_year >= [YearLow]
         AND d_year <= [YearHigh]
GROUP BY c_city, s_city, d_year
ORDER BY d_year ASC, revenue DESC
```

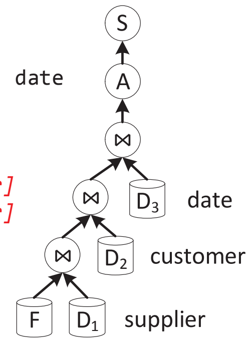


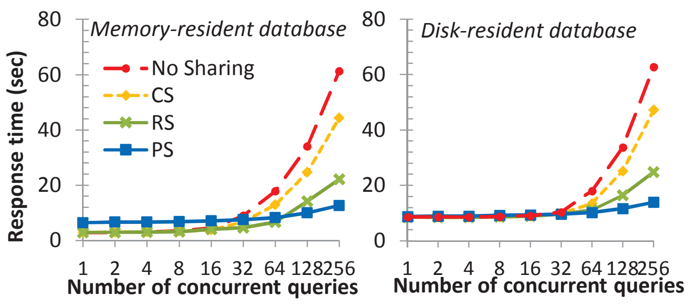
Figure 3.6 – The SSB Q3.2 SQL template and the query plan.

sensitivity analysis because we can adjust the similarity of the query mix to gain insight on the benefits of reactive sharing, and also, the GQP of CJOIN is the same for all experiments, with the same 3 shared hash-joins for all star queries. Queries are submitted at the same time, and are all evaluated concurrently. This single batch for all queries allows us to minimize query admission overheads for proactive sharing, and additionally allows us to show the best case for reactive sharing, as all queries with common sub-plans arrive surely inside the WoP of their pivot operators. We note that variable inter-arrival delays can eliminate sharing opportunities for reactive sharing, and refer the interested reader to the original QPipe paper [91] to review the effects of interarrival delays for different cases of pivot operators and WoP.

3.4.2 Impact of Concurrency

We start with an experiment that does not involve I/O accesses to study the computational behavior of the configurations. We store our database in a RAM drive. We evaluate multiple concurrent SSB Q3.2 instances for a scale factor 1. The predicates of the queries are chosen randomly, keeping a low similarity factor among queries and the selectivity of fact tuples varies from 0.02% to 0.16% per query. Figure 3.7 (left) shows the response times of the configurations, while varying the number of concurrent queries.

For this first experiment, we evaluate additionally the *No Sharing* variation of QPipe, without



Configuration Measurement	No Sharing	CS	RS	PS
Memory-resident database				
Avg. # Cores Used	20.6	19.7	18.8	3.47
Disk-resident database				
Avg. # Cores Used	20.9	19.8	17.1	3.49
Avg. Read Rate (MB/s)	106.7	74.5	97.7	156

Figure 3.7 – Experiment with memory-resident (left) and disk-resident (middle) database of SF 1. The table (right) includes measurements for the case of 256 concurrent queries.

reactive sharing, to confirm the implications of Section 3.3. For low concurrency, CPU resources are used sufficiently. Starting with as few as 32 concurrent queries, there is contention for CPU resources, due to the fact that our server has 24 cores and all analytical operators are evaluated separately. For 256 queries all cores are used at their maximum. The circular scans of *CS* improve performance by reducing contention for CPU resources and the buffer pool.

CS misses several sharing opportunities at higher operators in the query plans. *RS* can exploit them. Even though we use random predicates, the ranges of variables of the SSB Q3.2 template allows *RS* to share the first hash-join 126 times, the second hash-join 17 times, and the third hash-join 1 time, on average for 256 queries. Thus, it saves more CPU resources and results in lower response time than the circular scans alone.

The shared operators of *PS* offer the best performance, as they are the most efficient in saving resources. *PS* has an initialization overhead in comparison to the other configurations, attributed to its admission phase, which has a large part that pauses the pipeline (see Section 3.2.1). The shared hash-joins in the GQP can effortlessly evaluate many instances of SSB Q3.2. Nevertheless, admission and evaluation costs accumulate for an increasing number of queries, thus the *PS* line also starts to degrade. *PS* reduces the response time of the query-centric *No Sharing* by a factor of 6, and of the circular scans of *CS* by a factor of 4.

We do not depict *PS+RS*, as it has the same behavior as *PS*. As we noted in Section 3.2.2, our implementation of *PS+RS* supports sharing CJOIN packets with all predicates identical. This is rare due to this experiment's random selection predicates.

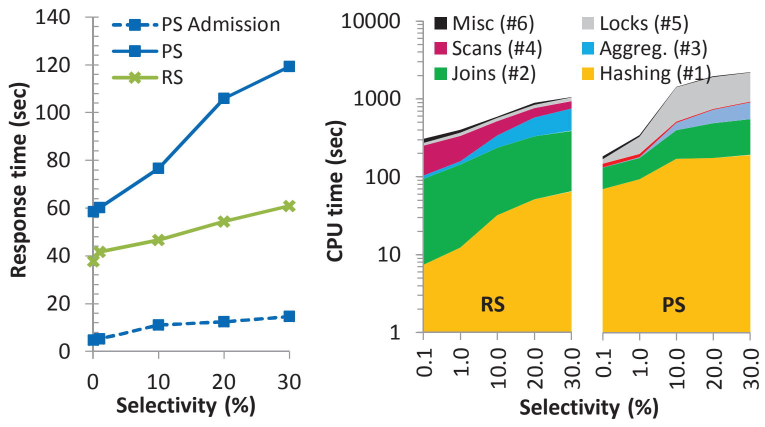
Our observations apply also to the same experiment with the database on disk, shown in Figure 3.7 (middle). Response times for low concurrency have increased, but not significantly for high concurrency because the workload becomes CPU-bound. For high concurrency, there is CPU contention for *No Sharing*, which de-schedules scanner threads regularly resulting in low I/O throughput. *CS* improves the performance. *RS* further improves performance by eliminating common sub-plans. The shared operators of *PS* still prevail for high concurrency. Furthermore, the overhead of the admission phase of CJOIN, that we observed for a memory-resident database, is masked by file system caches for disk-resident databases. We explore this effect in a next experiment, where we vary the scale factor.

Implications. Reactive sharing is able to eliminate common sub-plans. The shared operators of proactive sharing are more efficient in evaluating a high number of queries, in comparison to standard query-centric operators.

3.4.3 Impact of Data Size

In this section, we study the behavior of the configurations by varying the amount of data they handle. We perform two experiments: in the first, we vary the selectivity of fact tuples of queries, and in the second, the scale factor.

Impact of selectivity. We use a memory-resident database with scale factor 10. The query mix consists of 8 concurrent queries which are different instances of a modified SSB Q3.2 template. For the modified template, we select the maximum possible range for the year. Moreover, we extend the WHERE clause of the query template by adding more options for both customer and supplier nation attributes. For example, if we use a disjunction of 2 nations for customers and 3 nations for suppliers, we achieve a selectivity of $\frac{2}{25}\frac{3}{25} \approx 1\%$ of fact tuples. Nations are selected randomly over all 25 possible values and are unique in every disjunction, keeping a minimal similarity factor. The results are shown in Figure 3.8. In this experiment, there is no contention for resources and no common sub-plans. Thus, we do not depict *CS*, as it has the same behavior as *RS*, and we do not depict *PS+RS*, as it has the same behavior as *PS*.



Configuration Measurement	RS	PS
Avg. # Cores Used	17.79	18.86

Figure 3.8 – 8 queries with a memory-resident database of SF 10. The table includes measurements for 30% selectivity.

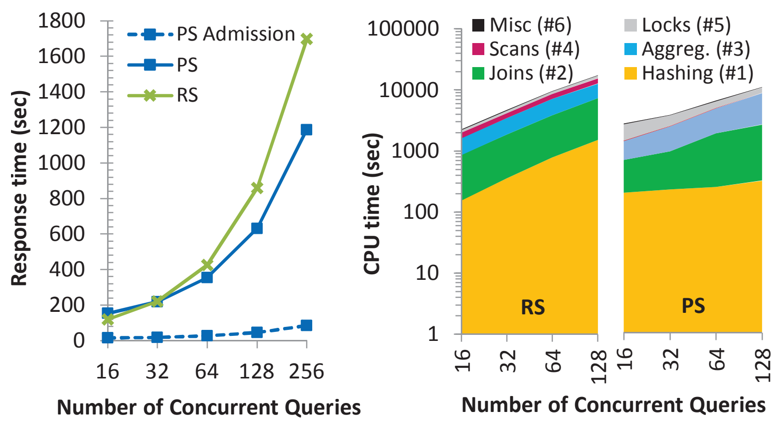
Our selectivity experiments provide more insight on the general behavior of the configurations. For this reason, we also include the time of the admission phase of CJOIN, and performance breakdown graphs. The latter show the CPU time of all cores, as measured with Intel VTune Amplifier, for different parts of the query evaluation. We compare the effect of sharing on the CPU time of hash-joins rather than analyze the bottlenecks of QPipe and CJOIN, which are largely dependent on implementation details. We further break down the CPU time of hash-joins to two categories. The first, shown as “Hashing”, includes the total CPU time of the `hash()` and `equal()` functions, which are the heart of the building and probing phases, and allow us to compare the effect of sharing, without strong side-effects from implementation details. The remaining CPU time of the hash-joins is shown as “Joins”.

Both *RS* and *PS* show a degradation in performance as selectivity increases, due to the increasing amount of data they need to handle. *PS*, however, is always worse than *RS*. This is due to three reasons mainly. Firstly, the cost of the admission phase of *PS* is increased, as more tuples are selected for referencing in the hash tables of the filters.

Secondly, the shared operators inherently entail a bookkeeping overhead, in comparison to standard query-centric operators. In our case, the additional cost of shared hash-joins includes the maintenance of larger hash tables for the union of the selected dimension tuples of all concurrent queries, and bitwise operations between the bitmaps of tuples. Query-centric operators do not entail these costs, and maintain a hash table for one query. The increased bookkeeping costs are reflected in the CPU time of the area under Joins of *PS*, which is more expensive than *RS* for all cases of selectivity. As the selectivity increases, the hashing CPU time of *RS* increases faster than *PS*, as it does not share parts of the hash-joins of the concurrent queries. We note that the bookkeeping overhead can be decreased significantly with careful implementation choices. DataPath [33] uses a single large hash table for all shared hash-joins, and techniques to decrease the maintenance and access costs for the hash table.

Thirdly, synchronization costs are a significant reason for the worse trend of *PS*. Threads contend while passing tuples through the pipeline. Nevertheless, synchronization costs are highly dependent on implementation. In DataPath or SharedDB, a shared operator in a GQP does not necessarily require multiple threads. Nevertheless, for low concurrency, the synchronization costs for a query are higher in a GQP than in the query-centric model, as a GQP tends to be much larger than the constituent query plans. For one query in a GQP, tuples not selected by it, but selected by other queries, need to pass through shared operators, and tuples selected by the query may need to pass by additional shared operators to accommodate other concurrent queries. This is also a reason why proactive sharing achieves better throughput for high concurrency, but may hurt the latency of queries, especially for low concurrency.

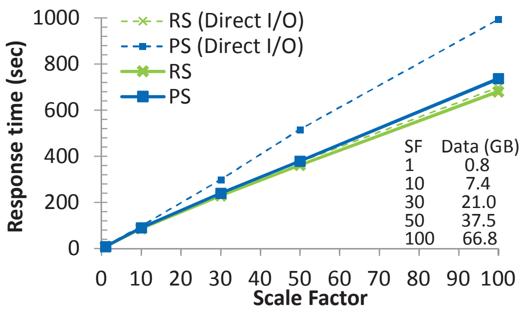
We used 8 queries to avoid CPU contention. For higher concurrency, shared operators still prevail, due to their efficiency in saving resources. Figure 3.9 shows the response times for the case of 30% selectivity. For high concurrency, the query-centric operators of *RS* contend for resources. This is also shown in the CPU times of the breakdown graph, which all scale (superlinearly) with the number of queries. *PS* is able to save more resources and outperform the query-centric operators. This is best reflected by the hashing CPU time, which stays at the same level, irrespective of the number of queries, as the hashing is shared.



Configuration Measurement	RS	PS
Avg. # Cores Used	22.86	17.73

Figure 3.9 – Memory-resident database of SF 10 and 30% selectivity. The table includes measurements for 256 queries.

Impact of scale factor. The same trade-off between shared operators and query-centric operators is observed by varying the scale factor. We use disk-resident databases and 8 concurrent queries with randomly varied predicates and selectivity between 0.02% and 0.16%. The results are shown in Figure 3.10. The response times of *RS* and *PS* increase linearly. Their slopes, however, are different. The reasons are the same as in our selectivity experiment.



Configuration Measurement	RS (Direct I/O)	PS (Direct I/O)	RS	PS
Avg. # Cores Used	5.96	1.68	5.38	2.47
Avg. Read Rate (MB/s)	97.16	70.01	215.58	204.71

Figure 3.10 – 8 concurrent queries with disk-resident databases. The table includes measurements for the case of SF 100.

We also show the response time of the two configurations by using direct I/O for accessing the database on disk, to bypass file system caches. This allows us to isolate the overhead of CJOIN's preprocessor. As we have mentioned, the preprocessor is in charge of the circular scan of the fact table, the admission phase of new queries, and finalizing queries when they wrap around to their point of entry. These responsibilities slow down the circular scan significantly. Without direct I/O, file system caches coalesce contiguous I/O accesses and read-ahead, achieving high I/O read throughput in sequential scans, masking the preprocessor's overhead.

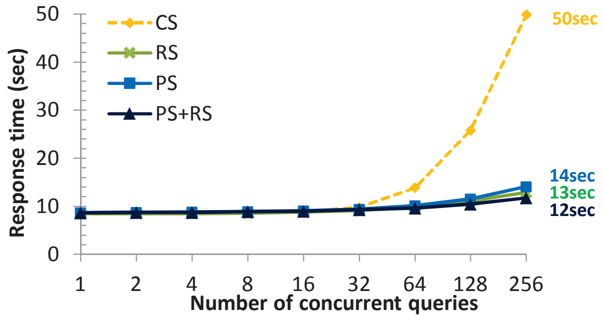
Implications. For low concurrency, proactive sharing has a bookkeeping overhead in comparison to query-centric operators. For high concurrency, however, the overhead is amortized.

3.4.4 Impact of Similarity

In this experiment we use a disk-resident database of scale factor 1. We limit the randomness of the predicates of queries to a small set of values: there are 16 possible query plans for instances of Q3.2. The selectivity of fact tuples ranges from 0.02% to 0.05%. In Figure 3.11, we show the response times of the configurations, varying the number of concurrent queries.

RS evaluates a maximum of 16 different plans and reuses results for the rest of similar queries. It shares the second hash-join one time, and the third hash-join 238 times, on average, for 256 queries. This leads to high sharing and minimal contention for computations. On the other hand, *CS* does not share operators other than the table scan, resulting in high contention.

Similarly, *PS* misses exploiting these sharing opportunities and evaluates identical queries redundantly. In fact, *RS* outperforms *PS*. *PS+RS*, however, is able to exploit them. For a group



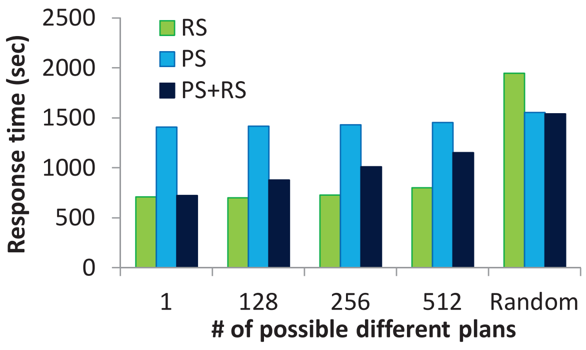
Configuration Measurement	CS	RS	PS	PS+RS
Avg. # Cores Used	20.32	9.6	2.5	2.34
Avg. Read Rate (MB/s)	74.37	120.2	130.18	130.15

Figure 3.11 – Disk-resident database of SF 1 and 16 possible plans. The table includes measurements for 256 queries.

of identical star queries, only one is evaluated by the GQP. *PS+RS* shares CJOIN packets 239 times on average for 256 queries. Thus, *PS+RS* outperforms all configurations.

To further magnify the impact of SP, we perform another experiment for 512 concurrent queries, a scale factor of 100 (with a buffer pool fitting 10% of the database), and varying the number of possible different query plans. Figure 3.12 shows the results. *PS* is not heavily affected by the number of different plans. For the extreme cases of high similarity, *RS* prevails. For lower similarity, the number of different plans it needs to evaluate is larger and performance is deteriorated due to contention for CPU resources. *PS+RS* is able to exploit identical CJOIN packets and improve the performance of *PS* by 20%-50% for cases with common sub-plans. For the case of identical queries, *PS+RS* reduces the response time of *PS* by 2x.

Implications. We can combine proactive and reactive sharing to eliminate redundant computations and improve the performance of shared operators in a GQP for a query mix that exposes common sub-plans.



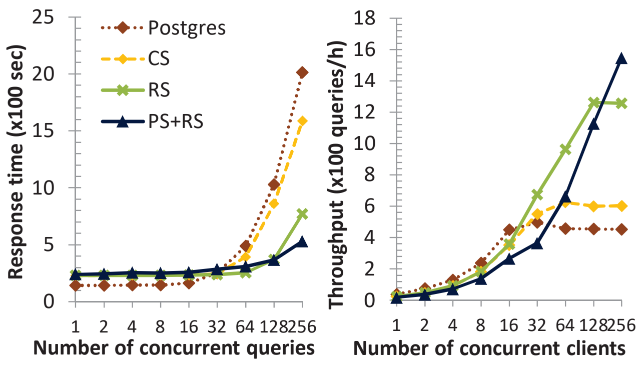
Different plans	1	128	256	512	Random
RS (1st hash-join)	1	18	49	106	362
RS (2nd hash-join)	0	95	156	196	82
RS (3rd hash-join)	510	381	287	188	5
PS+RS	510	384	287	190	12

Figure 3.12 – Evaluating 512 concurrent queries with a varying similarity factor, for a SF 100. The table includes the sharing opportunities of reactive sharing (average of all iterations).

3.4.5 SSB Query Mix

In this section we evaluate *Postgres*, *CS*, *RS*, and *PS+RS* using a mix of three SSB queries (namely Q1.1, Q2.1 and Q3.2), with a disk-resident database and a scale factor of 30. We use PostgreSQL 9.1.4 as another example of a query-centric execution engine that does not share among concurrent queries. We configure PostgreSQL to make the comparison with QPipe as fair as possible. We use 32 KB pages, large shared buffers that fit the database, ensure that it never spills to the disk and that the query execution plans are the same. The predicates for the queries are selected randomly and the selectivity of fact tuples is less than 1%. Each query is instantiated from the three query templates in a round-robin fashion, so all configurations contain the same number of instances for each query type.

Figure 3.13 (left) shows the response times of the configurations, while varying the number of concurrent queries. As *Postgres* is a more mature system than the two research prototypes, it attains a better performance for low concurrency. Our aim, however, is not to compare the per-query performance of the configurations, but their efficiency in sharing among a high number of concurrent queries. *Postgres* follows a traditional query-centric model of execution, and does not share among in-progress queries. *CS* results in a better performance due to circular scans. *RS* further improves performance with the elimination of any common sub-plans. *PS+RS* attains the best performance, as shared operators are the most efficient in sharing among a high number of concurrent queries.



Configuration / Measurement	Postgres	CS	RS	PS+RS
Response time experiment (256 queries)				
Avg. # Cores Used	19.68	20.64	21.6	15.2
Avg. Read Rate (MB/s)	16.9	63.2	111.9	137.4
Throughput experiment (256 clients)				
Avg. # Cores Used	20.16	21.6	21.6	16.8
Avg. Read Rate (MB/s)	16.7	63.2	106.6	138.2

Figure 3.13 – Disk-resident database of SF 30. Response time (left) and throughput experiment (middle), varying the number of concurrent queries and clients respectively. The table (right) includes measurements for both experiments.

Figure 3.13 (middle) also shows the throughput of the three configurations, by varying the number of concurrent clients. Each client submits three queries, which are instances of the three query templates. Proactive sharing is able to handle new queries with minimal additional resources. Thus, the throughput of *PS+RS* continues to increase. The throughput of the query-centric operators of *Postgres*, *CS* and *RS*, however, ultimately flattens with an increasing number of clients.

3.5 Conclusions

In this chapter we perform an experimental study to answer *when* and *how* an execution engine should share data and work across concurrent analytical queries. We review two state-of-the-art run-time sharing techniques: reactive sharing and proactive sharing. We perform an extensive evaluation of reactive and proactive sharing, based on their original research prototype systems.

Work sharing is typically beneficial for high concurrency because the opportunities for common work increase, and it reduces contention for resources. For low concurrency, however, there is a trade-off between sharing and parallelism, particularly when the sharing overhead is significant. We show that proactive sharing is not beneficial for low concurrency as shared operators in a GQP inherently involve a bookkeeping overhead compared to query-centric operators. For reactive sharing, however, we show that it can be beneficial for low concurrency as well, if the appropriate communication model is employed: we introduce SPL, a pull-based approach that scales better on servers with modern multi-core processors than the original push-based model of reactive sharing. SPL is a data structure that promotes parallelism by shifting the responsibility of sharing common results from the producer to the consumers.

Furthermore, we show that reactive and proactive sharing are two orthogonal techniques and their integration allows to share operators and handle a high number of concurrent queries, while also sharing any common sub-plans presented in the query mix. In conclusion, analytical query engines should employ query-centric operators with reactive sharing for low concurrency and proactive sharing enhanced by reactive sharing for high concurrency.

In this chapter, we show how to tackle the first issue of why conventional DBMS do not scale up efficiently on modern multi-core servers (see Chapter 1): sharing data and work across concurrent queries. In the following chapters, we tackle the second issue: making the execution engine NUMA-aware by assuming full control of scheduling and in-memory data placement. As a first step towards taking control of scheduling, the next chapter (see Chapter 4) explores task scheduling inside the execution engine of a main-memory DBMS.

4 Task Scheduling for Highly Concurrent Main-Memory Workloads

In this chapter, we show how the execution engine of a main-memory DBMS can employ task scheduling. This chapter is split into two sections. In Section 4.1, we show how we integrate our task scheduler in a main-memory DBMS: SAP HANA [72]. We illustrate how the task scheduler can handle blocking tasks in order to avoid underutilization of CPU resources, and how to avoid the scheduling overhead of excessive intra-query task parallelism. In Section 4.2, as an extension to this thesis, we investigate which factors affect the performance of mixed workloads by examining the commonalities and differences between two prominent main-memory DBMS: SAP HANA and HyPer [101]. We show that the main factors affecting the way mixed workloads utilize resources are data freshness, scheduling, and flexibility. Regarding scheduling specifically, we show that OLAP workloads tend to dominate over the OLTP workloads, and pinpoint the significance of workload management such as prioritization.

Publications. Parts of Section 4.1 have been published in [26, 158]. Parts of Section 4.2 have been published in [162].

4.1 Integrating Task Scheduling in a DBMS

The execution engine of a typical DBMS supports inter-query parallelism [121, 166] by using a single logical thread for each short-lived latency-sensitive transactional query. Long-running operations, such as complex transactional queries or analytical queries, may employ intra-query parallelism [121, 166] using more logical threads. For parallelizing highly concurrent workloads, simply issuing logical threads and leaving scheduling to the operating system (OS), can lead to performance impediments such as unexpected OS bugs [124], high thread creation costs, and numerous context switches. The context switches are incurred by the OS time sharing policy that handles the oversubscription of CPU resources by balancing the usage of a limited number of available hardware threads among a higher number of threads using time slices [12, 119].

DBMS typically employ a query admission control to limit the number of processed queries and avoid performance degradation due to overload [92]. A query admission control, however, is a mechanism that operates on a per-query level, and can only indirectly avoid an excessive number of threads. It does not control the CPU utilization of queries after they have been admitted. Task scheduling [12, 24, 37, 43, 94] can be an alternative or complementary solution, as it uses a number of threads to process all operations for the whole run-time of an application. Tasks encapsulate operations and are stored in task pools. Worker threads are employed by the task scheduler to process the tasks.

Moreover, task scheduling is well-suited for the recent wave of main-memory DBMS that forfeit disk-based storage in favor of performance (see Section 2.1). By removing I/O bottlenecks, main-memory DBMS can focus completely on optimizing CPU and memory utilization. Task scheduling can prove a powerful tool for main-memory DBMS, as it automates the efficient usage of CPU resources, especially of modern shared-memory multi-core processors [12, 37, 43, 142], and helps developers easily parallelize database operations.

Recent popular task scheduling frameworks include the OpenMP API [11, 37] and Intel Thread Building Blocks (TBB) [12]. Their main advantage is that developers express partitionable operations, that can be parallelized with a variable number of tasks, using a high level of abstraction such as data parallelism. The high level of abstraction helps to automatically adjust the task granularity of analytical partitionable operations, such as aggregations or hash-joins. The high level of abstraction, however, also turns out to be a disadvantage, as it cannot be used straightforwardly by already developed applications. Integration into a commercial DBMS would require a re-write of large portions of code, which is a process with significant cost and time considerations. Moreover, partitionable operations in commercial DBMS typically define their task granularity independently, without the use of a central mechanism for data parallelism. This is the common case, as optimizing the granularity of a single DBMS paritionable operation alone involves considerable research effort (see [42], for example, for partitioning in hash-joins). We show how to adjust task granularity in a main-memory DBMS, in a non-intrusive manner, without the need of a high level of abstraction. We supply partitionable operations with a hint reflecting recent CPU availability, that can be used to adjust their task granularity. Our experiments show that when partitionable operations use this concurrency hint, overall performance for analytical workloads is significantly improved.

Furthermore, recent task schedulers, e.g. Intel TBB, use a fixed number of worker threads, equal to the number of hardware threads, to avoid oversubscription of CPU resources. The fixed concurrency level, however, is only suited for CPU-intensive tasks that rarely block [13]. Tasks in DBMS can block due to synchronization. Thus, the fixed concurrency level can result in underutilization of CPU resources. We show how the task scheduler can detect the inactivity periods of tasks and dynamically adapt its concurrency level. Our scheduler gives control of additional worker threads to the OS when needed.

Contributions. In Section 4.1, we apply task scheduling to a commercial main-memory DBMS. Our experiments show the benefits of using task scheduling for scaling up main-memory DBMS over modern multi-core servers, to efficiently evaluate highly concurrent analytical workloads. Our main contributions are:

- We show that a fixed concurrency level for task scheduling is not suitable for a DBMS. Our scheduler adapts its concurrency level, by detecting blocked tasks, and giving control of additional worker threads to the OS to saturate CPU resources.
- We show that using a hint reflecting recent CPU availability helps to adjust the task granularity of partitionable analytical operations. The concurrency hint improves overall performance significantly in cases of high concurrency, by reducing costs related to communication, synchronization and bookkeeping.
- We show how we integrate our task scheduler into a prototype of SAP HANA [72], a commercial main-memory DBMS. We show that our task scheduler improves the performance of highly concurrent analytical workloads (TPC-H [15]) by up to 16%.

Outline. In Section 4.1.1, we show how we integrate our task scheduler in SAP HANA. Next, we present the general architecture of our scheduler in Section 4.1.2, how we handle blocking tasks using a flexible concurrency level in Section 4.1.3, and how we use concurrency hints to aid task creators of partitionable operations adapt their task granularity in Section 4.1.4. Finally, in Section 4.1.5, we show our experimental evaluation.

4.1.1 Integration in SAP HANA

For an overview of SAP HANA and its general architecture, please see Section 2.5. Figure 4.1 depicts the architecture and various components of SAP HANA.

There are three independent thread pools. (1) The *Dispatcher* is a simple task graph scheduler used typically for parallelizing partitionable analytical operations. (2) The *Executor* is a task graph scheduler that processes plans of operations that can potentially be distributed across servers. Plan nodes can use the Dispatcher for parallelizing on one server. (3) The *Receivers* are threads that process received network requests. Short-running transactions are typically completely executed within a Receiver. For more complex transactions, longer analytical or distributed queries, a Receiver may use the Executor or the Dispatcher.

We design our new scheduler to integrate the two thread pools of the Executor and the Dispatcher, which parallelize analytical and distributed operations. We do not integrate the thread pool of Receivers, so that we do not hurt the latency of any short-running transactions or queries. If a Receiver decides to parallelize using either the Executor or the Dispatcher, then it indirectly uses our task scheduler. Our integration solves the following problems. Firstly,

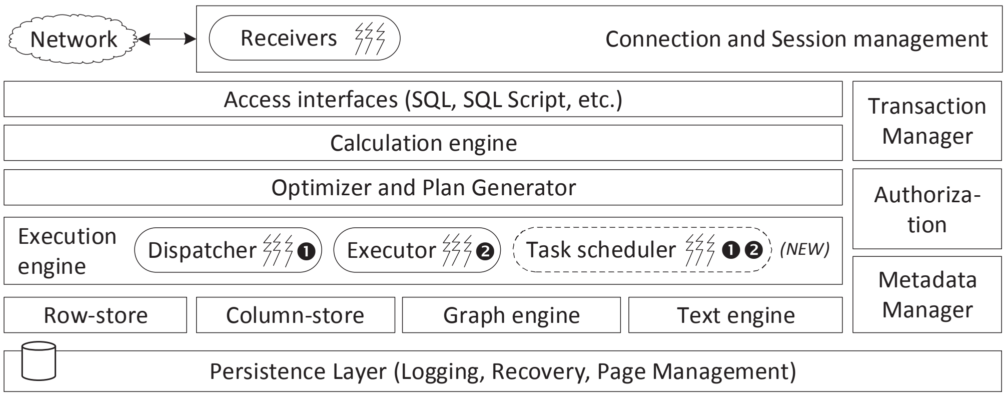


Figure 4.1 – Our task scheduler integrates two main thread pools of SAP HANA.

by integrating two threads pools, which are resource-intensive, we alleviate the problem of overcommitting CPU resources and decrease context switching costs. Secondly, DBMS administrators need to configure only one thread pool instead of two. Thirdly, developers need to know one thread pool implementation for parallelizing operations, and not two.

Our scheduler constitutes a new component in the general architecture of SAP HANA (see Figure 4.1), and is orthogonal to other components such as the Persistency Layer or the Transaction Manager. It is important to note that our scheduler does not compromise any transactional correctness or persistency semantics.

4.1.2 Task Scheduler Architecture

To support fast scheduling of all heterogeneous general-purpose tasks, we opt for a dynamic task scheduler that does not require or process a priori execution information about the tasks, except for potentially a directed acyclic graph (DAG) defining their correlations and ultimately their order of execution. The DAG can take any form, with the only restriction of having a single root node. This does not prevent the creation of single-node graphs. Each node in the task graph can contain any piece of code. A node can potentially spawn a new task graph, or issue itself again to the scheduler. We can encapsulate tasks that coordinate synchronization among themselves, since we take care to maintain a flexible concurrency level (see Section 4.1.3). Optionally the developer can assign a priority for the task graph, which results in a decreased or increased probability of being chosen for execution. The developer then dispatches the root node to the scheduler, and can wait for execution of the task graph or continue immediately.

The scheduler maintains two sets of queues, depicted in Figure 4.2. The first set contains one queue per priority and holds the root nodes of the submitted graphs that have not yet been initiated for execution. The second set contains queues that hold the non-root nodes to be executed. The second set actually constitutes the main distributed task pools for our scheduler. The task pools can further be sorted by node depth, in order to favor execution of deep-running graphs, or by the timestamp of the owning query, in order to favor execution of earliest queries. For our experiments of Section 4.1.5, we sort the task pools by node depth, because resource-intensive queries tend to create deep-running graphs, and we take care to finish these queries early, in order to free up the resources. We use distributed task pools to reduce synchronization contention. We create as many task pools as the number of sockets. We note that in the next chapter (see Chapter 5), we describe how we modify our task scheduler to support NUMA-aware task scheduling. More task pools can also be created if the number of hardware threads in one socket is high and results in synchronization contention for the task pool. Each worker thread is assigned to a specific task pool in a round-robin fashion according to its ordinal identifier. If the worker thread finds its assigned task pool empty, it starts querying other task pools, in a round-robin fashion, and steals tasks [94].

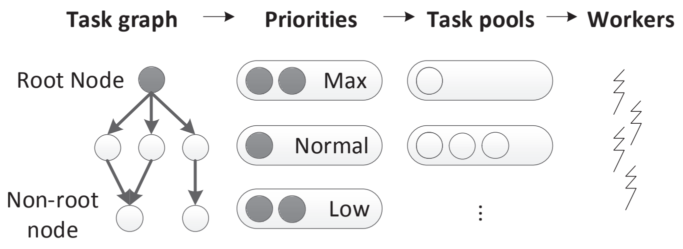


Figure 4.2 – The data structures used by the task scheduler.

When the task pools are empty, a free worker retrieves a root node from the queues of priorities, with a probability that favors prioritized root nodes. This probability is configurable. We note that in our experiments of Section 4.1.5, all tasks have the same priority. After executing the root node, the worker thread continues executing the first descendant for better data locality, while the rest of the descendants are dispatched randomly to the task pools for load-balancing. When the task pools are not empty, a free worker retrieves his next task from the task pools. When a non-root node is executed, the worker checks which descendants are ready for execution, takes upon the first of them and dispatches the rest to the task pools.

Integration. Our simple design allows to integrate the two thread pools of SAP HANA into our scheduler (see Section 2.5). We quickly bundle old tasks and generic blocks of code into tasks for our task scheduler. As is standard for task schedulers [12], we do not bundle I/O-bound operations into tasks. These operations are executed by separate threads that are handled by the underlying OS scheduler. Since these threads do not reserve any worker threads from our scheduler, we can keep the system busy with CPU-intensive tasks in the presence of I/O operations. It is easy to detect I/O-bound operations in main-memory DBMS, as general query execution is CPU-bound or memory-bound. Heavy I/O operations, such as savepoints, are only done periodically and in the background to minimize the disruption of the general performance of the database [72]. Thus, I/O-bound operations are traced mainly inside the persistence or network layer.

Watchdog thread. To control workers, but also to monitor the state of execution, we reserve an additional watchdog thread. The watchdog typically sleeps, but wakes up periodically to gather information and potentially control worker threads, similar to the notion of centralized scheduling [88]. We use light-weight mechanisms for monitoring, based on statistical counters, such as the number of waiting and executing tasks, how many tasks each worker thread has executed etc. These counters are updated using atomic instructions by each worker thread and the watchdog.

4.1.3 Dynamic Adjustment of Concurrency Level

Employing a number of worker threads equal to the number of hardware threads is suitable for task schedulers whose aim is to handle CPU-intensive tasks that do not block frequently [12]. Our aim, however, is to integrate already-developed general-purpose code into tasks. We need to handle tasks that can use synchronization primitives and locks. When tasks are inactive, we take care to overlap inactivity periods with additional worker threads and saturate CPU resources. Next, we describe a task's potential inactivity states, and how our scheduler handles them by adapting its concurrency level at run-time.

Blocked workers. The OS scheduler is the first to know when a thread blocks after a system call for a synchronization primitive. It then cedes the CPU to another thread waiting for its time slice. If we set a fixed number of worker threads equal to the number of hardware threads, blocked threads will not be overlapped by other threads, as the OS scheduler does not have knowledge of any other working threads in our application. This results in underutilization of CPU resources, as the OS scheduler could potentially schedule another worker thread while a worker thread blocks. Also, since we do not know how the developer synchronizes tasks, a fixed concurrency level can lead to potential deadlocks, if the interdependency edges between the nodes in a task graph are not correctly used. For example, if a node in a task graph requires a conditional variable from another node at the same level of the task graph, the latter node may not be scheduled in time if the nodes in the level are more than the hardware threads. Deadlocks can also happen if code synchronizes heavily between different task graphs.

To avoid deadlocks and underutilization of CPU resources, we argue that a scheduler handling general-purpose tasks should not use a fixed concurrency level. Our watchdog periodically checks for blocked worker threads, and activates additional worker threads, that get scheduled by the OS immediately and overlap the inactivity period. Thus, we cooperate with the OS by voluntarily adjusting the concurrency level, and giving control of additional worker threads to the OS when needed to saturate CPU resources. We exploit both the advantages of task scheduling and the OS scheduler: Task scheduling ensures that the number of working threads is small enough so that costly context switches are avoided. By dynamically adjusting the concurrency level, we exploit the capability of the OS scheduler to quickly cede the CPU of a blocked thread to a new worker thread.

To detect blocked threads efficiently, we do not use OS synchronization primitives directly. The DBMS can encapsulate these in user-level platform-independent data structures. For example, SAP HANA on Linux uses a user-level semaphore based on atomic instructions, that calls the “futex” facilities of Linux when needed. We leverage these user-level synchronization primitives to know when a worker is about to call a potential system call that could block.

Active concurrency level. We define:

concurrency level = total number of worker threads

The concurrency level is variable. There can be a number of inactive workers, such as blocked threads, and a number of active workers (see Figure 4.3). We are mainly interested, however, in keeping the total number of active workers as close as possible to the number of hardware threads, in order to saturate CPU resources. For this reason, we define:

active concurrency level = concurrency level – inactive workers

When threads resume from inactivity, they are considered again in the active concurrency level, which can at times be higher than the number of hardware threads.

Parked threads. In order to fix a high active concurrency level, when there are no free threads, the scheduler gets the chance to preempt a worker when it finishes a task. We cannot preempt a worker in the middle of a generic task, as it can be in a critical section and the consequences can be unpredictable. Instead of ending the thread, we keep it suspended, in a *parked* state. The watchdog is responsible for monitoring if the active concurrency level gets low and waking up parked threads. Parked threads overcome the costs of creating logical threads, which include the memory allocation of their stacks.

Other inactive threads. Apart from blocked and parked threads, we define two additional states of inactivity. Firstly, there can be tasks that wait for another task graph. This inactivity state is comparable to OpenMP’s suspend/resume points (e.g. `taskwait`) [37], or to TBB’s wait methods (e.g. `wait_for_all`) [12]. Secondly, we give the developer the opportunity to explicitly define a region of code as inactive. For both these cases of inactive threads, a new worker thread is activated immediately if allowed by the active concurrency level, instead of being activated by the watchdog. We note that while a worker thread is blocked, parked,

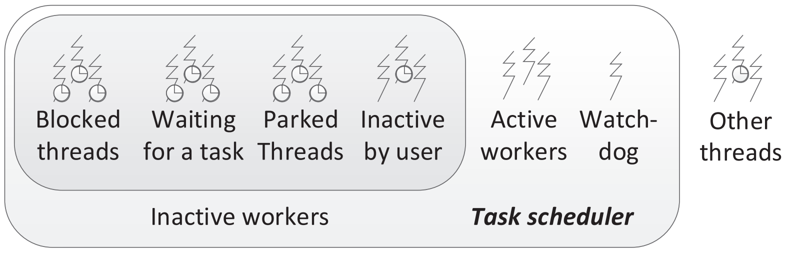


Figure 4.3 – The task scheduler’s types of worker threads.

or waiting for another task graph, it is also considered inactive by the OS scheduler. A code region, however, that is defined by the developer as inactive, pertains only to our scheduler's accounting for its active concurrency level, while the OS considers the relevant worker thread as runnable and schedules it.

All the aforementioned types of workers are shown in Figure 4.3. The total number of inactive workers is defined as:

$$\begin{aligned}\text{inactive workers} = {} & \text{blocked workers} + \text{inactive by user}\\ & + \text{workers waiting for a task graph} + \text{parked workers}\end{aligned}$$

Avoiding too many active threads. We note that activating additional worker threads in place of inactive workers may not always be beneficial. If the inactivity period of a worker is short, and the newly activated worker begins executing a large task, then when the first worker returns from inactivity, there will be two worker threads active. If this situation is repeated many times, the active concurrency level can get much higher than available hardware threads, leading to context switching costs from the OS scheduler.

For this reason, it is important to handle inactivity states carefully. The inactivity states where the developer specifies a code region as inactive, and where a task waits upon another task, are typically not too short. These inactivity states lower the active concurrency level, and immediately activate additional worker threads that increase the active concurrency level up to the number of available hardware threads. The duration of the inactivity state of blocked threads, however, is generally unknown, and can be too short. Thus, blocked tasks are handled differently: they only lower the active concurrency level, and do not immediately spawn additional workers. The active concurrency level is increased only when the watchdog checks it periodically and attempts to fix it by activating additional worker threads, or in case another inactive worker thread resumes activity in the meanwhile and thus increases the active concurrency level and is allowed to continue working on next tasks. Thus, too short block periods are typically hidden between the intervals of the watchdog and of active tasks. Even in a bad case when the active concurrency level gets too high, this is quickly fixed when active workers finish their current tasks and are preempted and parked in order to fix the active concurrency level.

To support our intuition, our experiments have an active concurrency level that is most of the time equal to the number of hardware threads (see Section 4.1.5). We note that the watchdog interval we use in our experiments is 20 ms. We have experimented with larger intervals as well, but have not noticed significant differences in the active concurrency level.

4.1.4 Dynamic Adjustment of Task Granularity

Partitionable operations can be parallelized using a variable number of tasks. Many analytical operations fall into this category, e.g., aggregations and hashing. If a column needs to be aggregated, it can be split into parts which can be processed in parallel independently. This is a classic example of data parallelism using a fork-join approach [36].

For this kind of partitionable operations, a number of tasks lower than the number of hardware threads, i.e., a coarse granularity of tasks, can underutilize CPU resources. A higher number of tasks (up to the number of hardware threads) means that the partitionable operation can potentially use more CPU resources and decrease its latency. Using a fine granularity, however, can potentially introduce additional costs for communication, synchronization and scheduling [3, 52, 122]. Thus, a balance is required for the task granularity.

Task schedulers like Intel TBB [12] can greatly help in case of partitionable operations. As the developer expresses partitionable operations through higher-level algorithmic structures and data parallelism, the framework employs a centralized mechanism for adjusting task granularity. In DBMS, however, partitionable operations do not necessarily use a central mechanism for data parallelism. This is because optimizing the granularity of a single paritionable operation alone involves considerable research effort (see [42], for example, for partitioning in hash-joins). There can be distinct components for partitionable operations that handle data parallelism and granularity independently. In SAP HANA, for example, each partitionable operation employs heuristics to find the right task granularity, based on factors such as data size, communication costs, and the number of hardware threads of the system.

In this thesis, we are not concerned with how each component calculates task granularity, but with how task granularity affects performance when numerous concurrent queries, possibly with additional partitionable operations, are being processed. The problem is that partitionable operations calculate the number of tasks irrespective of other concurrent tasks. In the worst case, every operation can dispatch a number of tasks equal to the number of hardware threads. Our experiments of Section 4.1.5 show that this practice results in a myriad of tasks in cases of high concurrency, and increased bookkeeping and scheduling costs. Task creators for partitionable operations need to adjust their task granularity by considering other concurrent tasks. To solve this problem, our scheduler provides information about the state of execution to partitionable operations that they can use for the calculation of the task granularity.

Concurrency hint. If a partitionable operation issues more tasks than can be handled by free worker threads at the moment, there is little to no benefit for parallelism, and redundant scheduling costs. The intuition is that in cases of high concurrency, when the system is fully loaded and free worker threads are scarce, partitionable operations should opt for a very coarse granularity in order to minimize the number of tasks to be processed. In SAP HANA specifically, if the concurrency hint is zero, analytical operations may opt for a single-threaded optimized code path instead of a parallel code path.

Our scheduler can provide task creators with information about the current availability of computing resources, as it has knowledge of the active concurrency level of the whole DBMS. Thus, it can give a hint to task creators about the maximum number of tasks they should create at the moment. The watchdog is responsible for calculating the *concurrency hint*, which is an exponential moving average of the free worker threads in the recent past. The free worker threads are defined as:

$$\textit{free worker threads} = max\{0, \text{number of hardware threads} - \text{active concurrency level}\}$$

The concurrency hint is defined as:

$$\textit{concurrency hint} = a * \text{free worker threads} + (1.0 - a) * \text{previous concurrency hint}$$
$$\text{where } 0 \leq a \leq 1.0$$

Due to the dynamic nature of our workers, which can change status often and quickly, an average can give better results than an absolute value. For our experiments, we use an exponential moving average, with equal weight for the free workers threads of the previous observations and the currently observed number of free worker threads (i.e., $a = 0.5$). The sampling rate is configured at 50 ms in our experiments, which provides reasonable smoothing over the recent past, and also quickly captures changes in the number of free worker threads. Due to the fact that the exponential moving average captures all past observations, we take care to reset it to the number of hardware threads when it surpasses a predefined threshold. This threshold is set to 90% of the number of hardware threads for our experiments.

4.1.5 Experimental Evaluation

We integrate our task scheduler in a prototype built on SAP HANA (SPS6), a commercial main-memory column-store DBMS. We use the SPS6 version when our integration originally took place, to show the contributions of our work that motivated the final integration of the thread pools in later versions of SAP HANA. We compare the following variations of our prototype:

- *Baseline*, without our task scheduler, and with the three different thread pools (see Section 4.1.1). This serves as our baseline.
- *Fixed*, which integrates two of the thread pools of *Baseline* into tasks for our task scheduler (see Section 4.1.1). This variation assumes workers blocked on synchronization primitives as working, and includes them in the active concurrency level of the scheduler. Also, this variation defines the concurrency hint as the number of hardware threads, to simulate the original behaviour.
- *Flexible*, which is like *Fixed*, but uses a flexible concurrency level by assuming workers blocked on synchronization primitives as inactive.

- *Hints*, which is like *Flexible*, but with the concurrency hint following the exponential moving average of free workers in the recent past. Partitionable analytical operations adjust their task granularity according to the concurrency hint. This variation is the best of our task scheduler.

We use the server of Figure 2.5. It has eight ten-core processors Intel Xeon E7-8870 at 2.40 GHz, with hyper-threading enabled, and 1 TB of RAM. The OS is a 64-bit SMP Linux (SuSE), with a 2.6.32 kernel. Unless stated otherwise, every data point in our graphs is an average of multiple iterations with a standard deviation less than 10%. Our measurements for context switches and CPU times are gathered from Linux. The total number of instructions retired are gathered from Intel Performance Counter Monitor [196]. For all experiments, we warm up the DBMS first and there are no thinking times. We make sure that all queries and clients are admitted, and we disable query caching because our aim is to evaluate the execution of the queries and not query caching.

We use a read-only variant of the TPC-H benchmark [15] with a scaling factor 10, stored in a column-store. We measure performance by varying the number of concurrent queries, and measuring the response time of each variation from the moment we issue the queries until the last query returns successfully. Queries are instantiated from the 22 TPC-H query templates in a round-robin fashion, with the same parameters for each query template for stable results, but without query caching. We start measuring from 32 concurrent queries, to include all query templates, up to 1024. The results are shown in Figure 4.4.

Fixed improves performance of *Baseline* by only 3% for high concurrency. The main improvement comes from reducing lock contention, as shown by the reduction in system CPU

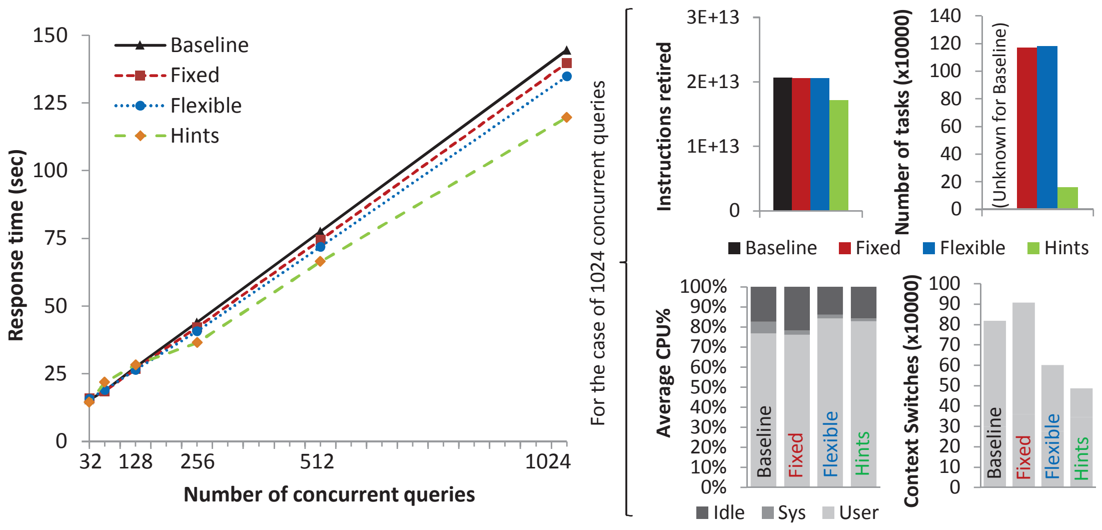


Figure 4.4 – Experiment with TPC-H queries. Measurements on the right-hand side are for the case of 1024 TPC-H concurrent queries.

time. The number of context switches has increased, even though we integrate all thread pools of *Baseline* into a single task scheduler. We attribute this to the fixed concurrency level. When workers block on synchronization primitives, the OS does not have knowledge of any additional workers to schedule. It replaces the time slice of a blocked worker with any non-CPU-intensive thread, outside the scheduler, with a small time slice. This is also reflected in the increased idle CPU time.

Flexible, which has a flexible concurrency level, overcomes this problem and improves performance of *Baseline* by 7%. When many worker threads block, the watchdog issues more worker threads and gives the chance to the OS scheduler to schedule CPU-intensive worker threads with new tasks and full time slices. That is reflected in the decreased idle CPU time, and the fewer context switches.

Hints results in the best performance improvement of *Baseline* by 16%. The coarser task granularity leads to a reduction of the total number of tasks by 86%. We achieve a significant reduction in unnecessary bookkeeping and scheduling costs, which is reflected in the 16% reduction of the total number of instructions retired. Furthermore, we corroborate previous related work that a coarser granularity results in less costs for synchronization and communication [3, 52, 122], since system CPU time is further decreased.

We note that for the case of 64 concurrent queries of Figure 4.4, the standard deviation for *Hints* is up to 30%. As mentioned in Section 4.1.4, this is due to the fact that in a few iterations, a partitionable operation that got a low concurrency hint was left in the end alone, underutilizing CPU resources and slightly prolonging the response time. Nevertheless, the benefit of the concurrency hint for the cases of high concurrency, on which we focus, is significant.

To better understand the effect of the flexible concurrency level and hints throughout the whole experiment, we show the timelines for *Fixed* and *Hints* for the case of 1024 queries in Figure 4.5. For *Fixed*, we notice the effect of bursts of too many tasks being issued to the scheduler. The redundant scheduling, communication, and bookkeeping costs of these bursts of numerous tasks result in erratic behavior of the CPU utilization. In contrast, the timeline for *Hints* presents a much smoother run-time. The majority of the tasks are issued by all queries

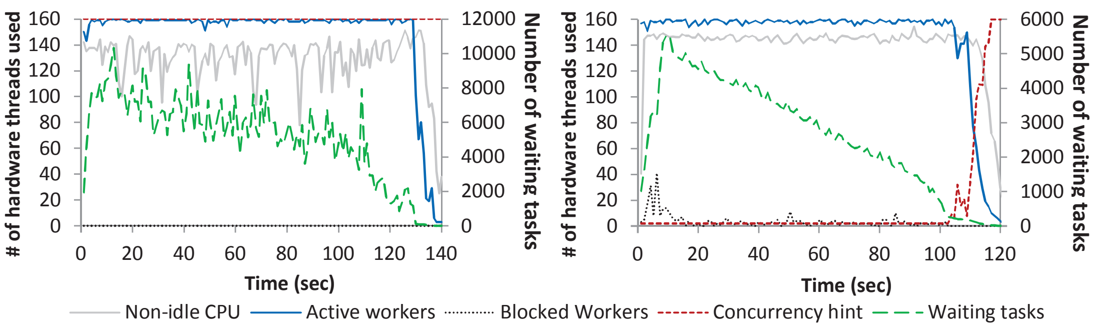


Figure 4.5 – Timelines for *Fixed* (left) and *Hints* (right), for the case of 1024 TPC-H queries.

in the beginning of the experiment, and they are gradually scheduled until the end of the experiment. The CPU utilization line is more stable.

4.2 Scaling Up Mixed Transactional and Analytical Workloads

In this section, we shortly extend the scope of this thesis to mixed OLAP and OLTP main-memory workloads. We analyze the performance of two main-memory databases that support mixed workloads, SAP HANA [72] and HyPer [101], while evaluating the mixed workload CH-benCHmark [59] and scaling the number of concurrent transactional and analytical clients. We identify that scheduling is one of the significant factors, among data freshness and flexibility, that affect how mixed workloads utilize resources.

Real-time reporting. Nowadays, the design gap between OLTP-oriented and OLAP-oriented DBMS or data warehouses is prominent, due to the increasing demands and performance requirements of big data applications [181] (see Chapter 1). OLTP-oriented DBMS, such as VoltDB [182] or IBM DB2, are typical row-stores that deliver high throughput for updates and index-based queries (see Section 2.1). OLAP-oriented DBMS, such as Vectorwise [209] or Sybase IQ [126] or DB2 BLU [167], are typical column-stores that deliver high performance for complex analytical queries. In exchange for high performance, they do not support transactional workloads or offer only a chunk-wise mechanism for loading data. As a result, data analytics queries run on an outdated version of operational data. This is unacceptable for real-time reporting, where organizations and enterprises are increasingly requiring analytics on fresh operational data to gain a competitive advantage or obtain insight about fast-breaking situations [16, 150]. Examples include online games that make special offers based on non-trivial analysis [49], liquidity and risk analysis, which benefits from fresh data while also requiring complex analytical queries [154], and fraud detection analyzing continuously arriving transactional data [144].

The need for real-time reporting necessitates the development of a new class of DBMS that can efficiently support mixed (OLTP and OLAP) workloads processing common data of a common schema [154]. Efficient processing means scaling OLTP clients to as many users as possible, with reasonably short response times [73], while, at the same time, servicing OLAP clients whose longer-running queries should be able to efficiently analyze the live operational data. In this section, we evaluate the performance of two prominent main-memory DBMS that support mixed workload: SAP HANA [72], and HyPer [101]. By examining their similarities and differences, we aim to identify the factors that affect the performance of mixed workloads while we scale the number of concurrent clients.

To evaluate mixed workloads, we prefer to not use benchmarks aimed for either OLTP or OLAP, such as TPC-C, TPC-W, TPC-H, TPC-DS [14] or OLTP-bench [67], because the mixed workload would work on disjoint datasets (as in our experiment of Section 4.1.5). As a new direction to benchmarking mixed workloads, we employ the CH-benCHmark [59], which considers concurrent OLAP and OLTP clients in a mixed workload on the same dataset, inspired by TPC-C and TPC-H. We find the CH-benCHmark an adequate solution since it allows to scale the number of concurrent transactional and analytical clients independently.

Scaling up mixed workloads. We identify three main factors that affect the performance of mixed workloads while we scale the number of concurrent clients: (a) *data freshness*, (b) *flexibility*, and (c) *scheduling*. In Figure 4.6, we sketch how we expect performance to be affected by these three factors.

Data freshness refers to how recent is the data that is processed by analytical queries. On the one hand, data can be stale, as is the case for typical data warehouses where operational data is periodically replicated. This separation, with a low level of data freshness, allows for various optimizations such as decoupling transactions and analytics, minimizing the interference between them, and having additional materialized views or indexes or shared execution plans (see Chapter 3) for analytics (which may be otherwise expensive to maintain with a high level of data freshness). On the other hand, as the refresh rate is increased, performance is compromised because we need to sustain the overhead of more frequent snapshots of the transactional data for the analytical workload, and respect transactional semantics for the concurrent OLTP workload.

Flexibility refers to the restrictions that a DBMS may impose on the transactional features or expressiveness in order to increase optimization choices to enhance performance. For example, a system can restrict flexibility by requiring that transactions are instantiated from templates that are known in advance, allowing for pre-compilation of transactions [182]. Another example is restricting interactivity, i.e., transactions cannot have multiple rounds of communication with a remote client, which allows optimizing execution [182]. Moreover, it favors techniques like just-in-time (JIT) compilation which, at the expense of a small compilation overhead, can improve the performance of ad-hoc queries [143].

Scheduling determines how, and the order in which transactions and analytical queries use the system's resources, including potential workload management techniques. For cases of

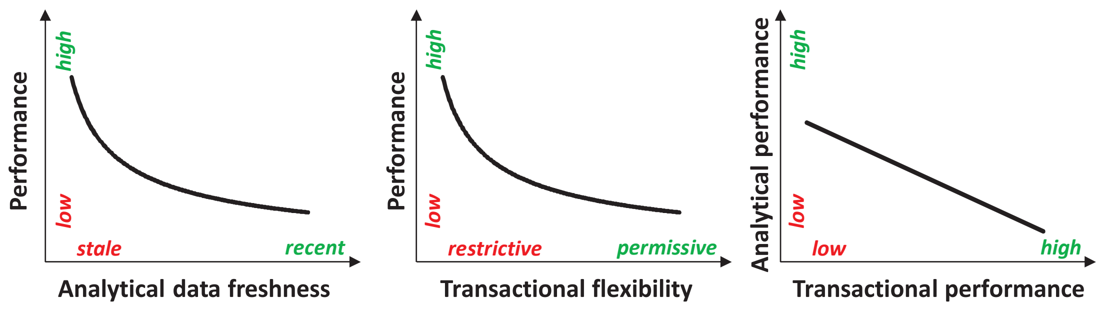


Figure 4.6 – Conceptual figures of how we expect the performance of mixed workloads to be affected by (a) data freshness, (b) flexibility, and (c) scheduling.

high concurrency with numerous OLTP and OLAP clients and a fully saturated system, the DBMS may opt to either favor transactions at the expense of analytical queries, or reversely.

Contributions. In Section 4.2, we survey, evaluate, and compare two state-of-the-art main-memory DBMS for mixed workloads: SAP HANA and HyPer. Through our analysis, we detail how (a) data freshness, (b) flexibility, and (c) scheduling affect the performance of mixed workloads while we scale the number of concurrent clients. The most significant findings of our experimental evaluation are:

- DBMS that maintain separate versions of the operational data for analytics, can suffer a decrease in performance of up to 40% for high refresh rates.
- DBMS which are optimized for the execution of less flexible or less expressive transactions, can achieve up to one order of magnitude better transactional throughput than DBMS optimized for flexible and interactive transactions.
- The absence of workload management in cases of high concurrency, that saturate the system, results in long-running and complex analytical queries overwhelming the system, and significantly hurting the performance of short-lived transactional workloads.

Outline. In sections 4.2.1 and 4.2.2, we describe how SAP HANA and HyPer, respectively, handle mixed workloads. In Section 4.2.3, we describe how we implement the CH-benCHmark. Our experimental evaluation is presented in Section 4.2.4.

4.2.1 Mixed Workloads in SAP HANA

For a review of how SAP HANA handles mixed workloads, please see Section 2.5. Here, we discuss shortly the issues of data freshness for analytical queries, how flexible are transactions, and how scheduling works in SAP HANA.

Data freshness. The fact that both analytical and transactional operations target the same data means that SAP HANA allows analytics to query the most recent version of operational data. As soon as an OLTP operation, e.g., updates data in the delta of a column, the new version is immediately available by the MVCC to upcoming analytical queries. Allowing OLTP and OLAP to target common data, however, comes with the cost of synchronization for the common data structures, such as the index of the delta's dictionary (see Section 2.5).

Flexibility. SAP HANA supports fully interactive ACID transactions [113], which can contain multiple round-trips to the client. Efficient and flexible support for distributed transactions is available. Upon first execution, queries are compiled and the cached plan is available in subsequent invocations of the same query. It supports multiple interfaces, including SQL and specialized languages [72].

Scheduling. SAP HANA employs a pool of threads is employed for servicing network clients and short-running transactions and queries, and our task scheduler for servicing analytical queries (see Section 4.1). Analytical queries are expressed as single tasks or as multiple tasks (intra-query parallelism) which are dispatched to the task scheduler. When the server is fully saturated, scheduling decides how transactions and analytical queries utilize resources. We show that the default configuration of SAP HANA favors analytical throughput over transactional throughput. By decreasing parallelism of analytical queries, however, we can increase transactional throughput to the detriment of analytical throughput (see Section 4.2.4).

4.2.2 Mixed Workloads in HyPer

HyPer is a research prototype main-memory relational DBMS that supports mixed OLTP and OLAP workloads [101]. The aim is to support high OLTP throughput, as well as efficient concurrent execution of OLAP workloads. The storage engine can be configured to be a row-store or a column-store. We use the column-store configuration.

OLTP clients are serviced serially with a single thread [101]. This avoids the usage of locks or latches for data structures, and, due to the absence of I/O, allows transactions to be executed in one-shot, uninterrupted and efficiently. Multiple threads for OLTP are supported if the schema is manually partitioned or the server supports hardware transactional memory [116]. We use the default single-threaded behavior.

For serving OLAP clients, HyPer uses an innovative way to provide snapshots of operational data. As shown in Figure 4.7a, OS- and hardware-supported virtual memory facilities are leveraged to create snapshots. Each arriving OLAP client forks the main OLTP process into another process, getting a virtual memory snapshot to work on. The lazy copy-on-update strategy ensures that a virtual page is not physically replicated, and OLTP and OLAP are reading the same physical page. The OS creates a new physical copy only in the case a transaction modifies a page. In this case, the parent OLTP process has the latest version, and the OLAP process refers to the older version of the page. The capability to update OLAP snapshots on demand in a single system is far more efficient than the usual two system setup (one for OLTP and one for OLAP), since data does not need to be replicated from system to the other.

Data freshness. Conceptually, HyPer's main OLTP process is similar to SAP HANA's delta and the OLAP processes are similar to versions of the main. Forking is similar to the merge operation. In contrast to SAP HANA, analytical queries read their snapshot and not the freshest data from the OLTP process. This allows decoupling of OLTP and OLAP, and synchronization overhead is avoided. Also, since the OLAP client can update its snapshot on demand, data freshness is customizable: on the one hand, the client can opt to take a snapshot and never update it, or, on the other hand, update its snapshot after every couple of queries. The downside of this tactic, however, compared to SAP HANA, is that, in the case that OLAP clients

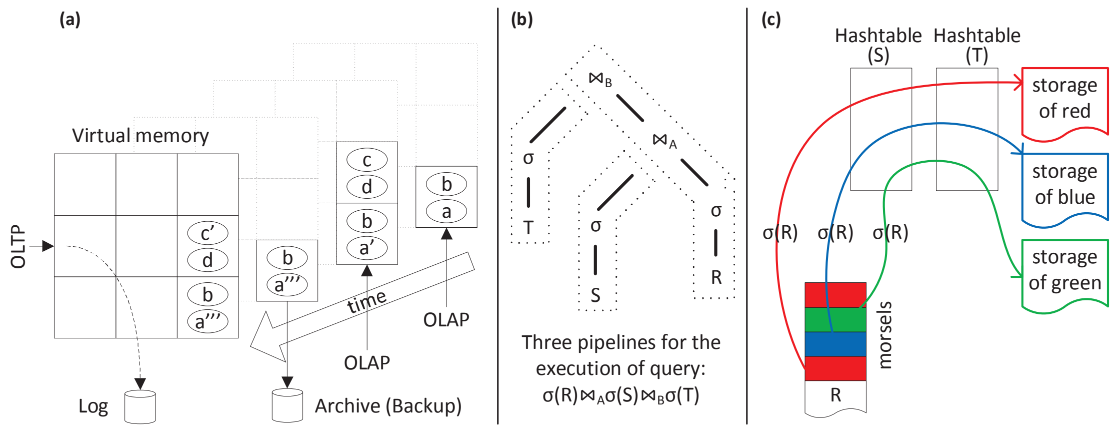


Figure 4.7 – (a) HyPer design using virtual memory facilities to create snapshots for analytics [101]. (b) Restricting flexibility to support further optimizations of query plans [117]. (c) Conceptual figure of scheduling for pipeline R of the query plan of (b) [117].

wish to keep their snapshots as fresh as possible, the virtual memory snapshot overhead is increased (see Section 4.2.4).

Flexibility. HyPer is optimized for the execution of prepared statements or precompiled transactions [101]. Ad-hoc queries and ACID transactions are both supported and compiled by a just-in-time (JIT) compiler. The overhead of the elaborate compilation may be amortized for multiple invocations of the same query or transaction, but can limit the scalability of short-lived ad-hoc OLTP.

HyPer restricts flexibility for clients on purpose to allow for further optimizations. For example, clients need to define if they are OLTP or OLAP clients. Also, for an OLTP client, the whole client transaction is performed in a single batch, i.e. there cannot be multiple round-trips to a client in a transaction. The restrictions for OLTP clients allow for, e.g., analysis of the transaction, regarding the accessed and updated tables, or the control-flow [143]. These optimizations, along with the serialization of transactions, can achieve significantly higher OLTP throughput (see Section 4.2.4). Read-only OLAP clients access a read-only snapshot; ad-hoc queries are fully supported because they are compiled as they arrive on the database server.

As shown in Figure 4.7b, the query plans created by the optimizer are composed of operator pipelines through which tuples are pushed. Pipelines are broken by operators that cannot be pipelined (e.g., a sort). By pushing tuples through a whole pipeline of several operators, performance can be significantly improved with JIT compilation, better data locality, and predictable branch layout [143].

Scheduling. HyPer includes a NUMA-aware (non-uniform memory access) task scheduler for queries. Each phase of a query is parallelized, and the scheduler takes care to distribute work evenly across sockets, using task stealing and elastic parallelism, and optimize for data locality [117]. In Figure 4.7c, we show an example of how the scheduler executes the probe phases of the hash-joins of pipeline R (of the query of Figure 4.7b), using three of the sockets of a server (depicted in different colors). Relation R is partitioned into small fragments, called *morsels*. A thread continuously takes a morsel from relation R, local to its socket, and passes it through the pipeline, probing the hash tables for relations S and T, finally storing locally the result. In comparison to SAP HANA, we show in our experiments (see Section 4.2.4), that HyPer's scheduling also favors analytical throughput over transactional throughput in cases of high concurrency and saturation.

4.2.3 Setting Up the CH-benCHmark

The CH-benCHmark builds upon the widely used TPC-C and TPC-H benchmarks [14]. TPC-C is used to analyze the performance of transactional workloads in a scenario of order processing, while TPC-H analyzes the performance of analytical workloads in the context of a wholesale supplier. The goal of the CH-benCHmark [59] is to combine TPC-C and TPC-H in a unified schema, in order to analyze the performance of the mixed OLTP and OLAP workload. Next, we give an overview of the CH-benCHmark, and how we adapt it.

Overview of the benchmark. The database schema of the CH-benCHmark is shown in Figure 4.8. The schema uses the nine tables of TPC-C and adds the tables `NATION`, `REGION`, and `SUPPLIER` from TPC-H. As in TPC-C, the size of the database scales with the number of warehouses. The integrated schema has the following changes over TPC-H and TPC-C:

- `NATION` contains 62 rows instead of 25, and `SUPPLIER` is fixed to 10,000 rows.
- `CUSTOMER` and `NATION` can be joined on columns `N_NATIONKEY` and `C_STATE`. Column `C_STATE`, however, is defined as a two-character code while `N_NATIONKEY` is defined as integer. To solve this mismatch, the following join condition was proposed in the original definition of the CH-benCHmark: `NATION.N_NATIONKEY = ASCII(SUBSTR(CUSTOMER.C_STATE,1,1))`. This is also the reason for increasing the number of entries in `NATION` from 25 to 62.
- Similarly, `SUPPLIER` and `STOCK` can be joined using the following condition: `SUPPLIER.SU_SUPPKEY = MOD(STOCK.S_W_ID * STOCK.S_I_ID, 10000)`.

Regarding the workload, the CH-benCHmark uses the five transactions defined in TPC-C for the OLTP workload. In contrast to TPC-C, an OLTP client randomly chooses a warehouse, and there is no correlation between the number of warehouses and the number of clients.

The OLAP workload is based on the 22 queries defined in TPC-H, but adapted to the modified schema (see Figure 4.8). A client either executes the OLTP workload or the OLAP workload. Hence, the number of clients for each type of workload can be scaled independently.

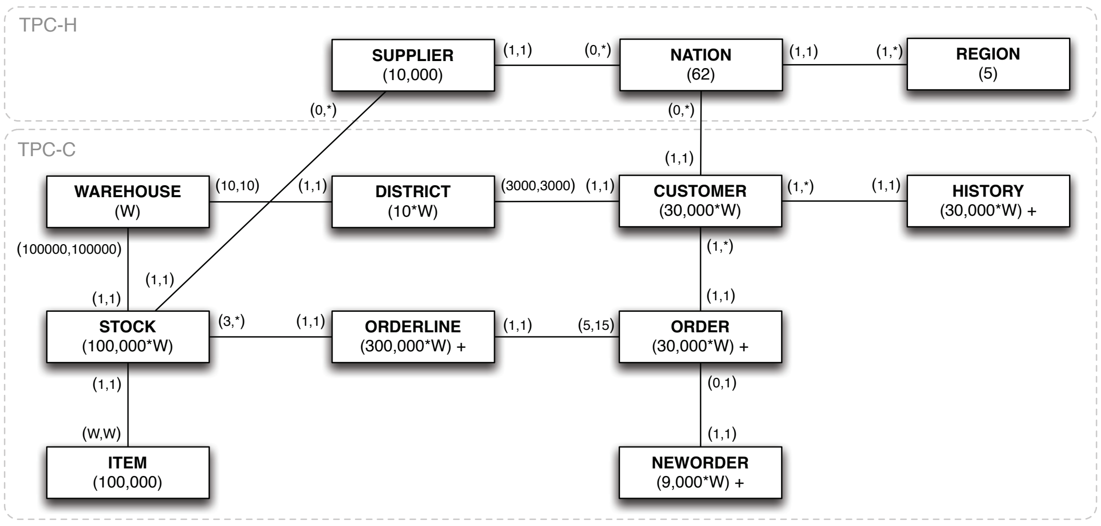


Figure 4.8 – Schema of the CH-benCHmark.

Adjusting the benchmark. The schemas of TPC-C and TPC-H were originally integrated in an ad-hoc fashion using expressions in the join conditions of the queries. For foreign-key relationships, as defined between tables `CUSTOMER` and `NATION` as well as tables `SUPPLIER` and `STOCK`, real-world schemas would avoid such expressions. This leads us to the decision to materialize the join expressions explicitly in the database because it allows us to use standard equi-joins for queries joining these tables. Thus, we introduce the following:

- A column `C_N_NATIONKEY` in table `CUSTOMER` computed as `ASCII(SUBSTR(CUSTOMER.C_STATE,1,1))`.
- A column `S_SU_SUPPKEY` in table `STOCK` computed as `MOD(STOCK.S_W_ID * STOCK.S_I_ID, 10000)`.

4.2.4 Experimental Evaluation

In this section, we evaluate and compare SAP HANA and HyPer (versions of June 2014) using the CH-benCHmark. First, we detail the experimental configuration, how we setup each system, and the performance metrics we measure. Then, we present the results of the experimental evaluation for SAP HANA and HyPer, while detailing the implications of the results: how data freshness, flexibility, and scheduling affect the performance of mixed workloads.

Experimental configuration. To execute the CH-benCHmark, we use the server of Figure 2.5. It has eight ten-core processors Intel Xeon E7-8870 at 2.40GHz, with hyper-threading enabled (for a total of 160 hardware threads), and 1TB of RAM. The OS is a 64-bit SMP Linux (SuSE), with a 3.0 kernel. We use "read committed" for the isolation level of SAP HANA (see Section 2.5). HyPer executes transactions using timestamp ordering with a single thread.

We connect to SAP HANA via the ODBC interface. We use SQL prepared statements for both OLTP and OLAP. Transactions are fully interactive and managed by standard ODBC calls. For HyPer, we use its available client interface, as it does not offer an ODBC interface. We use pre-compiled statements for OLTP (non-interactive), and send SQL statements for the OLAP workload. The caching of OLAP query plans in HyPer is similar to using prepared statements.

In our experiments we use a one minute warm-up period, followed by a five minutes period to collect throughput information. We use 100 warehouses, which amount to 6.7 GB of raw CSV files to be imported. We note that we observe similar trends for a higher number of warehouses. We are, however, more interested in assessing the scalability of concurrency than the increase in data size. We scale the number of OLAP and OLTP clients exponentially (power of 2) between 0 and 2^7 leading to 81 different combinations. Since the result of combination 0/0 is trivial, we are left with 80 combinations for the clients of the mixed workload.

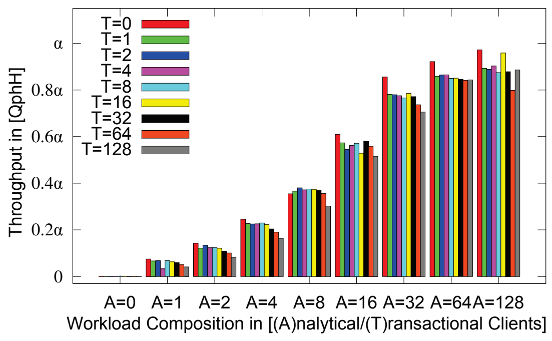
For an OLTP client, the benchmark reports throughput in tpmC, as defined in TPC-C, i.e., the number of successful new order transactions per minute. For an OLAP client, throughput is reported in QphH, i.e., the finished TPC-H queries per hour. Defining an aggregated metric for the whole benchmark is difficult in practice, and thus we follow the original benchmark proposal and analyze both measures separately.

For each system, we present a figure showing the analytical throughput of all combinations of OLTP and OLAP clients, and another figure showing the transactional throughput of all combinations. In this pair of plots, each experiment is displayed twice. As an example we refer the reader to Figure 4.9, where the black bar (T=32) in section A=8 represents a single experiment with 8 analytical (OLAP) and 32 transactional (OLTP) clients. We also measure the average CPU utilization of the host server as we increase the load.

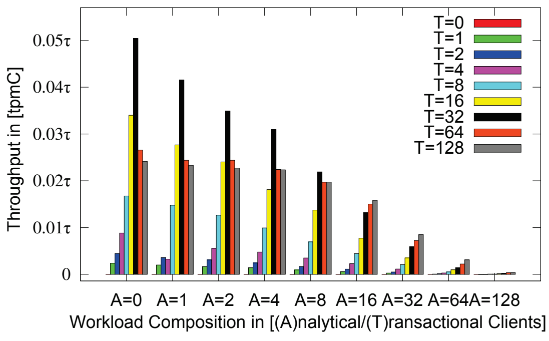
Due to legal reasons, we do not disclose absolute numbers. For this reason, all throughput results are normalized to undisclosed constants α for OLAP and τ for OLTP, where α and τ are the maximum observed throughput values for OLAP and OLTP respectively. This does not hinder us from showing the implications of our experiments, because our focus is on the scalability of the mixed workload as we increase the number of clients, and comparing SAP HANA and HyPer as to how they handle mixed workloads.

Experimental evaluation of SAP HANA. Figure 4.9 shows the performance of the default configuration of SAP HANA as we scale the mixed workload. Figure 4.9a shows how analytical throughput scales as we increase the number of analytical clients. For each case of analytical clients, we also show how analytical throughput scales as we increase the number of transactional clients. As shown in the figures, analytical throughput increases almost linearly up to 32 analytical clients. After that, as the system gets saturated (see Figure 4.9c), the increase of throughput levels out.

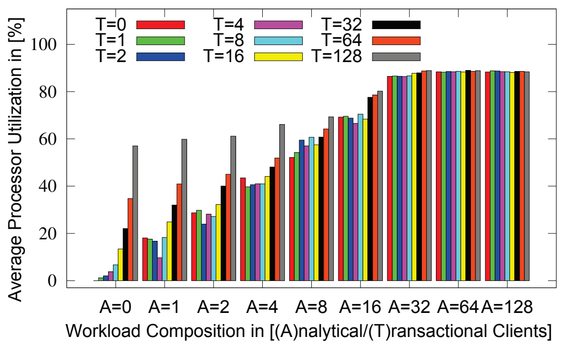
Figure 4.9b demonstrates the scaling behavior of the transactional throughput as we increase the number of analytical clients. For a small number of concurrent OLAP clients (up to 8),



(a) Analytical throughput



(b) Transactional throughput



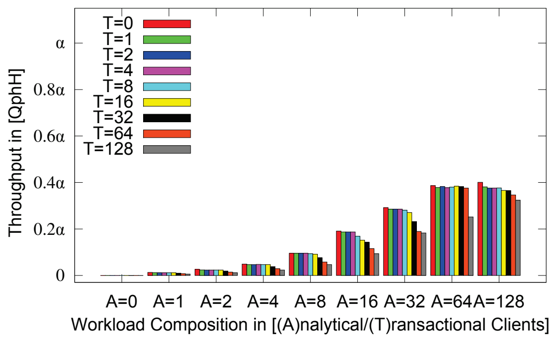
(c) Average CPU utilization

Figure 4.9 – Performance of the default configuration of SAP HANA.

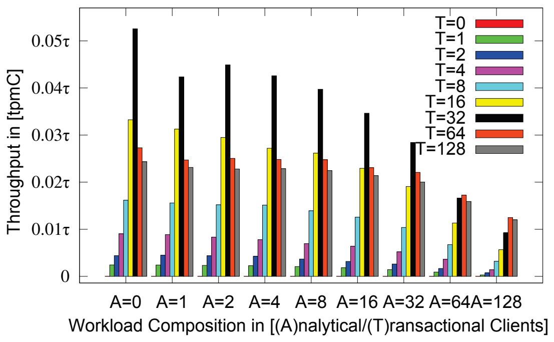
transactional throughput generally increases as we increase the number of OLTP clients up to 32, after which, OLTP throughput drops. This is due primarily to the fact that more and more transactions contend for modifying common data, resulting in higher abort rates, and, secondarily, in increased synchronization overhead (in the latches of the deltas' indexes). As we add more OLAP clients, overall transactional throughput is generally hurt, as it almost reaches zero throughput for the case of 128 concurrent analytical clients.

We call this scaling behavior the *house pattern*, due to the increasing OLAP throughput and the decreasing OLTP throughput as we increase the number of OLAP clients. This effect is intrinsic to the behavior of not distinguishing between short-lived transactions and complex analytical queries. The scheduler of SAP HANA employs the server's resources for analytical queries for long durations, and does not leave enough space for the continuously arriving short-lived OLTP transactions. As we add more OLAP clients, overall OLTP throughput decreases.

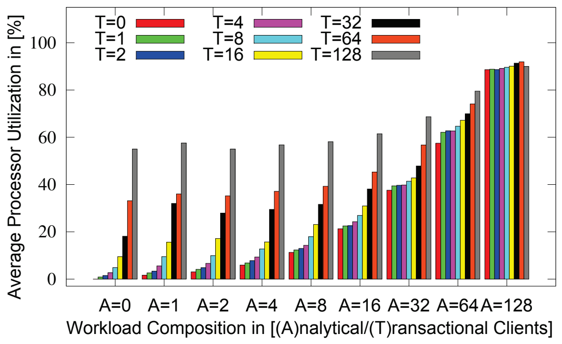
To reinforce our argument, we evaluate SAP HANA under a configuration which disables intra-query parallelism, and decreases the effect of analytical queries overwhelming execution. Figure 4.10 shows the results. OLTP transactions and OLAP queries are mostly executed with a single thread (or task) each. OLAP throughput is overall lower than the default configuration, since queries do not benefit from parallel execution any more. Still, OLAP throughput increases



(a) Analytical throughput



(b) Transactional throughput



(c) Average CPU utilization

Figure 4.10 – Performance of SAP HANA when intra-query parallelism is disabled.

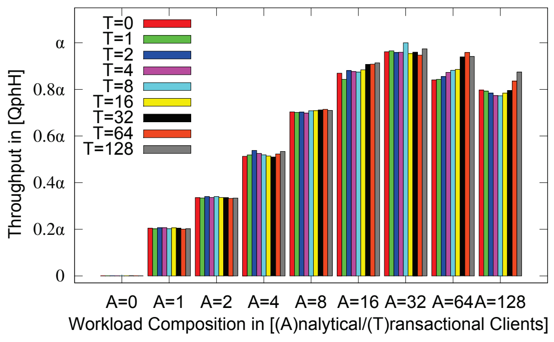
as we increase the number of OLAP clients. The positive effect is that OLTP throughput is overall improved in comparison to the default configuration. System utilization is lower than the default configuration, and is only saturated for 128 analytical clients.

Experimental evaluation of HyPer. In Figure 4.11 we show the experimental results for the most performant case of HyPer. In this case, we keep the initial snapshot for OLAP clients throughout the whole experiment duration, i.e., OLAP clients do not see any updates from the OLTP clients. This configuration minimizes the overhead of creating snapshots, and minimizes any interference between the OLTP and OLAP workloads.

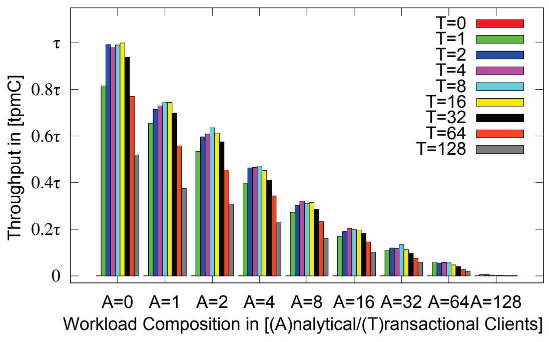
As we see in Figure 4.11a, the analytical throughput increases as we add more analytical clients, reaching the maximum at around 32 analytical clients. Additional analytical clients drop analytical throughput slightly, due to overwhelming the system with threads. Limiting the overall number of used threads, similar to SAP HANA's task scheduler, can avoid this effect. In comparison to SAP HANA, analytical throughput reaches almost the same maximum, indicating that both systems are similar in parallelizing and executing analytical queries. Also, analytical throughput is not affected by scaling the transactional clients. This is expected, since transactions are executed separately with a single thread.

Transactional throughput, as shown in Figure 4.11b, is significantly higher (up to an order of magnitude) than that of SAP HANA. This is attributed to several reasons including: (a) transactions are non-interactive whereas transactions in SAP HANA are interactive (with multiple round-trips to the client as defined in TPC-C), (b) transactions are pre-compiled for fast execution, and (c) a single thread executes transactions serially, avoiding any synchronization overhead. Conceptually, we can place HyPer to the left-most part of Figure 4.6b, and place SAP HANA to the right-most part of the figure.

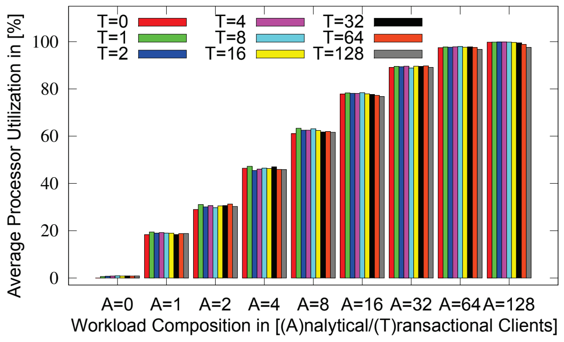
The trend of the OLTP throughput, however, is similar to SAP HANA. Firstly, we notice a similar drop in throughput for more than 32 OLTP clients, for most experiments. As with SAP HANA, numerous OLTP clients target common data, and result in high abort rates. Secondly, we also identify the same *house pattern* as in SAP HANA: while we increase the number of analytical clients, overall OLTP throughput drops and reaches almost zero for the case of 128 concurrent OLAP clients. Both SAP HANA and HyPer fall in the left-most part of Figure 4.6c: under cases of high concurrency and saturated resources, the scheduler favors analytics over transactions. This shows a need for advanced workload management for mixed workloads, that can enable the DBMS administrator to dynamically tip the scales of performance to either analytics or transactions, choosing a spot across the whole span of the line of Figure 4.6c.



(a) Analytical throughput



(b) Transactional throughput



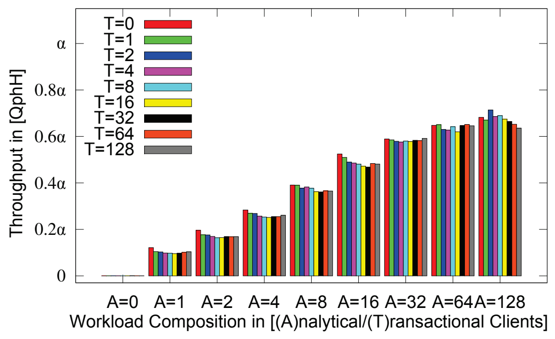
(c) Average CPU utilization

Figure 4.11 – Performance of HyPer with the lowest level of analytical data freshness.

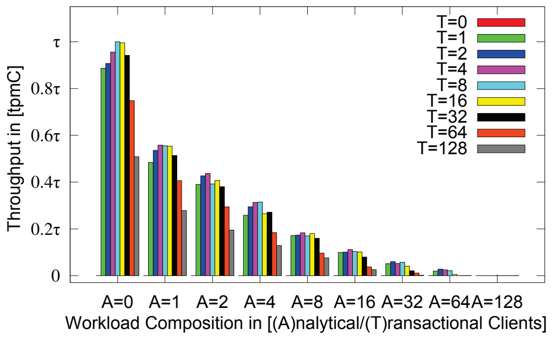
Next, we show how the performance is affected by a different level of data freshness. Figure 4.12 shows the performance of an intermediate level of data freshness, where every OLAP client takes a new snapshot from the OLTP process after executing all queries of TPC-H (after every 22 queries). Performance is overall decreased in comparison to the best performant case of the lowest level of data freshness, supporting our expectations (see Figure 4.6a). Analytical throughput is decreased by around 40%. Transactional throughput is decreased as soon as the first OLAP client is added, by around 30%. This is mainly due to the overhead of forking the OLTP process to create snapshots for OLAP clients. It actually interrupts the single-threaded OLTP process and presents an overhead.

We note that increasing the level of data freshness further is not desirable in HyPer because the extremely frequent forks at a fine granularity can significantly deteriorate performance. In such cases, a sort of "snapshot bundling" could be implemented to decrease the snapshot overhead at a small expense of data freshness: instead of every OLAP client forking the OLTP process, several OLAP clients can be batched and serviced on a single snapshot.

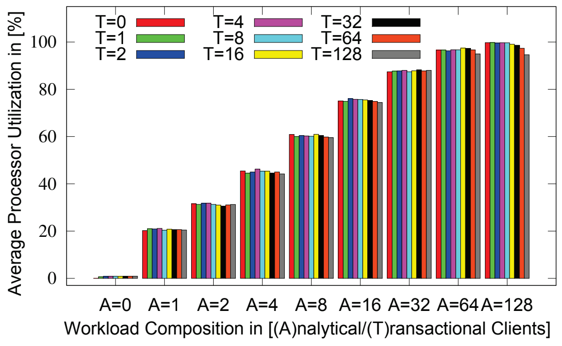
While SAP HANA aims for the highest level of data freshness, HyPer provides the opportunity to the DBMS administrator to choose the level of data freshness for analytics. This is a desirable property when it is acceptable not to consider the latest updates in reports. For cases where



(a) Analytical throughput



(b) Transactional throughput



(c) Average CPU utilization

Figure 4.12 – Performance of HyPer with an intermediate level of analytical data freshness.

extreme real-time reporting is required, SAP HANA's approach to executing both OLTP and OLAP workloads on common data structures can be better for analytical throughput.

4.3 Summary and Conclusions

In this chapter, we show how task scheduling can be practically employed in a main-memory DBMS. As tasks can use synchronization primitives, we show that the concurrency level of the task scheduler should not be fixed, but be flexible. When worker threads block, more worker threads should be issued, giving control of additional worker threads to the OS scheduler to saturate CPU resources. Furthermore, for partitionable analytical operations, we observe that task granularity can significantly affect scheduling costs in cases of high concurrency. For this reason, our scheduler gives a concurrency hint to the task creators of partitionable operations, reflecting the level of CPU contention. Using this hint, partitionable operations re-adjust their task granularity, to avoid excessive scheduling costs for high concurrency.

Furthermore, we shortly extend the scope of this thesis to mixed workloads and evaluate the performance of two state-of-the-art main-memory DBMS for mixed workloads: SAP HANA and HyPer. We evaluate the CH-benCHmark by scaling the number of concurrent transactional and analytical clients. Through our evaluation, we find that the most important factors that affect the performance of mixed workloads are (a) data freshness, i.e., how recent is the data that analytical queries are processing, (b) flexibility, i.e., optimizing the performance of transactions and queries by restricting interactivity and/or expressiveness, and (c) scheduling, i.e., how the DBMS utilizes resources for OLTP and OLAP clients.

Concerning data freshness, SAP HANA's design, where OLTP and OLAP clients target common data, is suited for cases where the highest level of data freshness is required, whereas HyPer's design is suitable for cases where the DBMS administrator wishes to toggle the trade-off between performance and data freshness. Concerning flexibility, we show that HyPer's less interactive statements allow for pre-compilation and achieve a very high transactional throughput. Finally, concerning scheduling, we show that both systems exhibit a "house pattern", i.e., increasing OLAP clients can significantly hurt OLTP throughput in cases of high concurrency and saturated resources. This behavior stresses the need for workload management in mixed workloads, where OLTP statements can be distinguished from OLAP statements and can be prioritized differently. The house pattern has been a motivation for the support of workload management features in SAP HANA [8].

Overall, this chapter shows that task scheduling is a technique that helps the DBMS decouple its scheduling from the OS. Intra-query parallelism can be achieved without unnecessary context switches due to avoiding overcommitting CPU resources. Meanwhile, task scheduling still allows opportunistic cooperation with the OS when needed, e.g., for supporting a flexible concurrency level. In the next chapter (see Chapter 5), we show how the data structures of the task scheduler can easily reflect the underlying topology of a multi-socket server, and support various NUMA-aware task scheduling and data placement strategies.

5 NUMA-Aware Task Scheduling and Data Placement Strategies

In this chapter, we analyze the performance of NUMA-aware task scheduling and data placement strategies for main-memory column-stores, using a prototype based on SAP HANA. Our analysis first focuses on the basic case of concurrent scans and is afterwards extended to aggregations and equi-joins. The two main insights of our analysis are that unnecessary data partitioning and inter-socket stealing of memory-intensive tasks involve a significant overhead. In Chapter 6 we use these two insights to develop a data placement and task scheduling strategy that adapts to the workload at run-time in order to balance utilization across sockets.

Publications. Parts of this chapter have been published in [161, 163].

5.1 Introduction

NUMA introduces new performance challenges for main-memory column-store DBMS, as communication costs vary across sockets and the bandwidth of the interconnect links is an additional bottleneck to be considered (see Section 2.5). The DBMS needs to become NUMA-aware by handling the placement of its data structures across sockets, and scheduling the execution of queries accordingly onto the sockets. Figure 5.1a shows the performance difference between a NUMA-agnostic and a NUMA-aware column-store as an increasing number of analytical clients issues scan-heavy queries, on the 4-socket server of Table 2.2. See Section 5.4 for more details on the experimental configuration. In this scenario, NUMA-awareness significantly improves throughput, by up to 5x. By avoiding inter-socket communication, the memory bandwidth of the sockets can be fully utilized, as shown in Figure 5.1b.

In the literature, there has been a recent wave of related work for NUMA-aware analytical DBMS (see Section 2.4.2). The majority employs a static strategy for data placement and scheduling. For example, HyPer [117] and ERIS [103] partition data across sockets and parallelize execution with a task scheduler. Each worker processes local tasks or steals tasks from other workers. The trade-offs between different data placement and task scheduling strategies have not been extensively analyzed yet.

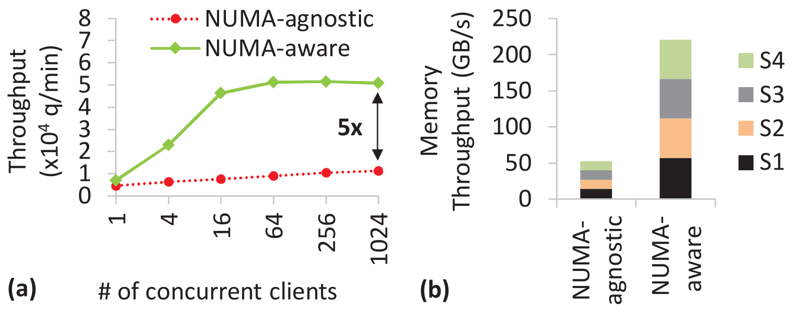


Figure 5.1 – (a) Impact of NUMA on the performance of concurrent clients issuing scan-heavy queries. (b) Memory throughput of the sockets for the case of 1024 clients.

Contributions. In this chapter, we describe and implement data placement and task scheduling strategies for main-memory dictionary-encoded column-stores. Through a sensitivity analysis, based on a prototype of SAP HANA, we identify the trade-offs for each strategy under various workload parameters. The main insights of our analysis are:

- We show that unnecessary partitioning can hurt throughput by up to 40% in comparison to not partitioning. Partitioning should be used for hot data when the workload is skewed, until socket utilization is balanced.
- We show that inter-socket stealing of memory-intensive tasks can hurt throughput by up to 15%. Memory-intensive tasks should not be stolen across sockets.

Outline. In Section 5.2, we detail the implementation and implications of data placement strategies. We present our NUMA-aware task scheduler in Section 5.3, and how concurrent scans are scheduled. Section 5.4 includes our sensitivity analysis of concurrent scans. Section 5.5 extends our analysis to aggregations and equi-joins. Finally, Section 5.6 contains the conclusions of this chapter.

5.2 Data Placement of a Main-Memory Column

Before continuing, please see Section 2.5 for an overview of the basic data structures of a main-memory dictionary-encoded column. Next, we describe three data placement strategies for these data structures, shown in Figure 5.2. In Table 5.1, we summarize which workload properties best fit each data placement and a few of their key characteristics.

Round-robin (RR). The simplest data placement is placing a whole column on a socket. This means that queries wishing to scan this column, or do index lookups, should run and parallelize within that socket to keep memory accesses local. A simple way to exploit this data placement for multiple columns is to place columns on sockets in a round-robin way.

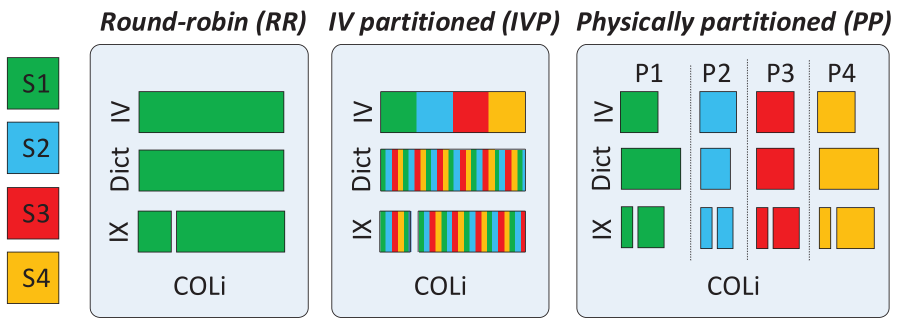


Figure 5.2 – Different data placements of a dictionary-encoded column on four sockets.

As we show in our experiments, RR is not performant for low concurrency, because a query cannot utilize the whole server, or for skewed workloads, because placing more than one hot column on a socket creates a hotspot on that socket. Additionally, our evaluation shows that for high concurrency, query latencies suffer a high variation in comparison to the following partitioning strategies.

Indexvector partitioning (IVP). To overcome the aforementioned negative implications of RR, we present a novel data placement, especially for scans, that partitions the IV across the sockets. This can happen quickly and transparently by using, e.g., `move_pages` in Linux, to change the physical location of the involved pages without affecting virtual memory addresses. A scan can be parallelized within the socket of each part, potentially using all sockets.

The disadvantage of IVP is that there is no clear choice how to place the dictionary or the IX. Unless the column has sorted values, the ordering of the *vid* in the IV does not follow the same ordering as the *vid* of the dictionary and the IX. Thus, we interleave them across the sockets, in order to average out the latency of memory accesses during materialization (converting qualifying *vid* to real values from the dictionary – see Section 5.3.1) and during index lookups.

As we show in the experiments, the disadvantage of IVP results in high-selectivity scans and index lookups suffering decreased performance. Although a high-selectivity scan can scan the parts of the IV locally, the dominating materialization phase involves potentially numerous remote accesses to the interleaved memory of the dictionary. Similarly, index lookups suffer from remote accesses to the interleaved index.

Physical partitioning (PP). To overcome the limitations of IVP, we can opt for an explicit physical partitioning of the table. PP can use a hash function on a set of columns, a range partitioning on a column, or a simple round-robin scheme [8, 105]. The table and its columns are split into the *table parts (TBP)* defined by the partitioning specification. PP is useful for improving the performance through pruning, i.e., skipping a part if it is excluded by the query predicate, and for moving parts in a distributed environment (see, e.g., SAP HANA [8] and Oracle [105]). In this thesis, we use PP to place each part on a different socket. Since we wish to evaluate NUMA aspects, we avoid exploiting the pruning capability in our sensitivity analysis.

Table 5.1 – Workload properties best fitted for each data placement, and key characteristics.

Data placement	Concurrency	Selectivities	Workload distribution	Latency distribution	Memory consumed	Readjustment
RR	High	All	Uniform	Unfair	Normal	Quick
IVP	All	Low (w/o index) & medium	Uniform & skewed	Fair	Normal	Quick
PP	All	All	Uniform & skewed	Fair	Potentially lower/higher	Slow

The advantage of PP is that each part of a column can be allocated on a single socket. A scan is split into each part. The materialization phase for each part takes the qualifying *vid* of the scan and uses the local dictionary to materialize the real values. In contrast to IVP, PP is thus performant for high-selectivity queries and index lookups as well.

The disadvantages of PP are three-fold. First, PP is heavy-weight and time-consuming to perform or repartition. The DBMS needs to recreate the components of all parts of the columns. The second disadvantage of PP is its potentially increased memory consumption. Although it results in non-intersecting IV across the parts of a column, the dictionaries and the IX of multiple parts may have common values. For large data types, e.g., strings, this can be expensive. There are, however, some cases when the memory consumption can decrease, and this becomes an advantage for PP. If the unique values in a part decrease substantially, the bitcase of the part can decrease. This can happen if the values are correlated to the partitioning scheme, e.g., if the values are sorted according to the PP range partitioning scheme. The third disadvantage is that analytical operations requiring to match values across the parts need a pre-processing phase to match the potentially different *vid* of the same real value across the parts. This disadvantage does not apply to scans, but applies to aggregations with a group-by and to equi-joins (see Section 5.5).

Other data placements. We note that the aforementioned data placements are not exhaustive. For example, one can interleave columns across a subset of sockets. Or, one can replicate some or all components of a column on a few sockets, at the expense of memory. Replication is an orthogonal issue. The three basic data placements we describe are a set of realistic choices. More importantly, through our experiments in this chapter, we show how their basic differences affect the performance of different workloads, and motivate a design that adapts the data placement to the workload at run-time.

5.2.1 Tracking Memory

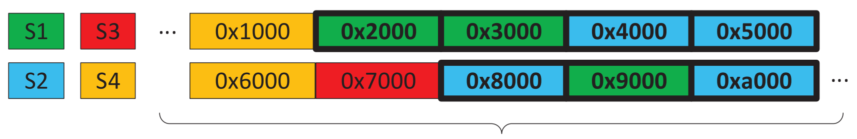
We need a way to expose a column's data placement. For *RR* and *PP*, we can simply use a socket identifier per column and partition. For *IVP*, however, a column's component (IV, dictionary, or IX) may exist on multiple sockets. For this reason, we design a novel data structure, *Page Socket*

Mapping (PSM), that summarizes the physical location of virtual address ranges. Figure 5.3 shows an example of a PSM. The figure depicts a piece of virtual memory consisting of ten 4 KB pages. Each box includes the base address of each page. The color signifies the socket where the page is physically allocated. Assume that we wish to track the physical location of virtual address ranges `[0x2000, 0x6000)` and `[0x8000, 0xb000)`. This example can represent a tiny column, without an index, placed with IVP, where the first range holds the IV, partitioned across sockets S1 and S2, and the second range holds the interleaved dictionary.

The PSM maintains an internal vector of *ranges*. Each range consists of a virtual page address, the number of subsequent pages, and the socket where they are physically allocated. If the range is interleaved, the interleaving pattern signifies the participating sockets, and the socket number denotes the starting socket. The ranges are sorted by the virtual address of their base page. We choose a vector of ranges to optimize for reading the PSM instead of updating it. Looking up the physical location of a pointer includes a quick binary search on the ranges' first pages, and, in case the range is interleaved, following the interleaving pattern. Furthermore, we maintain another vector that summarizes the number of pages on each socket.

When we add virtual address ranges to the PSM, it maps them to page boundaries, checks which pages are not already included, and calls `move_pages` on Linux to find out their physical location. The algorithm goes through their physical locations, collapsing contiguous pages on the same socket into a new range for the internal vector. It detects an interleaving pattern when every other page is allocated on a different socket, following the same recurring pattern. When the pattern breaks, the new interleaved range is inserted in the internal vector, and the algorithm continues. The summary vector is updated accordingly.

PSM objects support additional useful functionality: we can remove memory ranges, ask for a subset of the metadata in a new PSM, and get the socket where the majority of the pages are. We can also move a range to another socket or interleave it. The PSM uses `move_pages` to move the range, and update the internal information appropriately.



Vector of ranges	#1	#2	#3
First page address (64 bits)	0x2000	0x4000	0x8000
Number of pages (32 bits)	2	2	3
Socket (8 bits)	1	2	2
Interleaving pattern (256 bits)	0000...	0000...	1100...

Summary: pages per socket (256 · 32 bits)	3	4	0	0	···

Note: for simplicity, we display only the last 4 of the 16 hexadecimal characters of 64-bit addresses.

Figure 5.3 – Example of a PSM after adding the virtual memory ranges to track (bold lines).

The space used by a PSM depends on the number of stored ranges. For the indicative sizes of Figure 5.3, we assume that a range can contain a maximum of 2^{32} pages (or 16 TB for 4 KB pages), and that a server can have up to 256 sockets. The size of a PSM is $360 \cdot r + 8192$ bits, where r is the number of stored ranges. Let us examine the size of the metadata for a column on a 32-socket server, assuming we attach a PSM to the IV, dictionary, and IX of a column so that we can query their physical location.

If a column is placed wholly on a socket, then $r = 1$ for the IV and the dictionary, and $r = 2$ for the IX (contains 2 vectors). The metadata is 26016 bits, or 3 KB. If a column is placed with IVP across all sockets, then $r = 32$ for the IV, $r = 1$ for the interleaved dictionary, and $r = 2$ for the interleaved IX. The metadata is 37176 bits, or 5 KB. If a column is physically partitioned, with 32 parts, each part is wholly placed on a socket. The metadata is around 102 KB. The size of the metadata is not large compared to the typical sizes of columns (at least several MB). We note that one can decrease the space substantially by losing some accuracy and the capability of querying specific virtual addresses by keeping only the summary vector.

5.3 NUMA-Aware Task Scheduling

In this section, we describe how we modify our initial task scheduler presented in Section 4.1 to support NUMA-aware task scheduling. We then outline the different task scheduling strategies for concurrent scans, considering also the data placement strategies of Section 5.2.

Tasks need to be able to choose the socket to run on, and the task scheduler needs to expose the topology of the multi-socket server. Figure 5.4 depicts the design of our NUMA-aware task scheduler. Upon initialization, the scheduler divides each socket into one or more *thread groups*. Small topologies are assigned one thread group per socket, while larger topologies are assigned a couple of thread groups per socket. Hyperthreads (denoted HT in Figure 5.4) are grouped in the same thread group. Figure 5.4 depicts the thread groups for a socket of the 4-socket server of Table 2.2. The main purpose of multiple thread groups per socket is to decrease potential synchronization contention for the contained task priority queues.

Inside a thread group. Each thread group contains two priority queues for tasks. The first has tasks that can be stolen by other sockets. The second has "bound" tasks that have a *hard affinity* and can only be stolen by worker threads of thread groups of the same socket. As our experimental evaluation shows, supporting bound tasks is essential for memory-intensive workloads. The priority queues are protected by a lock. Lock-free implementations for approximate priority queues [31] can be employed for cases of numerous short-lived tasks where synchronization becomes a bottleneck. Each thread group maintains a number of worker threads, which are distinguished as active or inactive. These worker threads are handled similarly to Section 4.1.3 for maintaining the active concurrency level. The task scheduler keeps the active concurrency level of each thread group equal to the number of its H/W threads.

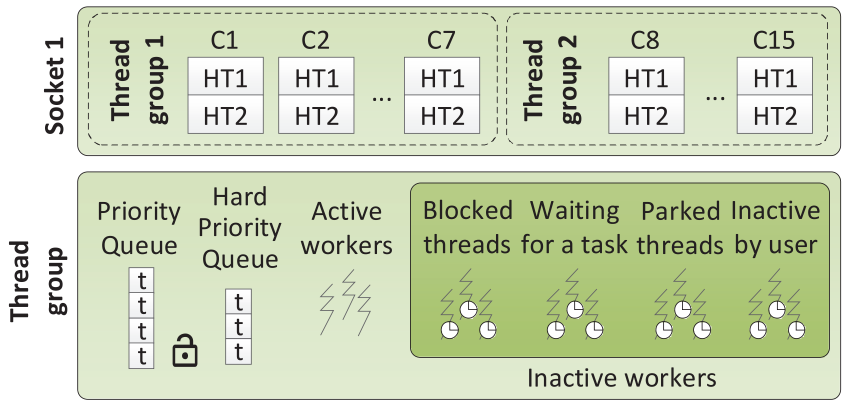


Figure 5.4 – The design of our NUMA-aware task scheduler.

Main loop of a worker thread. The worker firstly checks that it is allowed to run, by checking that the number of working threads is not larger than the number of the H/W threads of the thread group. If it is, the worker parks itself (see Section 4.1.3). If it is allowed to run, it peeks in the two priority queues to get the element with the highest priority. If there are no tasks in the thread group, it attempts to steal a task from the priority queues of the other thread groups of the same socket. If there are no tasks, it goes around the thread groups of all sockets, stealing tasks (not from the hard priority queues). If the worker thread finally has a task, it executes it, and loops again. If no task is found, it goes to park.

Watchdog. A watchdog thread wakes up periodically (similarly to Section 4.1) to gather statistics. It checks the active concurrency level of all thread groups. If a thread group is not saturated (meaning its active concurrency level is lower than its number of H/W threads), but has tasks, it signals parked worker threads, if there are any, else creates new worker threads. If a thread group is saturated, but has more tasks, it also monitors that, in order to signal parked threads in other non-saturated thread groups that can potentially steal these tasks.

Task priorities. We do not use the fixed priorities of our initial design (see Section 4.1). Instead, each task has a numeric priority which is a weighted combination of a user-defined priority and other information such as the depth of a task in its task graph and the timestamp of its related SQL statement. In this thesis, we do not set the user-defined priority of tasks [200]. The timestamp of a query, however, affects the priorities of its tasks. The older the timestamp, the higher the priority of related tasks. For our experimental evaluation, this also means that tasks generated during the execution of a query are handled more or less at the same time.

Task affinities. A task can have an affinity for a socket, in which case it is inserted in the priority queue of one of the thread groups of that socket. Additionally, the task can specify a flag for a hard affinity so that it is inserted into the hard priority queue. In case of no affinity, the task is inserted into the priority queue of the thread group where the caller thread is running (for potentially better cache affinity).

By default, for NUMA-agnostic workloads, we do not bind worker threads to the H/W threads of their thread groups, so that the OS schedules the NUMA-agnostic worker threads. We bind a worker thread to the H/W threads of its thread group only when it is about to handle a task with an affinity. And if the next task also has an affinity, the thread continues to be bound, otherwise it unbinds itself before running a task without an affinity. This gives us the flexibility to compare the OS scheduler, by not assigning affinities to tasks, against NUMA-aware scheduling by assigning affinities to tasks.

5.3.1 NUMA-Aware Scheduling of Scans

In this section we describe how scans are scheduled in a NUMA-aware fashion by considering the data placement. Tasks need to consult the PSM of the data they intend to process to define their affinity. In Figure 5.5 we show the execution phases of a query selecting data from a single column, assuming the column is placed using IVP: (a) finding the qualifying matches, and (b) materializing the output [118]. Next, we describe these phases.

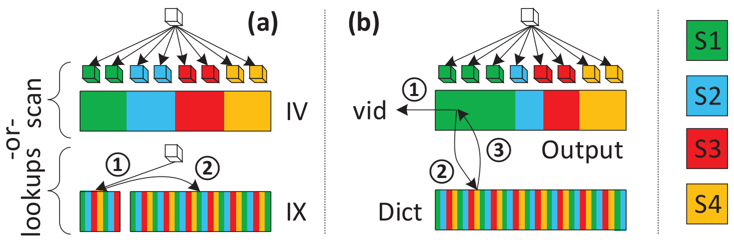


Figure 5.5 – Task scheduling (a) for a scan or index lookups to find qualifying matches, and (b) for the output materialization, for an IVP-placed column.

Finding the qualifying matches. Depending on the predicate's estimated selectivity, the optimizer may either scan the IV, or perform a few lookups in the index (if there is an index). For both cases, the query first needs to encode its predicate with *vid*. For a simple range predicate, the boundaries are replaced with the corresponding *vid*. If the predicate is more complex, a list of qualifying *vid* is built and used during the scan or the index lookups [195, 197].

In the case of a scan, it is parallelized by splitting the IV into a number of ranges and issuing a task per range. The task granularity is defined by the concurrency hint (see Section 4.1.4), to avoid too many tasks under cases of high concurrency, but also to opt for maximum parallelism under low concurrency. In the case of IVP, as in the example of Figure 5.5, we round up the number of tasks to a multiple of the partitions, so that tasks have a range wholly in one partition. We define a task's affinity by consulting the PSM of the IV for the task's range.

Index lookups are not parallelized. For each qualifying *vid*, the IX is looked up to find the qualifying positions of the IV. The affinity of the single task is defined as the location of the IX. If it is interleaved, as in the case of IVP, we do not define an affinity.

The qualifying matches can be stored in two potential formats [118, 197]. For high selectivities, a bitvector format is preferred where each bit signifies if the relevant position is selected. For low selectivities, a vector of qualifying IV positions is built. Both formats typically consume little space and we do not track their location on memory with a PSM.

Output materialization. Since we know the number of qualifying matches, we allocate the whole output vector. The materialization is parallelized in case of a high number of results. In such cases, we parallelize materialization similarly as the scan, by splitting the output into ranges and issuing a task per range. A task, for each qualifying position of its range, finds the relevant *vid* through the IV. Then it finds the real value by consulting the dictionary, and finally writes it to the output.

Because different partitions of the IV may produce more or less qualifying matches, the output may have unbalanced partitions. To define task affinities, we need a short preprocessing. Going through all qualifying matches to figure out the exact boundaries is costly. Thus, we divide the output vector length by a fixed number, e.g., the number of H/W threads of the server, to make fixed-sized regions, and find the location of their boundaries by consulting the PSM of the IV. We coalesce contiguous regions on the same socket to make up the final list of partitions (visualized in Figure 5.5). For each partition, we issue a correspondingly weighted number of tasks with the affinity of that partition's socket, taking care that the number of tasks does not exceed the concurrency hint, and that each partition has at least one task.

Figure 5.5 hints that we place the partitions of the output to their corresponding socket. Unfortunately, allocating a new output vector in order to specify its location turns out to have a bad performance. Especially for high-selectivity concurrent scans, it involves numerous page faults with heavy synchronization in the OS. This is one reason why SAP HANA implements its own allocators [8, 191] to reuse virtual memory. Furthermore, we note that using `move_pages` to move the partitions also runs into a similar problem in the OS. Thus, for concurrent workloads, re-using virtual memory for the output vectors, even if writes are remote accesses, is better than explicitly placing the pages of the output vectors.

Remaining data placements. Figure 5.5 describes how a scan is scheduled when the column is placed with IVP. In the case of RR, when a column is on one socket, the same scheduling is involved, but without figuring out the boundaries of the output vector's partitions. In the case of PP, the phase of finding qualifying matches occurs once per table part, concurrently. There is a single output vector, with a length equal to the sum of the lengths of the results of each table part. The preprocessing phase of the materialization happens once, in a similar way as in the case of IVP, by considering that partitions of the IV are now separate IV. The materialization phase occurs once per table part, concurrently, and each table part knows the region of the single output vector where to write the real values.

5.4 Experimental Evaluation of Concurrent Scans

In this section, we present a sensitivity analysis of concurrent scans for different data placement and task scheduling strategies, under various workload parameters. We use a prototype based on SAP HANA (SPS9). We extend the execution engine of SAP HANA with our NUMA-aware data placements (see Section 5.2) and task scheduling (see Section 5.3).

For all experiments, we warm up the DBMS first. We make sure that all clients are admitted, and we disable query caching. In several cases, we present additional performance metrics gathered from Linux, SAP HANA, and H/W counters (using the Intel Performance Counter Monitor tool [196]).

We generate a dataset with a large table, resulting in a 100 GB flat CSV file. It consists of 100 million rows, an ID integer column as the primary key, and 160 additional columns of random integers generated with a uniform distribution. We use bitcases 17 to 26 in a round-robin fashion for the 160 columns, to avoid scans with the same speed [195].

We use a Java application on a different server to generate the workload. The clients connect and build a prepared statement for each column: `SELECT COLx FROM TBL WHERE COLx >= ?  AND COLx <= ?`. The implications of our evaluation are relevant for queries that have a predicate on multiple columns or project multiple columns as well. In the former case, the first phase of Figure 5.5 is repeated (in parallel) for each column, to find the qualifying positions. In the latter case, the materialization phase of Figure 5.5 is repeated (in parallel) for each column.

Each client continuously picks a prepared statement to execute. There are no thinking times. The client does not fetch the results, otherwise the network transfer would dominate. We measure the achieved queries over 2 min and report the average throughput (TP) per minute. The additional performance metrics presented are averaged over the whole 2 minutes period. Every data point and metric presented is an average of 3 iterations with a standard deviation of less than 10%. The main server we use is the 4-socket Ivybridge-EX server of Table 2.2. The OS is a 64-bit SMP Linux 3.0.101 (SUSE Enterprise Server 11 SP3).

The data placement strategies we compare are:

- *Round-robin (RR)*. Each column is allocated on one socket, in a round-robin fashion.
- *Indexvector partitioning (IVP)*. Each column's IV is partitioned equally across the sockets.
- *Physical partitioning (PP)*. The table is physically partitioned according to ranges of the ID column. The number of equally-sized ranges is the number of the sockets. Each table part is placed on a different socket.

The task scheduling strategies we compare are:

- *OS*. We do not define task affinities, and we do not bind worker threads, leaving the scheduling to the OS.
- *Target*. We define task affinities. Tasks may be stolen.
- *Bound*. We define task affinities, and set the hard affinity flag for tasks. Inter-socket stealing is prevented.

The workload parameters we vary are:

- *Concurrency*. The number of clients in the workload.
- *Indexes*. Whether indexes can be used or not. In the majority of the experiments, we do not use indexes.
- *Selectivity*. The selectivity of the range predicates.
- *Column selection*. The probability that a column may be selected. Can either be uniform or skewed to make a subset of columns hot.

5.4.1 Uniformly Distributed Workload

In this section, we evaluate a uniform workload, i.e., clients pick a column to query, randomly with uniform distribution.

Impact of scheduling

This experiment aims to show the largest performance difference between NUMA-agnostic and NUMA-aware execution. We use *RR* to place the columns on the 4-socket server. Intra-query parallelism is enabled, and the selectivity of the queries is low (0.001%). Indexes are not used, thus scans are used, and the workload is memory-intensive. The throughput (TP) and relevant performance metrics are shown in Figure 5.6. Performance metrics include the CPU load, the number of tasks, and the number of tasks stolen across sockets. For the case of 1024 clients, performance metrics include the last-level cache (LLC) misses (local and remote), the memory throughput of each socket, the instructions per cycle (IPC), the total traffic through the QPI, and the total data (without the cache coherence) traffic through the QPI.

There is a 5x throughput improvement with the *Target* and *Bound* strategies, over the *OS*, mainly due to the improved memory throughput. The LLC misses, most of which are prefetched, are almost 5x more, and mostly local compared to the mostly remote misses of *OS*. The number of processed tasks is 5x higher, and IPC is also 5x higher due to faster memory accesses. QPI data traffic is reduced analogously, but cache-coherence traffic is generated indirectly, even with local accesses, and cannot be avoided.

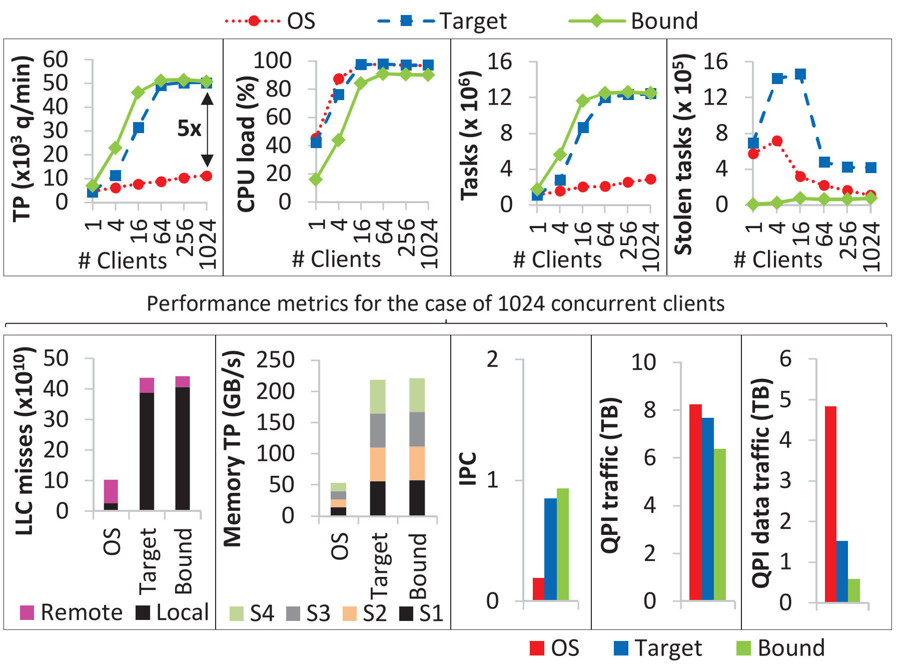


Figure 5.6 – Evaluating *OS*, *Target*, and *Bound* scheduling strategies, with *RR*-placed columns.

Overall, *Bound* achieves better throughput than *Target*. Although stealing improves CPU load, it hurts throughput for memory-intensive workloads due to the incurred remote accesses and stress on the remote memory controllers and the QPI. We revisit this effect later.

Implications. NUMA-awareness can significantly improve the performance of memory-intensive workloads. Memory-intensive tasks should be bound to the socket of their data.

Impact of the cache-coherence protocol

Figure 5.7 shows the results of the previous experiment on the 8-socket Westmere-EX server of Table 2.2. *Bound* decreases the QPI data traffic, but the total traffic is increased, due to

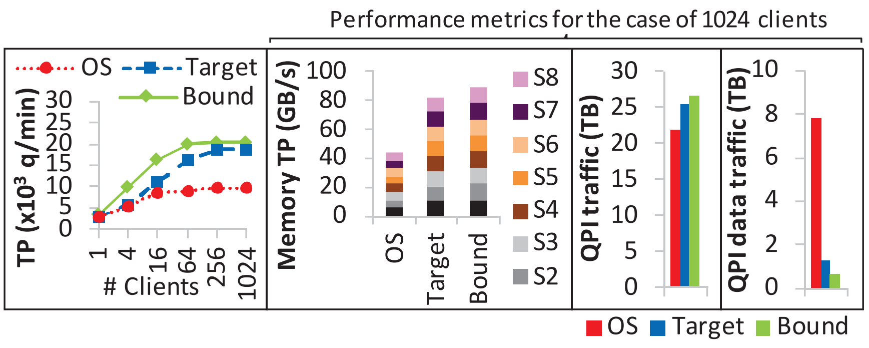


Figure 5.7 – As Figure 5.6, on the 8-socket Westmere-EX server.

the broadcast-based cache-coherence protocol (see Section 2.4.1). Due to the saturation of the QPI, we cannot fully exploit the memory bandwidth of all sockets simultaneously. Thus, *Bound* improves performance only by 2x compared to *OS*.

Implications. H/W characteristics, such as the cache-coherence protocol, can affect the NUMA impact we observe on different servers. The performance improvement of preferring local over remote accesses, however, still applies.

Impact of data placement

We continue with the previous experiment, using *Bound*, but with different data placements, on the 4-socket Ivybridge-EX server of Table 2.2. Figure 5.8 shows the results.

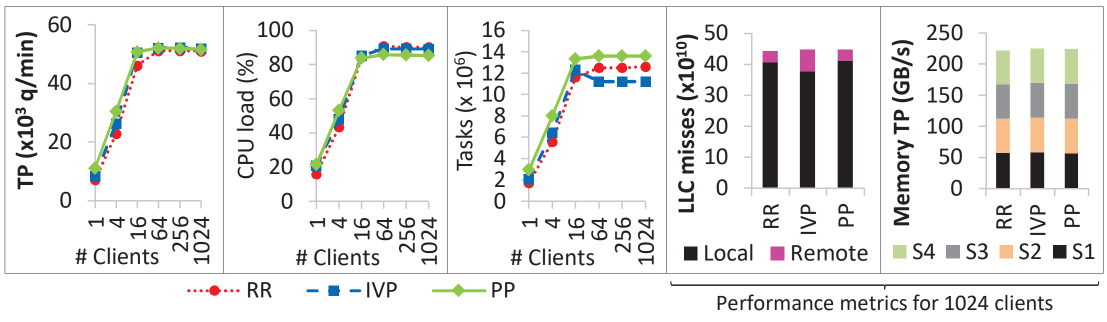


Figure 5.8 – The effect of the *RR*, *IVP*, and *PP* data placements.

All data placements reach the same throughput for high concurrency. *IVP* has slightly more remote accesses than *PP*, since the dictionary is interleaved. Although the same throughput is reached with parallelism, there is a difference between the data placements. In Figure 5.9, we show violin plots of the query latency distributions. All have the same average latency. *RR*, however, is unfair, with more variance. *IVP* and *PP* have most latencies closer to the average. This is because in *RR*, queries queue up and execute on the socket level. With *IVP* and *PP*, each query parallelizes across all sockets, and because the tasks are prioritized according to the query's timestamp, they complete approximately in the order they were received.

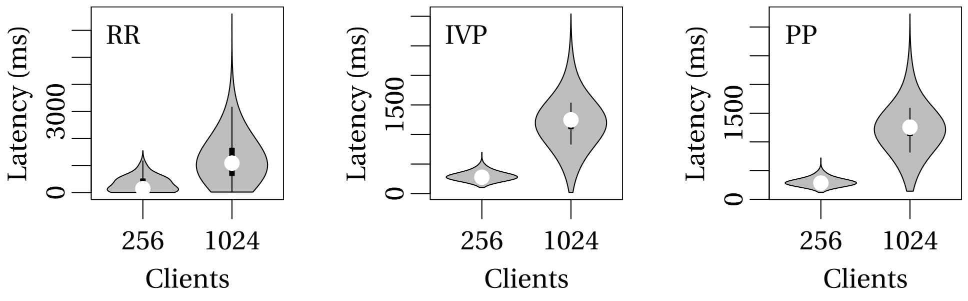


Figure 5.9 – Violin plots of the latency distributions.

Implications. Partitioning has a fair latency distribution.

Impact of scale on data placement

On the 4-socket server, partitioning achieves the same performance as *RR*. This is not the case on large-scale servers, where partitioning can involve a "convoy" effect: the slowest socket becomes a bottleneck, as detailed next. We evaluate the previous experiment on the 32-socket Haswell-EX server of Table 2.2. To showcase the convoy effect as prominent as possible, for this experiment each column is targeted by 7 clients (for a total of 1120 clients) and the concurrency hint is set to a minimum (each query executes a task graph with 1 task per partition). Figure 5.10 shows the results for the data placements and scheduling strategies.

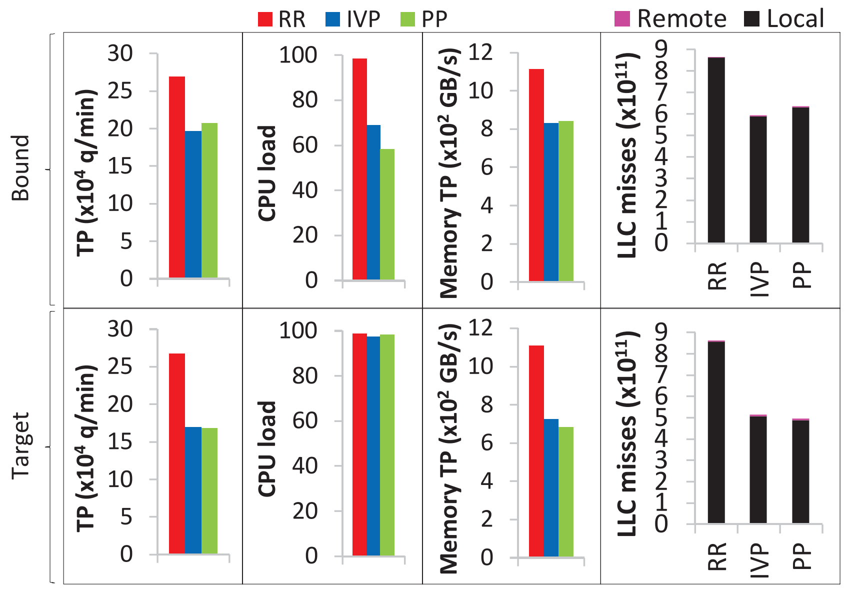


Figure 5.10 – Evaluating 1120 clients with different data placement and scheduling strategies, on a 32-socket server.

Bound is the best scheduling strategy since the tasks are memory-intensive. *RR* is the best data placement, as it fully utilizes CPU load because the sockets process scans independently. Partitioning, however, presents an overhead. This is not a NUMA effect due to a convoy effect. Load is not perfectly balanced across the sockets. There are also side-effects from the OS whose processes consume a substantial CPU load (56 H/W threads on average) during the experiments. A socket becomes the slowest and has always more CPU load and queued tasks than the rest of the sockets. Due to all queries needing to parallelize on all sockets, queries have to wait for the slowest socket. Overall CPU load, memory throughput and query throughput are effectively dropped for both *IVP* and *PP*.

Stealing can be a natural solution for balancing tasks and CPU load. For concurrent scans, however, stealing does not help because tasks are memory-intensive. Figure 5.10 shows that *Target* is able to saturate CPU load, but drops memory and query throughput for both *IVP* and *PP*. Stealing memory-intensive tasks should be avoided. We revisit this effect in Section 5.4.2 on the 4-socket server.

Implications. Unnecessary partitioning can cause a convoy effect for memory-intensive workloads on large-scale multi-socket servers.

Impact of selectivity

In this experiment, we vary the selectivity from 0.001% up to 10%. We enable indexes, evaluate 1024 clients, and use *RR* and *Bound*, on the 4-socket server. We note that *Target* achieves similar results since the workload is uniform and the concurrency is high. The results are shown in Figure 5.11.

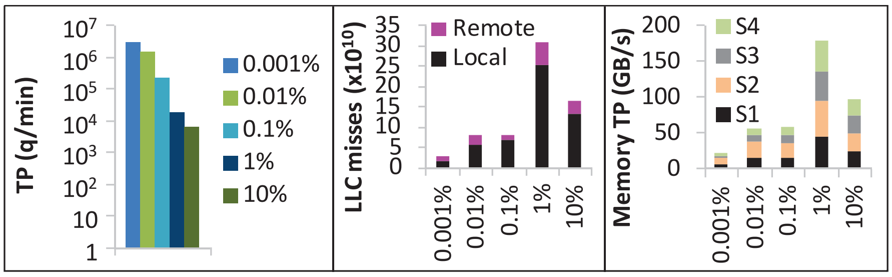


Figure 5.11 – Evaluating different selectivities.

As expected, throughput drops as we increase the selectivity. The optimizer chooses to perform index lookups for selectivities 0.001%-0.1%, as implied by the low memory throughput and the number of LLC misses. For larger selectivities, it chooses scans. For selectivity 1%, scans dominate the execution, as is shown by the high memory throughput and the large number of LLC misses. The workload is more memory-intensive. For selectivity 10%, however, the materialization phase dominates the execution. Since the materialization consists of random accesses due to the dictionary, it is less memory-intensive with less memory throughput and a lower number of LLC misses.

Implications. For a dictionary-encoded column, with an index, the selectivity changes the critical path of execution. It consists of index lookups for low selectivities with low memory throughput, memory-intensive scans for intermediate selectivities with high memory throughput, and less memory-intensive materializations for high selectivities with intermediate memory throughput.

5.4.2 Skewed Workload

In this section, we use a skewed workload. Clients have a 20% probability of choosing a random column from the first 80 columns of the dataset, and an 80% probability of choosing one from the remaining 80 columns. We continue using the 4-socket server.

Impact of stealing memory-intensive tasks

We perform the first experiment of Section 5.4.1. We use *RR*, and we intend to see the effect of scheduling strategies on performance. The results are shown in Figure 5.12.

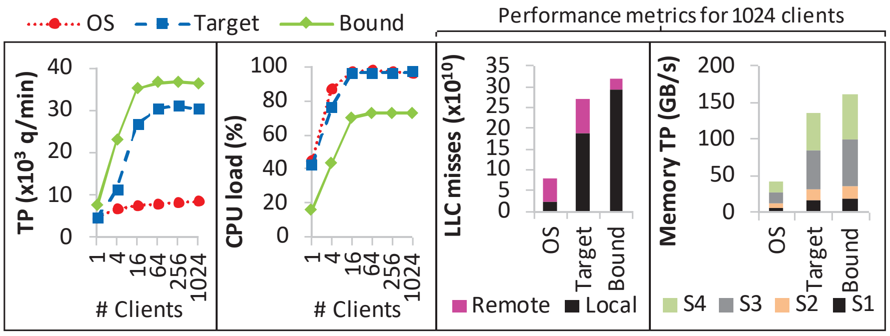


Figure 5.12 – Evaluating the *OS*, *Target*, and *Bound* strategies, with *RR*-placed columns.

Bound still achieves the best throughput, even though it underutilizes the server. As implied by the memory throughput, only two sockets contain the hot set of columns.

One would expect that *Target* achieves better throughput, since it utilizes more CPU resources. It decreases the throughput, however, by around 15%. The remote accesses overwhelm the already saturated two hot sockets and the interconnect network.

Implications. Inter-socket stealing of memory-intensive tasks can decrease overall throughput by up to 15%.

Impact of partitioning

To battle skewness, apart from collocating hot and cold columns, one can partition hot columns. Figure 5.13 shows the results of the previous experiment using *Bound*, but evaluating the different data placements and partitioning types. *IVP* and *PP* achieve the best throughput as *RR* in the case of the uniform workload of Section 5.4.1. Skewness is smoothed out since queries are parallelized across all sockets.

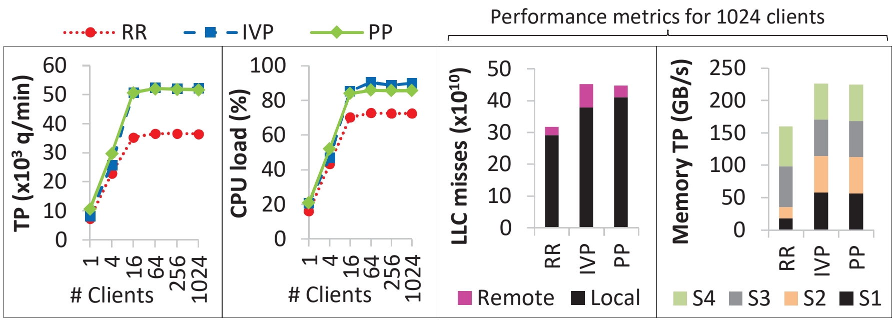


Figure 5.13 – Evaluating the *RR*, *IVP*, and *PP* data placements, with the *Bound* strategy.

Implications. Partitioning can significantly improve the throughput of skewed workloads.

Impact of partitioning type

One needs to consider two things for choosing between *IVP* and *PP*. Firstly, *PP* is expensive to perform as it recreates columns. *PP* on this dataset takes around 18 min, compared to 4 min for *IVP*, and consumes around 8% more memory because dictionaries contain recurrent values. Secondly, *IVP* interleaves the IX and the dictionary, which may be inefficient for index lookups and intensive materialization phases.

As a practical example, Figure 5.14 shows the results of the previous experiment with a high selectivity of 10%. Execution is dominated by the materialization phase, which involves random accesses to the dictionary. *PP* is better, since it involves more local accesses.

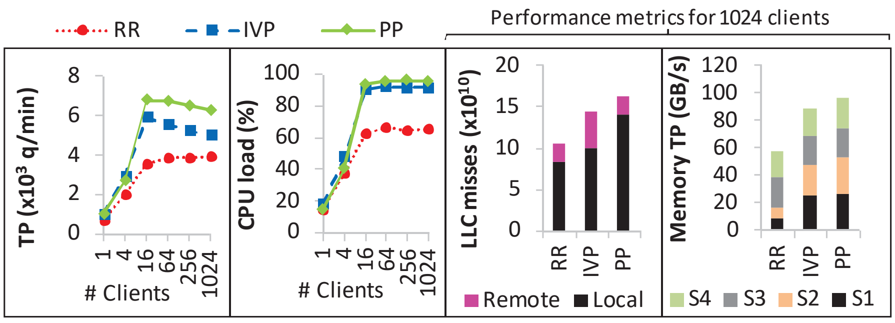


Figure 5.14 – As Figure 5.13, with a high selectivity.

Implications. To battle workload skewness, *IVP* is a quick solution. *PP* is slower to perform, but has the best performance for all skewed workloads with low and high selectivities.

Impact of stealing tasks that are not memory-intensive

Stealing can be helpful when tasks are not as memory-intensive as scans, without such a high memory throughput as scans. Figure 5.15 shows the results of the previous experiment with

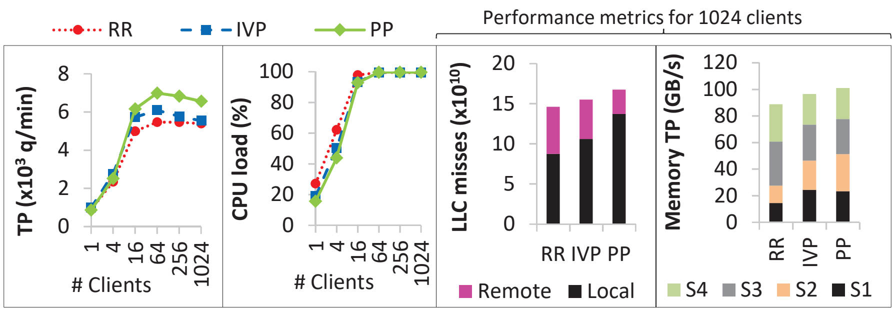


Figure 5.15 – As Figure 5.14, with the *Target* strategy.

high selectivities, but with the *Target* strategy. Stealing now does not hurt as in the case of memory-intensive tasks in the first experiment of Section 5.4.2.

Stealing does not improve *IVP* and *PP* since they were already saturating CPU resources, but improves *RR*, which now has full CPU load and achieves the same throughput as *IVP*. Stealing incurs remote accesses, and both *RR* and *IVP* are still worse than *PP* which results in more local accesses.

Implications. Inter-socket stealing should be allowed for tasks that are not memory-intensive.

5.5 NUMA-Aware Scheduling of Aggregations and Equi-Joins

To sum up our evaluation of concurrent scans, we remind that a scan has two phases: (a) a memory-intensive phase that scans the bit-compressed IV for qualifying rows for a given predicate, and (b) a less memory-intensive (with less memory throughput) phase that materializes the output values corresponding to the *vid* of the qualifying rows by consulting the dictionary. The main implication of our analysis for task scheduling is that stealing of memory-intensive tasks should be disallowed. The main implication about the performance of the data placement strategies is that physical partitioning (*PP*) results in the best performance and can battle skewness in the workload. However, it should not be abused on large-scale servers as unnecessary partitioning can create a convoy effect for memory-intensive tasks that should not be stolen. *PP* is better than *IVP* because the data structures of a part are wholly allocated on a socket, and the scans result in more local accesses. In this section, we continue our analysis to see the implications about task stealing and data partitioning for additional NUMA-aware operators: aggregations and equi-joins.

We use the distributed implementation of analytical operators to support NUMA-aware execution on a multi-socket server. For this reason, we support a simple data placement for aggregations and equi-joins: a socket identifier per *table part (TBP)*. We do not continue further with PSM and the *IVP* data placement. An additional drawback of *IVP* is that it does not capture any transient data structures, e.g., hash tables. This allows us to extend our analysis, in this and the next chapter, to aggregations and equi-joins by realizing the *RR* and *PP* data placement strategies at the level of tables and table parts. Our unit of data placement is a row-wise partition of a table. We assume that the organization of associated columns into tables is left to the administrator. Next, we describe our NUMA-aware implementation of aggregations and equi-joins.

Aggregations. An aggregation uses the scan's first phase to find the qualifying rows. Then it is parallelized with multiple tasks, where each task executes two phases: (a) aggregating using a local *hash table (HT)*, and (b) merging the local HT to a set of disjoint result HT [186]. The reason for the two phases is that they are interchanged potentially multiple times in order to avoid large local HT that do not fit in the processor's last level cache (LLC) [139, 186].

If there are multiple TBP, an additional phase precedes, that employs a global dictionary (used in the subsequent aggregation phases) by matching the *vid* of the qualifying rows of the group-by column from each TBP. Tasks are scheduled in a NUMA-aware fashion similar to the scans. Although local HT are placed on a task's socket, the global dictionary is accessed by all tasks, and is placed on one of the sockets of the involved TBP. Each disjoint result HT is placed on one of the involved sockets in a round-robin manner. During the merge phase, a task merges local HT that can lie on multiple sockets into a result HT that lies on its socket.

A visualized aggregation example. To better understand how NUMA-aware aggregations are executed, we depict in Figure 5.16 the following query:

```
SELECT NAME, COUNT(NAME) FROM TBL WHERE NAME <= "Dave" GROUP BY NAME.
```

In the example, we assume that the table is partitioned with round-robin partitioning into two TBP, and that the first is placed on Socket 1 and the second on Socket 2. Execution starts with the first phase of a scan, done concurrently on each TBP, in order to produce a bitvector that signifies which rows of each TBP qualify the query's range predicate. Since the same column is the group-by column, execution then proceeds to produce minimal dictionaries from the qualifying *vid* of each TBP, and merges them in parallel to create a global dictionary. It also keeps a mapping from the minimal dictionaries. Note how "Anna" has different *vid* in each TBP. The global dictionary can be placed on either socket.

Afterwards, the two aforementioned aggregation phases take place. During the first phase, a task continuously handles portions of a table part's scanned results and aggregates them into a local HT. If a local HT exceeds a predefined size, a new one is created and used. The predefined size is calculated on the basis of fitting the local HT of the concurrent aggregation

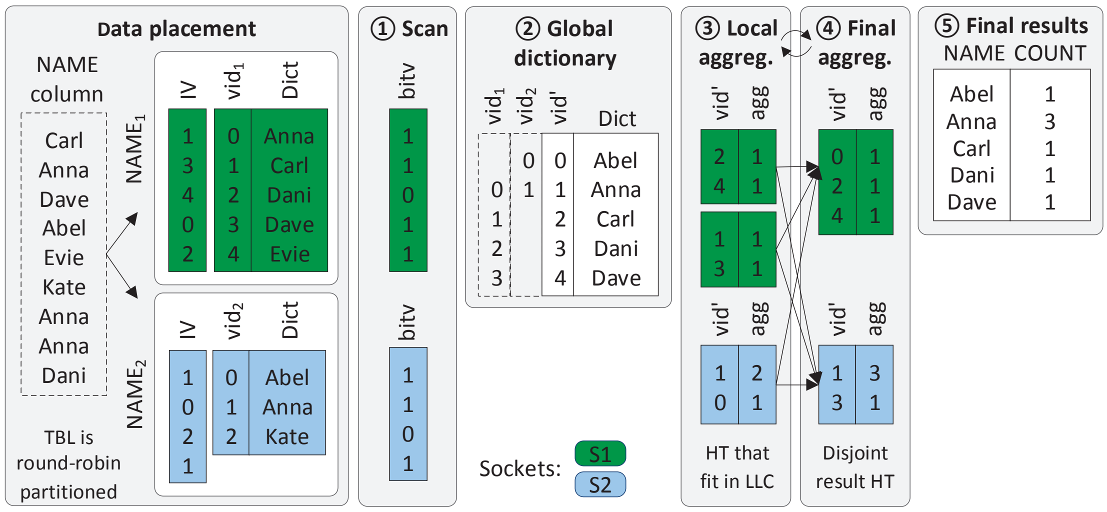


Figure 5.16 – NUMA-aware aggregation with a table partitioned across two sockets.

tasks on a socket's LLC [186, 139, 202]. For visualization purposes in Figure 5.16, we assume that the size has been set to two entries. Note how two local HT are created for the first TBP.

After all scanned data has been aggregated into local HT, the merge phase is initiated to calculate the final disjoint result HT. We note that the merge phase can also kick in earlier if there are numerous local HT created, in which case the two phases are interchanged multiple times until all data has been aggregated [186]. Each result hash table holds a disjoint subset of all qualifying *vid*. The number of result HT is less than or equal to the number of involved tasks. In this example, we assume two result HT. During the merge phase, a task handles the merging of all local hash tables, which can reside on multiple sockets, into one result HT. Finally, the final result set, to be fetched by the query, is constructed. It is constructed by appending the disjoint HT, and by consulting the global dictionary to convert the *vid* to their real values.

Equi-joins. The NUMA-aware implementation of equi-joins is similar to their distributed implementation [93, 178]. Consider an example of an equi-join of two tables with a selection predicate on the first table. The first table is scanned to find qualifying rows for the predicate, and the corresponding *vid* of the join column. A global dictionary is employed, similar to aggregations, to map the *vid* of the join columns between the tables. This is done by consulting the join columns' dictionaries. The join column of the second table is searched for the qualifying *vid* and rows. A potentially reduced set of *vid* is then used to filter out rows of the first table whose *vid* did not occur in the second table. The matched rows from both tables are joined, with the help of the global dictionary, to produce the final result. The steps are parallelized with multiple tasks and scheduled in a NUMA-aware fashion, best when the tables are on the same socket. If they are on different sockets, the tasks incur remote accesses when mapping the *vid* of the join columns, and during the final result production.

If there are multiple TBP for the tables, the aforementioned steps are done by visiting all involved TBP to map their *vid* using the global dictionary, and match the qualifying rows to produce the final result. For this reason, remote accesses can be increased considerably if the TBP reside on multiple sockets. There is one case when remote accesses are largely avoided. If the tables are partitioned on the joined column, and each pair of TBP (of the two tables) resides on the same socket, copartitioned equi-joins can be parallelized and executed locally on the sockets where the pairs of TBP reside. Our adaptive data placement in the next chapter (see Chapter 6) exploits this special case when the administrator has defined *table groups (TG)* (see Section 2.4.2) that contain the joined tables, and partitioning specifications on the joined column. Joins on other columns, or between tables outside a TG, are not guaranteed to have mostly local accesses. This is similar to distributed joins with partitioned tables [8].

5.5.1 The Overhead of Partitioning and Stealing

Here, we perform an experimental evaluation of physical partitioning and stealing for our NUMA-aware implementations of aggregations and equi-joins. We show that, unlike scans, partitioning has an overhead for aggregations and equi-joins. Also, we confirm once again that task stealing of memory-intensive tasks hurts throughput. Both of these implications point to a need for adaptivity at run-time.

Figure 5.17 shows the throughput (TP) of concurrent aggregations or joins under different cases of selectivity, available tables, task stealing, and partitioning (TBP/table). See Section 6.5 for more details on our methodology, dataset, and query types. Aggregations use query type `b`: each query picks a random table out of the available tables and aggregates a column with a group-by. Joins use a slightly modified version of query type `c`: a query picks a random pair of tables (either `TBL1`-`TBL2`, `TBL3`-`TBL4` etc.) to join on their `COL1` instead of the `ID` primary key (PK) column, with a filter predicate as well. In the case of *RR* (1 TBP/table), the table pairs are placed in a round-robin way around the sockets. In the case of *PP* (4 TBP/table), tables are partitioned on the PK, and the TBP of a table pair are collocated on the same socket. This configuration can show the worst overhead of partitioning. The server used is the 4-socket Ivybridge-EX server of Table 2.2. All cases saturate CPU load, apart from the case of 1 table with *RR* and *Bound*, which uses one socket.

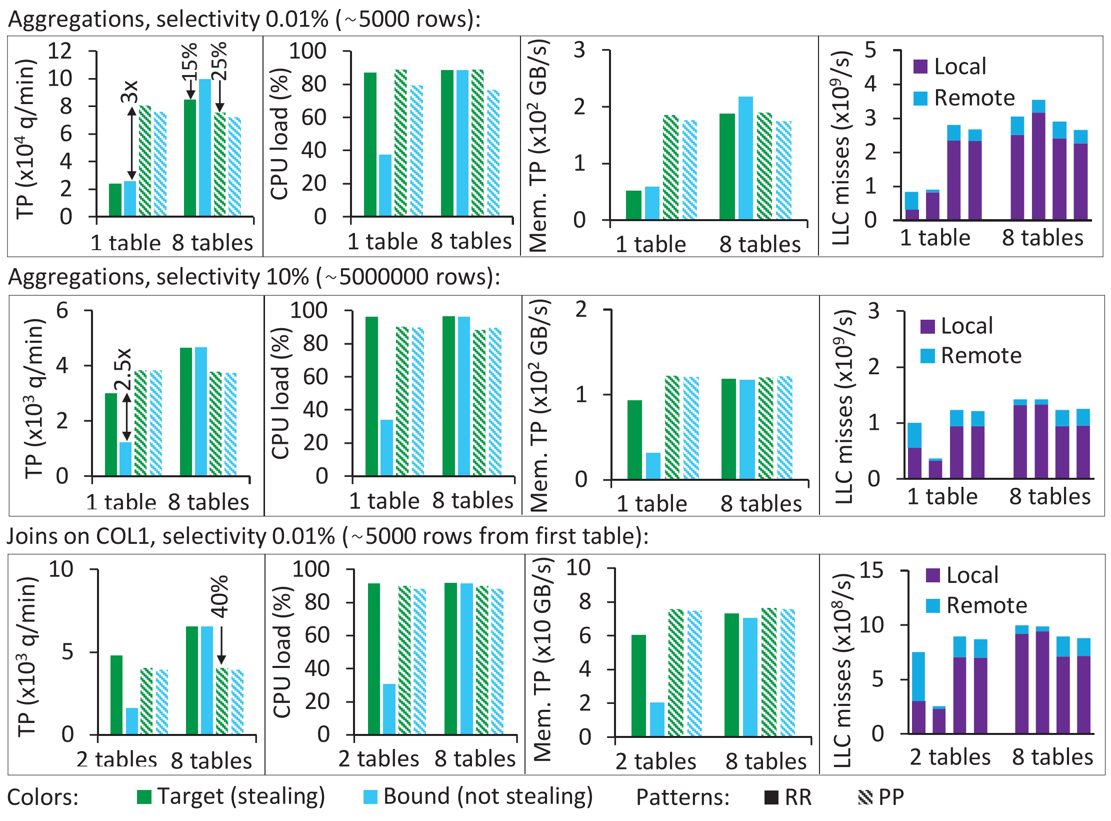


Figure 5.17 – 256 clients issuing aggregation or join queries on the 4-socket server, under different cases of selectivity, available tables, stealing, and partitioning. Each result of LLC misses belongs to the corresponding case of the left-hand side graphs.

Low-selectivity aggregations are dominated by IV scans. The workload is memory-intensive, as can be seen by the high memory TP (maximum 280 GB/s). As far as stealing is concerned, it helps saturate the CPU load (for the case of 1 table with *RR*), but it does not improve the TP as the workload is memory-intensive and stealing is unnecessary. In fact, it can hurt TP by up to 15% (see case of 8 tables and *Target* vs. *Bound*). As far as partitioning is concerned, it can greatly help improve the memory TP, improving the TP by up to 3x in case the workload is skewed (see case of 1 table and *RR* vs. *PP*). But when partitioning is unnecessary, it has an overhead of up to 25% (see case of 8 tables and *RR* vs. *PP*). The partitioning overhead is due to the need of a global dictionary, plus the remote accesses during the merge phase, and unnecessary scheduling overhead. We note that remote accesses during the merge phase are necessary irrespective of the implementation [202, 139], and depend on the number of groups involved (see Section 6.5.3 for a relevant experiment).

High-selectivity aggregations are dominated by the aforementioned aggregation phases. Due to the random accesses and hashing involved, the workload is not memory-intensive as it does not achieve a high memory TP. Stealing is now helpful. Stolen tasks do not run the risk of overwhelming the remote memory controller or the interconnects. It can help saturate CPU load and improve TP by up to 2.5x (see case of 1 table and *RR*). The implications about partitioning are the same: it can help when the workload is skewed, otherwise it has an overhead (see case of 8 tables and *RR* vs. *PP*). The reasons for the overhead are the same as in the case of low selectivity, with the remote accesses of the merge phase more pronounced.

Joins are also not memory-intensive and stealing helps. The overhead of unnecessary partitioning, however, is aggravated for joins that are not copartitioned. The overhead reaches up to 40% (see case of 8 tables with *RR* vs. *PP*). The overhead is due to mapping the *vid* of the TBP of both tables, and due to accessing TBP on all sockets for producing the final results.

Dictionary encoding. The overhead of partitioning includes the employment of a global dictionary for aggregations and joins. The reason for keeping different dictionaries across parts is due to the fact that SAP HANA supports mixed OLTP and OLAP workloads as well. This allows handling *vid* independently in each partition. Even with global *vid* across the partitions, however, or even without dictionary encoding at all, the need for adaptive data placement still exists since unnecessary partitioning requires remote accesses for aggregations with groups and joins that are not copartitioned.

Summary of implications. Summing up our analysis of scans, aggregations, and equi-joins, partitioning should be used for skewed workloads to balance utilization across sockets, and stealing should not be used for memory-intensive tasks. Both of these implications depend on the workload. For this reason, we adapt data placement and task scheduling to the workload at run-time in the next chapter (see Chapter 6).

5.5.2 TPC-H and SAP BW-EML Benchmarks

We further verify the basic implications of our analysis with two benchmarks. For the first benchmark, we use TPC-H (read-only) [15] with a scaling factor of 100 (around 100 GB flat files). We measure the throughput (queries per hour) of TPC-H Q1 instances with random parameters, which are continuously issued by 32 concurrent clients. The clients can saturate resources due to intra-query parallelism. We evaluate only Q1 to show a workload dominated by aggregations on a single table (lineitem). In the next chapter, we evaluate all TPC-H query templates (see Section 6.5.6).

For the second benchmark, we measure the throughput of the reporting load of SAP BW-EML [9, 40].[1] BW-EML is representative of realistic SAP BW (business warehouse) industrial workloads. It uses an application server to host BW users who query a database server. Throughput is measured in navigation steps (or queries) per hour. At the core of the data model, there are 3 *InfoCubes*, each modeling multidimensional data with an extended star schema [115]. The major part of the execution consists of queries which are dominated by scans and aggregations on the InfoCubes, and less by equi-joins. Every presented measurement uses the maximum number of users that can be serviced without timeouts. Our dataset has 1 billion records (around 800GB of flat files).

To evaluate BW-EML, we split the 32-socket Ivybridge-EX rack-scale server of Table 2.2 into two 16-socket ones, one hosting the application and the other the database server (running our prototype of SAP HANA). We use the database server for TPC-H as well. We evaluate the impact of different *PP* granularities, and the impact of inter-socket task stealing on the throughput. Granularity *PP2*, for example, means that the tables are physically partitioned into two partitions each. For each granularity, we distribute the partitions in a round-robin manner around the sockets. The case of one partition per table degenerates to *RR*. Due to legal reasons, throughput results are normalized to undisclosed constants c_1 and c_2, corresponding to the maximum observed throughput for TPC-H and BW-EML respectively. This does not hinder us from comparing the impact of the different data placement and scheduling strategies on the throughput. The results are shown in Figure 5.18.

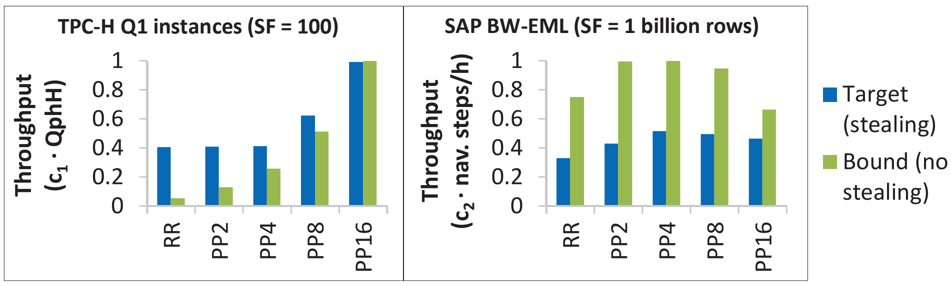


Figure 5.18 – TPC-H and SAP BW-EML with different *PP* granularities and scheduling strategies.

[1] We refer the reader to an IBM redbook [53] for a more detailed introduction to BW-EML and sample statements.

TPC-H is severely skewed, since Q1 queries one table. Increasing the number of partitions improves performance as queries are gradually executed locally on more sockets. There is minimal overhead from partitioning, since the workload is skewed, and the aggregations of Q1 have few groups. Our internal profiling indicates that Q1 is not memory-intensive (without a high memory throughput) as its execution is dominated by the multiplications of its aggregations. Due to this, stealing is better than not stealing. Without stealing, increasing the number of partitions results in utilizing more sockets, finally matching the throughput of stealing.

BW-EML, in contrast, has simpler aggregation expressions and is more scan-intensive. For this reason, not stealing is always better than stealing. Increasing the number of partitions up to 4 improves the performance of not stealing, since the 3 InfoCubes are partitioned across 12 sockets. Further partitioning, however, is unnecessary and incurs an overhead.

5.6 Summary and Conclusions

In this chapter, we analyze NUMA-aware data placement and task scheduling strategies for main-memory analytical workloads. For concurrent scans, we propose three alternative data placement strategies: *RR*, *IVP*, *PP*. With respect to task scheduling, we differentiate between allowing and not allowing inter-socket stealing of tasks. Our experimental analysis shows that *PP* achieves the best performance and can battle skewness in the workload. It should not, however, be abused on large-scale servers unnecessarily. Moreover, we show that stealing of memory-intensive tasks should be disallowed, otherwise it can hurt overall performance.

We continue our analysis with additional NUMA-aware analytical operators: aggregations and equi-joins. With respect to data placement, we compare *RR* and *PP*, and identify that *PP* involves an overhead for aggregations and equi-joins mainly due to the need of remote accesses and translating and matching *vid* across the dictionaries of the table parts. With respect to task scheduling, we confirm once again that inter-socket stealing of memory-intensive tasks should be disallowed.

To sum up, partitioning should be used for hot data when the workload is skewed in order to balance utilization across sockets, and stealing should be disallowed for memory-intensive tasks. In the next chapter, we use these implications to adapt the data placement and task scheduling to the workload at run-time (see Chapter 6).

6 Adaptive NUMA-Aware Task Scheduling and Data Placement

In this chapter, we build upon the implications of our analysis in the previous chapter to propose a data placement and task scheduling strategy that adapts to the workload at run-time. Our adaptive data placement algorithm tracks resource utilization, and when a utilization imbalance across sockets is detected, the algorithm corrects the imbalance by moving or physically repartitioning tables. Also, inter-socket task stealing is dynamically disabled for memory-intensive tasks that could otherwise hurt performance.

Publications. Parts of this chapter have been published in [163].

6.1 Introduction

In order to balance utilization across sockets, state-of-the-art systems [103, 117] partition data across sockets and employ task scheduling with inter-socket task stealing. In the previous chapter, our analysis showed that unnecessary partitioning can incur an overhead and that stealing memory-intensive tasks can hurt overall performance, depending on the workload. In this chapter, we use the implications of our analysis and adapt data placement and task scheduling to the workload at run-time, with the aim to balance resource utilization across sockets. We target highly concurrent workloads dominated by scans, aggregations, or equi-joins working on a single table or table group (see Section 2.4.2).

Our proposed design relies on tracking the history of CPU and memory bandwidth utilization at three levels (see Section 6.2): (a) tasks, (b) partitions of tables and table groups, and (c) sockets. When the execution engine detects a utilization imbalance across sockets, it either moves or repartitions tables in order to fix the imbalance (see Section 6.3). Moreover, it also finds cold partitioned tables to consolidate and disallows inter-socket stealing of memory-intensive tasks that would hurt performance (see Section 6.4).

Figure 6.1 shows a conceptual example of the most significant aspects of our adaptive techniques. The server has four fully interconnected sockets. The workload consists of numerous

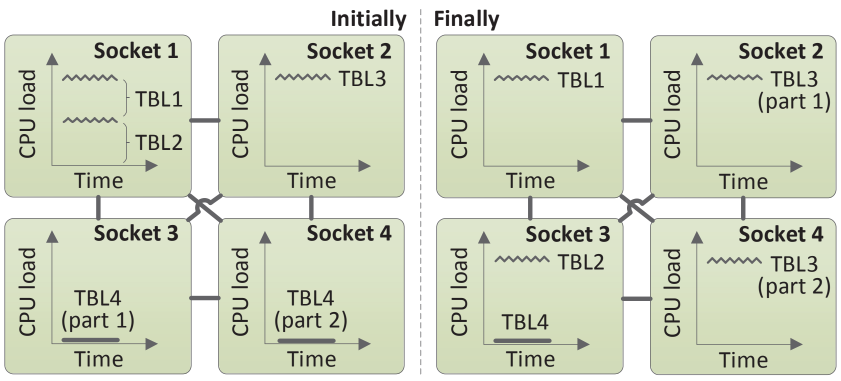


Figure 6.1 – A conceptual example of our adaptive data placement.

concurrent memory-intensive scans on three tables which are initially placed on two of the server's sockets. Task stealing is disallowed due to the memory intensity of the scans. Two of the sockets are fully utilized, and their memory bandwidth is saturated, while the remaining two sockets are idle. Our adaptive data placement detects the utilization imbalance, and takes actions to fix it. It moves table *TBL2* to socket 3, partitions *TBL3* across sockets 2 and 4, and finally merges the unutilized parts of $TBL4$. The final data placement is shown on the right-hand side of Figure 6.1. Socket utilization becomes balanced. The total memory throughput is 2x higher than initially, improving the workload's throughput by 2x (see Section 6.5.1).

Contributions. In this chapter, we present adaptive NUMA-aware techniques for main-memory column-stores. We adapt data placement and inter-socket task stealing to workloads dominated by operators working on a single table or table group (copartitioned tables), at run-time. Our implementation and experiments are based on our prototype of SAP HANA. Our contributions are the following:

- We present an adaptive data placement strategy that can improve throughput by up to 2x in comparison to partitioning tables across all sockets. Our adaptive strategy moves and repartitions tables at run-time in order to balance the utilization across sockets.
- We adapt inter-socket task stealing to the memory intensity of tasks, improving throughput by 1.1x-4x in comparison to always allowing stealing.
- To adapt data placement and task stealing, we need to know the system's utilization. We present a design that tracks the utilization history at the levels of (a) tasks, (b) partitions of tables and table groups, and (b) sockets.

Outline. Section 6.2 shows how we track the utilization at the level of tasks, table or table group parts, and sockets. Section 6.3 details our adaptive data placement. Section 6.4 describes how we adapt inter-socket task stealing to the memory intensity of tasks. Section 6.5 includes our experimental evaluation. Section 6.6 presents the conclusions of this chapter.

6.2 Tracking Resource Utilization

We track resource utilization (CPU load and memory throughput) across three levels: (a) tasks, (b) table parts (TBP) and table group parts (TGP), and (c) sockets. Figure 6.2 depicts our monitoring infrastructure with an example of concurrent low-selectivity scans dominated by the scan's memory-intensive first phase that scans the indexvector (IV).

Task classes. We organize tasks into classes that have similar functionality and memory throughput. We use a different class for the tasks of the first phase of a scan (IV scan), for the tasks of the second phase of a scan (materialization), for the aggregation tasks that interchange between the two aggregation phases, and for every step of a join (see Chapter 5). For each class, we track the observed memory throughput with a single exponential moving average.

Tasks in the same class should have similar memory throughput. We assume that classes are defined manually, as we do for our NUMA-aware operators. One can further specialize classes, e.g., by the involved predicates. As we show in Section 6.4, an aggregation's memory throughput can vary depending on the predicate's complexity. In our experiments in Section 6.5, we use rather typical predicates and the defined task classes can sufficiently capture the memory intensity of our NUMA-aware operators' different phases.

Scheduling. When a task is scheduled on its intended socket, where its associated TBP is placed, we aggregate its utilization on the level of the TBP and the socket. We atomically add the average memory throughput of the task's class to the memory throughput consumed by the TBP. We also atomically increment the CPU load (number of threads) utilized by the TBP and the socket. When the task either finishes or blocks in a synchronization primitive, we atomically subtract the memory TP we previously added and decrement the CPU load. With this method, we can keep track of the current local CPU load and the estimated local memory throughput of TBP and sockets. We ignore on purpose stolen tasks. The utilization of sockets that steal appear non-saturated in our metrics, as stealing is a temporary solution to balance CPU load until our adaptive data placement algorithm (see Section 6.3) fixes the imbalance of local utilization. In the example of Figure 6.2, we depict how the average memory TP of a task class is used when scheduling tasks to aggregate the utilization at the level of TBP and sockets.

Measuring memory throughput. A task class is needed to assign an estimated memory TP to a task when it is about to run. After the task ends, we calculate its consumed memory TP, and push back the value to the exponential moving average of its class with a low weight: $\text{classAverage} = 0.9 * \text{classAverage} + 0.1 * \text{newValue}$. In our prototype of SAP HANA, we calculate memory TP through H/W counters (integrating the Intel PCM tool [196]), and considered only for non-stolen tasks that have not blocked or moved to another core during execution. The formula is: $\text{memoryTP} = ((\text{localLLC}_{\text{end}} - \text{localLLC}_{\text{start}}) * 64) / (\text{timestampUS}_{\text{end}} - \text{timestampUS}_{\text{start}})$. This calculates the accessed bytes by associating every local LLC miss with a cache line (64 bytes) retrieval. Dividing by the task's duration, gives the average memory TP in MB/s. We ignore on purpose remote LLC, since we do not wish to track remote memory throughput in the task classes.

We note that the hardware counters of the Intel architecture codename Haswell do not capture LLC misses caused by the prefetcher [18]. However, we can still compare the relative memory throughput between task classes for supporting our adaptive task stealing (see Section 6.4). We note that this problem does not exist on the Intel architecture codename Ivybridge, which most of our experiments run on. Also, recent and upcoming Intel architectures, such as codename Broadwell and Skylake, support hardware counters that can capture the local memory throughput of a core [18, 196], and can be used in place of our formula.

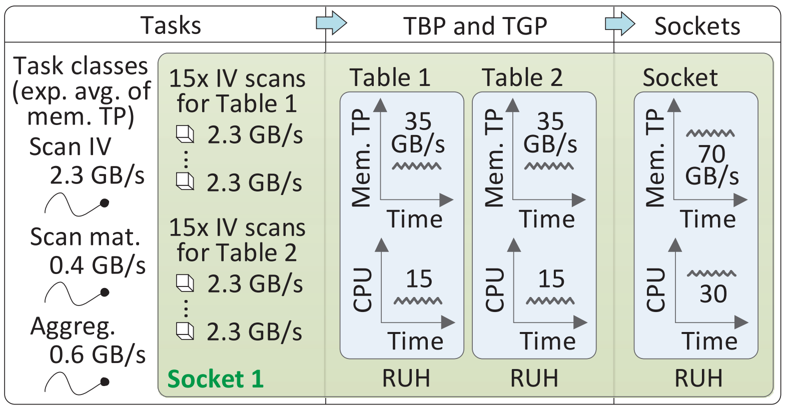


Figure 6.2 – Tracking the resource utilization of tasks, TBP and TGP, and sockets. The values are taken from instances of our experiments on our 4-socket server.

Table group parts (TGP). If a table group (TG) is defined for a group of copartitioned tables, tasks do not aggregate their utilization at the level of TBP, but at the level of *table group parts (TGP)*. For example, if the tables are partitioned with three TBP each, then we keep three TGP. Each one of the TGP tracks the aggregated utilization of the corresponding TBP of the tables. We note that we place the TBP of a TGP on the same socket, thus a TGP is associated with a single socket. If a table of the TG does not have a partitioning specification, it is considered as a single TBP and placed on the first TGP.

Resource utilization histories (RUH). Every TBP, TGP, and socket, has a RUH that keeps track of the memory throughput consumed and the number of threads used. The components of a RUH are shown in Figure 6.3. It contains the number of memory pages occupied by the TBP or TGP or socket, a pointer to the owner TBP or TGP or socket, the socket where the owner is placed, and two *history* objects that capture the recent utilization for memory throughput (`memh`) and threads (`cpuh`). Especially for the pages of a TGP, we do not sum the pages of its TBP, but keep the maximum pages of any of its TBP. This is because we later use the pages to estimate the moving or partitioning time, and we can move/partition a TG's TBP in parallel.

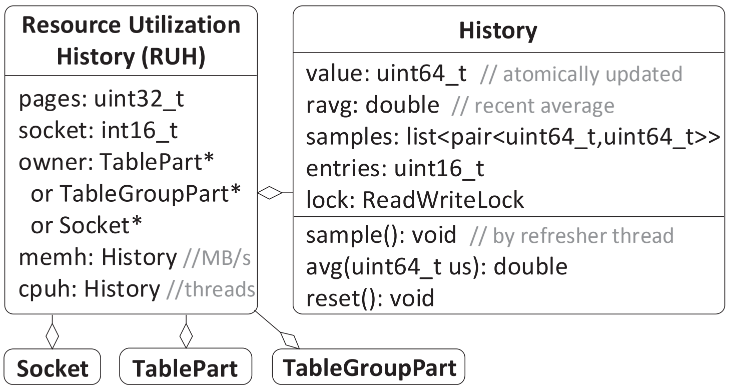


Figure 6.3 – Each table part (TBP), table group part (TGP), and socket is associated with a resource utilization history (RUH).

A History object contains the absolute value that, as explained before, is atomically incremented by a task when it is scheduled and decremented when it blocks or is de-scheduled. The absolute value is periodically sampled into a list of pairs of timestamps and values. We maintain one background *refresher thread* per socket, that periodically calls `sample()` on the histories of its socket and of a subset of all TBP and TGP. The method `sample()` appends a pair with the current timestamp and value to the back of the samples list, and increments the entries. If the entries grow over a specified limit, it deletes the pair at the front of the list. In our current implementation, a refresher thread runs every 100 ms, and the limit of the samples list is set to 3000 entries, which means that it can reach up to around 47 KB, holding samples for roughly the last 5 minutes.

We note that tasks atomically modify the absolute values of the RUH of TBP or TGP only. A socket's absolute values are periodically calculated by the corresponding refresher thread by aggregating the absolute values of all TBP and TGP placed on the socket. This is to avoid numerous atomic operations at the level of a socket, since many TBP and TGP can be placed on it, and because we use a socket's utilization mostly for monitoring purposes. The adaptive data placement algorithm of Section 6.3 calculates the sockets' utilization by aggregating the utilization of TBP and TGP, in order to work on a single "snapshot" of how the sockets' utilization is composed by the involved TBP and TGP.

The adaptive data placement algorithm makes heavy usage of the `avg(uint64_t us)` function of the RUH, which iterates the sample list to find the first entry that is not older than the given microseconds, and starts averaging the entries up to the last entry (each entry is given a weight equal to the microseconds passed since the previous entry), finally returning the average utilization. If the given microseconds would require an entry older than the first entry of the samples list, we assume that the value of the utilization during that period is equal to the value of the first (oldest) entry.

Due to synchronization issues with the refresher threads, a read-write lock is used by both the refresher threads (which write) and the adaptive data placement algorithm (which reads).

There are minimal synchronization issues due to the periodicity of the threads, and the fact that every refresher thread processes a different socket and a different subset of TBP and TGP.

Finally, we also keep a shortcut average value for the last microseconds that correspond to the period with which the adaptive data placement algorithm runs. At the end of every `sample()` call, the recent average (`ravg`) is updated by calling `avg()` with the period of the adaptive data placement algorithm. The algorithm can use this value immediately, without having to calculate it with `avg()`. This is a simple performance enhancement.

6.3 Adaptive Data Placement

First, we describe the abstract workflow of our adaptive data placement algorithm. Then we gradually delve into algorithmic details.

6.3.1 Abstract Workflow

Our adaptive data placement's main component is the *Data Placer (DP)* background thread. Figure 6.4 shows its workflow. The first time tables are loaded into memory, they can be placed across sockets in a round-robin manner. DP runs periodically to monitor the workload and automatically takes care to either move or repartition tables to fix a utilization imbalance.

DP focuses on balancing CPU utilization, under the constraint of not creating a memory bandwidth bottleneck. We remind that we refer to local-only utilization as tracked in Section 6.2. We balance the utilization between sockets with saturated CPU resources and colder

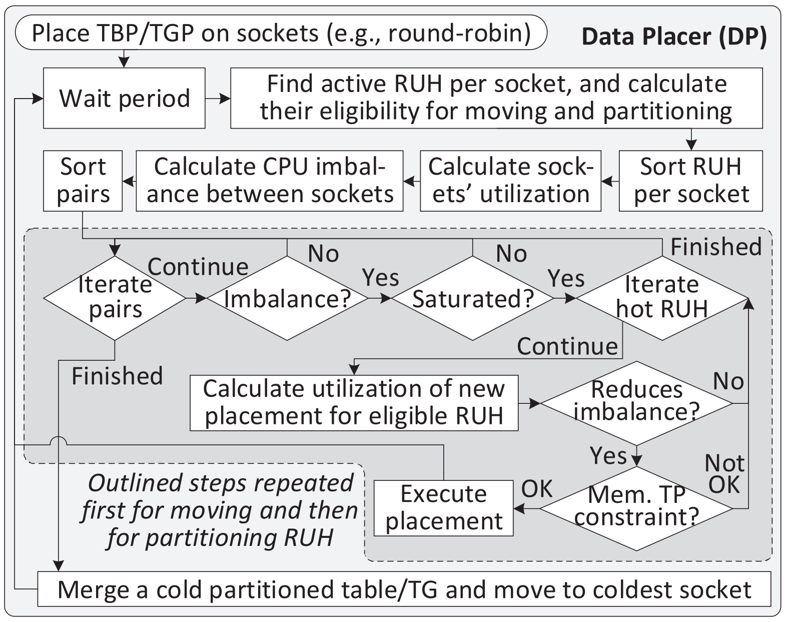

Place TBP/TGP on sockets (e.g., round-robin)
Data Placer (DP)
Wait period
Find active RUH per socket, and calculate their eligibility for moving and partitioning
Sort pairs
Calculate CPU imbalance between sockets
Calculate sockets' utilization
Sort RUH per socket
Iterate pairs
Continue
No
Imbalance?
Yes
No
Saturated?
Yes
Finished
Iterate hot RUH
Continue
Finished
Calculate utilization of new placement for eligible RUH
Reduces imbalance?
No
Yes
Not OK
OK
Execute placement
Mem. TP constraint?
Outlined steps repeated first for moving and then for partitioning RUH
Merge a cold partitioned table/TG and move to coldest socket

Figure 6.4 – Abstract workflow of the Data Placer (DP).

sockets, by moving or partitioning tables. This strategy allows tables that were previously on saturated sockets to potentially increase their utilization using free worker threads on colder sockets (due to intra-query parallelism), increasing the total system utilization.

At every period, DP gets a snapshot of the active RUH (of TBP and TGP) and their recent utilization. It then calculates their eligibility for moving and partitioning, depending on whether their past utilization has been stable. Then, DP sorts the RUH within each socket by their recent utilization in descending order, aggregating them as well to calculate the recent utilization of the sockets. Afterwards, DP calculates the CPU utilization imbalance between all pairs of sockets, and sorts the pairs.

For every pair of sockets, DP investigates whether a new placement can reduce the imbalance. DP proceeds only if the imbalance is over a threshold, and if the hot socket is saturated. If the hot socket is not saturated, the TBP and TGP cannot increase their utilization by exploiting free worker threads on the cold socket. DP iterates the RUH of the hot socket, and examines whether moving or partitioning an eligible RUH's owner (the corresponding TBP or TGP) reduces the imbalance. If additionally it does not create a memory bandwidth bottleneck, DP proceeds to move or partition the RUH's owner.

The outlined steps in Figure 6.4 are first executed while considering moving an eligible RUH's owner. If none can be moved across all socket pairs, we repeat the steps considering partitioning an eligible RUH's owner. This ensures we first prefer moving over partitioning, to avoid any unnecessary overhead of partitioning (see Section 5.5.1).

In case DP did not move or partition a RUH's owner, it goes on to see if there are any cold partitioned tables to merge. This optional step can give the opportunity to cold tables previously partitioned to not suffer the partitioning overhead in case they are again utilized in the future (see Section 6.5.3 for a relevant experiment).

Table 6.1 – Configurable parameters used in our algorithms.

Symbol	Description	Our value
c_p	Period of the Data Placer (DP) algorithm	1 second
c_e	Eligibility threshold for the divergence between the past and recent utilization of a RUH	30%
c_i	Acceptable imbalance threshold between the utilization of a pair of sockets	40% of socket cores
c_s	Lower threshold for considering a socket's utilization saturated	70% of socket cores

Next, we detail how the eligibility of RUH is calculated (see Section 6.3.2), how we reduce the utilization imbalance by moving or partitioning (see Section 6.3.3), and finally how all pieces are put together in DP's algorithm (see Section 6.3.4). The parameters used in our algorithms are summarized in Table 6.1. The values shown are used for our experiments. They are not absolute and can be modified to fit other systems, implementations and use cases. We assume that sockets have the same number of H/W threads and maximum memory TP.

This is a standard assumption for NUMA servers, as processors are typically the same, and administrators are advised to keep sockets balanced (same number and type of DIMM).

6.3.2 Information and Eligibility of RUH

At every period, DP finds the RUH of all TBP and TGP, takes a snapshot of their recent utilization and calculates their eligibility to be moved or partitioned. For every RUH, this information is stored in an *InfoRUH* object, which is defined in Algorithm 1, and calculated through the function `calculateInfoRUH`.

The algorithm first stores the recent CPU and memory throughput utilization of the RUH (lines 2-3). The RUH is deemed active if its utilization is non-zero (line 4). DP continues to calculate the eligibility of the RUH for moving or partitioning. A RUH is deemed eligible if its average utilization in the past does not diverge much from its recent utilization. The amount of time we look into the past depends on the implementation of the move or partition operation.

Algorithm 1 Calculate information and eligibility of a RUH

```
    struct InfoRUH:
        RUH;  // pointer to corresponding RUH object
        recentCpu;  // recent CPU utilization
        recentMem;  // recent memory throughput utilization
        isActive;  // whether the recent utilization is non-zero
        canMove;  // eligible for moving
        canPartition;  // eligible for partitioning
 1: function CALCULATEINFORUH(corresponding RUH)
 2:    recentCpu ← RUH.cpuh.ravg
 3:    recentMem ← RUH.memh.ravg
 4:    isActive ← (recentCpu > 0 and recentMem > 0)
 5:    usMove ← (RUH.pages * speed of moving) * 2
 6:    pastCpu ← RUH.CPUH.AVG(usMove)
 7:    pastMem ← RUH.MEMH.AVG(usMove)
 8:    canMove ← (| recentCpu - pastCpu | < $c_e$ * recentCpu) and
                 (| recentMem - pastMem | < $c_e$ * recentMem)
 9:    canPartition ← canMove and 2 * current partitions ≤ sockets
10:    if canPartition then
11:       usPartition ← usMove + (RUH.pages * speed of partitioning)
12:       pastCpu ← RUH.CPUH.AVG(usPartition)
13:       pastMem ← RUH.MEMH.AVG(usPartition)
14:       canPartition ← (| recentCpu - pastCpu | < $c_e$ * recentCpu) and
                         (| recentMem - pastMem | < $c_e$ * recentMem)
```

For the time to look into the past in the case of moving, we first calculate the time required to move the RUH's owner (line 5), by multiplying its pages with the speed of moving (microseconds per page). The speed of moving and partitioning are calculated at start-up by moving or partitioning a simple mock-up table to another socket, without a concurrent workload. See Table 6.2 for the speeds of the servers we use in our experiments. The speeds are rough estimates. One can improve accuracy by specializing the speeds by socket, or the concurrent workload, or a table's characteristics such as the number of columns, data types, etc. However, we do not need to be precise, since the aim of our eligibility calculations is to disallow instant actions by DP and not delaying them for long.

In our current implementation, queries need to wait while a TBP is moved. We use SAP HANA's functionality to unload a TBP from memory and reload it on the desired socket. We do not use use Linux's `move_pages`, because it can mess up the statistics of SAP HANA's NUMA-aware memory allocators [191]. Due to queries waiting during the move, we double the time to look into the past (line 5). This is optional and simply prolongs the amount of time to look into the past. Conceptually, the additional time corresponds to the time required to "recover" the utilization which drops to zero during the move. We then calculate the average past utilization of both CPU and memory TP (lines 6-7). The RUH is eligible for moving if the past utilization is within a threshold of the recent utilization (line 8).

The algorithm then continues similarly for calculating the eligibility of partitioning (lines 9-14). There are three differences. First, we require that the RUH is also eligible for moving (line 9). This is to enforce the preference of DP to having first considered moving the RUH's owner before considering partitioning it. Second, the amount of time to look into the past consists of partitioning plus the time required to move the new partitions to the correct sockets (line 11). We use SAP HANA's partitioning commands, which, contrary to the implementation of moving, creates the partitions in the background and allows queries [8]. Third, we limit the number of new partitions to the number of sockets to avoid excessive partitioning (line 9).

We note that when we partition a TBP or TGP, we partition the corresponding table or TG into double their previous number of partitions. The reasons why we double the partitions are two. First, partitioning is more time-consuming than moving. Since we decide to partition, we can immediately have double number of partitions and give the algorithm more parts that can potentially be moved later. Second, repartitioning with a number of partitions that is a multiple of the previous number of partitions is fast since each existing partition can be split separately and concurrently [8].

As an illustrative example, Figure 6.5 depicts two RUH, one that is eligible for partitioning, and one that is not. Both RUH have similar recent utilizations. The utilization of the first one, however, is not stable in the past, and is thus ineligible for partitioning yet.

Our adaptive techniques consider the average past utilization to estimate the upcoming utilization and balance it across sockets. The eligibility criteria ensure that the past utilization has been stable. We note that it is possible to extend the prediction of the upcoming utilization by using more sophisticated forecasting techniques [46], which can potentially allow for a more diverse set of eligible workloads.

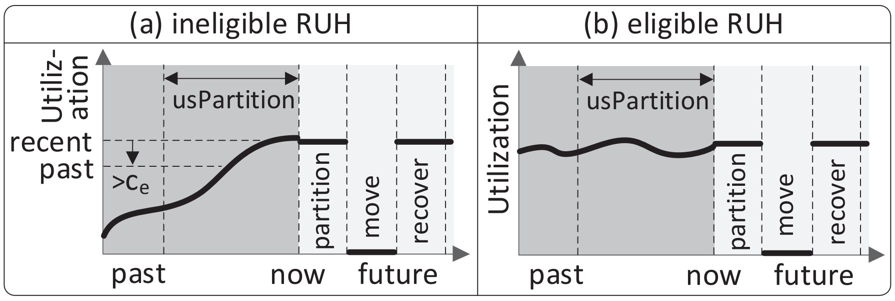


Figure 6.5 – Conceptual examples of calculating the partitioning eligibility of (a) an ineligible RUH, and (b) an eligible RUH.

6.3.3 Reducing the Utilization Imbalance

The purpose of balancing the CPU utilization between sockets is to allow tables to increase their utilization by exploiting free threads on cold sockets when moved or partitioned out of saturated sockets. When considering moving or partitioning a tables, however, we assume the worst case that it does not increase its utilization. This allows us to be on the safe side when calculating the new utilization imbalance, and truly decrease it with every move or partition.

Let us denote the utilization imbalance between two sockets at some timestamp t_n as $imb(t_n)$. If our algorithm does nothing, the imbalance stays the same. If our algorithm moves or partitions a RUH's owner, the imbalance decreases: $imb(t_{n+1}) \leq imb(t_n)$. The sequence $imb(t_n)$ is monotonically decreasing with a lower bound of 0. According to the monotone convergence theorem, the sequence will converge. In our case, since we limit partitioning to a number of partitions capped by the number of sockets, the imbalance may converge to a non-zero value. Also, we set a lower threshold for the imbalance (see Table 6.1), below which DP does nothing. We note that even if tables increase their utilization after the new placement, the resulting imbalance cannot exceed the previous one (see proof after Algorithm 2).

The algorithm for reducing the utilization imbalance of a pair of sockets is presented in Algorithm 2. As mentioned, DP at every period calculates a snapshot of the recent utilization of the system's RUH by creating InfoRUH objects. It also aggregates the utilization of every InfoRUH to calculate the snapshot of the recent utilization of every socket. For every socket, this information is stored in an *InfoSocket* object, which is defined in Algorithm 2 as well.

Function `reduceImbalance` receives two InfoSocket objects and a strategy (move or partition). First, it discerns which socket is the hotter one and which is the colder one (lines 2-3). It then gets the cores and maximum memory throughput of a socket (lines 4-5). These are calculated once on a server (see Section 6.5 and Table 6.2). The current imbalance is calculated (line 6). If the hot socket's utilization is over our saturation threshold and the imbalance is over our threshold (see Table 6.1), the function proceeds (line 7). It iterates the RUH of the hot socket (line 8), and examines whether the utilization imbalance can be reduced by either moving

Algorithm 2 Reduce the utilization imbalance between two sockets

```
     struct InfoSocket:
         recentCpu;  // recent CPU utilization
         recentMem;  // recent memory throughput utilization
         infoRUH[];  // InfoRUH of TBP and TGP placed on the socket
 1: function ReduceImbalance(InfoSocket S1, InfoSocket S2, strategy)
 2:     S1 ← hotter socket of the two, according to recentCpu
 3:     S2 ← colder socket of the two, according to recentCpu
 4:     maxCpu ← H/W threads of a socket
 5:     maxMem ← Maximum memory throughput of a socket
 6:     imb ← S1.recentCpu - S2.recentCpu
 7:     if S1.recentCpu > $c_s$ * maxCpu and imb > $c_i$ * maxCpu then
 8:         for all S1.infoRUH do
 9:             if strategy = move and infoRUH.canMove then
10:                 S1'cpu ← S1.recentCpu - infoRUH.recentCpu
11:                 S2'cpu ← min(S2.recentCpu + infoRUH.recentCpu, maxCpu)
12:                 imb' ← | S1'cpu - S2'cpu |
13:                 S2'mem ← S2.recentMem + infoRUH.recentMem
14:                 if imb' < imb and S2'mem ≤ maxMem then
15:                     Move TBP or TGP to S2
16:                     return true
17:             else if strategy = partition and infoRUH.canPartition then
18:                 S2freeCpu ← maxCpu - S2.recentCpu
19:                 halfRUH ← min(infoRUH.recentCpu / 2, S2freeCpu)
20:                 S1'cpu ← S1.recentCpu - infoRUH.recentCpu + halfRUH
21:                 S2'cpu ← S2.recentCpu + halfRUH
22:                 imb' ← | S1'cpu - S2'cpu |
23:                 S2'mem ← S2.recentMem + (infoRUH.recentMem / 2)
24:                 if imb' < imb and S2'mem ≤ maxMem then
25:                     Partition table or TG into 2 · current partitions, and move all new partitions to
                        their original sockets, apart from one partition of S1 which is moved to S2
26:                     return true
27:     return false
```

(lines 9-16) or partitioning an eligible RUH's owner (lines 17-26). Conceptual examples of the calculations are shown in Figure 6.6.

In case of moving, the sockets' new CPU utilization is calculated (lines 10-11). For the cold socket, we cap the CPU utilization by the socket's cores (line 11). The new CPU imbalance is calculated (line 12). The new memory bandwidth of the cold socket is calculated (line 13), without capping it. If the new CPU imbalance is less than the original, and we do not create a memory bandwidth bottleneck on the cold socket (line 14), we move the TBP or TGP to the cold socket (line 15), and return true (line 16). All TBP of a TGP are moved concurrently.

In case of partitioning, we calculate the cold socket's free threads (line 18). Each new partition's CPU utilization is half of the original utilization, capped by the cold socket's free threads (line 19). Then, we calculate the sockets' new CPU utilization (lines 20-21) and imbalance (line 22). The cold socket's new memory bandwidth is calculated (line 23), without capping it. If the

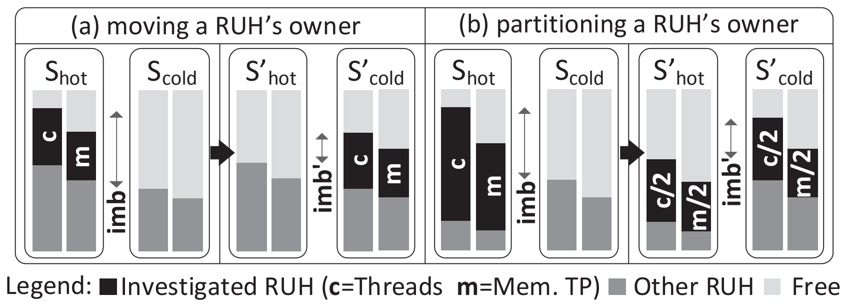

(a) moving a RUH's owner
(b) partitioning a RUH's owner
S_hot
S_cold
S'_hot
S'_cold
c
m
imb
imb'
c/2
m/2
Legend: ■ Investigated RUH (c=Threads m=Mem. TP) ■ Other RUH ■ Free

Figure 6.6 – Calculating the utilization imbalance of a pair of sockets before and after (a) moving, and (b) partitioning.

new CPU imbalance is less than the original, and we do not create a new memory bandwidth bottleneck on the cold socket (line 24), we partition the RUH's owner (line 25), and return true (line 26). We note that we partition all tables of a TG concurrently. After partitioning, we move concurrently all new TBP or TGP to their original sockets, apart from one of the hot socket which is moved to the cold socket.

Proof of imbalance reduction even when the utilization of RUH increase

From here until the end of this Section, we prove that for a stable workload, even if RUHs' owners increase their utilization after the new placement, the resulting imbalance cannot exceed the previous one. When the hotter socket is saturated, the RUHs' owners that were there may have not been at their maximum utilization. They may be able to utilize more free worker threads due to intra-query parallelism. As soon as a RUH's owner is moved or partitioned, the utilization of either the RUH's owner or the other RUHs' owners in the hotter socket can increase. Next, we prove that even if the utilization of RUHs' owners increases, the resulting imbalance is still less than the original imbalance before moving or partitioning, keeping the monotonic decrease of the imbalance and its convergence.

Moving a RUH's owner without increasing utilization. Figure 6.7a shows what Algorithm 2 calculates, by considering that the utilization of RUHs' owners does not increase. Using the figure's variables, the original and the resulting imbalance is:

$$imb_1 = ruh + used_h - used_c$$

$$imb_2 = |\,(ruh + used_c) - used_h\,|$$

If $(ruh + used_c) > used_h$, then the RUH's owner is moved if

$$imb_2 < imb_1 \Leftrightarrow ruh + used_c - used_h < ruh + used_h - used_c \Leftrightarrow used_c < used_h$$

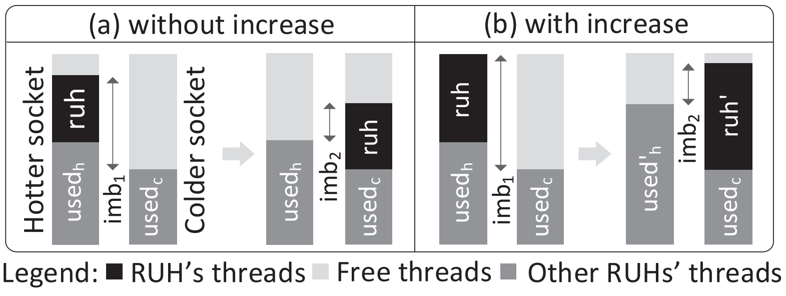


Figure 6.7 – Calculating the imbalance considering (a) that the utilization of RUHs' owner does not increase, and (b) that it actually increases after moving a RUH's owner.

If $(ruh + used_c) < used_h \Leftrightarrow ruh < used_h - used_c$, then the RUH's owner is moved if

$$imb_2 < imb_1 \Leftrightarrow used_h - ruh - used_c < ruh + used_h - used_c \Leftrightarrow 0 < ruh$$

This means that $0 < ruh < used_h - used_c \Rightarrow used_c < used_h$ also in this case. Thus, our algorithm moves the RUH's owner if $used_h > used_c$, irrespective of the RUH's utilization.

Moving a RUH's owner with increasing utilization. Figure 6.7b shows the case when the utilization of RUH may increase after the algorithm decides to move a RUH's owner. A requirement is that the hotter socket needs to be saturated, so that the RUH on the hotter socket can increase their utilization. After moving the RUH's owner to the colder socket, both the moved RUH's owner and the RUHs' owners on the hotter socket may have increased utilization.

The requirements in this case are: (a) $used_c < used_h$, which needs to apply, as we showed before, for the algorithm to move the RUH's owner, (b) $ruh' \geq ruh$, and (c) $used'_h \geq used_h$. The resulting imbalance is:

$$imb_2 = |\,(ruh' + used_c) - used'_h\,|$$

If $(ruh' + used_c) > used'_h$, then we need to prove that

$$\begin{aligned} imb_2 < imb_1 &\Leftrightarrow ruh' + used_c - used'_h \leq ruh + used_h - used_c \\ &\Leftrightarrow ruh' - ruh \leq used_h - used_c + (used'_h - used_c) \\ &\Leftarrow ruh' - ruh \leq used_h - used_c \end{aligned}$$

Which is always true, since the moved RUH's owner can maximally increase its utilization by $used_h - used_c$ (see Figure 6.7b).

If $(ruh' + used_c) < used'_h$, then we need to prove that

$$\begin{aligned} imb_2 < imb_1 &\Leftrightarrow used'_h - ruh' - used_c \leq ruh + used_h - used_c \\ &\Leftrightarrow used'_h - ruh' \leq ruh + used_h \\ &\Leftrightarrow used'_h - ruh' \leq \text{socket's H/W threads} \end{aligned}$$

Which is always true, since both quantities can maximally increase their utilization to a socket's H/W threads. Thus, we proved for the case of moving that even if the utilization of the RUHs' owners increases, the resulting imbalance is equal or less than the original imbalance.

Partitioning a RUH's owner without increasing utilization. Figure 6.8a shows what Algorithm 2 calculates, by considering that the utilization of RUHs' owners does not increase. We assume that with hash (or round-robin) partitioning, both new partitions will have the same utilization. The original and the resulting imbalance is:

$$imb_1 = ruh + used_h - used_c$$

$$imb_2 = |\,(used_h + (ruh/2)) - (used_c + (ruh/2))\,|$$

If $used_h + (ruh/2) > used_c + (ruh/2) \Leftrightarrow used_h > used_c$, then the RUH's owner is partitioned if

$$\begin{aligned} imb_2 < imb_1 &\Leftrightarrow (used_h + (ruh/2)) - (used_c + (ruh/2)) \\ &< ruh + used_h - used_c \Leftrightarrow 0 < ruh \end{aligned}$$

If $used_h + (ruh/2) < used_c + (ruh/2) \Leftrightarrow used_h < used_c$, then the RUH's owner is partitioned if

$$\begin{aligned} imb_2 < imb_1 &\Leftrightarrow (used_c + (ruh/2)) - (used_h + (ruh/2)) < ruh + used_h - used_c \\ &\Leftrightarrow used_c - used_h < (ruh/2) \end{aligned}$$

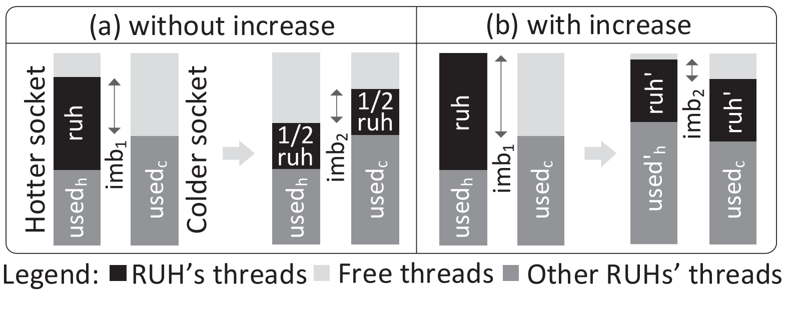


Figure 6.8 – Calculating the imbalance considering (a) that the utilization of RUHs' owners does not increase, and (b) that it actually increases after partitioning a RUH's owner.

Thus, our algorithm partitions a RUH's owner if $used_h > used_c$, but if $used_h < used_c$ it partitions only if $used_c - used_h < (ruh/2)$.

Partitioning a table part with increasing utilization. Figure 6.8b shows the case when the utilization of the RUHs' owners may increase after the algorithm partitions a RUH's owner. Similarly to the moving case, the requirement is that the hotter socket needs to be saturated, so that the RUHs' owners on the hotter socket can increase their utilization. After partitioning, both the partitioned RUH's owner and the RUHs' owners on the hotter socket may have increased utilization. The requirements are that (a) $ruh' \geq ruh$, and (b) $used'_h \geq used_h$.

Let us first examine the case when $used_h > used_c$. As shown before, in this case our algorithm always partitions the RUH's owner. The resulting imbalance is:

$$imb_2 = |\,(used'_h + ruh') - (used_c + ruh')\,| = |\,used'_h - used_c\,|$$

Since we know that $used'_h \geq used_h > used_c$, the imbalance is:

$$imb_2 = used'_h - used_c$$

We need to prove that

$$\begin{aligned} imb_2 < imb_1 &\Leftrightarrow used'_h - used_c \leq ruh + used_h - used_c \\ &\Leftrightarrow used'_h \leq ruh + used_h \\ &\Leftrightarrow used'_h \leq \text{socket's H/W threads} \end{aligned}$$

Which is always true. Let us now examine the case when $used_h < used_c$. The additional requirement is that $used_c - used_h < (ruh/2)$, which, as shown before, needs to apply for the algorithm to partition the RUH's owner. The resulting imbalance is:

$$imb_2 = |\,used'_h - used_c\,|$$

If $used_c > used'_h$, then we need to prove that

$$\begin{aligned} imb_2 < imb_1 &\Leftrightarrow used_c - used'_h \leq ruh + used_h - used_c \\ &\Leftrightarrow 2 \cdot used_c - used'_h - used_h \leq ruh \\ &\Leftarrow 2 \cdot used_c - used_h - used_h \leq ruh \\ &\Leftrightarrow used_c - used_h \leq (ruh/2) \end{aligned}$$

Which is true as one of the requirements. Finally, in the case that $used_c < used'_h$, then we need to prove that

$$\begin{aligned} imb_2 < imb_1 &\Leftrightarrow used'_h - used_c \leq ruh + used_h - used_c \\ &\Leftrightarrow used'_h \leq ruh + used_h \\ &\Leftrightarrow used'_h \leq \text{socket's H/W threads} \end{aligned}$$

Which is always true, as $used'_h$ can maximally increase its utilization to the socket's cores. Thus, we proved for the case of partitioning that even if the utilization of the RUHs' owners increases, the resulting imbalance is equal or less than the original imbalance.

6.3.4 Data Placer

Our final Algorithm 3 implements the data placer (DP). After waiting for a period (line 2), DP goes through all TBP and TGP (line 3). For every one, it calculates its InfoRUH (line 5), and if it is active, DP adds it to the appropriate InfoSocket, aggregating its recent utilization as well to the socket (line 7). At this point, we have a snapshot of the recent utilization in the past period. DP then sorts the RUH of every socket by their recent CPU utilization in descending order (line 8). This makes intensely used RUH to be considered first for moving or partitioning.

Algorithm 3 Data Placer

```
 1: while true do
 2:   wait $c_p$
 3:   InfoSocket[] ← initialize an InfoSocket object for every socket
 4:   for all loaded TBP and TGP do
 5:     InfoRUH ← CALCULATEINFORUH(RUH)
 6:     if InfoRUH.isActive then
 7:       Add and aggregate InfoRUH to the appropriate InfoSocket
 8:   Sort the InfoRUH in every InfoSocket by their recentCpu
 9:   list<tuple<InfoSocket, InfoSocket, imbalance>> ← empty list
10:   for all pairs of InfoSocket do
11:     Add new tuple(S1, S2, | S1.recentCpu - S2.recentCpu |) to list
12:   Sort list by imbalance
13:   for all pairs of InfoSocket in the sorted list do
14:     if REDUCEIMBALANCE(S1, S2, move) then
15:       goto line 2
16:   for all pairs of InfoSocket in the sorted list do
17:     if REDUCEIMBALANCE(S1, S2, partition) then
18:       goto line 2
19:   for all partitioned tables and TG in the catalog do
20:     pages ← sum pages of all RUH of table or TG
21:     usMove ← (pages * speed of moving) * 2
22:     usPartition ← usMove + (pages * speed of partitioning)
23:     if all RUH have 0 average utilization during last usPartition then
24:       Merge and move to the coldest socket in the background
```

DP then calculates the imbalance of every pair of sockets (lines 9-11). The pairs of sockets are sorted by their imbalance (line 12), so that we first examine the pair with the greatest imbalance. For all pairs (line 13), we try to reduce their imbalance by moving a RUH's owner (line 14). If nothing is moved, we try again to reduce their imbalance by partitioning a RUH's owner (lines 16-18). If DP moves or partitions a RUH's owner, it goes back to waiting.

If DP does not move or partition a RUH's owner, it attempts to merge cold partitioned data. It iterates through all partitioned tables and TG in the system catalog (line 19). DP checks whether a table or TG is cold by looking into its past utilization. The amount of time to look into the past is calculated (lines 20-22) as in Algorithm 1 for the case of partitioning, with the difference of summing the pages of all RUH (since merging creates a single partition). If the average past CPU and memory TP utilization of all RUH is zero (line 23), a background request is initiated (line 24) which merges the table, or all involved tables in case of a TG, and moves it to the coldest socket. We use a `set` internally to keep track of tables or TG undergoing merging in order to consider them ineligible for moving or partitioning until their merging completes.

Finally, we note that DP balances primarily the CPU utilization under the constraint of not creating memory bandwidth bottlenecks. This is to allow newly placed data to potentially increase their utilization. Since we balance local-only CPU utilization, this can indirectly balance memory TP as well as shown in many of our experiments in Section 6.5. As an extension, one may wish DP to continue explicitly balancing memory TP after CPU utilization is balanced, under the constraint of not increasing the CPU imbalance.

6.4 Adaptive Task Stealing

As we showed in Chapter 5, stealing memory-intensive tasks can hurt overall performance. Here, we pinpoint the switching point when tasks become memory-intensive enough that they should not be stolen across sockets. The switching point depends on the hardware, the implementation, and the workload. For this reason, we propose a calibration experiment that the DBMS can run once on a server to find the switching point. In order to fully control the memory intensity of tasks, we avoid the SQL layer of the DBMS, and use immediately the task scheduler on simple data structures.

Calibration experiment. We place four 4 GB vectors of randomly generated `doubles` on each of the half sockets of the server. Thus, half of the sockets can have local accesses, while the rest will either steal remote tasks or stay idle. The workload consists of one client thread per vector. Each client continuously issues a query that sums the elements of its corresponding vector. We measure the total throughput (TP). The client parallelizes the query with a number of tasks equal to the number of hardware threads in a socket. The tasks are given equi-sized ranges of the vector to sum. The reason we use four vectors per socket, and one client thread per vector, is to have enough tasks to saturate the local socket and more tasks that can potentially be stolen by another socket (that does not have data).

In order to control the memory intensity (in terms of memory throughput) of the summation, we raise each element of the vector to a varying power `n`: $sum = v_1^n + v_2^n + v_3^n$... We implement each task's summation using a for loop to raise its element to the desired power `n`. As we increase `n`, we increase the time spent in the for loop, thus increasing the CPU intensity.

Figure 6.9 shows the results of the calibration experiment for the three servers we use in our experiments (see Section 6.5). For all servers, disallowing stealing results in the best TP for lower values of `n`, since the summation is more memory-intensive. This is also shown by the system's memory TP, which almost saturates the memory bandwidth of half the sockets. It is also shown by the average memory TP of the task class (tasks belong to a single class in this experiment).

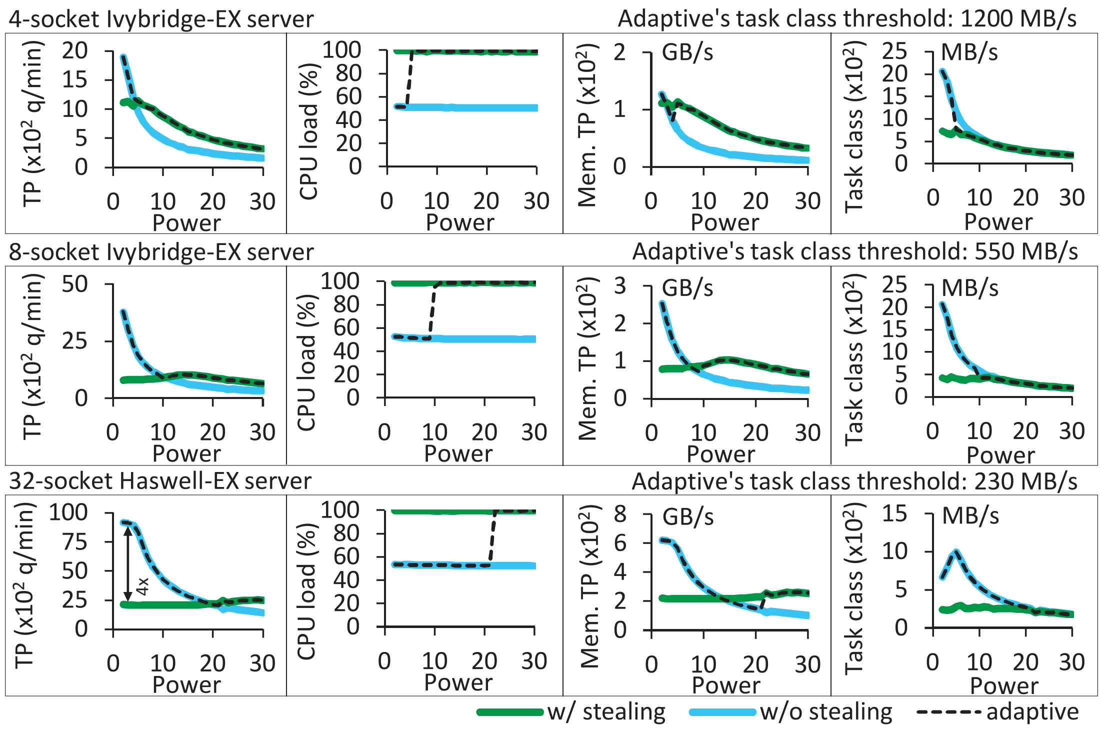


Figure 6.9 – Calibration experiment for three NUMA servers.

We note that disallowing stealing only achieves a 50% CPU load, since half of the sockets have data. But it is better for memory-intensive workloads than allowing stealing, achieving up to 4x better TP (on the 32-socket server). Stealing has 100% CPU load, but saturates the interconnect network and overwhelms the already saturated remote memory controllers. The overwhelmed memory controllers achieve much lower overall memory TP (for both local and remote tasks) than the case of disallowing stealing.

As `n` increases, the workload becomes more CPU-intensive, and the memory TP of the system and the task class finally starts decreasing. In terms of CPI (cycles per instruction), e.g., on the 4-socket server for the case when stealing is disabled, it starts at 0.83 and gradually increases up to 1.25 (for n = 30). At a switching point, stealing becomes better, and remote tasks can be satisfied through the interconnects and the remote memory controllers sufficiently.

At the switching point, we mark the memory TP of the task class as the threshold for stealing vs. not stealing. Our adaptive task stealing uses this threshold at run-time. If the exponential average of a task class becomes higher than the threshold, stealing is disallowed for this task class. If the exponential average becomes lower than the threshold, stealing is allowed.

Figure 6.9 shows the threshold pinpointed for each server, and also shows the adaptive task stealing that uses this threshold. The adaptive line achieves the best throughput for all cases of power n, successfully allowing stealing at the switching point.

Finally, we note that the calibration experiment can be further extended to specialize the switching point for different number of sockets having data and different CPU utilization per socket. An adaptive technique for finding the switching point at run-time, or disallowing memory-intensive tasks if a socket is saturated, is not ideal as we cannot gauge whether the memory bandwidth of a socket is saturated due to sufficient stealing or hurt due to overwhelming stealing. Our current calibration experiment is sufficient for our use cases and experiments, as it roughly finds out the switching point for the average case where half of the server's sockets have active data.

6.5 Experimental Evaluation

We first present our experimental configuration. Then, we present results of a custom benchmark and finally of a read-only variant of the TPC-H benchmark.

Experimental configuration. We use a prototype built on SAP HANA (SPS11) with our NUMA-aware task scheduler. We add support for tracking resource utilization (see Section 6.2), and employ our adaptive NUMA-aware data placement (see Section 6.3) and task stealing (see Section 6.4).

For all experiments, we warm up the DBMS, we admit all clients and disable result caching. LLC misses, CPU load, and memory throughput (TP) are gathered from Linux and H/W counters (integrating Intel PCM [196]). The utilizations of RUH are the local-only utilizations

Table 6.2 – Further characteristics of the NUMA servers of Table 2.2 that we use in this chapter's experimental evaluation.

Statistic \ Server	4x15-core Intel Xeon E7-4880v2 (Ivybridge-EX) at 2.50GHz	8x15-core Intel Xeon E7-8880v2 (Ivybridge-EX) at 2.50GHz	32x18-core Intel Xeon E7-8890v3 (Haswell-EX) at 2.50GHz
Stealing threshold	1200 MB/s	550 MB/s	230 MB/s
Move us/page	59	63	60
Partition us/page	109	123	129

we track (via the same H/W counters). The imbalance metric corresponds to the maximum imbalance between any two sockets, when calculated by DP. The imbalance fluctuates since the averaged sampled utilizations of RUH also fluctuate, but in general is decreased with a stable workload. Results shown with a timeline consist of a single run, while any data points are averages of at least three iterations with a standard deviation <10%. Table 6.2 shows further characteristics of the servers that we use in the following experiments.

Custom benchmark. Our dataset has 64 tables (`TBL1-64`). For each table we generate a CSV file of 50 million rows, around 3.2 GB, for a total of 204 GB files. Each table has an `ID` integer column (PK), 8 additional columns (`COL1-8`) of random integers (uniform distribution), and a partitioning specification (hash) on `ID`. The 8 columns have bitcases 17 to 24, so as to have different number of unique values. Each experiment mentions the initial table placement.

The workload is generated with a Java application on a different server. Clients continuously issue queries and we measure the total throughput (TP). At each experiment, we mention how many clients are used, which query type(s) they issue, which table(s) they target, and which selectivity they use. The possible query types are:

(a) `SELECT COLx FROM TBLy WHERE COLx >= ? AND COLx <= ?`. The client selects a random column from its target table. The query involves both scan phases mentioned in Section 5.3.1.

(b) `SELECT COL1, SUM(TO_DOUBLE(COLx)) FROM TBLy WHERE COLx >= ? AND COLx <= ? GROUP BY COL1`. The client selects a random column (`COL2-8`) from its target table to aggregate and group-by `COL1`. This query involves the aggregation phases mentioned in Section 5.5. We choose `COL1` for the group-by because it has the least number of unique values. We cast to double to avoid potential numeric overflow errors.

(c) `SELECT TBLz.COLx FROM TBLy, TBLz WHERE TBLy.ID = TBLz.ID AND TBLy.COLx >= ? AND TBLy.COLx <= ?`. The client joins two target tables on the `ID` column. A random column is selected to filter and project. This query involves the equi-join steps mentioned in Section 5.5. With multiple table parts, the equi-joins are copartitioned.

Before each experiment begins, we let clients build a prepared statement for each query they can issue. There are no thinking times. The clients do not fetch results, in order to not let the network transfer dominate. Each TP value in a timeline corresponds to the slope of the achieved queries during the previous 30 seconds.

6.5.1 Adaptive Data Placement

The first experiment realizes the introductory example of Figure 6.1. TBL1 and TBL2 are placed on socket S1, TBL3 on S2, and TBL4 is partitioned across S3 and S4. Each of the tables TBL1-3 are targeted by 64 clients executing query (a) with a low selectivity (0.001%) for 5 minutes. Figure 6.10 shows the timelines of the throughput (TP), the utilization imbalance, and other performance measurements such as H/W counters and our tracked utilization (RUH).

At the beginning, only S1 and S2 execute queries as shown by their RUH. Queries are dominated by the scan's first phase ("IV-Scan"). Tasks are memory-intensive as shown by the task class's memory TP, which is over the stealing threshold. That is why adaptive stealing disallows stealing, and most LLC misses are local. As shown by the tables' RUH, TBL1 and TBL2 share S1, while TBL3 fully utilizes S2.

DP recognizes the imbalance, but does not take action because the TBP are not yet eligible to be moved or partitioned. DP searches the catalog to find TBL4 which is partitioned and cold (thus not shown in Figure 6.10), and at 16 sec starts a background request to merge it. The merge finishes at 64 s, and the single TBP is moved to S4 (a cold socket) at 106 s. The merge and move contribute to the small bump in the CPU load and memory TP of S3 and S4. Another

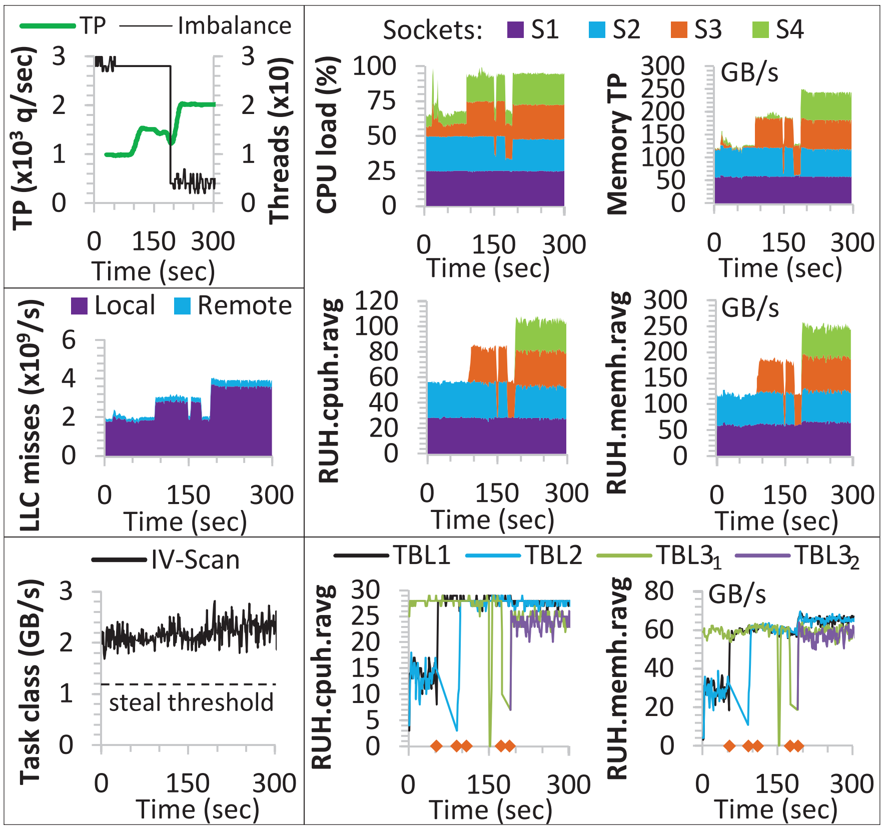


Figure 6.10 – Adaptive data placement of three active tables (4-socket server).

reason for their increased CPU load is that their worker threads attempt to steal tasks from other sockets, but tasks are memory-intensive and cannot be stolen in this experiment. Since S3 and S4 do not process any queries, we do not account this busy CPU load in their RUH.

At 53 s (see the orange markers on the timeline of the tables' RUH graphs), DP examines the pair of S1 and S3. It fixes their imbalance by moving TBL2, which has become eligible for moving, to S3. The move completes at 91 s. Overall TP and memory TP are increased.

Next, DP detects that there is still a utilization imbalance because 3 sockets are utilized and S4 is not (as shown by its RUH). At 108 s, DP examines the pair of S2 and S4. It decides to fix their imbalance by partitioning TBL3, which is eligible for partitioning, into two parts. At 174 s, partitioning completes, and the two TBP are moved to S2 and S4 concurrently, which completes at 190 s.

After that point, the imbalance is decreased within our configured threshold, and there are no more actions. In comparison to the beginning of the experiment, overall memory TP is 2x more, and TP is also 2x more.

6.5.2 Adaptive Task Stealing

To show the effect of adaptive task stealing, we use scans of varying selectivity. We place TBL1 on S2 and TBL2 on S4. Adaptive placement is disabled. Each of the tables is targeted by 256 clients executing query (a) with the specified selectivity. Half of the sockets have local tasks, while the other half would need to steal. For each selectivity, we execute 5 min runs of: enabled stealing for all tasks, disabled stealing for all tasks, and adaptive stealing. We report each run's average throughput (TP). The results are shown in Figure 6.11.

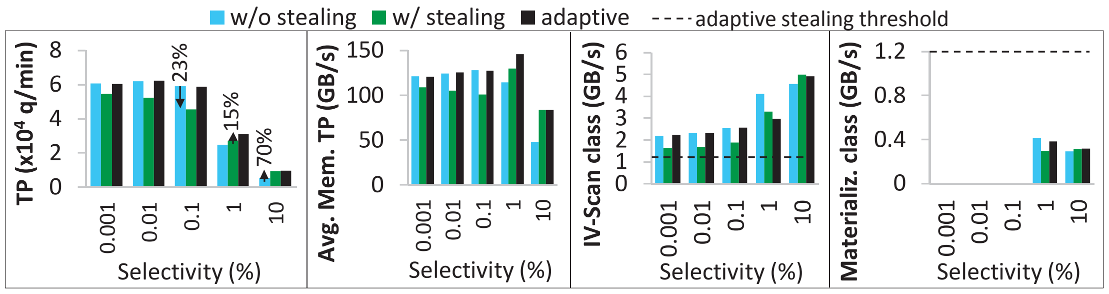


Figure 6.11 – Experiment showing how adaptive task stealing disallows stealing of memory-intensive classes (4-socket server).

For low selectivities, the scan's first phase ("IV-scan") dominates. Tasks are memory-intensive and stealing hurts TP by up to 23% for the case of 0.1% selectivity. As selectivity increases, the scan's second phase (materialization) dominates and is parallelized. The fewer IV-scan tasks can utilize more memory bandwidth on their socket. The dominating materialization tasks are less memory-intensive (with less memory throughput), due to their random accesses to the dictionary, and thus stealing helps improve throughput by up to 70% for the case of 10% selectivity. Adaptive stealing achieves the best throughput of either stealing or not stealing in all cases of selectivity. It can also, e.g., for the case of 1% selectivity, further improve performance by 15%. This is due to disallowing stealing of IV-scan tasks, and allowing stealing of materialization tasks, instead of taking a static strategy for all task classes.

6.5.3 Partitioning Overhead

Here, we show how our adaptive data placement can avoid the overhead of unnecessary partitioning, focusing on aggregations. We initially partition TBL1-8 across all sockets of the 8-socket server. The experiment has three consecutive 5min phases. In the first, each table is targeted by 8 clients executing query (b) with a high selectivity (10%). The second phase has no activity. The third phase is the same as the first. Adaptive task stealing is enabled. Most tasks are not very memory-intensive due to high selectivity and can be stolen. Figure 6.12 shows the results.

During the second phase, all tables are merged. During the third phase, DP moves tables that happened to be merged on the same socket to balance utilization. TP reaches 1.7x of the TP of the first phase, because there is no partitioning overhead, and the server can be saturated with non-partitioned tables. This is also shown by the improved CPU load, memory TP and local LLC misses.

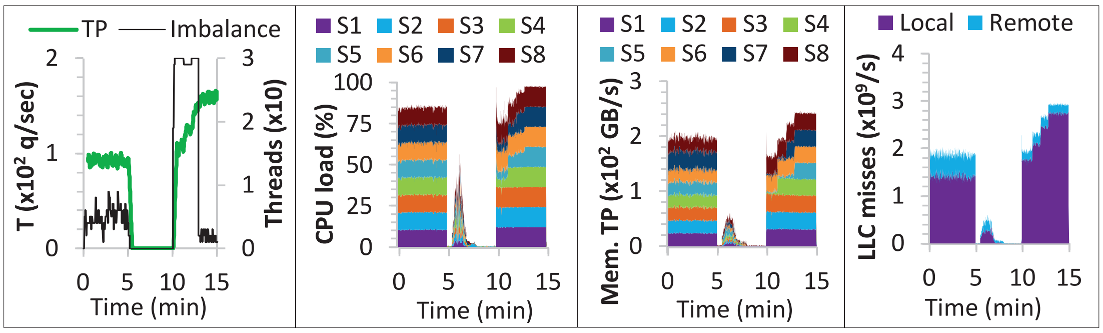


Figure 6.12 – The overhead of partitioning for aggregations (8-socket server).

Impact of groups. Next, we detail how the partitioning overhead for aggregations increases as the number of groups increases. This is an extension of the implications we showed in Section 5.5.1. We use TBL1-8 either partitioned (8 TBP/table) or non-partitioned (1 TBP/table), placed round-robin across the sockets. Adaptive placement is disabled. Each table is targeted by 8 clients issuing a variation of query (b) that selects COL8 with a high selectivity (10%), and groups-by a different column. As we group-by COL1 through COL7, the bitcase of the group-by column increases, and so the number of groups increases. Figure 6.13 shows the results. Each run is 5 minutes. We also include a case without a group-by, and show the percentage of remote LLC misses out of all LLC misses.

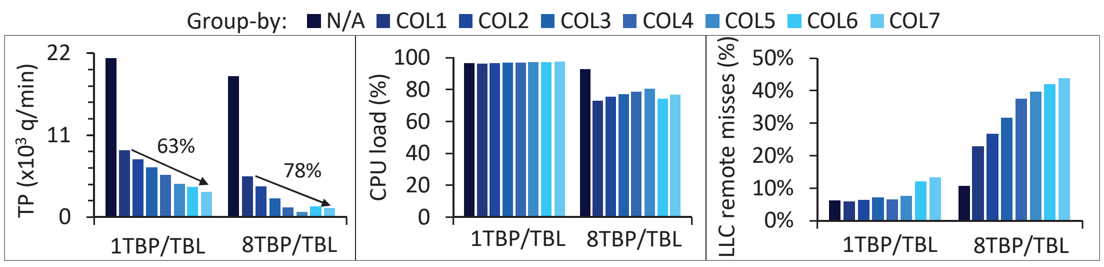


Figure 6.13 – The impact of the number of groups (8-socket server)

For the case without grouping, both data placements perform similarly, with mostly local accesses. As the number of groups increases, TP drops since we need to work on more hash tables. For 1 TBP/table, merges happen locally to each socket, while for 8 TBP/table there is the cost of the global dictionary and merges may need to access 7 other sockets. For this reason, the drop in TP from `COL1` to `COL7` for 8 TBP/table is worse (78% drop) than for 1 TBP/table (63% drop). This is also reflected in the percentage of LLC remote misses. Additionally, there is a decline in CPU load in the partitioned case due to scheduling overhead. For several group-by cases, the TP of 1 TBP/table is more than 2x than the TP of 8 TBP/table.

We note that the TP for the last two group-by columns in the partitioned case shows an increase jump due to an internal SAP HANA optimization threshold for the number of groups. This optimization does not affect the NUMA implications for the aggregations, since the percentage of remote accesses continues to increase with the number of groups.

6.5.4 Changing Workload

Here, we show a workload with three consecutive phases. We place 8 tables on each socket of the 8-socket server (all 64 tables are placed). Clients execute query (a) with low selectivity (0.001%). The first phase lasts 15 min, and only `TBL56` (on S7) and `TBL64` (on S8) are targeted by 512 clients each. The second phase lasts 5 min, and all 64 tables are targeted by 16 clients each. The third phase lasts 10 min, and only `TBL1-4` (on S1) and `TBL9-12` (on S2) are targeted by 128 clients each. Figure 6.14 shows the results.

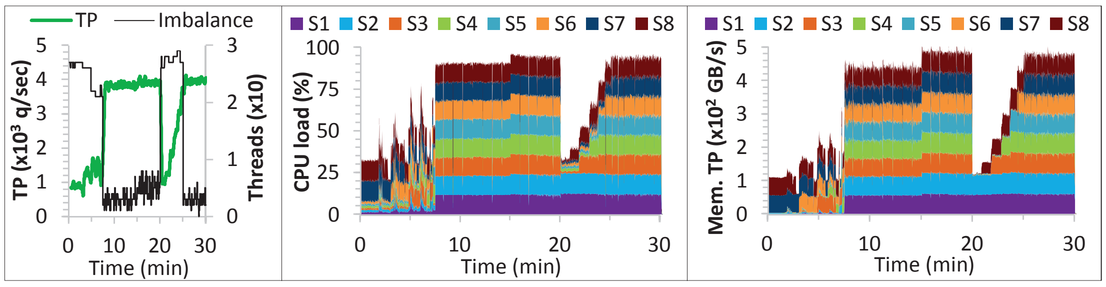


Figure 6.14 – Three different workload phases (8-socket server).

During the first phase, DP gradually partitions the hot tables to fill all sockets and reach maximum TP. During the second phase, all tables become hot, TP stays at the maximum, and DP takes no actions. At the third phase, only two sockets are used, and DP gradually moves their hot tables to fill all sockets and reach maximum TP. This experiment shows how our adaptive algorithms handle changing workloads.

6.5.5 Workload Mix

Here, we have an initial complex placement, a stable workload mix, and show that DP gradually reaches a final stable state. We use TBL1-7, initially placed on 4 sockets of the 8-socket server as shown in Figure 15. We use 128 clients, continuously issuing a random query out of the queries shown in Figure 6.15. We define a TG for TBL1-2 and another TG for TBL3-5, thus the queries' joins between these tables are copartitioned. Figure 6.15 shows the results.

Clients' queries in format <query type, selectivity, target table(s)>:

▪ (a), 1%, TBL1	▪ (a), 0.1%, TBL6	▪ (b), 1%, TBL3	▪ (b), 0.1%, TBL7
▪ (b), 1%, TBL6	▪ (c), 1%, TBL1-2	▪ (c), 1%, TBL3-4	▪ (c), 0.1%, TBL4-5

Initial and final placement of tables on sockets in format TBLx$_{\text{part-number}}$:

S1	S2	S3	S4	S5	S6	S7	S8
-	TBL6	-	TBL1 TBL2	-	-	TBL3$_1$ TBL4$_1$ TBL5$_1$	TBL3$_2$ TBL4$_2$ TBL5$_2$ TBL7

➡

S1	S2	S3	S4	S5	S6	S7	S8
TBL3$_3$ TBL4$_3$ TBL5$_3$ TBL7	TBL6$_1$	TBL6$_2$	TBL1$_1$ TBL2$_1$	TBL3$_2$ TBL4$_2$ TBL5$_2$	TBL3$_4$ TBL4$_4$ TBL5$_4$	TBL3$_1$ TBL4$_1$ TBL5$_1$	TBL1$_2$ TBL2$_2$

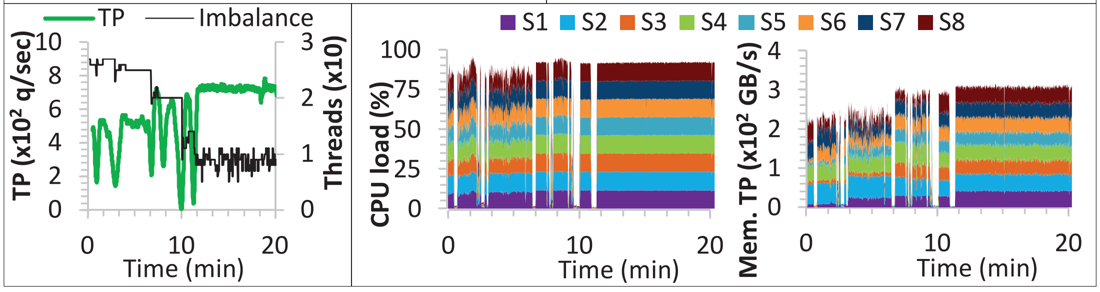


Figure 6.15 – Balancing the utilization of a stable workload mix (8-socket server).

Initially, four sockets have memory throughput. We note that CPU load is saturated, as there are tasks that are not very memory-intensive and can be stolen. DP moves and partitions tables to reach the placement shown in Figure 6.15 with a balanced utilization. All sockets finally have memory throughput and mostly local accesses, improving throughput by 44% vs. the initial throughput. Since in this experiment all clients issue all queries, there are drops in throughput while DP holds an exclusive lock for moving a table.

6.5.6 TPC-H benchmark

Here, we show the TPC-H (read-only) [15] benchmark with a scaling factor 30 (30 GB files). We use 512 clients, each continuously issuing a random query out of the 22 templates. We set

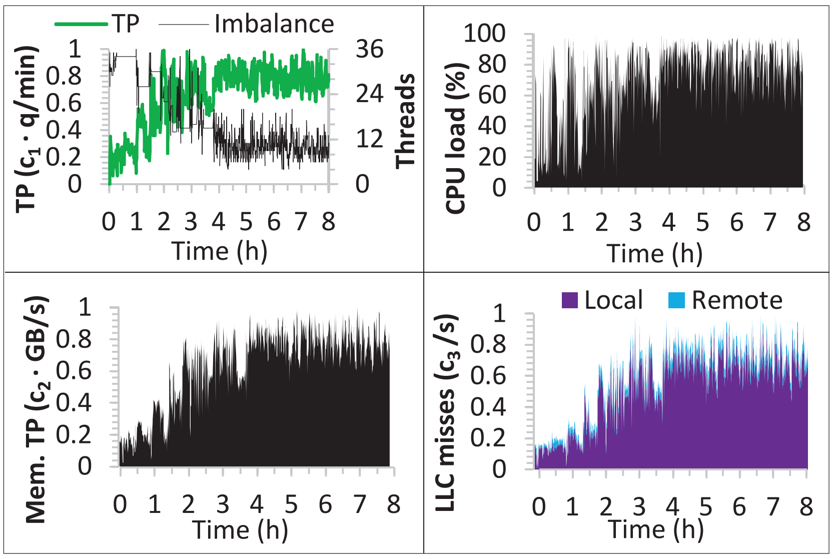


Figure 6.16 – TPC-H throughput run (32-socket server).

our configurable parameters c_p (DP period) to 10 sec in order to better capture the longer-running queries of TPC-H, and c_s (saturation threshold) to 40% because the workload does not constantly saturate CPU resources. The tables are initially placed on a single socket of the 32-socket server. Adaptive task stealing is enabled. We define a partitioning specification for all tables except nations and regions, and a TG for lineitems and orders so that they are copartitioned on the orderkey columns. The results are shown in Figure 6.16, which are normalized due to legal reasons with undisclosed constants to the maximum observed values. This does not hinder us from showing DP's impact.

Initially, one socket has memory TP. We note that more sockets have CPU load, as there are tasks that are not very memory-intensive and can be stolen. Query throughput (TP) is only around 0.2. DP gradually moves and repartitions tables and the TG to balance utilization. Finally, all sockets have data, the TG of lineitems and orders has 32 partitions, customers have 4 partitions, partsupp has 8 partitions, while the remaining tables are not partitioned. All sockets have memory TP and mostly local accesses. TP reaches around 0.8. Assuming the initial TP corresponds to the typical case when an administrator simply loads the dataset, our adaptive data placement helps improve TP by 4x.

6.6 Conclusions

In this chapter, we confirm once again that a static strategy for data placement and task scheduling should not be employed. We show that unnecessary partitioning involves an overhead of up to 2x in comparison to not partitioning. For this reason, we develop an adaptive data placement algorithm that can track a utilization imbalance across sockets, and can move or repartition tables at run-time to fix the imbalance. Furthermore, we show that inter-socket stealing of memory-intensive tasks can hurt throughput by up to 4x in comparison to not stealing. For this reason, we develop an adaptive technique that disallows stealing at run-time for tasks whose memory intensity exceeds a threshold for a NUMA server.

7 Conclusions and Future Directions

Modern multi-core servers provide increasing parallelism but with non-uniform memory access architectures. In this thesis we identify the major inhibitors for the performance of typical execution engines while scaling up concurrent analytical workloads on modern multi-socket multi-core servers. We claim that the execution engine needs to be redesigned in order to exploit the increasing concurrency of modern analytical workloads by employing sharing techniques across concurrent queries, and consider the non-uniformity of modern multi-socket servers by employing adaptive data placement and task scheduling techniques. The insights and techniques described in this thesis contribute toward this goal.

This chapter summarizes our contributions, discusses future directions, and highlights the impact of this thesis.

7.1 Thesis Summary

In this thesis we identify two main dimensions that inhibit the efficient scalability of highly concurrent analytical workloads on modern multi-core servers: the absence of sharing data and work across concurrent queries, and the absence of NUMA-awareness in the execution engine. Next, we detail our contributions with respect to these two dimensions.

Sharing. Query-centric execution engines, that execute each query independently of other queries, miss numerous opportunities for sharing data and work across concurrently running queries. We categorize state-of-the-art run-time sharing techniques into reactive and proactive sharing techniques. Reactive sharing identifies and shares common sub-plans across concurrent queries. Proactive sharing builds a global query plan with shared operators to evaluate the whole query mix. We integrate reactive and proactive sharing in the same system and perform an extensive sensitivity analysis of reactive vs. proactive sharing. We corroborate previous work on that proactive sharing can effortlessly handle a large number of concurrent queries with similar query plans. Our main contributions are a pull-based model for reactive sharing, and recognizing that reactive and proactive sharing are orthogonal sharing techniques that can be combined. Applying reactive sharing on top of proactive sharing takes the best of the two worlds and their combination is best suited for highly concurrent workloads.

NUMA-awareness. An execution engine that is not aware of the non-uniformity of multi-socket servers can suffer significant performance degradation due to increased memory access latencies and potential bandwidth bottlenecks in the interconnects and the sockets' memory controllers. NUMA-awareness is especially important for the numerous modern main-memory DBMS whose performance is directly affected by memory accesses (see Section 2.1). We claim that the execution engine needs to coordinate data and thread placement in order to efficiently utilize a multi-socket server.

In regard to data placement, we evaluate different strategies for placing data across sockets. We can place data on one socket or partition it across multiple sockets if the workload is skewed. We identify that always employing partitioning is not ideal, as partitioning involves an overhead for executing analytical operators. For this reason, we propose an adaptive data placement technique that employs partitioning only when needed under a skewed workload in order to balance utilization across the sockets of a multi-socket server.

In regard to thread placement, we claim that the execution engine should employ a task scheduler that can reflect the non-uniformity of the underlying hardware. Each socket has its own worker threads and task queues. We can queue a task to the socket where its processed data is placed. If there is skew in the workload, task stealing can balance CPU load across the sockets. We identify, however, that inter-socket task stealing of memory-intensive tasks can decrease overall performance by overwhelming the interconnects and the already saturated memory controllers. For this reason, we propose an adaptive task stealing technique that disallows inter-socket stealing of memory-intensive tasks.

7.2 Future Directions

Looking ahead, we can identify several opportunities for extending our work in this thesis.

In the realm of run-time sharing, it would be interesting to explore the benefits for OLTP workloads as well. SharedDB [77] supports sharing for OLTP and OLAP workloads, but its batched execution is not well suited for long-running queries. It has been suggested that MVCC and snapshot isolation can be exploited to share across the same snapshots of data [47]. Furthermore, reactive and proactive sharing do not exploit common sub-expressions as proposed by multiple-query optimization [175]. An analysis of the challenges and potential benefits of combining common sub-expressions with reactive and/or proactive sharing is missing. Moreover, we show in this thesis that proactive sharing is not beneficial for low concurrency in comparison to the query-centric model. The turning point, however, when proactive sharing becomes beneficial needs to be pinpointed. It would be interesting to develop a prediction model similar to reactive sharing [98], but for proactive sharing. Additionally, it would be interesting to employ and evaluate run-time sharing techniques in the processing model suggested by HyPer [117, 143]. The state-of-the-art run-time sharing techniques (see Section 2.2) adapt the traditional Volcano [82] style iterator model of processing, pipelining tuples across operators using intermediate buffers. In contrast, tuples in HyPer are passed through whole operator pipelines without intermediate buffers.

In the realm of NUMA-awareness, we assume in this thesis that all sockets have the same processor and memory characteristics. Although this is largely true for today's multi-socket servers, heterogeneity may become standard in future servers. An interesting extension of our work would be to consider sockets with different characteristics such as heterogeneous processors and different types of memories such as upcoming non-volatile memories. As an example of mainstream heterogeneous processors, Intel plans to integrate CPU and FPGA [87, 165]. NUMA-awareness is needed to consider where data is placed in relation to the specialized processing done with the FPGA. Non-volatile memories have different characteristics than main memory, such as different write latencies [133]. NUMA-awareness on a server with both main memory and non-volatile memory may need to be specialized, e.g., to store base tables on non-volatile memory and process queries, that write intermediate results, on the main memory of sockets near the related base tables.

On large-scale servers with hundreds of sockets, another interesting extension of our work would be for our adaptive data placement algorithm to take multiple actions at the same time, e.g., moving or partitioning tables to balance the utilization of multiple socket pairs. Further optimization opportunities would be to consider placing table partitions on nearby sockets at the expense of not aggressively reducing the utilization imbalance between far-away sockets. Moreover, another opportunity would be an automated way of recognizing associated tables, the dominant join predicates, and forming table groups for copartitioning the tables, by tracking the workload at run-time. This would require the adaptation of an offline solver for distributed data placement (see Section 2.4.2) or specially for copartitioned joins [203], for execution at run-time. Energy efficiency is another aspect that becomes increasingly important nowadays. It would be interesting to investigate how power management techniques, such as dynamic voltage-frequency scaling, can be integrated with our adaptive NUMA-aware techniques in order to improve the energy efficiency of analytical workloads [160].

Finally, although sharing and NUMA-awareness are orthogonal dimensions, an opportunity lies in integrating and combining both dimensions in the same system to explore how NUMA-aware sharing can further improve the performance of analytical workloads on multi-socket servers. The implications of this thesis can provide valuable insight. For example, memory-intensive operators, with a high memory throughput, could be avoided to be shared across sockets, in order to avoid bandwidth bottlenecks, while less memory-intensive operators could be shared across sockets.

7.3 Impact

This thesis is a first step towards improving the scalability of concurrent analytical workloads on modern multi-socket multi-core servers. Our analysis on run-time sharing techniques show that sharing can improve the performance of query-centric execution engines by up to 6x. We also set the basis for combining reactive and proactive sharing techniques. Our novel application of reactive sharing to proactive sharing can further improve the performance of proactive sharing by up to 2x. Our adaptive NUMA-aware data placement and task scheduling strategies pave the way for designing execution engines that consider the non-uniformity of the underlying hardware and adapt to the workload in order to efficiently utilize all the sockets of a multi-socket server. Our experiments show that our adaptive data placement can improve performance vs. static data partitioning by up to 2x, while our adaptive task stealing can improve performance vs. always stealing by up to 4x.

The implications of this thesis are expected to become increasingly important as multi-socket multi-core servers keep scaling up. Indeed, there is already a 256-socket rack-scale server available (see SGI UV 3000 [21]), and future servers may efficiently scale up by dispensing cache coherency across sockets and supporting a fast communication protocol across sockets [148]. In the battle between scaling up vs. scaling out, there is no clear winner. Scaling up can be better suited for several analytical workloads in terms of performance and cost than scaling out [32, 50]. Scale-up servers can also be used in scale-out infrastructures. Multi-socket servers, starting with as few as two sockets, are already used in the Amazon EC2 cloud [17].

Thus, the impact of both sharing and NUMA-awareness is expected to become more prominent. As more and more computing resources become available, the DBMS is expected to service an increasing number of concurrent queries. A query-centric execution engine will miss an increasing number of opportunities for sharing across concurrent queries, saving resources, and further improving its performance. Moreover, with an increasing number of sockets, the significance of our adaptive NUMA-aware techniques is also expected to become more prominent. We should not simply partition data and employ task stealing across the sockets for all workloads. This thesis suggests that a better approach on multi-socket servers is to adapt to the workload. In typical scenarios with multiple tables, partitioning should be judiciously used up to the point of balancing utilization across the sockets of the server. Also, task stealing should be used to balance load across the sockets, but should be avoided for memory-intensive tasks that can cause a bottleneck in the interconnects and the remote memory controllers.

Bibliography

[1] An introduction to the Intel QuickPath Interconnect, January 2009. http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html. 2.4.1, 2.4.1, 2.4.1

[2] Intel Xeon Processor E7 Family Uncore Performance Monitoring, April 2011. http://www.intel.com/content/www/us/en/processors/xeon/xeon-e7-family-uncore-performance-programming-guide.html. 2.4.1

[3] Intel articles - Granularity and Parallel Performance, February 2012. http://software.intel.com/en-us/articles/granularity-and-parallel-performance. 2.3, 4.1.4, 4.1.5

[4] Computing at the European Organization for Nuclear Research (CERN), August 2012. http://cds.cern.ch/record/1997391. 1

[5] The data deluge. *Nature Cell Biology*, 14(8):775–775, August 2012. ISSN 1465-7392. 1

[6] Intel Xeon Processor E7 v2 2800/4800/8800 Product Family - Datasheet - Volume Two, March 2014. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e7-v2-datasheet-vol-2.pdf. 2.4.1

[7] SAP BusinessObjects BI 4 - Sizing Guide, February 2014. http://scn.sap.com/docs/DOC-33126. 1

[8] SAP HANA Platform SPS 11 Administration Guide, December 2015. http://help.sap.de/hana_platform. 2.4.2, 4.3, 5.2, 5.3.1, 5.5, 6.3.2

[9] SAP Enhanced Mixed Load (BW-EML) benchmark, February 2015. http://global.sap.com/campaigns/benchmark/appbm_bweml.epx. 5.5.2

[10] Red Hat Enterprise Linux 7 - Product Documentation - Performance Tuning Guide - Chapter 3. CPU, March 2015. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Performance_Tuning_Guide/chap-Red_Hat_Enterprise_Linux-Performance_Tuning_Guide-CPU.html. 2.4.1

[11] The OpenMP API specification for parallel programming. October 2016. `http://www.openmp.org/`. 2.3, 4.1

[12] Intel Thread Building Blocks – Documentation – User Guide – The Task Scheduler – Task-based Programming, June 2016. `http://threadingbuildingblocks.org/documentation`. 1.3.2, 2.3, 4.1, 4.1.2, 4.1.3, 4.1.3, 4.1.4

[13] Intel Thread Building Blocks – Documentation – User Guide – The Task Scheduler – When Task-Based Programming Is Inappropriate, August 2016. `https://software.intel.com/en-us/node/506101`. 4.1

[14] Transaction Processing Performance Council (TPC). `http://www.tpc.org`, October 2016. 4.2, 4.2.3

[15] TPC-H Benchmark: Standard Specification, v. 2.17.1. October 2016. `http://www.tpc.org/tpch/`. 3.3, 3.4.1, 4.1, 4.1.5, 5.5.2, 6.5.6

[16] SAP HANA Live for SAP Business Suite, April 2016. `http://help.sap.com/hba`. 4.2

[17] Amazon EC2 dedicated hosts and instance types, August 2016. `https://aws.amazon.com/ec2/instance-types/` and `https://aws.amazon.com/ec2/dedicated-hosts/`. 7.3

[18] Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D. Order Number: 325462-059US., June 2016. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`. 6.2

[19] SAP HANA Data Distribution Optimizer Administration Guide, March 2016. `http://help.sap.com/hana_options_dwf`. 2.4.2

[20] SAP HANA Platform Core SPS 12 - sap hana sql and system views reference - transaction management statements, October 2016. `https://help.sap.com/saphelp_hanaplatform/helpdata/en/20/a3ae8a75191014b039b36c43e2029d/content.htm`. 2.5

[21] SGI UV server line data sheets, July 2016. URL `http://www.sgi.com/products/servers/uv/`. 2.4.1, 7.3

[22] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 671–682. ACM, 2006. 2.1

[23] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, page 967. ACM Press, 2008. 2.1

[24] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The Data Locality of Work Stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–12. ACM, 2000. 2.3, 4.1

[25] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB)*, pages 1110–1121, 2004. 2.4.2

[26] A. Ailamaki, T. Scheuer, I. Psaroudakis, and N. May. Parallel execution of parsed query based on a concurrency level corresponding to an average number of available worker threads, 2016. US Patent 9,329,899. 3, 4

[27] Anastasia Ailamaki, Erietta Liarou, Pinar Tözün, Danica Porobic, and Iraklis Psaroudakis. How to Stop Under-utilization and Love Multicores. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 189–192. ACM, 2014. 1.1, 2.4.1

[28] Anastasia Ailamaki, Erietta Liarou, Pinar Tözün, Danica Porobic, and Iraklis Psaroudakis. How to Stop Under-Utilization and Love Multicores. In *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE)*, 2015. 1.1

[29] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: A Hands-free Adaptive Store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1103–1114. ACM, 2014. 2.5

[30] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems. *Proc. VLDB Endow.*, 5 (10):1064–1075, June 2012. 2.4.1, 2.4.2

[31] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The SprayList: A Scalable Relaxed Priority Queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 11–20. ACM, 2015. 5.3

[32] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. Scale-up vs Scale-out for Hadoop: Time to Rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC)*, pages 20:1–20:13. ACM, 2013. 7.3

[33] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Perez. The datapath system: A data-centric analytic processing engine for large data warehouses. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 519–530. ACM, 2010. 1.3.1, 2.2, 2.2.3, 2.2.4, 3.1.1, 3.1.4, 3.2.1, 3.4.3

[34] Gupta Ashish and Inderpal Singh Mumick. *Materialized Views*. MIT Press, 2nd edition, 1999. ISBN 0471200247, 9780471200246. 2.2

[35] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.*, 1(2):97–137, June 1976. 2.1

[36] Siu-lun Au and Sivarama P. Dandamudi. The Impact of Program Structure on the Performance of Scheduling Policies in Multiprocessor Systems. *Int'l Journal of Computers and Their Applications*, 3(1):17–30, 1996. 4.1.4

[37] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Ernesto Su, Priya Unnikrishnan, and Guansong Zhang. *A Proposal for Task Parallelism in OpenMP*, pages 1–12. Springer Berlin Heidelberg, 2008. 2.3, 4.1, 4.1.3

[38] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, September 2013. 2.4.2

[39] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 29–44. ACM, 2009. 2.4.2

[40] Thomas Becker and Tobias Kutning. Searching for the Best System Configuration to Fit Your BW Needs?, November 2012. `http://scn.sap.com/docs/DOC-33705`. 5.5.2

[41] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A Case for NUMA-aware Contention Management on Multicore Systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC)*. USENIX Association, 2011. 1.1, 1.2, 2.3, 2.4.1, 2.4.2

[42] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 37–48. ACM, 2011. 2.3, 4.1, 4.1.4

[43] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216. ACM, 1995. 1.3.2, 2.3, 4.1

[44] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but Effective Techniques for NUMA Memory Management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP)*, pages 19–31, 1989. 2.4.2

[45] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 212–221, 1991. 2.4.2

[46] George Edward Pelham Box, Gwilym Jenkins, and Gregory Reinsel. *Time Series Analysis, Forecasting and Control*. John Wiley & Sons, Inc., 4th edition, 2008. ISBN 0471200247, 9780471200246. 6.3.2

[47] George Candea, Neoklis Polyzotis, and Radek Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *Proc. VLDB Endow.*, 2(1):277–288, August 2009. 1.2, 1.3.1, 2.2, 2.2.3, 2.2.4, 3.1, 3.1.1, 3.1.4, 3.2.1, 3.2.2, 3.2.2, 3.3, 7.2

[48] George Candea, Neoklis Polyzotis, and Radek Vingralek. Predictable performance and high query concurrency for data analytics. *The VLDB Journal*, 20(2):227–248, April 2011. 2.2, 2.2.3, 2.2.4, 2.2.4, 3.1.1, 3.1.4, 3.2.1, 3.2.2, 3.3

[49] Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, and Walker White. Fast Checkpoint Recovery Algorithms for Frequently Consistent Applications. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 265–276. ACM, 2011. 4.2

[50] Xiang Cao, Kewal Keshaorao Panchputre, and David Hung-Chang Du. Accelerating data shuffling in mapreduce framework with a scale-up numa computing architecture. In *Proceedings of the 24th High Performance Computing Symposium (HPC)*, pages 17:1–17:8, 2016. 7.3

[51] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *ACM Sigmod record*, 26(1):65–74, 1997. 1

[52] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The impact of synchronization and granularity on parallel systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*, pages 239–248. ACM, 1990. 2.3, 4.1.4, 4.1.5

[53] W.J. Chen, B. Blaser, M. Bonezzi, P. Lau, J.C. Pacanaro, M. Schlegel, A. Zaka, A. Zietlow, and IBM Redbooks. *Architecting and Deploying DB2 with BLU Acceleration*. IBM Redbooks, 2014. ISBN 9780738440125. 1

[54] Rada Chirkova and Jun Yang. *Materialized Views (Foundations and Trends in Databases)*. Now Publishers Inc, 2nd edition, 2012. ISBN 0471200247, 9780471200246. 2.2

[55] Hong-Tai Chou and David J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the 11th International Conference on Very Large Data Bases (VLDB) - Volume 11*, pages 127–141. VLDB Endowment, 1985. 2.2

[56] John Cieslewicz and Kenneth A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 339–350. VLDB Endowment, 2007. 2.2.3

[57] John Cieslewicz and Kenneth A. Ross. Data Partitioning on Chip Multiprocessors. In *Proceedings of the 4th International Workshop on Data Management on New Hardware (DaMoN)*, pages 25–34. ACM, 2008. 2.3

[58] Latha S. Colby, Richard L. Cole, Edward Haslam, Nasi Jazayeri, Galt Johnson, William J. McKenna, Lee Schumacher, and David Wilhite. Redbrick Vista: Aggregate Computation and Management. In *Proceedings of the Fourteenth International Conference on Data Engineering (ICDE)*, pages 174–177. IEEE Computer Society, 1998. 2.2, 3.1.1

[59] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. The Mixed Workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems (DBTest)*, pages 8:1–8:6. ACM, 2011. 4.2, 4.2.3

[60] Cathan Cook. Database Architecture: The Storage Engine, 2001. `http://msdn.microsoft.com/library/aa902689(v=sql.80).aspx`. 2.2

[61] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd, and Toshiba Corporation. Advanced Configuration and Power Interface Specification (ACPI) Revision 5.0 Errata A, 2013. `http://www.acpi.info`. 2.4.1

[62] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in Multi-query Optimization. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 59–70. ACM, 2001. 2.2

[63] Dinesh Das, Jiaqi Yan, Mohamed Zait, Satyanarayana R. Valluri, Nirav Vyas, Ramarajan Krishnamachari, Prashant Gaharwar, Jesse Kamp, and Niloy Mukherjee. Query optimization in Oracle 12c database in-memory. *Proceedings of the VLDB Endowment*, 8: 1770–1781, 2015. 2.1

[64] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 381–394. ACM, 2013. 1.2, 2.4.2

[65] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–48. ACM, 2013. 2.4.1

[66] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. Anti-caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.*, 6(14):1942–1953, September 2013. 2.1

[67] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.*, 7(4):277–288, December 2013. 4.2

[68] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of OpenMP Task Scheduling Strategies. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism (IWOMP)*, pages 100–110. Springer-Verlag, 2008. 1.3.2, 2.3

[69] Kapali P. Eswaran. Placement of Records in a File and File Allocation in a Computer. In *Proceedings of the IFIP Congress on Information Processing*, pages 304–307, 1974. 2.4.2

[70] Babak Falsafi and David A. Wood. Reactive NUMA: a design for unifying s-COMA and CC-NUMA. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 229–240, 1997. 2.4.2

[71] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.*, 40(4):45–51, January 2012. 2.5

[72] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012. 1.3.2, 2.1, 2.5, 4, 4.1, 4.1.2, 4.2, 4.2.1

[73] Daniela Florescu and Donald Kossmann. Rethinking Cost and Performance of Database Systems. *SIGMOD Rec.*, 38(1):43–48, June 2009. 4.2

[74] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, December 1992. 2.1

[75] Minos N. Garofalakis and Yannis E. Ioannidis. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, pages 296–305. Morgan Kaufmann Publishers Inc., 1997. 2.4.2

[76] Balazs Gerofi, Masamichi Takagi, Yutaka Ishikawa, Rolf Riesen, Evan Powers, and Robert W. Wisniewski. Exploring the Design Space of Combining Linux with Lightweight Kernels for Extreme Scale Computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, pages 5:1–5:8. ACM, 2015. 2.4.2

[77] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. SharedDB: Killing One Thousand Queries with One Stone. *Proc. VLDB Endow.*, 5(6):526–537, February 2012. 1.3.1, 2.2, 2.2.3, 2.2.4, 3.1.1, 3.1.4, 3.2.1, 7.2

[78] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. Shared Workload Optimization. *Proc. VLDB Endow.*, 7(6):429–440, February 2014. 2.2.3

[79] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. Deployment of Query Plans on Multicores. *Proc. VLDB Endow.*, 8(3):233–244, November 2014. 2.4.2

[80] Jana Giceva, Gerd Zellweger, Gustavo Alonso, and Timothy Rosco. Customized OS Support for Data-processing. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN)*, pages 2:1–2:6. ACM, 2016. 2.4.2

[81] Lukasz Golab, Marios Hadjieleftheriou, Howard Karloff, and Barna Saha. Distributed Data Placement to Minimize Communication Costs via Graph Partitioning. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 20:1–20:12. ACM, 2014. 2.4.2

[82] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering (ICDE)*, pages 209–218. IEEE Computer Society, 1993. 2.1, 7.2

[83] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 4:287–326, 1979. 2.3

[84] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1st edition, 1992. ISBN 1558601902. 1

[85] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE: A Main Memory Hybrid Storage Engine. *Proc. VLDB Endow.*, 4(2):105–116, November 2010. 2.5

[86] Tim Gubner. Achieving many-core scalability in Vectorwise. 2014. Master thesis at the Technische Universität Ilmenau. `http://homepages.cwi.nl/~boncz/msc/2014-Gubner.pdf`. 2.4.2

[87] PK Gupta (Intel). Xeon+FPGA Platform for the Data Center. The Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2015). URL `http://www.ece.cmu.edu/calcm/carl/`. 7.2

[88] B. Hamidzadeh and D. J. Lilja. Dynamic scheduling strategies for shared-memory multiprocessors. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS)*, pages 208–215, May 1996. 2.3, 4.1.2

[89] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July 2011. 1.1

[90] Stavros Harizopoulos and Anastassia Ailamaki. A case for staged database systems. In *In Proceedings of 1st Conference on Innovative Data Systems Research (CIDR)*, 2003. 2.2.2

[91] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 383–394. ACM, 2005. (document), 1.2, 1.3.1, 2.1, 2.2, 2.2, 2.2.1, 2.2, 2.2.2, 3.1.1, 3.1.3, 3.3, 3.3, 3.4.1

[92] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. Architecture of a database system. *Found. Trends databases*, 1(2):141–259, February 2007. 4.1

[93] Gerhard Hill and Andrew Ross. Reducing Outer Joins. *The VLDB Journal*, 18(3):599–610, June 2009. 5.5

[94] Ralf Hoffmann, Matthias Korch, and Thomas Rauber. Performance evaluation of task pools based on hardware synchronization. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC)*, pages 44–. IEEE Computer Society, 2004. 2.3, 4.1, 4.1.2

[95] Judith Hurwitz, Alan Nugent, Fern Halper, and Marcia Kaufman. *Big Data For Dummies*. For Dummies, 1st edition, 2013. ISBN 1118504224, 9781118504222. 1

[96] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012. 2.5

[97] F. Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. Decoupling Contention Management from Scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 117–128. ACM, 2010. 2.3

[98] Ryan Johnson, Stavros Harizopoulos, Nikos Hardavellas, Kivanc Sabirli, Ippokratis Pandis, Anastasia Ailamaki, Naju G. Mancheril, and Babak Falsafi. To Share or Not to Share? In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 351–362. VLDB Endowment, 2007. 1.2, 3.1.3, 3.1.4, 3.3, 3.3, 3.3, 7.2

[99] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT)*, pages 24–35. ACM, 2009. 2.1, 3.1.2, 3.2.2, 3.4.1

[100] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 439–450. Morgan Kaufmann Publishers Inc., 1994. 2.2

[101] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE)*, pages 195–206. IEEE Computer Society, 2011. (document), 2.1, 4, 4.2, 4.2.2, 4.7, 4.2.2

[102] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., 2nd edition, 2002. 1, 2.2.3, 2.2.4

[103] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workload. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 74–85, 2014. 1.3.2, 2.4.1, 2.4.2, 5.1, 6.1

[104] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, December 1999. 2.3

[105] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T. H. Lee, J. Loaiza, N. Macnaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zait. Oracle Database In-Memory: A dual format in-memory database. In *IEEE 31st International Conference on Data Engineering (ICDE)*, pages 1253–1258, April 2015. 2.5, 2.5, 5.2

[106] Christoph Lameter, Bill Hsu, Marc Sosnick-Pérez, David Bacon, Rodric Rabbah, Sunil Shukla, Andrew Danowitz, Kyle Kelley, James Mao, and John P. Stevenson. NUMA (Non-Uniform Memory Access): An Overview. *ACM Queue*, 11(7):40, 2013. 1.1, 1.2, 2.4.1, 2.4.1

[107] C. A. Lang, B. Bhattacharjee, T. Malkemus, S. Padmanabhan, and K. Wong. Increasing Buffer-Locality for Multiple Relational Table Scans through Grouping and Throttling. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 1136–1145, 2007. 2.2

[108] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. Massively Parallel NUMA-aware Hash Joins. In *International Workshop on In-Memory Data Management and Analytics (IMDM)*, 2013. 2.4.2

[109] Richard P. LaRowe, Jr., Mark A. Holliday, and Carla Schlatter Ellis. An analysis of dynamic page placement on a numa multiprocessor. In *Proceedings of the 1992 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 23–34, 1992. 2.4.2

[110] Richard P. LaRowe Jr and Carla Schlatter Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. In *ACM Transactions on Computer Systems (TOCS)*, volume 9, pages 319–363, 1991. 2.4.2

[111] Per-Ake Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. Enhancements to SQL Server Column Stores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1159–1168. ACM, 2013. 2.1, 2.4.2, 2.5, 2.5

[112] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, pages 241–251. ACM, 1997. 2.4.1, 2.4.1

[113] J. Lee, Y. S. Kwon, F. Färber, M. Muehle, C. Lee, C. Bensberg, J. Y. Lee, A. H. Lee, and W. Lehner. SAP HANA distributed in-memory database system: Transaction, session, and metadata management. In *IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1165–1173, April 2013. 2.5, 2.5, 4.2.1

[114] Rubao Lee, Xiaoning Ding, Feng Chen, Qingda Lu, and Xiaodong Zhang. MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases. *Proc. VLDB Endow.*, 2(1):373–384, August 2009. 2.4.2

[115] Thomas Legler, Wolfgang Lehner, and Andrew Ross. Data Mining with the SAP NetWeaver BI Accelerator. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 1059–1068. VLDB Endowment, 2006. 5.5.2

[116] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *2014 IEEE 30th International Conference on Data Engineering (ICDE)*, pages 580–591, March 2014. 4.2.2

[117] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 743–754. ACM, 2014. (document), 1.3.2, 2.4.1, 2.4.2, 2.5, 4.7, 4.2.2, 5.1, 6.1, 7.2

[118] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. Speeding Up Queries in Column Stores: A Case for Compression. In *Proceedings of the 12th International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, pages 117–129. Springer-Verlag, 2010. 2.1, 2.5, 5.3.1, 5.3.1

[119] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the Cost of Context Switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS)*. ACM, 2007. 4.1

[120] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. NUMA-aware algorithms: the case of data shuffling. In *6th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013. 2.4.2

[121] Ling Liu and M. Tamer Zsu. *Encyclopedia of Database Systems*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 0387355448, 9780387355443. 4.1

[122] Hans-Wolfgang Loidl and Kevin Hammond. On the granularity of divide-and-conquer parallelism. In *Proceedings of the 1995 International Conference on Functional Programming (FP)*, pages 135–144. British Computer Society, 1995. 2.3, 4.1.4, 4.1.5

[123] Tom Lovett and Russell Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*, pages 308–317, 1996. 2.4.1

[124] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. pages 1–16. ACM Press, 2016. ISBN 978-1-4503-4240-7. 4.1

[125] Hongjun Lu and Kian-Lee Tan. Dynamic and Load-balanced Task-Oriented Datbase Query Processing in Parallel Systems. In *Proceedings of the 3rd International Conference on Extending Database Technology: Advances in Database Technology (EDBT)*, pages 357–372. Springer-Verlag, 1992. 1.3.2

[126] Roger MacNicol and Blaine French. Sybase IQ Multiplex - Designed for Analytics. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB) - Volume 30*, pages 1227–1230. VLDB Endowment, 2004. 4.2

[127] Zoltan Majo and Thomas R. Gross. Matching Memory Access Patterns and Data Placement for NUMA Systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO)*, pages 230–241, 2012. 2.4.2

[128] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. MQJoin: Efficient Shared Execution of Main-memory Joins. *Proc. VLDB Endow.*, 9 (6):480–491, January 2016. 2.2.3

[129] Jaydeep Marathe and Frank Mueller. Hardware Profile-guided Automatic Page Placement for ccNUMA Systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 90–99, 2006. 2.4.2

[130] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. Feedback-directed Page Placement for ccNUMA via Hardware-generated Memory Traces. *J. Parallel Distrib. Comput.*, 70 (12):1204–1219, December 2010. 2.4.2

[131] Sebastian Mattheis, Tobias Schuele, Andreas Raabe, Thomas Henties, and Urs Gleim. Work Stealing Strategies for Parallel Stream Processing in Soft Real-time Systems. In *Proceedings of the 25th International Conference on Architecture of Computing Systems (ARCS)*, pages 172–183. Springer-Verlag, 2012. 2.3

[132] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies (FAST)*, pages 115–130. USENIX Association, 2003. 2.2

[133] S. Mittal, J. S. Vetter, and D. Li. A Survey Of Architectural Approaches for Managing Embedded DRAM and Non-Volatile On-Chip Caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1524–1537, June 2015. 7.2

[134] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP)*, pages 185–197. ACM, 1990. 2.3

[135] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *44th International Conference on Parallel Processing (ICPP)*, pages 739–748, 2015. 2.4.1

[136] G.E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998. 1.1

[137] Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant Scheduling. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 422–431. Society for Industrial and Applied Mathematics, 1993. 2.3

[138] Niloy Mukherjee, Shasank Chavan, Maria Colgan, Dinesh Das, Mike Gleeson, Sanket Hase, Allison Holloway, Hui Jin, Jesse Kamp, Kartik Kulkarni, Tirthankar Lahiri, Juan Loaiza, Neil Macnaughton, Vineet Marwah, Atrayee Mullick, Andy Witkowski, Jiaqi Yan, and Mohamed Zait. Distributed Architecture of Oracle Database In-memory. *Proc. VLDB Endow.*, 8(12):1630–1641, August 2015. 2.4.2

[139] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-Efficient Aggregation: Hashing Is Sorting. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1123–1136. ACM, 2015. 5.5, 5.5, 5.5.1

[140] Girija J. Narlikar. Scheduling Threads for Low Space Requirement and Good Locality. In *Proc. of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 83–95, 1999. 2.3

[141] Pat O Neil, Betty O Neil, and Xuedong Chen. The Star Schema Benchmark (SSB), Revision 3. 2009. 3.4.1

[142] Daniel Neill and Adam Wierman. On the Benefits of Work Stealing in Shared-Memory Multiprocessors, 2011. Project Report at Carnegie Mellon University. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.126.3498&rep=rep1&type=pdf`. 2.3, 4.1

[143] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011. 4.2, 4.2.2, 7.2

[144] Tho Manh Nguyen, Josef Schiefer, and A. Min Tjoa. Sense & Response Service Architecture (SARESA): An Approach Towards a Real-time Business Intelligence Solution and Its Use for a Fraud Detection Application. In *Proceedings of the 8th ACM International Workshop on Data Warehousing and OLAP (DOLAP)*, pages 77–86. ACM, 2005. 4.2

[145] Thu D. Nguyen, Raj Vaswani, and John Zahorjan. *Using runtime measured workload characteristics in parallel processor scheduling*, pages 155–174. Springer Berlin Heidelberg, 1996. 2.3

[146] Dimitrios S. Nikolopoulos, Constantine D. Polychronopoulos, Theodore S. Papatheodorou, Jesús Labarta, and Eduard Ayguadé. Scheduler-Activated Dynamic Page Migration for Multiprogrammed DSM Multiprocessors. *J. Parallel Distrib. Comput.*, 62 (6):1069–1103, June 2002. 2.4.2

[147] Lisa Noordergraaf and Ruud van der Pas. Performance Experiences on Sun's Wildfire Prototype. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC)*, 1999. 2.4.2

[148] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out numa. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–18, 2014. 7.3

[149] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *Int. J. High Perform. Comput. Appl.*, 26(2):110–124, May 2012. 2.3, 2.4.2

[150] Carl Olofson and Henry Morris. Blending transactions and analytics in a single in-memory platform: Key to the real-time enterprise. Technical report, International Data Corporation (IDC), February 2013. IDC 239327. 4.2

[151] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 297–306. ACM, 1993. 2.2

[152] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition.* Springer, 2011. ISBN 978-1-4419-8833-1. 2.4.2

[153] M. A. Palis, Jing-Chiou Liou, and D. S. L. Wei. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):46–55, Jan 1996. 2.3

[154] Hasso Plattner. A Common Database Approach for OLTP and OLAP Using an In-memory Column Database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 1–2. ACM, 2009. 4.2

[155] D. Porobic, E. Liarou, P. Tözün, and A. Ailamaki. ATraPos: Adaptive transaction processing on hardware Islands. In *IEEE 30th International Conference on Data Engineering (ICDE)*, pages 688–699, March 2014. 2.4.2

[156] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki. OLTP on Hardware Islands. *Proc. VLDB Endow.*, 5(11):1447–1458, July 2012. 1.1

[157] Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. Sharing Data and Work Across Concurrent Analytical Queries. *Proc. VLDB Endow.*, 6(9):637–648, July 2013. 1.2, 1.3.1, 1, 2, 3

[158] Iraklis Psaroudakis, Tobias Scheuer, Norman May, and Anastasia Ailamaki. Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads. In *Proceedings of the Fourth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2013. 1.2, 1.3.2, 3, 4

[159] Iraklis Psaroudakis, Manos Athanassoulis, Matthaios Olma, and Anastasia Ailamaki. Reactive and proactive sharing across concurrent analytical queries. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 889–892. ACM, 2014. 1.3.1, 2, 3

[160] Iraklis Psaroudakis, Thomas Kissinger, Danica Porobic, Thomas Ilsche, Erietta Liarou, Pinar Tözün, Anastasia Ailamaki, and Wolfgang Lehner. Dynamic Fine-grained Scheduling for Energy-efficient Main-memory Queries. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN)*, pages 1:1–1:7. ACM, 2014. 7.2

[161] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Scaling Up Concurrent Main-memory Column-store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *Proc. VLDB Endow.*, 8(12):1442–1453, August 2015. 1.3.2, 4, 5

[162] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling. In Raghunath Nambiar and Meikel Poess, editors, *6th TPC Technology Conference on Performance Characterization and Benchmarking (TPCTC) 2014. Revised Selected Papers*, pages 97–112. Springer International Publishing, 2015. 1.3.2, 3, 4

[163] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *Proc. VLDB Endow.*, 10(2), 2016. 1.3.2, 5, 5, 6

[164] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. Main-memory Scan Sharing for Multi-core CPUs. *Proc. VLDB Endow.*, 1(1):610–621, August 2008. 2.2, 3.1.3, 3.1.4, 3.3

[165] Ravi Rajwar, Martin Dixon, and Ronak Singhal. Specialized Evolution of the General Purpose CPU. In *Seventh Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015. 7.2

[166] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 3rd edition, 2002. ISBN 0072465638. 1.2, 4.1

[167] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proc. VLDB Endow.*, 6(11):1080–1091, August 2013. 2.1, 2.4.2, 4.2

[168] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating Physical Database Design in a Parallel Database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 558–569. ACM, 2002. 2.4.2

[169] Nicholas Roussopoulos. View Indexing in Relational Databases. *ACM Trans. Database Syst.*, 7(2):258–290, June 1982. 1.3.1, 2.2

[170] Nick Roussopoulos, Chungmin M. Chen, Stephen Kelley, Alex Delis, and Yannis Papakonstantinou. The ADMS Project: Views "R" Us. *IEEE Data Eng. Bull.*, 18(2), 1995. 2.2

[171] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 249–260. ACM, 2000. 2.2

[172] Philip Russom. High-Performance Data Warehousing. *The Data Warehousing Institute (TDWI)*, October 2012. TDWI Best Practices Report. `http://tdwi.org/research/2012/10/tdwi-best-practices-report-high-performance-data-warehousing.aspx`. 1

[173] Giovanni Maria Sacco and Mario Schkolnick. Buffer management in relational database systems. *ACM Trans. Database Syst.*, 11(4):473–498, December 1986. 1.3.1

[174] Jochen Seidel. Job-Scheduling in Main-Memory Based Parallel Database Systems. 2010. Diploma thesis at the Karlsruhe Institute of Technology (KIT). 1.1

[175] Timos K. Sellis. Multiple-query Optimization. *ACM Trans. Database Syst.*, 13(1):23–52, March 1988. 2.2, 7.2

[176] Junho Shim, Peter Scheuermann, and Radek Vingralek. Dynamic Caching of Query Results for Decision Support Systems. In *Proceedings of the 11th International Conference on Scientific and Statistical Database Management (SSDBM)*. IEEE Computer Society, 1999. 2.2, 2.2.1

[177] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 731–742. ACM, 2012. 2.5

[178] O. Steinau and J. Hartmann. Method for calculating distributed joins in main memory with minimal communicaton overhead, 2006. US Patent App. 11/018,697. 5.5

[179] Per Stenström, Truman Joe, and Anoop Gupta. Comparative Performance Evaluation of Cache-coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 80–91, 1992. 2.4.2

[180] Radu Stoica, Justin J. Levandoski, and Per-Ake Larson. Identifying Hot and Cold Data in Main-memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE)*, pages 26–37. IEEE Computer Society, 2013. 2.1

[181] Michael Stonebraker and Ugur Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 2–11. IEEE Computer Society, 2005. 1, 2.1, 4.2

[182] Michael Stonebraker and Ariel Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013. 4.2, 4.2

[183] Jie Tao, Martin Schulz, and Wolfgang Karl. SIMT/OMP: A Toolset to Study and Exploit Memory Locality of OpenMP Applications on NUMA Architectures. In *Proceedings of the 5th International Conference on OpenMP Applications and Tools: Shared Memory Parallel Programming with OpenMP (WOMPAT)*, pages 41–52, 2005. 2.4.2

[184] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Using Hardware Counters to Automatically Improve Memory Performance. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC)*, 2004. 2.4.2

[185] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Hardware monitors for dynamic page migration. *J. Parallel Distrib. Comput.*, 68(9):1186–1200, September 2008. 2.4.2

[186] F. Transier, C. Mathis, N. Bohnsack, and K. Stammerjohann. Aggregation in parallel computation environments with shared memory, 2012. US Patent App. 12/978,194. 5.5, 5.5

[187] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 392–403, June 1995. 1.1

[188] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proc. VLDB Endow.*, 2(1):706–717, August 2009. 2.2, 3.1.1

[189] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 279–289, 1996. 2.4.2

[190] Vish Viswanathan. Intel Memory Latency Checker, November 2013. `https://software.intel.com/en-us/articles/intelr-memory-latency-checker`. 2.4.1

[191] Mehul Wagle, Daniel Booss, Ivan Schreter, and Daniel Egenolf. *NUMA-Aware Memory Management with In-Memory Databases*, pages 45–60. Springer International Publishing, 2016. 5.3.1, 6.3.2

[192] David Watts and Duncan Furniss. *IBM eX5 Portfolio Overview*. Sixth edition, April 2013. 2.4.1

[193] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., 2001. ISBN 1-55860-508-8. 2.5

[194] Thomas Willhalm. Independent Channel vs. Lockstep Mode – Drive your Memory Faster or Safer, July 2014. `https://software.intel.com/en-us/blogs/2014/07/11/independent-channel-vs-lockstep-mode-drive-you-memory-faster-or-safer`. 2.4.1

[195] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units. *Proc. VLDB Endow.*, 2(1):385–394, August 2009. 2.1, 2.5, 5.3.1, 5.4

[196] Thomas Willhalm, Roman Dementiev, and Patrick Fay. Intel Performance Counter Monitor - A better way to measure CPU utilization, August 2012. `http://software.intel.com/articles/intel-performance-counter-monitor`. 4.1.5, 5.4, 6.2, 6.5

[197] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. Vectorizing database column scans with complex predicates. In *4th International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 1–12, 2013. 2.1, 2.5, 5.3.1

[198] Jason Williamson. *Getting a Big Data Job For Dummies*. For Dummies, 1st edition, 2015. ISBN 9781118903407. 1

[199] Kenneth M. Wilson and Bob B. Aglietti. Dynamic Page Placement to Improve Locality in CC-NUMA Multiprocessors for TPC-C. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC)*, pages 33–33, 2001. 2.4.2

[200] Florian Wolf, Iraklis Psaroudakis, Norman May, Anastasia Ailamaki, and Kai-Uwe Sattler. Extending Database Task Schedulers for Multi-threaded Application Code. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 25:1–25:12. ACM, 2015. 5.3

[201] Johannes Wust, Martin Grund, Kai Hoewelmeyer, David Schwalb, and Hasso Plattner. *Concurrent Execution of Mixed Enterprise Workloads on In-Memory Databases*, pages 126–140. Springer International Publishing, 2014. 2.3

[202] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. Scalable Aggregation on Multicore Processors. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware (DaMoN)*, pages 1–9. ACM, 2011. 5.5, 5.5.1

[203] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. Locality-aware Partitioning in Parallel Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 17–30. ACM, 2015. 7.2

[204] Steffen Zeuch and Johann-Christoph Freytag. QTM: Modelling Query Execution with Tasks. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 34–45, 2014. 2.3

[205] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7): 1920–1948, July 2015. 2.1

[206] Zheng Zhang, Marcelo Cintra, and Josep Torrellas. Excel-NUMA: Toward Programmability, Simplicity, and High Performance. *IEEE Trans. Comput.*, 48(2):256–264, February 1999. 2.4.2

[207] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, December 2012. 2.3

[208] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer New York. ISBN 978-1-4419-8833-1 978-1-4419-8834-8. 2.4.2

[209] Marcin Zukowski and Peter A. Boncz. Vectorwise: Beyond Column Stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012. 2.4.2, 4.2

[210] Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sandor Heman. MonetDB/X100 – a DBMS in the CPU cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005. 2.1

[211] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 723–734. VLDB Endowment, 2007. 1.3.1, 2.2, 3.1.1

Curriculum Vitae: Iraklis Psaroudakis

Age:	Birth year: 1987 (Greece)	Address:	Please e-mail me to ask for address
Website:	http://www.kingherc.com/	E-mail:	kingherc@gmail.com

Objective: I am a technology enthusiast with a firm theoretical background and strong practical skills in software and web development. I completed my PhD in computer science at EPFL (Lausanne, Switzerland). After finishing my PhD thesis, I am interested in finding a challenging job in the technology industry.

Education

Sep. 2011 – Dec. 2016	**École Polytechnique Fédérale de Lausanne (EPFL)**, Lausanne, Switzerland. **PhD**. School of Computer and Communication Sciences, Data-Intensive Applications Laboratory (DIAS) lab. In cooperation with the SAP HANA database team (Walldorf, Germany) during Jan. 2013 – Sep. 2016. Thesis title: "Scaling Up Concurrent Analytical Workloads on Multi-Core Processor Servers". Core research areas: ▪ Sharing data and work across concurrent analytical queries. Showed how two state-of-the-art sharing techniques, sharing common sub-plans and sharing query operators, can be combined to further improve performance by up to 2x. ▪ Adaptive NUMA-aware data placement and task scheduling. Showed how to adapt data placement and task stealing to the workload to improve the performance of main-memory column-stores for analytical workloads by up to 4x.
Sep. 2005 – Jul. 2010	**National Technical University of Athens (NTUA)**, Athens, Greece. **5-Year Diploma** (MSc equivalent) in Electrical & Computer Engineering. ▪ Major: computer science (information systems). ▪ GPA: **8.70/10.00**. Ranking: inside **10%** of the class.
Jun. 2005	**Private Senior School Pagkritio Ekpaideutirio**, Heraklion, Greece. Degree **19.6/20.00**.

Work experience

Aug. 2012 – Dec. 2012	**Software developer** (Internship) – SAP S.E., Walldorf, Germany. C++ developer for the thread and task scheduling framework of the SAP HANA database.
Nov. 2010 – Aug. 2011	**Computer programmer and analyst** (Private) – Greek army, Greece. Completed my military obligations as a private in the body of information technology.
Jan. 2009 – Jun. 2010	**Software and web developer** (Under student fellowship) – Institute for the Management of Information Systems (IMIS), Athens, Greece. Did my diploma thesis "Application of Object Relational Mapping on the environment for the management of webpages of courses of NTUA", supervised by Prof. Timos Sellis.

Publications and conferences

VLDB 2017	**I. Psaroudakis**, T. Scheuer, N. May, A. Sellami, A. Ailamaki. "Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores".
VLDB 2015	**I. Psaroudakis**, T. Scheuer, N. May, A. Sellami, A. Ailamaki. "Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement".
SSDBM 2015	F. Wolf, **I. Psaroudakis**, N. May, A. Ailamaki, K.-U. Sattler. "Extending database task schedulers for multi-threaded application code".
ICDE 2015	A. Ailamaki, E. Liarou, P. Tözün, D. Porobic, **I. Psaroudakis**. "How to Stop Under-Utilization and Love Multicores". (tutorial)
TPCTC 2014	**I. Psaroudakis**, F. Wolf, N. May, T. Neumann, A. Böhm, A. Ailamaki, K.-U. Sattler. "Scaling up Mixed Workloads: a Battle of Data Freshness, Flexibility, and Scheduling".
SIGMOD 2014	A. Ailamaki, E. Liarou, P. Tözün, D. Porobic, **I. Psaroudakis**. "How to Stop Under-Utilization and Love Multicores". (tutorial)
SIGMOD 2014	**I. Psaroudakis**, M. Athanassoulis, M. Olma, A. Ailamaki. "Reactive and Proactive Sharing Across Concurrent Analytical Queries". (demo)
DaMoN 2014	**I. Psaroudakis**, T. Kissinger, D. Porobic, T. Ilsche, E. Liarou, P. Tözün, A. Ailamaki, W. Lehner. "Dynamic Fine-Grained Scheduling for Energy-Efficient Main-Memory Queries".
ADMS 2013	**I. Psaroudakis**, T. Scheuer, N. May and A. Ailamaki. "Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads".
VLDB 2013	**I. Psaroudakis**, M. Athanassoulis, and A. Ailamaki. "Sharing Data and Work Across Concurrent Analytical Queries".

Computer skills

Programming & IT skills	Excellent	▪ C++. Boost C++ libraries. Debugging with gdb. ▪ C# for Microsoft's .NET Framework using Visual Studio. ▪ Java for desktop and web development. ▪ Web development: HTML, CSS, JavaScript, AJAX, jQuery. ▪ Performance analysis using perf and Intel VTune Amplifier. ▪ SQL. DBMS: Microsoft SQL Server, MySQL, PostgreSQL, SAP HANA. ▪ Software IDEs: Eclipse, NetBeans, Visual Studio, Dreamweaver, vim.
	Good	▪ LAMP (Linux, Apache, MySQL, PHP) for web development. ▪ Windows Server 2008, its network infrastructure and tools. ▪ Learned during university: ML, Prolog, Matlab (for image & signal analysis), assembly (for microprocessors 8085/6, AVR and DOS), UML.
Certifications		▪ ETS Graduate Record Examinations (**GRE**) scores: V. 530, Q. 800, A. 4.5. (Oct. 2010) ▪ Microsoft Certified Technology Specialist (**MCTS**) certifications: .NET Framework 3.5 (ASP.NET, Windows Forms, ADO.NET, Windows Communication Foundation Applications), SQL Server 2008 (Database Development), Windows Server 2008 (Applications Infrastructure, Configuration). To validate, visit https://mcp.microsoft.com/Anonymous/Transcript/Validate and enter 857732, MCPpr0ven. ▪ Adobe Certified Associate for Rich Media Communication using Adobe **Flash**. ▪ European Computer Driving License (**ECDL**) Core Certificate.

Personal skills

Languages	▪ **English** – Fluent (Certificate of Proficiency in English, University of Cambridge). ETS Test of English as a Foreign Language (TOEFL) score: 112 (internet-based), 2010. ▪ **French** – Conversational (Diplôme d' Études en Langue Française 2nd Degré). ▪ **Greek** – Native speaker.
Competences	▪ **Social**: Frequent cooperation with teammates for the successful completion of projects. ▪ **Problem-Solving:** Worked on projects which required extensive analysis and research. ▪ **Ability to multitask:** Worked in more than one project at the same time with success.
Interests	Technology and video games. Trips, photographing, driving and tennis. Music and movies.

Seminars and other events

27-31 Jan. 2014	CUSO Winter School on Data-Centric Systems, Veysonnaz, Switzerland.
7-12 Apr. 2013	Summer School "Implementation Techniques for Data Management Software", Dagstuhl, Germany
28 Jun. – 2 Jul. 2010	Onassis Foundation Science Lecture Series 2010 in Computer Science, Heraklion, Greece Attended as a Scholarship Recipient Student
9-13 Nov. 2008	Microsoft TechEd Europe, Berlin, Germany. Participated as a Microsoft Student Partner.
Apr. 2003	European Parliament, Strasbourg, France Won a scholarship (via an essay competition) by Euroscola for a one-week educational visit

Awards and distinctions

2015	Ranked 2nd place in the ACM SIGMOD programming contest
2011	Fellowship student for the first year of my PhD at EPFL
2010	Onassis Foundation Scholarship Recipient to attend the Onassis Science Lecture Series 2010
2007	Imagine Cup (global student competition by Microsoft) – IT Challenge – Ranked 10th

Memberships

2015 – now	IEEE (Institute of Electrical and Electronics Engineers) member
2014 – now	ACM (Association for Computing Machinery) member
2010 – now	Member of the Technical Chamber of Greece (TEE)
2007 – 2011	Microsoft Student Partner in Greece Active participation in the StudentGuru.gr community. For my presentations, please visit my personal website. Awarded with MSDN Ultimate subscriptions.