Thèse n° 8975

Efficient Concurrent Analytical Query Processing using Data and Workload-conscious Sharing

Présentée le 12 mai 2023

Faculté informatique et communications
Laboratoire de systèmes et applications de traitement de données massives
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Panagiotis SIOULAS

Acceptée sur proposition du jury

École
polytechnique
fédérale
de Lausanne

2023

Words are never ‘only words’; they matter because
they define the contours of what we can do.
— Slavoj Žižek

To my family and friends, who made this possible.

Acknowledgements

Ph.D. is a personal experience, but it is by no means solitary. Thus, I would like to take the opportunity to thank my fellow travelers and companions in this part of my life.

First, I would like to thank my advisor, *Anastasia Ailamaki*. Anastasia's drive for perfection, attention to detail, and incisiveness has helped me grow and mature both as a researcher and a person. Also, I am grateful for the abundance of available resources that Anastasia has provided, whether that be hardware to run experiments or conferences on the other side of the world. More importantly, she has been an excellent team builder: by creating an ecosystem of wonderful, talented, young researchers in DIAS, she has given us a catalyst for brewing wild, forward-thinking, and innovative ideas.

Next, I thank the jury members who reviewed this thesis. Their feedback and constructive criticism have been extremely valuable in shaping this work. I thank the external reviewers, *Donald Kossmann* and *Volker Markl*, whose innovative work laid the groundwork for this thesis. Furthermore, I thank the internal reviewer, *Anne-Marie Kermarrec*, for helping me improve this work's quality and positioning. Finally, I thank *Carmela Troncoso* for serving as the jury's president and thus making it possible.

Aristotle remarked that "one swallow does not a summer make". I would humbly add that one researcher alone rarely innovates in a vacuum. During my five-and-a-half-year-long stay with DIAS and the internship before that, I had the honor of working alongside worthy and considerate colleagues. I thank *Danica*, *Mano*, *Raja*, *Angele*, *Mira*, *Matt*, *Eleni*, *Giorgo*, *Utku*, *Stella*, *Giagko*, *Perikli*, *Haoqiong*, *Kosta*, *Christina*, *Aunn*, *Viktor*, *Eleni*, *Anna*, *Hamish*, *Dimitra*, *Erika*, *Maggy* and *Fabienne*. As I worked with some of them more closely, some special thanks are in order: *Mano*, *Raja* and *Perikli* for mentoring me during my first year and teaching me how to conduct research. *Matt* and *Stella* for being great TA team leaders, excellent role models, and, more importantly, kind, supportive, cheerful, and overall wonderful people. *Viktor* for being motivational after every setback and rejection that came my way. *Giagko* and *Eleni* for sharing with me (pun intended) the excitement of working on the line of research presented in this thesis. Finally, I owe special thanks to the junior researchers I worked with: *Vladimir*, *Vaggeli*, *Giorgo*, and *Pedro*.

Another old proverb states, "all work and no play makes Jack a dull boy". Thankfully, I did not become dull; my friends in Lausanne, Athens, and beyond made my days enjoyable and memorable throughout these years. I thank my friends for the great moments we shared, for celebrating with me during my highs, and for their support and thoughtfulness during my lows. Thank you (or rather, I cannot thank you enough) *Kosta, Christo, Kosta, Panayioti, Stella, Sofia, Aggeliki, Angele, Matt, Stelio, Foivo, Giorgo, Alexandra, Dimitra, Ergys, Ivi, Angelina, Chrystalleni, Sylvia, Konstantine, Konstantine, Amalia, Stelio, Niko, Dimitra, Anna, Alexandre, Viktor, Viktor, Kristina, Tiziano,* and *Ceyhun.*

Lastly, I want to thank my family: my father *Thodori,* my mother *Kyparisso,* and my brother *Taso.* I thank them for their loving support since my first breath. I thank them for teaching me the value of hard and honest work and for shaping me to be the person I am. And I thank my parents for working hard to provide us with the opportunity to define our future.

Abstract

Analytical workloads are evolving as the number of users surges and applications that submit queries in batches become popular. However, traditional analytical databases that optimize-then-execute each query individually struggle to provide timely responses under high concurrency even when tuning databases for the target data and workload: response time is increased as a function of concurrency. Thus, high concurrency jeopardizes the interactivity, hence also the usability, of real-time applications.

Work-sharing databases reduce the total processing time across queries. However, existing planning, execution, and database-tuning strategies for work-sharing databases suffer from redundant processing in their chosen operator orders, data access methods, and techniques for handling recomputation. For real-time workloads that are ad-hoc, selective, or recurring, redundant processing results in performance bottlenecks that put timeliness at risk.

This thesis addresses the inefficiency in choosing operator orders, accessing data, and reusing materialized results in work-sharing databases. It introduces planning, execution, and database-tuning strategies that reduce redundant processing by holistically optimizing plans for the characteristics of the data, the query batches, and the workload patterns. Our goal is to enable timeliness for processing highly concurrent workloads using work sharing.

To choose efficient operator orders, we propose RouLette, a specialized engine that optimizes and processes Select-Project-Join queries using runtime adaptation. RouLette incrementally explores alternative plans with different sharing opportunities by combining reinforcement learning with feedback from monitoring the execution of explored plans. Also, RouLette introduces optimizations that reduce the overhead of adaptive execution. Thus, it both explores more candidate plans and evaluates them more accurately while maintaining low overhead.

To reduce processing time for data access and filtering when processing a batch of queries, we introduce SH2O. SH2O focuses on efficiency and scalability. First, it identifies multidimensional data regions where filtering decisions are invariant and uses spatial indices to perform shared access to the regions. Second, to mitigate overhead when the number of regions is large, SH2O employs two optimization strategies that: i) choose which filters to replace in a cost-based manner, ii) specialize for the different data properties and indices across partitions.

Finally, we propose ParCuR, a framework for effectively and efficiently reusing materialized results in work-sharing databases. ParCuR optimizes reuse across three axes: i) to address interference between work sharing and reuse, it introduces sharing-aware materialization and reuse policies, ii) to reduce the overhead for reuse, it builds access methods that eliminate frequent predicates, and iii) to maximize the usability of materialized results, it introduces a hybrid partitioning-materialization scheme that enables partial reuse while making efficient use of the available storage budget.

All in all, this thesis redesigns work-sharing databases by specializing shared execution for the target data, query batches, and workload patterns. It significantly expands the applicability of work-sharing databases, enabling them to provide timely responses to real-time, highly concurrent applications that produce unpredictable, selective, and recurring workloads.

Keywords: database management systems, analytical query processing, access method, query optimization, materialization, reuse, work sharing, concurrent query processing

Résumé

Les charges de travail analytique évoluent, à mesure que le nombre d'utilisateurs augmente et que les applications qui soumettent des requêtes par lots, deviennent de plus en plus populaires. Cependant, les bases de données analytiques traditionnelles, qui optimisent-puis-exécutent chaque requête individuellement, ont du mal à fournir des réponses rapides dans des conditions de simultanéité élevée, même lors du réglage des bases de données pour les données et la charge de travail cibles : le temps de réponse augmente comme une fonction de la simultanéité. Ainsi, une concurrence élevée compromet l'interactivité, donc aussi la facilité d'utilisation, des applications en temps réel.

Les bases de données de travail partagé réduisent le temps de traitement total à travers des requêtes. Cependant, les stratégies de planification, d'exécution et de réglage de la base de données existantes pour les bases de données de partage de travail souffrent d'un traitement redondant dans leurs ordres d'opérateur choisis, les méthodes d'accès aux données et les techniques de traitement du recalcul.

Cette thèse aborde l'inefficacité dans le choix des ordres des opérateurs, l'accès aux données et la réutilisation des résultats matérialisés dans les bases de données de travail partagé. Il introduit des stratégies de planification et d'exécution qui réduisent le traitement redondant en optimisant de manière globale en tenant en compte les caractéristiques des données, les lots de requêtes et les modèles de charge de travail. Notre objectif est de respecter les délais tout en gérant plus de requêtes en utilisant le travail partagé.

Pour choisir des ordres d'opérateurs plus efficaces, nous proposons RouLette, un moteur spécialisé qui optimise et traite les sous-requêtes Select-Project-Join en utilisant l'adaptation à l'exécution. RouLette explore progressivement des plans alternatifs avec différentes opportunités de partage en combinant l'apprentissage par renforcement avec les commentaires de la surveillance de l'exécution des plans explorés. Ainsi, il oriente l'exploration vers des plans efficaces pour le lot de requêtes et les données cibles. De plus, RouLette introduit des optimisations qui réduisent la surcharge de l'exécution adaptative. Ainsi, il explore à la fois plus de plans candidats et les évalue plus précisément, tout en maintenant une faible surcharge.

Pour réduire le temps de traitement pour l'accès aux données et le filtrage lors du traitement d'un lot de requêtes, nous introduisons SH2O. SH2O se concentre sur l'efficacité et la scalabilité. Tout d'abord, il identifie les régions de données multidimensionnelles où les décisions de filtrage sont invariantes et utilise des indices spatiaux pour effectuer un accès partagé aux régions. Deuxièment, pour atténuer la surcharge lorsque le nombre de régions est important, SH2O utilise deux stratégies d'optimisation qui : i) choisissent les filtres à remplacer en fonction des coûts, ii) se spécialisent pour les différentes propriétés de données et les indices à travers les partitions.

Enfin, nous proposons ParCuR, un cadre permettant de réutiliser efficacement et effectivement les résultats matérialisés dans les bases de données de travail partagé. ParCuR optimise la réutilisation sur trois axes : i) pour résoudre les interférences entre le travail partagé et la réutilisation, il introduit des politiques de matérialisation et de réutilisation sensibles au partage, ii) pour réduire la surcharge d'accès et de filtrage des données, il construit et utilise des méthodes d'accès qui éliminent les prédicats fréquents, et iii) pour maximiser la facilité d'utilisation des résultats matérialisés, il introduit un schéma hybride de partitionnement-matérialisation qui permet une réutilisation partielle tout en utilisant efficacement le budget de stockage disponible.

Dans l'ensemble, cette thèse reconçoit les bases de données de travail partagé en spécialisant l'exécution partagée pour les données cibles, les lots de requêtes et les modèles de charge de travail. Elle élargit considérablement l'applicabilité des bases de données de travail partagé, leur permettant de fournir des réponses opportunes aux applications hautement simultanées en temps réel qui produisent une charge de travail imprévisible, sélective et récurrente.

Contents

Acknowledgements **i**

Abstract (English/Français) **iii**

List of Figures **xiii**

1 Introduction **1**

1.1 Scaling Concurrent Processing 2

1.2 Limitations to Efficiency in Work Sharing 3

1.2.1 Global Plan Selection 4

1.2.2 Data Access 5

1.2.3 Reuse 6

1.3 Data and Workload-conscious Work Sharing 7

1.3.1 Goal: Efficiency under High Concurrency 7

1.3.2 Contributions: Adapting Work Sharing to the Data and Workload 8

1.3.3 Thesis Outline 10

2 Background **11**

2.1 Work sharing 11

2.1.1 Applications 11

2.1.2 Architecture of Work-sharing Databases 13

2.1.3 Admission Policy 13

2.1.4 Data Access 14

2.1.5 Shared Execution 15

2.1.6 Sharing-aware Optimization 20

2.2 Data and Workload-conscious Optimizations 22

2.2.1 Query Optimization 22

2.2.2 Access Methods 26

2.2.3 Materialization and Reuse 28

2.3 Reinforcement Learning 30

3 Scalable and Efficient Sharing-aware Optimization 31

3.1 Limitations in Sharing-aware Optimization 33

3.1.1 Coverage Limitation 33

3.1.2 Accuracy Limitation 34

3.2 RouLette Architecture 34

3.3 Learned Adaptation Policy 38

3.3.1 Policy-based Planning 39

3.3.2 Learning Policy Decisions 42

3.3.3 Specialized Q-learning Implementation 44

3.4 Adaptive Multi-query Executor 46

3.4.1 Efficient Shared Operators 46

3.4.2 Optimizations for Adaptive Processing 49

3.5 Experimental Evaluation 50

3.5.1 Response Time Evaluation 51

3.5.2 Quality of Learned Planning 58

3.5.3 Effect of Optimizations 62

3.5.4 Multi-core Execution 62

3.6 Summary 63

4 Efficient Shared Data-Access **65**

4.1 The Data-access Bottleneck 68

4.2 Multidimensional Shared Access 69

4.2.1 Computing Single Query-set Regions 70

4.2.2 Index-based Access to Regions 71

4.2.3 Hybrid Execution 73

4.2.4 Consolidation 74

4.2.5 Supported Predicates and Extensions 74

4.3 Selecting Probing Attributes 74

4.3.1 Cost Model 75

4.3.2 Enumerating Candidate Attribute Sets 78

4.4 Partition-oriented Access 78

4.5 SH2O-aware Data Organization 80

4.5.1 Iterative Partitioning 81

4.5.2 Index Selection 83

4.6 Implementation 83

4.7 Experimental Evaluation 84

4.7.1 End-to-end Performance for Analytics 85

4.7.2 Efficient Multidimensional Data Access 86

4.7.3 Scaling using Attribute Selection 90

4.7.4 Decoupling Dimensions using Partitioning 92

4.8 Summary 94

5 Effective and Efficient Reuse in Work-Sharing Environments **95**

5.1 Reuse in Shared Execution 98

5.1.1 Recomputation Bottleneck . . . 98
5.1.2 Pitfalls of Combining Reuse and Work Sharing . . . 98
5.1.3 Harmonizing Reuse and Work Sharing . . . 100
5.1.4 Putting It All Together . . . 102
5.1.5 ParCuR's Scope . . . 103
5.2 Tuning ParCuR's state . . . 104
5.2.1 Workload-driven Partitioning . . . 104
5.2.2 Materialization Policy . . . 108
5.2.3 Building Access Methods . . . 114
5.3 Reuse-aware Shared Execution . . . 115
5.3.1 Data and Filter Skipping . . . 115
5.3.2 Partitioned Execution . . . 116
5.3.3 Injecting Materializations in Global Plans . . . 116
5.4 Implementation . . . 120
5.4.1 Tuning the Cost Model . . . 120
5.4.2 Enforcing Reuse Decisions to Policy . . . 120
5.4.3 Ingestion from Materializations . . . 120
5.4.4 Materializing Results . . . 121
5.4.5 Tuner Utilities . . . 121
5.4.6 Tuning Partitioning . . . 121
5.4.7 Limitations . . . 121
5.5 Experimental Evaluation . . . 122
5.5.1 Impact of Reuse in Global Plans . . . 123
5.5.2 Sharing-aware Materialization Policy . . . 125
5.5.3 Effect of workload shift in reuse . . . 128
5.5.4 Macro-benchmarks . . . 131

5.6 Summary 131

6 Conclusion and Future Outlook **133**

6.1 Data and Workload-conscious Work Sharing: What we did 133

6.2 Efficient Work-Sharing Databases: Next steps 134

Bibliography **150**

List of Figures

1.1 Example use case: supporting batch processing for a dashboard 3

1.2 Challenges for efficient work sharing and the contributions of this thesis that address the challenges. Each contribution is denoted as *(X)* such that we refer to it in the text 9

2.1 Flow of queries through a work-sharing database 13

2.2 Global plan for Q1 and Q2 (example in text) 16

2.3 Example of shared selection 18

2.4 Example of probing a predicate index 19

2.5 Example of shared join 20

2.6 Tradeoff between efficiency and the scalability of sharing-aware optimization mechanisms 21

2.7 Example for Symmetric Hash-join 24

2.8 Example of 3-way Symmetric Hash-join using SteMs 25

3.1 Missed sharing opportunities 31

3.2 RouLette operates as a special engine alongside a database. We present the architecture through a three-query example. 35

3.3 Multi-step optimization for join-phase 38

3.4 Execution of selection-phase 47

3.5 Execution of join-phase 48

3.6 Symmetric join pruning 49

3.7 Micro-benchmarks: a) coverage limitation, b) accuracy limitation, c) adaptivity 53

3.8 Sensitivity analysis for varying a) concurrency, b) selectivity, c) number of joins, d) query shapes 55

3.9 Comparison with offline sharing for small batch 56

3.10 Batches of large queries with correlated data (JOB) 57

3.11 Evaluation of learned policy for batches of JOB queries 58

3.12 Evaluation of learned policy for dynamic query admission 59

3.13 Synthetic schema for 4 chains and 9 relations in total 59

3.14 (a)-(h) Episode cost comparison: measured vs estimated 60

3.15 Comparison of learned and selectivity-based policies for synthetic schema . . . 60

3.16 Impact of optimizations and profiling for batch of JOB queries 61

3.17 Impact of optimizations and profiling for batch of sensitivity analysis queries . 61

3.18 Scalability to the number of cores 62

3.19 Comparison with concurrent query execution 62

4.1 Impact of concurrency on batch response time for batches of select-aggregate queries. The results in Figure 4.8 corroborate the conceptual graph 66

4.2 From queries to predicate index ranges 70

4.3 Trie-based grid index 72

4.4 Overview of SH_2O 73

4.5 Applying SH_2O at partition-granularity 79

4.6 SSBM and TPC-H benchmarks with scale factor 10 85

4.7 Effect of joint selectivity on data-access 87

4.8 Effect of concurrency on response time 88

4.9 Effect of the number of filter attributes on response time 89

4.10 Effect of data correlation on response time 90

4.11 Scalability for wide tables 91

4.12 Cost of building the spatial index 91
4.13 Scalability with the number of predicate index's ranges 92
4.14 Impact of partitioning on response time 93
4.15 Preprocessing cost for partitioning 93

5.1 ParCuR harmonizes reuse and work sharing to speed up recurring batches . . 96
5.2 Motivational example: work sharing introduces novel challenges for reuse . . . 99
5.3 ParCuR's workflow in a) the offline tuner and b) the online executor 103
5.4 Two-query example for subquery vectors 104
5.5 Example showing the effect of reuse on a global plan 109
5.6 Impact of reuse for a varying number of filter attributes 123
5.7 Impact of reuse for a varying number of joins per query 124
5.8 Impact of reuse for varying query selectivity 124
5.9 Impact of budget for workloads A and B 126
5.10 Impact of budget for workloads B-P1 and B-P2 127
5.11 Impact of workload shift in the filtering attributes 128
5.12 Impact of workload shift in the query templates 129
5.13 Impact of workload shift in the predicates of query patterns 129
5.14 Macro-benchmarks: a) SSBM b) TPC-H 130

1 Introduction

Real-time analytical workloads require processing a large number of queries at the same time due to the widespread adoption of applications that inherently submit queries in batches and the growing number of application users. On the one hand, applications, such as dashboards, notebooks, and data pipelines, that have become popular among analysts for presenting and automating analysis produce their output by processing multiple queries at once. Dashboards, for instance, present insights using several, sometimes even hundreds, visualizations [120]. Populating these visualizations requires processing the corresponding analytical queries. Similarly, in organizations such as cloud providers, pipelines process thousands of queries to cover analytical needs [33]. On the other hand, with the number of users growing, workloads become increasingly concurrent. Even for ad-hoc analysis, multiple users can jointly produce tens of queries per second [14]. Thus, applications need to operate and remain usable under high concurrency.

The onus of conserving the usability of real-time applications for highly concurrent workloads falls on the analytical databases and the infrastructure that support them. To be usable, real-time applications require timeliness. They are sensitive to wall clock response time, as it is disruptive for users [77]. For example, in visual data exploration, even a response time longer than 500 milliseconds impacts user productivity [68]. Meanwhile, concurrency needs to be transparent to timeliness: real-time applications require that timeliness holds for their workload's degree of concurrency. Applications in the industry commonly express requirements in terms of both the timeliness and the degree of concurrency to sustain. Dashboards in Youtube require responses within tens of milliseconds to tens of thousands of queries per second [14]. Similarly, Uber's use cases require processing thousands of queries per second with subsecond response times [31]. Furthermore, Meta's infrastructure for interactive analytics is designed for processing tens of concurrent wall clock-sensitive queries within seconds or minutes [107]. Hence, to effectively support such use cases, analytical databases need to use query processing techniques that, given the available infrastructure, achieve fast responses even when processing tens or hundreds of concurrent queries at any given time.

1.1 Scaling Concurrent Processing

Over several decades, analytical databases have been optimized for *query-at-a-time (QaT)* execution. QaT databases meet performance requirements by striving for maximum *efficiency* for each individual query; they use execution strategies that minimize processing time when evaluating each query. To this end, QaT databases are *data and workload-conscious*. They optimize execution for the data and the queries at hand using techniques such as query optimization, indexing, and materialization, which can drastically reduce the processing time by orders of magnitude.

However, as the number of concurrent queries is increased, efficiency for individual queries is insufficient for maintaining timeliness using fixed hardware resources. During concurrent execution, queries compete for limited hardware resources, such as CPU time. Databases give each query only a fraction of the total resources throughout its duration, e.g., by time-sharing the CPU across concurrent queries [124]. The result of contention is that the response time for individual queries deteriorates as concurrency is increased [93, 124]. Hence, reasoning about timeliness requires a notion of efficiency as a function of concurrency – this notion subsumes both the query processing time and the rate at which performance deteriorates.

The impact of concurrency on efficiency is critical for the timeliness and cost-efficiency of processing analytical queries. After some degree of concurrency that depends on the database's efficiency for the given workload and the application's requirements, response time becomes prohibitively long. Even worse, when the database cannot sustain the incoming query rate (i.e., queries arrive faster than they are processed), response time becomes unbounded. Under such conditions, analytical databases fail to meet the requirements of real-time applications. Then, to restore timeliness for applications, analytical databases need to scale up or out to larger infrastructure. However, running and operating large clusters for processing a high volume of analytical queries is expensive [51]. Furthermore, the demand for machine hours of infrastructure grows as a function of the number of concurrent queries, thus resulting in ever-increasing monetary costs.

Work sharing is an alternative to the QaT model that improves efficiency when processing concurrent queries with commonalities in data and computation; overlapping computations make up a substantial portion of the processing time in production workloads, e.g., in Microsoft's analytics clusters [51]. By exploiting overlapping data and work between queries, work sharing reduces extra processing for each additional query and, hence, blunts the surge in response time as concurrency is increased. Then, work sharing results in significantly lower response time and contributes towards cost-efficient real-time processing for highly concurrent analytical workloads. However, as we discuss in the next section, existing techniques suffer from redundant processing that introduces performance bottlenecks and puts timeliness at risk. In this thesis, we focus on minimizing the redundant processing, hence maximizing the efficiency of work-sharing databases in a scale-up in-memory execution environment.

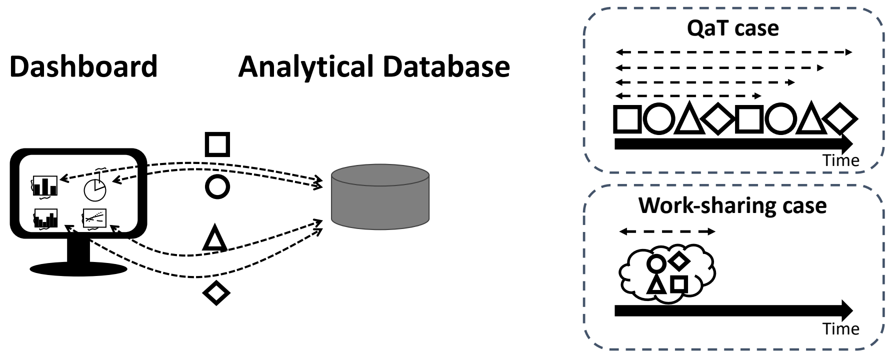


Figure 1.1: Example use case: supporting batch processing for a dashboard

Example: Figure 1.1 presents the above concepts through an example of a dashboard use case. To minimize response time, dashboards submit queries to the backend analytical databases in parallel, e.g., using multiple connections at the same time [120]. Assume that a dashboard with four visualizations refreshes its connection to the data source. Then, for each of the four visualizations, it submits a separate query (represented with a square, a circle, a triangle, and a diamond, respectively). We consider two cases: i) using a QaT database and ii) using a work-sharing database. A QaT database optimizes and processes each query individually. The concurrently executing queries share the available hardware resources, e.g., they alternately time-share the CPU until they are finished (timeline in QaT case represents scheduling slots, dashed lines above each query show the query's duration) hence their response time is increased compared to running in isolation. By contrast, a work-sharing database performs a shared computation for the batch of queries (the shape that contains queries in the work-sharing case). While work sharing's processing time is longer compared to that of any individual query running in isolation, it mitigates the effect of concurrency and thus results in lower response time. However, when work-sharing databases suffer from redundant processing, their ability to process the queries in a tight timeframe is undermined. Addressing redundancy reduces query response time under high concurrency.

1.2 Limitations to Efficiency in Work Sharing

The key idea behind work sharing is to eliminate duplicate data access and computation across queries. To do this, state-of-the-art work-sharing databases, such as SharedDB [35] and DataPath [3], evaluate multiple concurrent queries using a common data flow graph of operators, the *global query plan*. Each query is processed by a subplan in the global plan, and queries with overlapping subplans share work at tuple granularity. Each intermediate tuple belongs to one or more queries, which work sharing keeps track of. Each operator processes its input tuples once for all the queries they belong to and produces the corresponding output. Then, output tuples flow to consumer operators based on their own query-membership metadata. Hence, to eliminate duplicate computation and improve efficiency, work sharing exploits overlapping subplans and, also, overlapping tuples.

Efficiency for a set of queries is defined by the corresponding global plan's processing time. The processing time depends on the global plan's operators. For each set of queries, there are multiple candidate global plans which differ in terms of the processing time that they require. Furthermore, some candidate global plans are applicable only when databases have auxiliary state such as indices and materialized results. Choosing global plans and the available auxiliary state, thus, determines the efficiency of work-sharing databases and is essential for timely responses.

Choosing the global plan and the auxiliary state that minimize processing time is challenging. Global plans incur redundant processing as work-sharing databases i) fail to consider alternative global plans that require significantly less processing time than selected plans, ii) lack operators that minimize their respective part of the processing for participating queries, and iii) miss optimization opportunities due to the overhead that prospective optimizations incur when applied in a work-sharing environment. We identify the above sources of redundancy in plan selection, data access, and recomputation. The redundancy increases processing time and thus jeopardizes timeliness.

1.2.1 Global Plan Selection

Each concurrent query corresponds to a different relational algebra expression that the database needs to evaluate. Choosing an efficient global plan depends both on the chosen subplans for each query and the work sharing between them. First, as in QaT databases, choosing an efficient query plan for the target expression based on the data distribution and the query's predicates and computations is critical for processing time. For example, efficient join orders require orders of magnitude lower processing time than inefficient ones [67, 85]. Second, operator orders across different queries determine overlaps, and thus work sharing, in the global plan – overlaps eliminate duplicate work across queries. Hence, to choose an efficient global plan, work-sharing databases require sharing-aware query optimization that holistically reduces the total processing time for a batch of queries.

Challenge: Sharing-aware optimization either chooses inefficient plans or is time-consuming. Exhaustive approaches [36, 88, 106, 110] choose a global plan that they estimate to be optimal. They holistically consider alternative plans for different queries in terms of the processing time of their operators and the overlap between plan combinations. However, such approaches have a vast search space, and thus, they are best suited for optimizing queries that match a small set of templates, offline. An alternative that enables fast planning decisions is to choose plans by using heuristics [3, 12, 35, 44]. Nevertheless, heuristics suffer from inefficient plans because they underutilize sharing opportunities or because they use operator orders that are expensive for the given data and queries, e.g., they fail to exploit data correlations that aggressively filter intermediate results in joins and minimize processing. In both cases, heuristics choose plans that require high processing time.

Implication: Ad-hoc workloads, such as queries in exploratory or interactive data analysis, are unpredictable and only known at runtime. In the dashboard example, user actions result in one or more queries, e.g., by adding a new visualization or by filtering a data source that is linked to multiple visualizations, respectively. The set of submitted queries is ad-hoc, and, moreover, it is combined with other queries running on the same backend database to form ad-hoc batches. For ad-hoc workloads, exhaustive optimization is a poor fit as i) optimizing a-priori is not an option because the workload is unknown in advance and ii) it is too time-consuming to perform at runtime – it takes tens of seconds for a moderately-sized query batch [36]. The viable solution is to choose the global plan using heuristics. However, heuristics choose inefficient global plans that miss sharing and query optimization opportunities. Inefficient plans significantly increase response time and put timeliness at risk.

1.2.2 Data Access

Different queries analyze different subsets of the data, which they specify using predicates on relations. Efficiently accessing the data that each query requires is critical for low response time. To selectively retrieve the data that satisfies the predicates of each query and prune out redundant data, analytical databases use techniques that exploit data organization, such as index accesses and data skipping [114]. Moreover, work sharing introduces an additional dimension: sharing common accesses to amortize the respective cost. Hence, in a work-sharing environment, efficient data access requires eliminating overfetching of both redundant data and overlapping data across queries.

Challenge: Existing data access strategies suffer from redundant accesses. Selective access methods have limited sharing opportunities, especially when queries use different access methods. Then, as concurrent queries independently access the same data multiple times, the total required processing is also increased. By contrast, scans over the full data enable work sharing for data access and filters [94, 122] but access redundant tuples and eagerly process the filters regardless of data organization. Eagerly processing the full data is time-consuming and imposes a lower bound for response time. Kester et al. [59] argue for choosing between indices and shared scans based on the workload's selectivity and concurrency. However, a binary selection between the two techniques only chooses the best between two inefficient options. Hence, access path selection blunts but fails to address redundant processing during data access.

Implication: Selective workloads comprise queries that process a small subset of the full data each. The required subsets across different queries may overlap. For example, consider the case where a user's dashboard visualizes their dedicated data, e.g., sales in a specific region. The data processed collectively during a refresh is a small fraction of the full data. Each visualization can have various additional predicates hence data across visualizations overlaps but is not identical. Data access is critical for the response time of selective queries. However, under high concurrency, both shared scans and index accesses fail to retrieve the required subsets of data within a tight timeframe, e.g., a few milliseconds for interactive analysis. Hence, data access becomes a performance bottleneck for real-time processing.

1.2.3 Reuse

Applications that generate queries using templates or SQL features, such as views, produce workloads with recurring patterns. The same subqueries frequently recur with different parameters. Recurring patterns enable databases to eliminate frequent computations by precomputing and materializing the corresponding results in advance and then by using the materialized results to answer queries. QaT databases exploit this opportunity using techniques such as caching, recycling, and materialized views and subexpressions [48, 51, 100, 109, 136]. As global plans contain multiple time-consuming operators, exploiting reuse opportunities is critical for timeliness.

Challenge: In a work-sharing environment, reuse is ineffective and inefficient. On the one hand, reuse becomes ineffective because both its impact and its usability are decreased. The impact of reuse depends on the computations that it eliminates. Work sharing hinders reuse from eliminating shared operators, e.g., after answering one of their downstream computations in the global plan using a materialized result, as long as the results of the operators are still required for other downstream processing. Also, work sharing competes with reuse for eliminating overlapping work across concurrently executing queries and, thus, shrinks the marginal benefit of reuse, which is the headroom for reuse-related overhead. Furthermore, the usability of reuse depends on whether materialized results subsume the queries at hand (i.e., a materialized result contains all the required data to answer the corresponding query). Materialized results eliminate a shared operator only when they subsume all the queries that participate in the operator. As the number of queries is increased, and especially during workload shifts, the probability that materialized results subsume all of their corresponding queries in the global plan is decreased, and thus reuse fails to eliminate the respective shared operators for non-subsumed queries.

On the other hand, shared execution over materialized results is inefficient due to high filtering overhead. Work-sharing databases need to evaluate the predicates from all the tables that contribute to the view. However, in shared execution, the relative cost of filters is particularly high compared to other operators [70]. Even worse, the processing time for filters on relatively small tables, such as dimensions in a star schema, is amplified when materialized results are significantly larger. Filtering overhead is such that it exceeds the processing time for full recomputation using work sharing, hence reuse deteriorates performance.

Implication: Recurring workload is predominant in static dashboards, periodic reports, and web applications but also occurs due to SQL constructs such as views. One such use case in the dashboard example is iteratively refreshing multiple visualizations simultaneously, using different parameters every time, e.g., changing their common data source's filters in an exploratory manner. In QaT databases, materialized results are critical for optimizing performance for recurring workloads. However, in a work-sharing environment, using out-of-the-box techniques for materialization and reuse deteriorates performance and is a poor option for processing queries. Thus, work-sharing databases opt for full recomputation, which processes global plans with several heavyweight operators and suffers from long response time.

1.3 Data and Workload-conscious Work Sharing

This thesis addresses the inefficiency of global plans in work-sharing databases. It introduces planning, execution, and database-tuning strategies that holistically optimize each global plan to reduce the total processing time rather than the processing time of any particular query. To this end, the strategies exploit the characteristics of the data, the global plan's queries, which they view comprehensively as a batch, and the workload patterns across time. The end goal is to enable work-sharing databases to provide timely responses for highly concurrent workloads and to resolve the bottlenecks for scaling real-time applications that produce ad-hoc, selective, and recurring workloads.

Thesis Statement

As the number of concurrent queries in real-time applications is increased, response time explodes. Existing work-sharing databases mitigate the effect of concurrency but incur substantial redundant processing due to inefficient data access and reuse, and suboptimal global operator orders. Holistic strategies for the optimization and execution of global plans, and database tuning that adapt to the data and workload help minimize redundancy, hence also the response time, for concurrent workloads.

1.3.1 Goal: Efficiency under High Concurrency

Increasing work sharing's efficiency requires a paradigm shift. Work-sharing databases need to use, across all processing steps, planning, execution, and database-tuning strategies that holistically optimize processing for the data, the query batches, and workload patterns at hand.

Sharing-aware optimization is, by definition, concerned with holistically reducing processing time for the target data and query batch. However, commonly used heuristic approaches forfeit holism and thus suffer either from inefficient operator orders or from missing sharing opportunities. Work-sharing databases require planning strategies that restore the exploration and evaluation of alternative global plans while maintaining the fast planning for hundreds of queries that heuristics offer. At the same time, evaluation needs to be judicious and accurate based on the correlations that are present in the data and the query batch. Doing so enables work-sharing databases to choose global orders that significantly reduce processing time, especially for ad-hoc workloads.

Also, as both scans and indices overfetch data during shared execution, work-sharing databases require an access strategy that selectively retrieves the necessary data for processing each query batch and only once across queries. The predicates of participating queries collectively define the required data. Efficiently accessing the necessary data requires an appropriate data organization and a mechanism that exploits the organization to retrieve the necessary data with minimum access and filtering overhead. By tuning data organization for the target workload and then, for each subsequent query batch, by adapting access to the batch's predicates and the data, work-sharing databases can address the data-access bottleneck and enable timely processing for highly concurrent selective workloads.

Finally, addressing recomputation requires addressing the ineffectiveness and inefficiency of reuse in work-sharing environments. Effective reuse requires i) strategically employing materialization and reuse to maximize the cost of eliminated operators in each global plan and hence minimize the plan's total processing time and ii) relaxing subsumption requirements so that shared execution benefits even when materialized results only partially cover queries. Moreover, efficient reuse is tied to efficient data access to materialized results. Hence, materialization and reuse techniques require an overhaul by adapting to the data, query batches, and workload patterns so that they are harmonized with work-sharing databases. Harmonizing reuse enables drastically reducing response time for recurring workloads.

1.3.2 Contributions: Adapting Work Sharing to the Data and Workload

Data and workload-conscious planning and execution strategies for work-sharing databases require rethinking the query processing stack across three axes: i) logical optimization strategies, ii) query processing and data access abstractions and iii) physical design optimization. First, to make judicious and accurate cost-based decisions that reduce processing time, work-sharing databases need to formulate and solve optimization problems for each planning subproblem. Second, work-sharing databases need novel execution strategies and operators that can efficiently actuate the decisions of optimization strategies (e.g., for implementing novel access strategies). Third, work-sharing databases need to support auxiliary structures, such as partitions, indices, and views, and a tuner that chooses which auxiliary structures accelerate a target workload. We organize the contributions of this thesis in three building blocks RouLette, SH_2O and ParCuR, which address the challenges in global plan selection, data access, and reuse, respectively.

Figure 1.2 shows how each building block affects the query processing stack. Each numbered box corresponds to a contribution. Horizontal lines separate query processing in layers, and each layer is affected by one or more contributions. We discuss each of the numbered contributions in the text below.

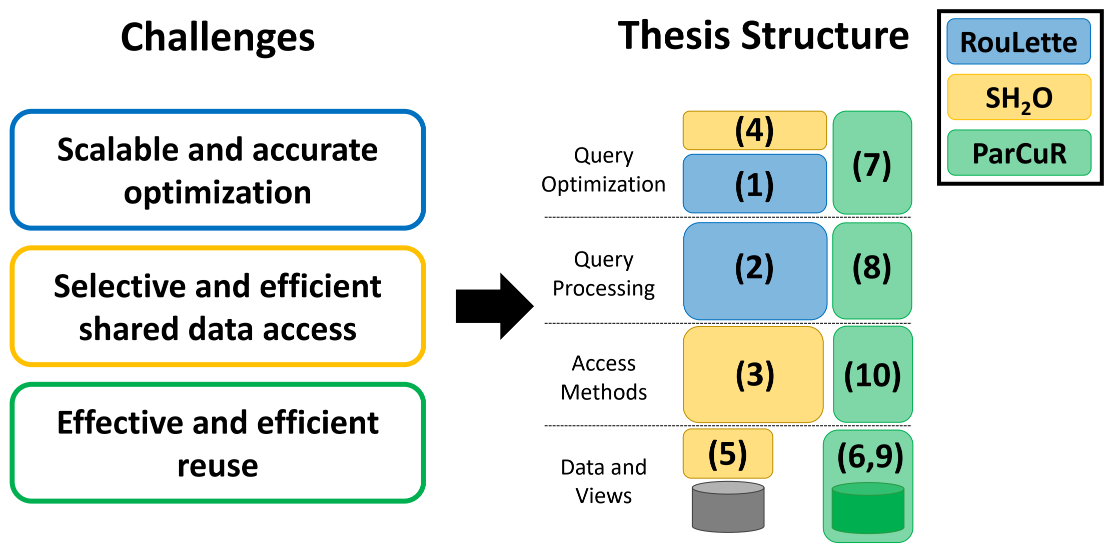


Figure 1.2: Challenges for efficient work sharing and the contributions of this thesis that address the challenges. Each contribution is denoted as *(X)* such that we refer to it in the text

The thesis makes the following key contributions:

Scalable and accurate sharing-aware optimization: We introduce RouLette [111], a specialized intelligent engine for multi-query execution that addresses sharing-aware optimization for Select-Project-Join (SPJ) subqueries. RouLette overcomes, through runtime adaptation, the shortcomings of heuristic sharing-aware optimization. It focuses on three problems when optimizing the global operator order: i) bounded optimization overhead, ii) accurate estimates for the cost of shared execution and judicious planning decisions, and iii) low adaptation overhead. RouLette mitigates the optimization overhead and accurately estimates the total processing time across queries by using adaptive query processing (**box (1)**): it immediately starts execution and iteratively reoptimizes the global plan by observing the outcome of execution and adapting the plan accordingly. Adaptive query processing incrementally explores the search space of candidate plans and, to judiciously choose an efficient global plan that minimizes total processing time, it makes operator ordering decisions using a novel reinforcement learning algorithm based on Q-learning. Finally, RouLette introduces query processing optimizations that minimize the overhead of adaptive query processing, making adaptation viable in practice (**box (2)**). By choosing more efficient plans, RouLette significantly improves response time compared to heuristic sharing-aware optimization.

Selective and efficient shared data-access: We propose SH_2O, a novel data-access operator that overcomes the limitations of both shared scans and index accesses. SH_2O is based on the insight that, for every set of filters, the data space comprises multidimensional regions where filtering decisions are invariant across all tuples. Thus, shared access to the regions that are required by at least one query amounts to i) avoiding data that all queries filter out, ii) accessing data only once for all interested queries, and iii) eliminating the corresponding filtering costs. SH_2O efficiently and selectively performs the shared access to the regions, regardless of their boundaries, by using range queries on spatial indices **(box (3))**. SH_2O accesses data by balancing between exploiting the regions and handling dimensionality **(box (4))**. First, it selects a subset of the filters to replace with multidimensional accesses such that it hits a sweet spot between the overhead for fine-grained index accesses and for post-filtering. Second, if the data is partitioned and indexed at partition granularity, it adapts the access strategy to each partition's local data organization and predicate patterns. Based on the above properties, we propose a partitioning and indexing scheme that minimizes SH_2O's processing time for a target workload **(box (5))**. SH_2O reduces the required processing time for shared data access and filtering.

Effective and efficient reuse for work sharing: We propose ParCuR, a framework that harmonizes reuse with work sharing in order to maximize the impact and applicability of materializations and minimize overheads. ParCuR adapts reuse to work sharing in three aspects: i) To achieve a higher decrease in the processing time of global plans through materialization **(box (6))** and reuse **(box (7))**, ParCuR introduces novel materialization and reuse policies that account for interference from work sharing. ii) To improve the usability of materialized results and to avoid performance cliffs when queries are partially covered, especially during workload shifts, ParCuR employs partial reuse. It implements partial reuse through partitioning and by making materialization and reuse decisions at partition granularity. Thus, we introduce an execution model that plans and processes each partition independently from other partitions **(box (8))**. As partitioning creates a dependency with materialization, ParCuR introduces a novel partitioning scheme that improves storage budget utilization **(box (9))**. iii) to reduce the filtering overhead when reusing materialized results, ParCuR builds and uses access methods to eliminate frequent filters **(box (10))**. ParCuR makes reuse beneficial instead of detrimental for work sharing and drastically reduces response times for recurring workloads.

1.3.3 Thesis Outline

Chapter 2 provides an overview of the areas that we build upon: work sharing, data and workload-conscious optimizations that QaT databases use, that is, query optimization, indexing and reuse, and reinforcement learning. Next, Chapter 3 presents a sharing-aware optimization paradigm for choosing global plans by learning which opportunities minimize processing time from monitoring the execution of ongoing queries. After that, Chapter 4 presents a sharing-aware data access operator that retrieves required tuples by collectively using the predicate of query batch along with existing spatial indices. Then, Chapter 5 presents a sharing-aware materialization and reuse framework that co-optimizes the data layout, the materializations, and reuse decisions to reduce recurring computations. Finally, Chapter 6 concludes the thesis and presents directions for future research.

2 Background

This thesis introduces novel data and workload-conscious optimizations that improve the performance of work sharing. In this chapter, we present a brief overview of the two overarching areas that the thesis touches upon, work sharing data and workload-conscious optimizations, and discuss reinforcement learning. First, we present the query processing stack for work-sharing databases and outline related work. Next, we discuss techniques that take advantage of data and workload characteristics to optimize individual queries, specifically query optimization, access methods and reuse. Finally, we briefly present reinforcement learning, which we later use in Chapter 3.

2.1 Work sharing

In this section, we present an overview of work sharing. The section is laid out as follows: First, we present applications where sharing is typically used in Section 2.1.1. Then, from Section 2.1.2 onward, we discuss the architecture of work-sharing databases and the main mechanisms used in the literature.

2.1.1 Applications

Work sharing has been applied to use cases where data processing systems execute a large number of concurrent queries. Such use cases include OLAP, OLTP, stream processing, scheduled queries, distributed frameworks, and machine learning workload. We briefly present the related work for each use case:

OLAP: OLAP databases process several concurrent queries that come from different users and applications. In such cases, databases can share common computation across queries to reduce the total processing time and increase throughput. The opportunity for sharing has motivated a significant body of work. Initial approaches focus on the problem of multi-query optimization [88, 106, 110], which entails finding a global plan that minimizes processing time. Nevertheless, multi-query optimization approaches cannot optimize a large number of queries due to the high complexity of the problem and suffer from execution-layer overhead. More recent efforts build work-sharing databases such as QPipe [44], CJOIN [12], DataPath [3], SharedDB [35], and RouLette [111] that can support a large number of queries and use execution strategies that maximize the benefit of work sharing. Subsequent sections analyze work-sharing databases in more detail.

OLTP: Transactional workload is highly concurrent. As transactional workload usually contains a small number of distinct statements [99], there is significant overlap; hence work sharing is beneficial for improving OLTP throughput. SharedDB [35]'s batched execution model supports OLTP workloads in addition to OLAP workloads. In OLTPShare, Rehrmann et al. [99] merge batches of READ statements by using a statistics-driven batching policy that maximizes throughput without introducing unnecessary delays. Follow-up work from Rehrmann et al. [98] studies the mergeability of multi-statement READ/WRITE transactions in different isolation levels.

Stream processing: To increase data throughput when processing a large number of streaming queries, stream processing engines also use work sharing. Stream processing engines share computation for various types of queries. TelegraphCQ [13, 69] and NiagaraCQ [17] use work sharing for continuous queries, whereas AStream [56] and AJoin [55] use work sharing across short-lived ad-hoc streaming queries. Stream processing engines use work-sharing techniques similar to the ones used in work-sharing OLAP databases.

Scheduled queries: Scheduled queries are queries that need to produce results by a given deadline. They are processed using an incremental execution model that is similar to stream processing [118]. Scheduled queries often operate over the same data and thus have common computation. iShare [119] judiciously shares work between queries with different deadlines: even though it exploits work sharing, it avoids pushing the whole shared execution to meet the tightest deadline. By doing so, iShare reduces resource consumption.

Distributed data analysis: Distributed frameworks such as MapReduce [22], Spark [134], and DryadLINQ [133] express data processing as jobs. Different concurrent jobs can have overlapping computations. Frameworks such as MRShare [86], PigReuse [11], Nectar [41] and the work from Wang et al.[126] identify and exploit common computation between jobs to reduce the total cost of processing.

Analytics and ML: Work sharing also accelerates analytics and ML workload. LMFAO [103] addresses analytics, such as regression and data cubes, that result in batches of aggregates over the same join. It uses a layered architecture that exploits work-sharing, parallelism, and code specialization. Moreover, Derakhshan et al. [23, 24] targets common computation in data science and ML workload. Specifically, they employ materialization and reuse to reduce redundant computations in collaborative environments [23] and data science pipelines [24].

This thesis focuses on work sharing for OLAP applications. While the insights and contributions have merit for work sharing in general, the design assumptions, implementation,

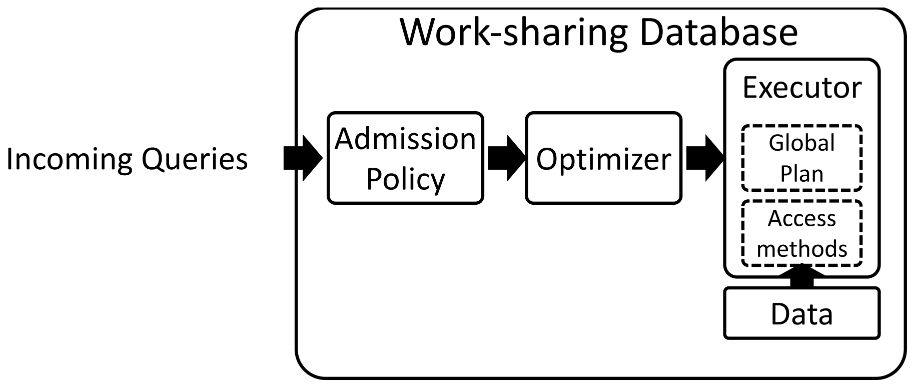


Figure 2.1: Flow of queries through a work-sharing database

and evaluation are OLAP-specific. Hereupon, further discussion assumes an analytical query processing scenario.

2.1.2 Architecture of Work-sharing Databases

We present an overview of the state-of-the-art in work sharing for OLAP workloads. By exploiting overlapping work across queries, work sharing reduces the total processing time and increases throughput. To achieve this, work-sharing databases differ from QaT databases on i) the admission policy, ii) the data-access methods, iii) the execution engine, and iv) the optimizer.

Figure 2.1 shows the flow of incoming queries through the work-sharing database's components. We illustrate the architecture of work-sharing databases in abstract terms to fit the description of diverse systems such as QPipe [44], CJOIN [12], DataPath [3], SharedDB [35], and RouLette [111]; we stress differences between systems in subsequent sections. First, queries are handled by the system's admission policy which hands them over to the optimizer individually or in batches. Second, the optimizer reacts to admitted queries by choosing a *global (query) plan*, which processes all ongoing queries and expresses the sharing of common operators. Third, the access methods read data from storage and forward it to the global plan. Fourth, the executor processes the global plan by pushing data from the access methods through the plan's operators.

2.1.3 Admission Policy

Work sharing is applicable when databases process multiple queries concurrently. The concurrency in a database depends on i) the application and ii) the database's admission policy. While applications such as dashboards submit queries in batches, other applications such as ad-hoc analysis submit individual queries. As the application's characteristics are beyond the database's control, to present the design space in work-sharing databases, we focus on the admission policy.

Work-sharing databases either start executing queries immediately as they arrive or delay them and execute them in batches. On the one hand, immediate admission enables sharing data and work with already running queries, and the waiting time it incurs is minimal. However, opportunities for sharing operators that belong to already running queries are limited for a wide range of relational operators due to timing dependencies [44] and, in addition, overlap fully depends on the workload's query submission patterns. On the other hand, batched execution, by eliminating timing dependencies, maximizes sharing opportunities between queries that belong to the same batch. Moreover, by offering full knowledge of concurrently executing queries, batching permits additional logical optimizations. Nevertheless, it introduces waiting time for each query and precludes work sharing between queries in different batches.

The choice of admission policy presents a trade-off for the design of work sharing. QPipe [44], CJOIN [12], and DataPath [3] use immediate admission to reduce processing time with minimum waiting time for each query. SharedDB [35] introduces batched execution to maximize work sharing and minimize total processing time. It composes a batch of incoming queries while the previous batch is running and then executes it as soon as the previous batch is finished. Despite their differences, both approaches reduce total processing and follow similar execution models.

2.1.4 Data Access

Existing work-sharing databases have used two different approaches to access data: i) shared scans and ii) shared index probes. We present the two approaches and subsequently discuss their limitations.

Shared scans: Sharing scan operators optimizes bandwidth utilization for the storage medium where data resides. Traditionally, disk-based databases suffered from the I/O bottleneck, and thus amortizing data access time across queries was critical. For this reason, several commercial systems, such as Microsoft SQL Server, RedBrick and Teradata, support scan sharing [44]. Cooperative scans [138] further extend disk-based scan sharing by both maximizing bandwidth and minimizing average latency for queries requesting uneven data ranges. Also, in addition to disk-based databases, in-memory databases use scan sharing to mitigate the memory bandwidth bottleneck when processing queries using multi-core CPUs [94].

Work-sharing databases use scan sharing both for improving data-access bandwidth and for providing input to downstream shared operators. Crescando [122] integrates shared scans with efficient shared selections to achieve fast and predictable performance for Select-Aggregate concurrent queries. Recent work-sharing databases either use Crescando [35, 70] or implement shared scans and filters in a similar way [3, 111]. We discuss the implementation of shared selections in detail in Section 2.1.5.

Shared index probe: Work-sharing databases can also share index probes, although only SharedDB [35] implements this access method. It uses a technique that batches index probes to improve instruction and data cache locality and to produce a shared result set across participating queries [34]. Outside the context of work sharing for OLAP, OLTPShare [99] also merges OLTP read-only queries into bulk index lookups. Also, in the context of information filtering applications, Fischer and Kossmann [30] propose a technique that includes merging identical probes into one index lookup and avoiding overlapping accesses between consecutive probes.

Limitations: Both shared scans and indices result in time-consuming data access for highly concurrent workloads. On the one hand, scans access the full data and, in addition, incur high filtering overhead. On the other hand, using multiple index probes that serve different queries, e.g., QaT probes or when using different indices for different sets of queries, results in redundant accesses and restricts downstream work sharing. Furthermore, while shared index probes reduce data access, they require post-filtering for the probe's results, which incurs high overhead in a work-sharing environment. Also, shared index probes depend on the properties of each index structure; they have not been studied for workloads that use predicates on multiple attributes, which complicate index traversal patterns hence the detection of overlaps.

In this Thesis: We introduce a novel data access operator, SH_2O, that overcomes the limitations of existing strategies: i) SH_2O identifies a shared multidimensional access pattern that replaces a shared scan followed by a set of shared filters. It actuates the access pattern using a spatial index and thus avoids accessing redundant data and processing the replaced filters. ii) Any spatial index that supports range queries can actuate the multidimensional access pattern. SH_2O exposes overlaps between queries regardless of index traversal patterns and generalizes for workloads where different queries use filters on different attributes. iii) SH_2O handles the dimensionality problem that emerges when evaluating numerous predicates over multiple attributes: it introduces a cost model-based optimization framework that decides which filters to replace with the multidimensional access pattern and, for each partition, it exploits local data organization and predicate patterns if available. We complete the framework by proposing a data organization strategy that minimizes SH_2O's processing for a target workload.

2.1.5 Shared Execution

Work-sharing databases accelerate query batches by exploiting overlapping computation across queries. To do so, they rely on i) the *global query plan* and ii) the *Data-Query model.* In this section, we present the two concepts. For ease of presentation, we use a simple example in which the work-sharing database processes two queries:

```
Q1: SELECT SUM(X) FROM A,B,C
    WHERE A.1=B.1 AND A.2=C.2 AND A.4 < 10 AND B.5 < 20

Q2: SELECT SUM(X) FROM A,B,D
    WHERE A.1=B.1 AND A.3=D.3 AND A.4 < 20 AND B.5 < 10
```

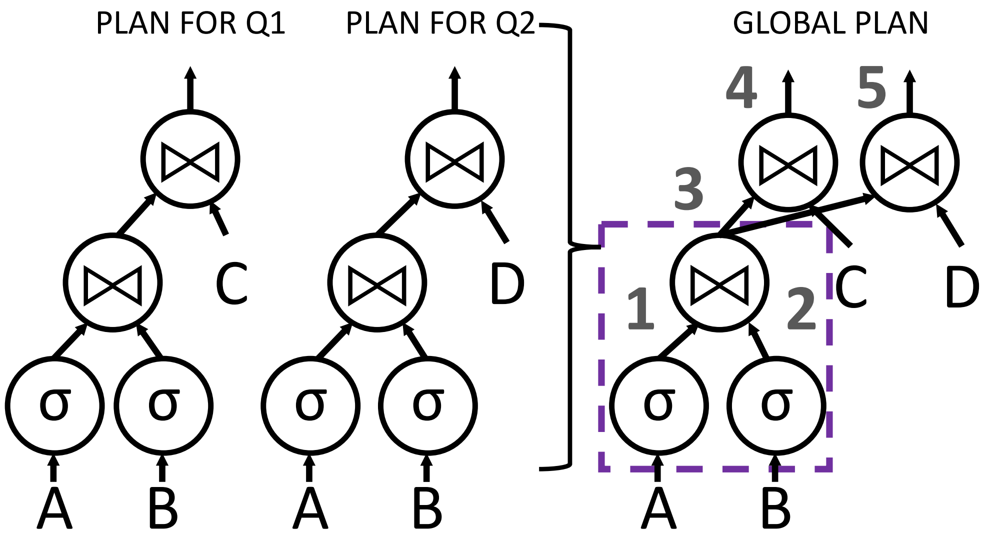


Figure 2.2: Global plan for Q1 and Q2 (example in text)

Global Plan

The global plan expresses sharing opportunities among different queries and is the common denominator of work-sharing databases explicitly [3, 35, 70, 111] or implicitly [12, 44]. It is a directed acyclic graph (DAG) of relational operators that process tuples for one or more queries and multi-cast their results to one or more parent operators. In each of its roots, the global plan produces the results of participating queries. Figure 2.2 shows the global plan for Q1 and Q2. For ease of reference, each operator is labeled with a number. Operators 1-3 process $\sigma(A) \bowtie \sigma(B)$ for both queries. Then, operator 3 sends results to operators 2 and 3, which serve Q1 and Q2, respectively. Different predicates between Q1 and Q2 are handled during multi-casting using the Data-Query model (discussed in the next paragraph). Downstream computation is independent for each query and at the two roots, the global plan produces the results for Q1 and Q2. By processing each operator of the global plan only once, the database shares work across queries (e.g., processing $\sigma(A) \bowtie \sigma(B)$ between Q1 and Q2) and reduces the overall processing time.

Data-Query model

The Data-Query model enables efficient work sharing between queries with different selection predicates. In such cases, using standard relational operators requires pushing down the union of predicates – in our example, $A.4 < 20\ AND\ B.5 < 20$ – below common operators and post-filtering after the queries branch off – $A.4 < 10\ AND\ B.5 < 20$ for Q1 and $A.4 < 20\ AND\ B.5 < 10$ for Q2. This rewrite is inefficient [62] as i) it produces and processes redundant tuples that do not belong to any query and ii) it filters data several times within the plan. For example, it is possible that operator 1 joins a “probe” tuple that belongs only to Q1 with a “build” tuple that belongs only to Q2 and filters it out afterwards.

The Data-Query model addresses these two inefficiencies by abstracting away predicates: instead, it annotates each tuple with a *query-set* that indicates to which queries the tuple contributes. Concretely, the model expresses tuples as

$$a = (a_1, a_2, \ldots, a_n, \underline{a_q})$$

where $a_1, a_2, \ldots, a_n$ are attributes and a_q is the set of queries a belongs to. Query-sets can be implemented in various ways such as lists of query-ids or bitsets [70].

Specialized *shared* operators exploit Data-Query model to eliminate redundant intermediates and processing. They process both the actual input tuples and their query-sets, and produce the union of output tuples across participating queries for the same operator, again in Data-Query model. Each output tuple’s query-set correctly marks its membership to each query, and if the tuple does not belong to any query, the operators drop it immediately.

We discuss how Data-Query model is used in global plans and present shared selections and joins, the shared operators that are supported in all databases that use the Data-Query model. SharedDB, due to its batched execution model, also supports Sort, TopN, and Group-By.

Data-Query model in global plans: Several work-sharing databases [3, 12, 35, 70] use global plans that comprise shared operators and represent intermediates using the Data-Query model. The difference in using shared operators lies in multi-casting their output to their parent operators. For each tuple, they decide which parent operators require the tuple using the query-set. Thus, the Data-Query model affects the flow of data through the global plan.

Shared Selection: Shared selection evaluates predicates for one or more queries and updates the query-set accordingly. Let a be an input tuple, $\mathcal{Q}$ be the set of running queries, and $\mathcal{Q}_{sat}(a)$ the set of queries such that $q \in \mathcal{Q}_{sat}(a)$ if and only if the predicate of q in the operator is satisfied or q has no predicate in the operator. Shared selection excludes queries with false predicates by updating a_q to $a_q \cap \mathcal{Q}_{sat}(a)$. Figure 2.3 shows processing for a shared selection for the running example. The input tuple satisfies the predicate of Q2 therefore the output tuple’s query-set is $\{Q2\}$.

The performance of shared selections is critical because i) as selections are the first operators after scans due to pushdown, they process a large fraction of the input, and ii) they need to process potentially hundreds of predicates belonging to different queries. A naive implementation would update each tuple’s query-set by going over all the predicates of the concurrent queries and checking which are satisfied and which are not. However, this algorithm is linear to the number of concurrent queries and introduces significant overhead in highly concurrent processing. To make shared selections efficient, prior work [35, 69, 122] proposes using predicate indices (PIs).

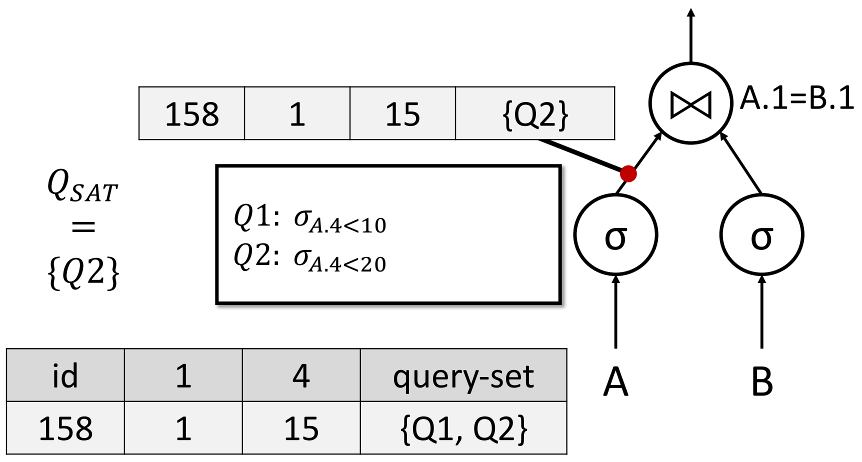


Figure 2.3: Example of shared selection

Unlike conventional indices, which are preconstructed, PIs are built at runtime, and their lifetime is the duration of the concurrent queries. Rather than index data, a PI indexes the predicates of the queries on a specific set of attributes. Predicates that are not covered by any PI are not indexed and are handled separately. To cover the selections of all queries, multiple PIs may be used.

Global plans evaluate predicates on different attributes by using multiple PIs. For each tuple they process, shared selections probe PIs, using the tuple's corresponding attributes, to identify satisfied predicates and set the tuple's query-set accordingly. If the query-set becomes empty, they discard the tuple. Figure 2.4 illustrates the process of probing a predicate index for the above example; the probe identifies that $A.4 \in (-\infty, 20)$ and the tuple belongs to Q2. After passing through shared selections, tuples have query-sets that represent the results of the predicates for all queries.

The implementation of PIs differs based on i) how they group predicates and merge probe results and ii) the data structures they use. In terms of merging strategies, TelegraphCQ and Crescando propose two different approaches. TelegraphCQ uses grouped filters. Each grouped filter builds one index for all the predicates on a specific attribute [69]. Internally, it uses a different data structure to organize each type of predicate (e.g. equality, less-than, etc). Grouped filters evaluate conjunctions of predicates by finding unsatisfied predicates from each attribute and removing them from the query-set. By contrast, Crescando indexes at most one predicate per query and thus partitions queries across PIs. When probing each PI, it retrieves a set of queries with at least one satisfied predicate; then, it evaluates the rest of the predicates only for these queries. Then, it produces each tuple's query-set by computing

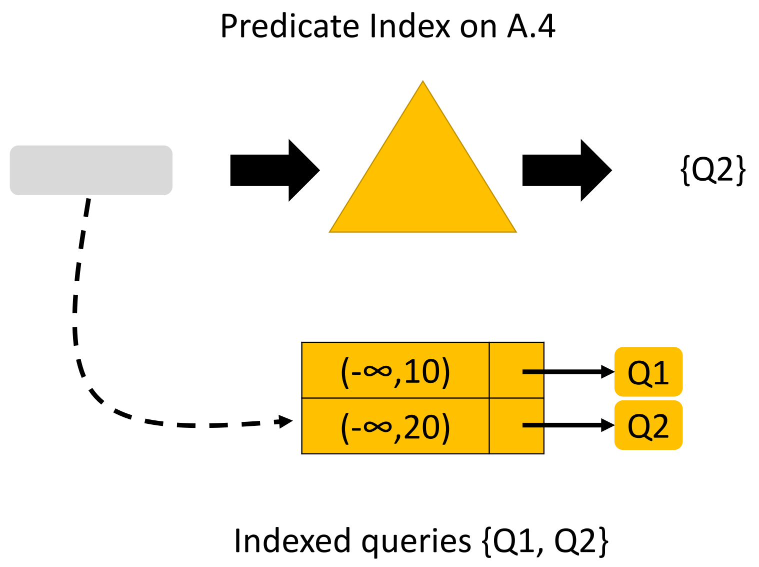


Figure 2.4: Example of probing a predicate index

satisfied queries across each PI and by merging the results. Thus, both approaches require several probes to compute each tuple's query-set.

Finally, in terms of data structures, different indices are used based on the type of indexed predicates. For example, both TelegraphCQ and Crescando use hash-tables for equality predicates, whereas for range predicates, they use binary trees and an R-tree, respectively. Both systems use PIs that index predicates on one attribute each.

Shared Join: Shared join matches Data-Query model tuples from its inputs based on a predicate. For each match between tuples a^1 and a^2, the join produces a new shared tuple a^3 that belongs to the intersection of the query-sets of the matching tuples, $a^1_q \cap a^2_q$. If the intersection is empty, the join drops the match. Figure 2.5 shows a shared join between tuples in Data-Query model. The match with a non-intersecting query-set is dropped, whereas for the other match, the query-set is the intersection of query-sets ($\{Q2\}$.

Work-sharing databases can support different implementations for shared joins, such as hash-join, sort-merge, and nested-loop joins. MQJoin [70] is a state-of-the-art dedicated shared hash-join algorithm. By minimizing redundant work, and by using main-memory bandwidth and multi-core CPUs efficiently, MQJoin maximizes its data processing rate and can efficiently process hundreds of concurrent queries.

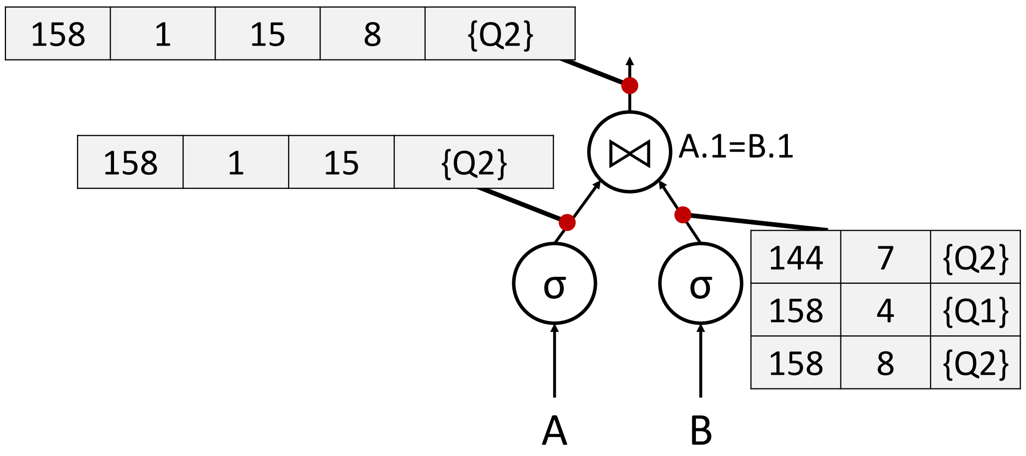


Figure 2.5: Example of shared join

2.1.6 Sharing-aware Optimization

In work-sharing databases, the optimizer chooses the global plan for processing the submitted concurrent queries. To do this, work-sharing databases use different mechanisms to detect sharing opportunities. We classify the optimizer mechanisms using the following taxonomy: *online sharing*, which quickly detects opportunities at runtime, and *offline sharing*, which uses exhaustive sharing-aware optimization. Figure 2.6 evaluates the mechanisms of existing approaches on i) the efficiency they achieve based on the opportunities they detect and exploit and ii) their ability to optimize (i.e., scale to) a large number of ad-hoc queries in real-time. We focus on ad-hoc queries to factor out a-priori optimization for predictable workloads. The two classes of mechanisms have complementary strengths and weaknesses. Online sharing scales to large numbers of ad-hoc queries but misses opportunities. By contrast, offline sharing maximizes the benefits of work sharing but has high complexity.

Online sharing: In order to make timely planning decisions at runtime, online sharing makes heuristic sharing decisions that are locally optimal in some sense. Online sharing approaches differ in terms of when and how they choose to exploit sharing opportunities. QPipe [44] and DataPath [3] detect and choose opportunities to exploit every time a new query is admitted. QPipe detects common subplans between the QaT plans of incoming and ongoing queries, whereas DataPath incorporates each incoming query into the global plan of already-optimized concurrent queries such that additional cost is minimum. CJOIN [12] and CACQ [12, 62] continuously evaluate and choose opportunities at operator level while the queries are running. Both systems reorder operators at runtime based on selectivity in an effort to minimize processing time. Finally, AJoin [55] sporadically triggers an approximate cost-based

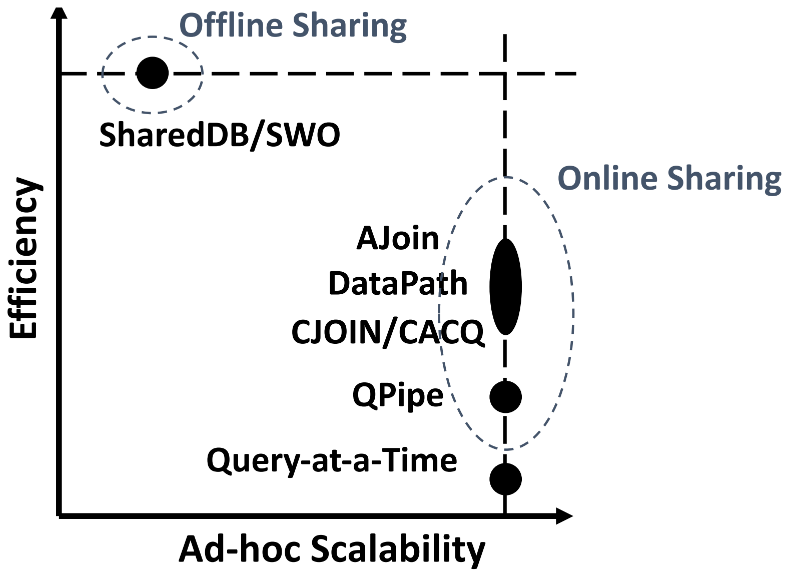


Figure 2.6: Tradeoff between efficiency and the scalability of sharing-aware optimization mechanisms

optimization algorithm inspired by Iterative Dynamic Programming [61] to produce a global plan for ad-hoc stream queries.

Online sharing approaches can be suboptimal because they miss sharing opportunities or make inefficient planning decisions. On the one hand, they explore a limited search space of global plans. For example, QPipe's global plan is determined by individual query plans, whereas DataPath's search space of global plan depends on the order of query arrivals. On the other hand, they evaluate sharing opportunities inaccurately. For instance, operator-level techniques evaluate the immediate effect of each operator hence they miss operator correlations and the long-term effects of planning. Therefore, online sharing can choose an inefficient global plan and hence result in a long response time. Furthermore, with the exception of QPipe which does not use the Data-Query model and thus misses opportunities between queries with different predicates, online sharing approaches miss different opportunities and choose inefficient plans under different conditions hence they are incomparable in terms of efficiency.

Offline sharing: Offline sharing uses exhaustive sharing-aware optimization to find a global plan that minimizes processing time for a batch of queries. The problem was originally formulated as Multi-query Optimization (MQO) [106]. MQO algorithms explore alternative plans for the batch's queries to choose an efficient global plan. However, they use standard relational operators and hence process subexpressions that subsume all individual participating queries, instead of using the Data-Query model. Shared Workload Optimization (SWO) [36] adapts the problem formulation for work-sharing databases that support the Data-Query model. It optimizes a batch of prepared statements, that lack specific predicates, and identifies shareable operators between the statements rather than subexpressions that exactly match.

Offline sharing is challenging due to its very high complexity; solving the optimization problem can be prohibitively time-consuming. Exhaustive MQO algorithms [88, 106, 110] explore a doubly exponential space. Even heuristic algorithms [101] require significant time to optimize only tens of queries at once. The same holds for SWO, which requires tens of seconds to optimize the queries of the TPC-H benchmark [36]. For this reason, exhaustive sharing-aware optimization is a better fit for moderately-sized predictable workloads – then it can run offline, and the result can be reused.

In this Thesis: Online sharing chooses suboptimal global plans, whereas offline sharing depends on exhaustive sharing-aware optimization, which is prohibitively time-consuming to perform at runtime. This thesis introduces RouLette, a novel approach that overcomes the shortcomings of online sharing while maintaining scalability to highly concurrent ad-hoc workloads. RouLette explores a wider search space of candidate plans compared to query-oriented approaches such as QPipe and DataPath and uses a reinforcement learning-based heuristic that chooses more efficient global plans than selectivity-based approaches.

2.2 Data and Workload-conscious Optimizations

This thesis proposes data and workload-conscious optimizations for work sharing in three axes: query optimization, data access, and reuse. In this section, we summarize related work for QaT execution in these areas. First, we present existing work in Query Optimization in Section 2.2.1. Then, we provide an overview of work that optimizes data access in Section 2.2.2. Finally, we discuss work in materialization and reuse in Section 2.2.3.

2.2.1 Query Optimization

Relational query languages, such as SQL, express queries using a high-level declarative description. Query optimization plays the critical role of transforming the query's declarative description into a *query plan*, an efficient implementation that uses interconnected relational operators. The challenge is that query optimizers choose between a large number of equivalent query plans, all of which implement the query. This is a high-stakes decision because an inefficient plan can be several orders of magnitude slower than the optimal one [67].

A broad body of work exists on query optimization. In this section, we briefly present two critical components for query optimization i) search space enumeration and ii) estimates for cardinalities and cost. Then, we discuss two of its subdomains that are relevant to this thesis i) adaptive processing and ii) learned query optimizers.

Search space enumeration: Query optimization chooses between a large number of equivalent query plans hence a significant body of work focuses on efficiently enumerating candidate query plans. In their influential work, Selinger et al. [105] introduce a dynamic programming algorithm for enumerating the plans for Select-Project-Join (SPJ) queries. More general frameworks such as Volcano [40] and Cascades [38] provide more general, extensible optimizer architectures that combine algebraic transformations with dynamic programming. Still, despite the use of techniques such as dynamic programming, the search space and the computational complexity is exponential. Thus, query optimization scales poorly for query plans with many operators. The scalability limitation has motivated enumeration strategies such as Iterative Dynamic Programming [61] and the adaptive optimization from Neumann and Radke [85], that produce efficient, but not necessarily optimal plans by pruning the search space and thus reducing complexity.

Cardinality and cost estimates: To evaluate candidate plans during enumeration, optimizers estimate the cost of each operator. As the cost depends on each operator's input size, evaluation requires estimating the cardinality of intermediate results as well. For this purpose, optimizers typically use statistics to estimate cardinalities [92] and sophisticated cost models that, by processing cardinality estimates, compute metrics such as CPU, I/O, and communication costs [15]. However, accurately estimating cardinalities is challenging due to data correlations, especially across joins [67], and often insufficient statistics [4]. Work on both adaptive optimization and learned query optimizers strives to improve estimation.

Adaptive Query Processing: Adaptive processing targets unpredictable environments with limited statistics and highly correlated data, where the traditional query optimization paradigm performs poorly [26]. It adapts planning by exploiting information collected at runtime in a feedback loop. Based on the adaptation frequency and mechanism, there exist several classes of adaptive processing. Continuous adaptation reorders the operators in query plans at runtime. Often, continuous adaptation schemes are driven by a special operator, the eddy [4], which monitors the input and output of operators, and optimizes operator order accordingly. Symmetric joins [128] and SteMs [96] further enhance eddies-based adaptation with additional reordering opportunities and increased adaptability. Another alternative is progressive query optimization [75], which uses a special operator, CHECK, as a point for validating cardinalities and reoptimizing queries. If CHECK detects significant misestimation, it triggers reoptimization which then exploits both the actual measured cardinality and partial results to speed up processing. An additional option is proactive reoptimization [5], which focuses on detecting the need for adaptation early during execution. To do this, it accounts for uncertainty using estimate intervals, uses the estimate intervals during query optimization, and employs sampling-based statistics collection. Finally, LEO [113] introduces a paradigm where optimizers improve their future estimates using information collected as a byproduct of the execution of previous queries. In all approaches, optimizers make more informed, judicious planning decisions.

We further discuss the adaptive techniques that are used in this thesis: symmetric hash joins, eddies, and State Modules.

Symmetric Hash-join: Traditional hash-joins impede revisiting planning decisions at runtime [26], as the choice of build relations and the execution order is static. First, adapting the choice

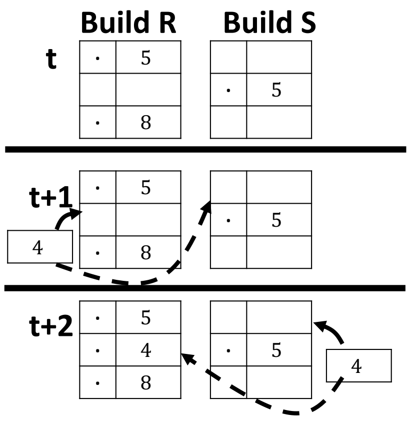


Figure 2.7: Example for Symmetric Hash-join

of build relations is expensive because it requires changing the existing state. Second, they impose scheduling constraints because build-side processing precedes probe-side and join result processing. The *symmetric hash-join (SHJ)* operator [43, 128] addresses both issues by treating inputs equally; it processes tuples from both inputs in any interleaved order.

To process tuples in any interleaved order, SHJ builds hashtables on both inputs. It inserts every input tuple in the respective hashtable, then probes the other relation's hashtable for matches. SHJ produces each result tuple when the matching tuples from both sides have been consumed. Figure 2.7 shows an example of matching two tuples. SHJ processes *R*'s tuple with key 4, inserts it in *R*'s hashtable, and, without matches, probes *S*. Next, SHJ processes *S*'s tuple, also with key 4, inserts it in *S*'s hashtable, and, to match with *R*'s preceding tuple, probes *R*. Thus, SHJ enables out-of-order processing but increases materialization cost and footprint.

SHJ also generalizes to joins with *n* input relations. For each input tuple, it probes n-1 hashtables. It decides the probe order at runtime, and hence it seamlessly switches between different left-deep plans. N-ary SHJ has been used in both stream processing [26, 123] and robust query processing [10, 66].

Eddies: An *eddy* operator [4] plays the role of the query optimizer during runtime adaptation. To do this, it chooses the order of the running query's operators at tuple granularity at runtime. It controls how tuples flow through operators by acting as a router at the center of the execution: for each input tuple or intermediate tuple, it chooses an operator to consume the tuple, and the consumer operator returns any produced tuples to the eddy for further processing. Hence, eddy observes the input and the output of each operator and, by using a runtime policy, it adapts its decisions and thus the effective operator order.

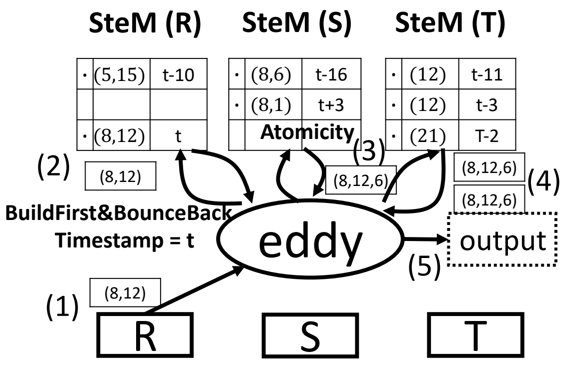


Figure 2.8: Example of 3-way Symmetric Hash-join using SteMs

Eddies can adapt the operator order as long as the query plan comprises commutative and symmetric operators, such as SHJ. However, even in that case, accumulated operator state limits adaptability. For example, inserted tuples in an SHJ are immutable, and thus they remain at the build side of the effective plan. These tuples cannot produce any new intermediate tuples until they are probed. Hence, the state makes routing history-dependent [25].

State Modules (SteMs) [96] enhance the adaptability of eddies by guaranteeing history-independence. A SteM is an index that stores tuples for each base relation. It exposes two operations, *insert(a)* and *probe(a)*: insert stores tuple a in the SteM and probe joins, based on a key, with previously inserted tuples. When used with eddies, SteMs store tuples at the endpoints of a join and avoid materializing intermediate tuples. Then, simplifying state management enables the eddy to adapt access methods, join algorithms, and join spanning trees at runtime.

SteMs can implement n-ary SHJ [96]. Figure 2.8 shows a running example for a 3-way SHJ. SteMs serve as hash tables, whereas the eddy dynamically reorders probes. The eddy first inserts each input tuple to its relation's SteM, e.g., *R*, producing an insertion timestamp. The insert operation returns the tuple back to the eddy. Then, the eddy atomically probes other SteMs e.g. $SteM_S$, then $SteM_T$, using the input tuple's timestamp to ensure a total order between tuples (atomicity). The sequence of probes produces the join between the input tuple and any previously processed tuples from other relations.

In this Thesis: RouLette acts as a query optimizer for SPJ subqueries. It adopts an adaptive processing paradigm based on eddies and SteMs. As the eddy produces global plans using fast decisions, the planning cost is linear to the plans' size hence scalable. However, existing adaptive processing techniques are insufficient for reducing the total processing time because i) eddy policies make inaccurate planning decisions, and ii) adaptive processing introduces runtime overhead. RouLette enhances adaptive processing by improving planning using a learned policy and by introducing execution techniques that reduce overhead.

Learned Query Optimizers: Query optimization suffers from the limited accuracy of estimates, inefficient search space exploration, and the need for significant hand-tuning. Learned query optimizers, by training on the data and workload at hand, aim to achieve low-error estimates and to identify efficient data and workload-specific heuristics. They learn to optimize both individual components such as join order enumeration [63, 74, 132], cardinality and selectivity estimates [28, 60, 84, 131], and full-blown rule-based optimizers [72, 73, 83]. The above approaches are trained offline and can improve optimization across a sequence of queries. By contrast, SkinnerDB [121] proposes an alternative approach that, to achieve regret-bounded query evaluation, learns from query execution at runtime. It splits execution into slices and tries different join orders across slices, thus evaluating progress and learning efficient plans. Thus, SkinnerDB bridges ideas in reinforcement learning and adaptive processing.

In this Work: To match the plan quality of query optimization in RouLette, the eddy needs to choose efficient operator orders. Existing selectivity-based techniques for reordering operators are greedy hence often suboptimal. To refine operator ordering and minimize the total processing time, RouLette uses reinforcement learning because it can model the correlations and long-term effects of planning decisions.

RouLette uses reinforcement learning differently than learned query optimizers [63, 73, 74, 132] for QaT databases. Learned query optimizers are trained offline and improve planning throughout a sequence of queries. By contrast, RouLette learns to order operators throughout the lifetime of queries. Learning is completely online and discards information after queries finish processing. We make this design choice for two reasons: i) planning decisions for a query batch do not generalize for seemingly similar future batches because they depend on the batch's predicates, which are rarely the same, and ii) the exact same batch recurs less often than the exact same subquery and thus learning has to rely on fewer offline samples.

RouLette is more similar to SkinnerDB [121], which bridges reinforcement learning and adaptive processing. However, it is different from SkinnerDB in that it targets sharing-aware optimization. RouLette explores different candidate global plans at runtime and learns which global plans reduce the total processing time by observing execution outcomes. Also, RouLette introduces execution techniques that minimize the overhead of adaptive processing.

2.2.2 Access Methods

Efficient data access is a significant factor in providing fast responses to queries. To this end, databases organize their data such that they provide *access methods*, that is, data structures and algorithms that can reduce the retrieved data for a target workload. This thesis applies two types of data organization, indexing, and partitioning.

Indices: An index is a data structure that enables queries to selectively access data based on a predicate. Indices show significant diversity: they differ in terms of the predicates they support, their granularity, their performance characteristics, their storage overhead, the overhead for building and updating them, etc. For example, hash-indices efficiently support point queries (i.e., queries with an equality predicate), whereas B+-trees support both point and range queries. Furthermore, both hash-indices and B+-trees map each value to one or more positions in the data, whereas other indices such as zonemaps [39] store a synopsis of a data region (e.g. min-max statistics) which allows pruning out the corresponding region during scans. Finally, indices can be clustered when they determine the order of data or unclustered otherwise.

Significant efforts on indexing focus on multidimensional data [32]. Such work includes spatial indices, such as R-tree [42] and k-d tree [9], as well as space-filling curves [80], which enable efficient range queries over multidimensional data. In Chapter 4, we draw a relationship between data access for work sharing and multidimensional data.

When processing a query, a database can use either a full scan or one of the available indices. Access method selection is a fundamental decades-long problem in query optimization [15, 105]. Databases typically make a decision based on the available statistics and the query's predicate. Kester et al. [59] demonstrated that choosing between a scan and an index depends not only on statistics but also on concurrency due to the opportunity for scan sharing. This result already exposes the friction between data and workload-conscious optimizations for individual queries and work sharing.

Partitioning: Partitioning is an alternative to indices for accelerating selective queries. By precomputing a compact set of aggregates [39, 78] per partition, analytical databases enable data skipping [114], a technique that prunes out partitions that contain data that is redundant for the query at hand. To minimize data access, state-of-the-art approaches partition the data by formulating an optimization problem based on the workload's predicates. This problem is proven to be NP-hard, and common solutions involve approximate heuristics [114, 115], and reinforcement learning [45, 130].

In this Thesis: SH_2O addresses efficient data access for highly concurrent workloads. It applies to work-sharing databases that use Data-Query model as a replacement for shared scans that are followed by a chosen set of shared filters. SH_2O outperforms shared scans because it offers both selective access and sharing and amortizes downstream filtering costs. At the same time, SH_2O advances work on shared index probes in terms of optimizing for the Data-Query model, handling predicates on multiple attributes at once regardless of index traversal patterns, and handling dimensionality. Finally, SH_2O is more efficient than data skipping-based approaches for shared data access, as it avoids overfetching data when partition boundaries are misaligned with query predicates and avoids excessive post-filtering.

SH_2O takes a different view of selecting an access strategy compared to access path selection. Instead of choosing the best available access path, it selects which filters to replace based on the batch's access patterns and the data organization. Thus, SH_2O uses an optimizable strategy that goes further at optimizing the access path based on both selectivity and concurrency.

Also, this thesis introduces two partition-oriented execution techniques and two partitioning algorithms. First, we identify the benefit of specializing SH_2O for each data subspace when accessing partitioned data and propose a partition/index selection algorithm that chooses a data organization that minimizes SH_2O's processing time for a target workload. Second, to improve the usability of materialized results in ParCuR, we implement partial reuse through partitioning. Doing so introduces a dependency between materialization footprint and data layout. Hence, we propose a novel partitioning scheme that improves storage budget utilization. In ParCuR, we also use partitioning to eliminate the computation of frequent predicates and reduce the overhead of reuse; however, the insights of SH_2O also apply in this use case.

2.2.3 Materialization and Reuse

When processing a predictable set of queries, databases reduce their processing time and each query's response time using materialization. They precompute and store views and then use each materialized view to answer one or more queries at runtime. Materialization poses an optimization problem that involves three aspects: i) choosing views to materialize, ii) choosing views to reuse, and iii) handling updates. We elaborate on each aspect of the problem and then discuss a subarea related to the thesis, partial materialization and reuse.

Choosing materializations: This aspect requires strategically choosing materializations such that they fit in an allocated storage budget. Thus, the choice needs to decide on a limited number of views that brings the maximum benefit in processing time. The decision constitutes an optimization problem that takes in workload information and results in a materialization decision. The problem of selecting materializations occurs in multiple applications with different constraints. Some common variants that make different assumptions about the explored search space, the workload information, and the timing of the decision are:

- **View selection:** Given a budget of resources and a target workload, view selection [53, 71, 100, 135] finds which set of views to materialize in order to minimize runtime processing time. Candidate views include intermediate results both in each query's chosen plan as well as in alternative equivalent plans.
- **Subexpression selection:** Subexpression selection [51, 52, 136] is a constrained version of view selection that improves optimization time. It restricts the search space by considering materialization candidates only among the intermediate results that occur in each query's chosen plan.
- **Recycling:** Recycling [48, 81, 89, 117] is a variant of materialization selection that chooses and actuates materializations online, as the queries arrive. Thus, recycling lifts the assumption of a-priori knowledge of a target workload.
- **Semantic caching:** Semantic caching [16, 21, 27, 109] focuses on materializing the end-results of queries. This is different from physical caching, which focuses on efficiently using buffer pool pages to cache parts of the base data.

Using materializations: This aspect involves deciding which available materializations minimize each query's cost. The mechanism for making this decision needs to determine i) if it is possible to rewrite the query such that the materialization subsumes its intermediate results and ii) the processing time after applying each rewrite. All in all, the mechanism chooses the rewrite, if any, that minimizes processing time.

Handling updates: Another dimension is maintaining materializations when handling updates. View maintenance introduces techniques that address the freshness-update cost trade-off differently. The taxonomy [64] includes i) recomputation vs incremental view maintenance, ii) immediate vs deferred maintenance, and iii) online vs offline maintenance. Another approach is to invalidate materializations that are affected by updates [48]. This thesis strictly focuses on read-only OLAP workloads, and thus, this aspect is beyond the scope.

Partial materialization and reuse: A set of materialization and reuse techniques focus on increasing reusability for shifting workloads. Partial materialization stores the most impactful [21, 27, 129] or hottest subset of rows [37, 137]. Then, at runtime, reuse exploits overlaps with materialized results and recomputes the rest. As a result, both the hit rate and the utilization of storage are improved.

In this Thesis: Work-sharing databases exploit overlapping work between queries in order to reduce the total cost of processing, but, for every query batch, incur full recomputation from scratch. ParCuR redesigns materialization and reuse for work-sharing databases to eliminate recurring computation more effectively and hence reduce response time.

ParCuR specifically addresses subexpression selection in the context of work-sharing environments; it makes decisions on which subexpressions to materialize and reuse based on which shared operators they eliminate in the global plans of a target workload. Work-sharing affects the impact of reuse, the usability of materialized subexpressions, and the reuse overhead; hence ParCuR adapts the data layout, the materialization policy, and the reuse policy for subexpression selection and introduces efficient reuse strategies. While extending semantic caching, recycling, and view selection for shared execution is beyond the scope of this thesis, the insights for optimizing materialization and reuse for work sharing hold across the different variants of the materialization problem.

ParCuR also adopts partial reuse through partitioning in order to increase the usability of materialized subexpressions during workload shifts. Similar approaches include chunk-based semantic caching [21, 27], partially materialized views [137], partially-stateful dataflow [37], and separable operators [129]. However, in QaT approaches, outstanding concurrent computation results in deteriorating response time as a function of concurrency. ParCuR both reuses available materializations and uses work-sharing to mitigate the impact of concurrency on response time.

2.3 Reinforcement Learning

Reinforcement learning is a paradigm for optimizing decision-making in environments that are modeled as Markov Decision Processes. In this section, we briefly present Markov Decision Processes and Q-learning.

Markov Decision Processes: Reinforcement learning applies to problems that are expressed as Markov Decision Processes (MDPs). An MDP models problems as multi-step processes. At each step, an actor observes the current state s and chooses an action $a \in A(s)$. The action's result is a reward $R(s,a)$ and a change of state to s'. The state space, each state's set of actions, the reward, and state transitions define the MDP. Eventually, the actor observes a terminal state, and the process finishes. Reinforcement learning algorithms find a decision-making policy that maximizes the cumulative reward that the agent observes throughout the process.

Q-learning: Q-learning [127] is a reinforcement learning algorithm for finding optimal policies. Q-learning has two desirable properties: i) it learns the optimal policy instead of evaluating the currently used policy, i.e. a randomized policy that interweaves exploratory decisions with the estimated best decisions. ii) to converge, it only requires that all state-action pairs continue to be visited [116]. A randomized policy guarantees property ii).

Q-learning approximates a function $Q : S \times A \to \mathbb{R}$ that evaluates the quality of decision a at state s, i.e., the expected cumulative reward in future steps. During decision-making, it uses a policy to choose an action in each state. For example, an ϵ-greedy policy chooses, with probability 1-ϵ, the action a that maximizes $Q(s,a)$ and a random action otherwise. Q-learning later uses the observed rewards to refine the approximation of Q. Given two hyper-parameters, learning rate μ and discount rate γ, it updates:

$$Q(s,a) \leftarrow Q(s,a) + \mu(r_t + \gamma \max_{a' \in A(s')} Q(s',a') - Q(s,a))$$

By repeating the MDP's process multiple times, Q-learning iteratively refines the approximation of Q. Q-learning estimates the future cumulative reward for each action hence also the action that maximizes rewards.

In this Thesis: RouLette uses Q-learning with an ϵ-greedy policy to choose efficient global plans. Specifically, it uses the table-based implementation that stores the values of Q in a sparse table instead of using a neural network. By exploiting two problem-specific properties to reduce the state space, RouLette uses an optimized instance of Q-learning for sharing-aware optimization.

3 Scalable and Efficient Sharing-aware Optimization

The growing need to provide real-time insights to a rising number of stakeholders that operate on the same infrastructure significantly increases the analytical query load. Even applications such as interactive analysis result in tens to hundreds of concurrently executing ad-hoc queries [14, 50, 107]. State-of-the-art analytical engines that follow a query-at-a-time execution model are a poor fit for such high query-load scenarios because, as concurrency is increased, query response time suffers. On the contrary, work-sharing databases [12, 35, 44] handle multi-query processing more efficiently by taking advantage of shared data and work across queries to reduce the amount of work to perform. In that case, the benefit depends on the sharing opportunities that the database detects and exploits. Nevertheless, there is no silver bullet for choosing opportunities that minimize processing.

Depending on the mechanism that chooses opportunities, sharing is either online [3, 12, 44] or offline [36]. *Online sharing* detects opportunities, such as common subexpressions, between queries at runtime. Although the detection overhead is low (e.g., matching subplans [35, 44]), online sharing finds only a subset of the opportunities in the workload. Figure 3.1 demonstrates this limitation. To minimize the processing time for each individual query, query optimization produces query plans (1) and (3). The plans share the first join, $R \bowtie S$. However, there exist equivalent plans (2) and (4) with permuted join orders that can share $R \bowtie S \bowtie U$, thus reducing the total processing time. The permutation constitutes a missed opportunity

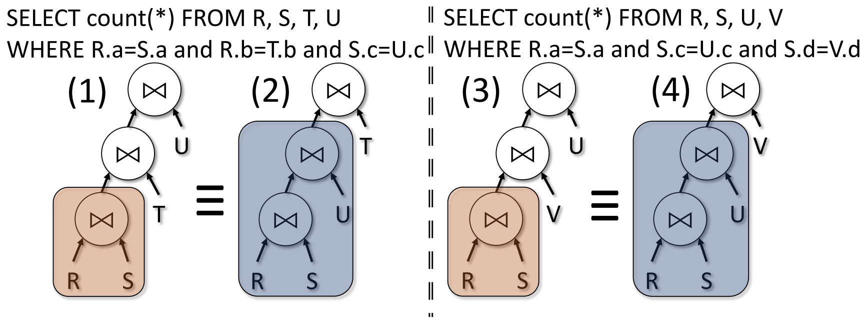

SELECT count(*) FROM R, S, T, U
WHERE R.a=S.a and R.b=T.b and S.c=U.c
SELECT count(*) FROM R, S, U, V
WHERE R.a=S.a and S.c=U.c and S.d=V.d
(1)
(2)
(3)
(4)
R
S
T
U
V
≡

Figure 3.1: Missed sharing opportunities

for the online sharing mechanism that matches subplans. *Offline sharing* optimizes batches of queries by using exhaustive sharing-aware optimization to form the global query plan that minimizes the total processing time. Thus, offline sharing discovers opportunities that online sharing misses. However, exhaustive sharing-aware optimization is a high-complexity problem that takes several seconds to process batches as small as few tens of queries [36]. As it lies in the critical path of execution, it obstructs offline sharing to scale to hundreds of queries, especially in ad-hoc workloads. Therefore, depending on the use or absence of exhaustive sharing-aware optimization, existing systems either forfeit support for highly concurrent workloads or miss substantial sharing opportunities.

We preserve scalability and increase the benefit from opportunities compared to online sharing. On the one hand, scalability requires avoiding the required time for exhaustive sharing-aware optimization being paid in full. This requirement contradicts the *optimize-then-execute* paradigm that most databases adopt. A continuously adaptive paradigm that, by using fast heuristics, reoptimizes queries at runtime is a better fit for highly concurrent workloads as it moves optimization out of the critical path by permitting execution to proceed alongside plan refinement. On the other hand, to increase the benefit of work sharing, the heuristics need to explore alternative candidate global plans which exploit opportunities that online sharing fails to detect. By monitoring execution outcomes, intelligent heuristics can steer exploratory decisions toward efficient global query plans and hence identify global plans that increase benefit.

We present RouLette, a novel intelligent engine that detects and exploits shareable work among Select-Project-Join (SPJ) subqueries through runtime adaptation. RouLette operates in fine-grained episodes. During each episode, it performs work for multiple ongoing queries, monitors the cardinalities of intermediate results, and adjusts the plan by using a learned heuristic. The learned heuristic estimates, by using reinforcement learning, which planning decisions decrease the total processing time, hence steering adaptation toward more efficient plans compared to existing online sharing mechanisms. By continuously adapting the global plan, RouLette reduces work while preserving scalability.

We make the following contributions:

- We present a work-sharing paradigm that both reduces the total processing time and overcomes the scalability limitation of exhaustive sharing-aware optimization. By using adaptive processing, RouLette explores and exploits opportunities at runtime, thus addressing the drawbacks of online and offline sharing.

- Existing heuristics [6, 125] for runtime planning make greedy decisions that result in suboptimal global plans. We design a sharing-aware learned heuristic that approximates, by using reinforcement learning, the decisions that minimize total processing time, and thus it overcomes the limitations of existing heuristics and produces efficient global plans.

- We identify performance bottlenecks inherited by i) adaptive processing, that is, materializing join state and intermediate tuples, and ii) shared operators, that is, filter comparisons and routing. We propose novel optimizations, i.e., symmetric join pruning, adaptive projections, and locality-conscious routers, that improve hardware utilization and scalability.

3.1 Limitations in Sharing-aware Optimization

RouLette addresses two limitations in existing sharing-aware optimization mechanisms: i) the coverage limitation in online sharing, i.e., missed sharing opportunities in alternative global plans, and ii) the accuracy limitation, i.e., choosing suboptimal plans due to inaccurate estimates for a global plan's processing time. We discuss the two limitations in this section and the mechanisms that they affect. The experiments in Section 3.5.1 show that RouLette effectively addresses the presented cases.

3.1.1 Coverage Limitation

Online sharing mechanisms miss sharing opportunities that are absent from their candidate global plans. Thus, mechanisms that explore limited opportunities for reordering operators, such as finding matching subplans, used in QPipe [44], and incremental planning, used in DataPath [3], suffer more from missing sharing opportunities. We demonstrate the coverage limitation through an example. Consider n queries:

```
Q1: SELECT count(*) FROM R,S,T1 WHERE R.a=S.a and R.b=T1.b and T1.x < P1
Q2: SELECT count(*) FROM R,S,T2 WHERE R.a=S.a and R.b=T2.b and T2.x < P2

⋮

Qn: SELECT count(*) FROM R,S,Tn WHERE R.a=S.a and R.b=T1.b and Tn.x < Pn
```

For query Q_i, the QaT plan first processes the join $R \bowtie T_i$ and then the join $(R \bowtie T_i) \bowtie S$. Both QPipe's and DataPath's mechanisms fail to detect any sharing opportunities between the queries. QPipe stitches together the QaT plans and hence completely misses the opportunity to share $R \bowtie S$. DataPath considers alternative ways for incorporating each query Q_i to the global plan of queries $\{Q_1, \ldots, Q_{i-1}\}$, but opts for the QaT operator order. As none of the previous queries processes $R \bowtie S$ first, choosing the QaT operator order minimizes the increase in the global plan's cost. Thus, both mechanisms miss plans that share $R \bowtie S$, which can reduce the total processing time. Furthermore, the example also generalizes for larger shared joins $R \bowtie S_1 \bowtie \ldots \bowtie S_m$, in which case reordering operators brings a higher processing time decrease. By exploring alternative global plans, RouLette identifies operator orders that expose beneficial sharing opportunities and increase work sharing.

3.1.2 Accuracy Limitation

Choosing efficient plans requires making accurate planning decisions that reduce processing time. On the contrary, online sharing mechanisms choose inefficient plans that are significantly more time-consuming when they make suboptimal planning decisions due to i) greedy heuristics that inaccurately capture processing time savings or ii) inaccurate statistics.

The online sharing mechanisms used in TelegraphCQ [69] and CJOIN [12] suffer from inaccurate decisions due to their greedy nature. They both reorder operators based on selectivity; hence, they use selectivity as a proxy for total processing time. However, when data has correlations, choosing the join order based on selectivity can be orders of magnitude more time-consuming [85]. Thus, greedy mechanisms result in inefficient global plans.

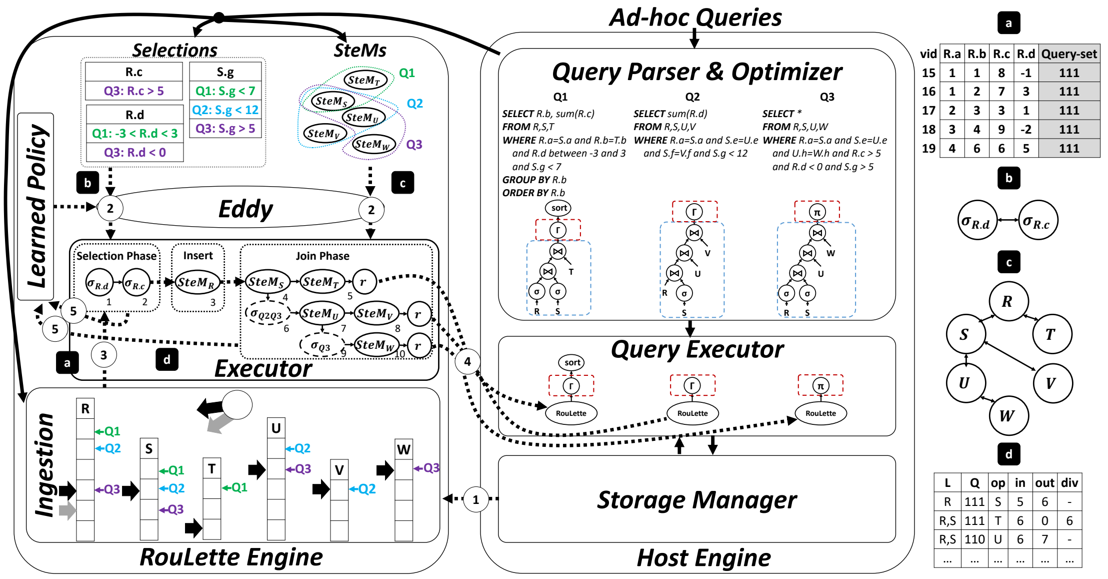
Also, inaccurate cardinality and query overlap estimation results in inefficient plans. The input and output cardinalities of each shared operator range between being identical to the cardinalities of participating queries (i.e., full overlap) and being the sum of the cardinalities across the same queries (i.e., inputs and outputs of the queries are disjoint). Then, for both the cost estimate and the actual processing time of each shared operator, the difference between the extreme cases (i.e., full overlap and no overlap) is high. Hence, making assumptions about query overlaps, such as statistical independence, can prompt planning decisions to miss sharing opportunities that reduce execution time or, worse, to underestimate the cost of time-consuming operator orders. Note that inaccurate estimates also affect offline sharing approaches; however, improving offline estimates using approaches such as LEO [113] is beyond the scope of this work.

RouLette addresses the accuracy limitation by i) estimating the total processing time, and ii) by doing so based on execution outcomes. By using reinforcement learning, RouLette learns which local reordering decisions reduce the cumulative cost for downstream operators. Furthermore, it learns from execution outcomes; thus, its decisions consider both data correlations and the actual overlaps between queries. As a result, RouLette chooses more efficient operator orders.

3.2 RouLette Architecture

We introduce RouLette, a specialized intelligent engine for efficiently executing multiple SPJ subqueries at once. By continuously adapting the global plan to sharing opportunities, it reduces the required time for processing the subqueries. Thus, RouLette optimizes the global operator order, without time-consuming offline sharing-aware optimization.

Figure 3.2 shows RouLette's architecture. RouLette accelerates processing for a host analytical engine. The host processes concurrent queries from different users and applications. It delegates SPJ subqueries to RouLette for shared execution and then collects the results for further processing. RouLette works separately alongside the host because i) it uses a different



```
SELECT R.b, sum(R.c)
FROM R,S,T
WHERE R.a=S.a and R.b=T.b
   and R.d between -3 and 3
   and S.g < 7
GROUP BY R.b
ORDER BY R.b
```

```
SELECT sum(R.d)
FROM R,S,U,V
WHERE R.a=S.a and S.e=U.e
   and S.f=V.f and S.g < 12
```

```
SELECT *
FROM R,S,U,W
WHERE R.a=S.a and S.e=U.e
   and U.h=W.h and R.c > 5
   and R.d < 0 and S.g > 5
```

vid	R.a	R.b	R.c	R.d	Query-set
15	1	1	8	-1	111
16	1	2	7	3	111
17	2	3	3	1	111
18	3	4	9	-2	111
19	4	6	6	5	111

L	Q	op	in	out	div
R	111	S	5	6	-
R,S	111	T	6	0	6
R,S	110	U	6	7	-
...	...	...	...	...	...

Figure 3.2: RouLette operates as a special engine alongside a database. We present the architecture through a three-query example.

processing paradigm (adaptive instead of optimize-then-execute), and ii) it controls its data access, state, and execution. Hence, the interface between the host and RouLette consists of the delegation of subqueries, the collection of results, and data access.

RouLette splits subquery processing into *episodes*. In each episode, RouLette plans-then-processes the operators of ongoing subqueries for an input vector and analyzes execution to refine planning in future episodes. Episodes are the quantum of planning; that is, plans change only across episodes. They process shared work for all ongoing subqueries and map 1-1 to input vectors. Subqueries finish after RouLette processes all their input.

Numbered dotted lines in Figure 3.2 show the data flow between RouLette's components in each episode.

1. *Ingestion* pulls a vector from the host's storage into RouLette.
2. An *eddy* within RouLette chooses the episode's plans for selections and joins using a *learned policy*.
3. The *executor* carries out, by processing the episode's plans for the vector, the eddy's decisions and produces SPJ results.
4. The executor pipelines results to host-side operators (e.g., GROUP BY, outer queries) for further processing.
5. The eddy uses execution metadata to refine the learned policy.

We present RouLette's components using the example of the three queries in Figure 3.2.

Query Optimizer: The optimizer processes incoming queries and produces plans. Then, it delegates one or more SPJ subqueries (blue boxes), which naturally occur at the bottom of plans, to RouLette. In the host, by replacing subqueries with *RouLette sources*, delegation transforms the original plans and assigns the transformed plans to the host's executor. RouLette sources represent intra-RouLette processing and pipeline SPJ results to their consumer (i.e., parent) operator (red boxes). As RouLette does not preserve interesting orders, the optimizer, during transformation, also adds any required operators, such as a *sort*, to the transformed plans.

On RouLette's side, delegated subqueries are dispatched either online or in batches. Dispatch updates the predicate list and the join list (at the top of RouLette in Figure 3.2) and notifies ingestion about new queries. After dispatch, RouLette starts processing the subqueries. In the example, the host delegates Q1, Q2, and Q3 one after the other.

Ingestion: Ingestion provides RouLette with input vectors from the host's storage. It is designed for two desirable properties: i) to ensure that all ongoing queries make progress, and ii) to enable sharing between incoming and ongoing queries when using online dispatch. To satisfy i), it concurrently scans relations that ongoing queries require in round-robin order, whereas to satisfy ii), it uses circular scans [44, 138].

In each episode, ingestion chooses i) a relation to access and ii) the relation's vector to access. Hence, ingestion uses a relation iterator and a vector iterator for each relation. In the example, it chooses R, then R's 4^{th} vector, and finally advances the two iterators. As scans are circular, retrieving the last vector of a relation (e.g., R's 6^{th} vector) will move the iterator back to the start (e.g., R's 1^{st} vector).

At any given moment, each ongoing query accesses data from one or more of the ingestion's scans. As we discuss in Section 3.4.2, RouLette's join processing benefits from accessing relations in a partial order. Ingestion enforces this partial order by attaching each query to a scan only after it has finished all preceding circular scans.

Ingestion also transforms the input into Data-Query model. By recording the position of each scan whenever a new query is attached to it, ingestion keeps track of each scan's active queries, i.e., queries that have not completed the circular scan. To translate the input to Data-Query model, it annotates tuples with the set of active queries. RouLette represents query-sets using bitsets. A set i^{th} bit means that the tuple belongs to Q_i e.g. if $Q1$, $Q2$, and $Q3$ are active, the tuple is annotated with 111. Figure 3.2a shows the resulting vector. When a query's circular scans are all finished, it becomes inactive, and hence ingestion signals the consumer with *end-of-input*.

SteMs: RouLette uses SteMs to enable operator reordering and out-of-order scans. They store and index tuples, making them accessible across episodes without limiting future operator orders. Thus, RouLette implements a history-independent multi-query n-ary symmetric join.

To address performance and parallelization bottlenecks in SteMs, RouLette introduces novel optimizations, i.e., symmetric join pruning, scalable versioning, and adaptive projections, which we discuss in Sections 3.4.1 and 3.4.2.

Eddy and Learned Policy: The eddy handles planning within each episode and adaptation across episodes. It produces global plans that process multiple subqueries for one episode each and analyzes the plans' execution to produce more efficient plans in future episodes. Hence, the eddy uses batching-based policies [26] that produce plans at episode-granularity. Compared to existing adaptation techniques, it produces more efficient plans because it uses novel learned policies that can accurately model global plan costs.

RouLette uses selection push-down. As joins are much more time-consuming, to reduce their input, plans process first the selections and then the joins. Hence, each episode has two separate plans, the selection phase that processes shared selections and the join phase that comprises SteM probes, routing selections, and output routers. Figure 3.2 shows the two plans inside the executor.

To produce each phase's plan, the eddy chooses an operator order using a multi-step optimization algorithm. Multi-step optimization uses learned policies and ordering constraints (Figures 3.2b and 3.2c, presented in Section 3.3.1) to incrementally build the plan from unordered operators. We discuss optimization in Section 3.3.1.

To continuously refine the learned policies used in multi-step optimization, the eddy uses reinforcement learning. By monitoring the input and output of operators, it collects an execution log that records the processed operator and queries, the operator history, and the size of input and output (Figure 3.2d). At the episode's end, to update the policies and improve planning in future episodes, it processes the log using a tailor-made variant of Q-learning that reduces the problem's state space. We discuss learning in Sections 3.3.2 and 3.3.3.

Executor: The executor contains a thread pool of RouLette workers. Each worker concurrently undertakes a different episode and synchronizes with other workers through shared SteMs. It processes the episode's plans for the ingested vector mapped to the episode as follows. First, it processes the selection phase, thus filtering the tuples' query-sets. Second, it inserts the selection phase's results into the base relation's SteM (e.g. $SteM_R$) to make the join symmetric. Third, it processes the join phase for the selection phase's results to produce SPJ results. Fourth, using routers, it sends SPJ results to respective RouLette sources, which pipeline tuples to host operators. Each processed episode contributes to completing the subqueries.

In all steps, RouLette's operators use the Data-Query model and serve one or more queries. RouLette introduces novel algorithms for efficiently processing selections and routing at scale and, to maximize sharing, adopts an MQJoin-like implementation [70] for SteM operations. We defer discussing operator implementation until Section 3.4.

The worker processes the phases using vectorized execution. It processes each operator for an

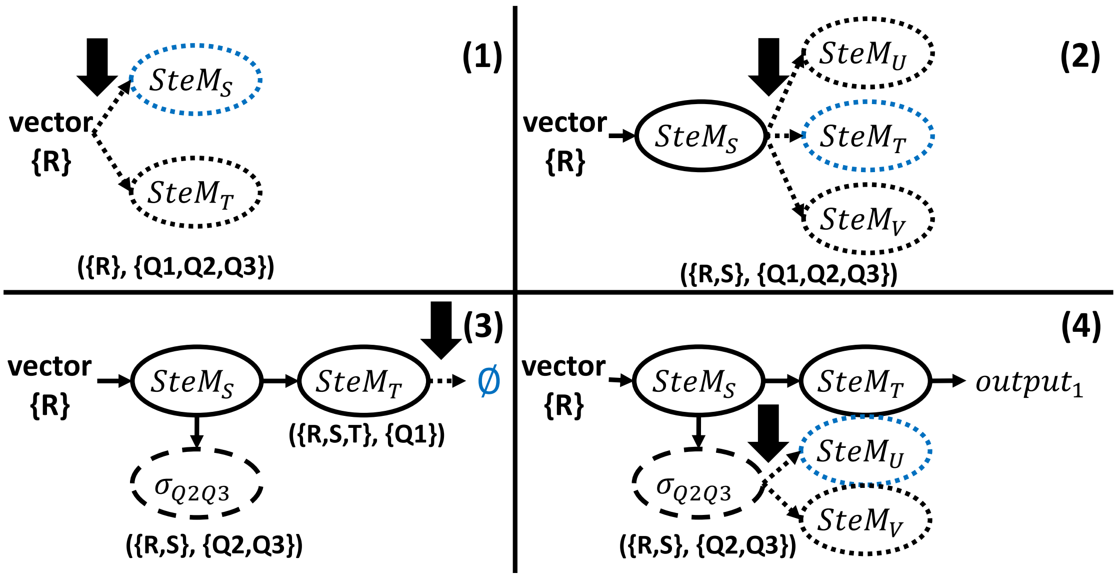


Figure 3.3: Multi-step optimization for join-phase

input vector and sends the output vector to one ($\sigma_{R.d}$'s case) or two operators ($SteM_S$'s case). When one operator follows, the worker executes it next. When two operators follow, to bound the footprint of pending vectors, it executes all operators in the probe subplan first, then all operators in the selection subplan, e.g. after $SteM_S$ probe, the data flow is i) $SteM_S \rightarrow SteM_T$, ii) $SteM_T \rightarrow Q1$'s RouLette source, iii) $SteM_S \rightarrow \sigma_{Q2,Q3}$, e.t.c. In Figure 3.2, numbers next to operators show the execution order. RouLette also implements multi-casting to more than two operators by composing two-output operator patterns one after the other.

Our prototype targets in-memory analytics. It uses columnar data and late materialization. Vectors consist of virtual ID (vID) tuples in PAX-layout [2], and operators reconstruct mini-columns for required attributes on demand. Input vectors contain 1024 tuples, whereas intermediate vectors can contain an arbitrary number of tuples by using a chunked array.

The design reduces the footprint of vectors and SteMs. As both data structures are in-memory, their footprint imposes an upper bound to the dataset size that RouLette can process. This thesis studies sharing in an in-memory environment and, as we discuss in Chapter 6, we leave handling larger-than-memory shared data structures as future work.

3.3 Learned Adaptation Policy

We examine planning in RouLette. We present how a policy produces global plans in Section 3.3.1, express planning as an MDP in Section 3.3.2, and propose a novel learning algorithm in Section 3.3.3.

3.3.1 Policy-based Planning

The eddy optimizes plans using policy decisions. Starting from the plan's input, each decision chooses the operators that process an intermediate vector. One or two chosen operators produce equally many new vectors. Eventually, the plan contains all operators for all queries. The decision sequence is a *multi-step optimization.* It runs at each episode's start and chooses a plan until the episode's end.

In this section, we present multi-step optimization. We follow the algorithm's first four steps for the running example in Figure 3.3.

Terminology

We first define the terms *dependency graph*, *lineage*, *operator query-set*, *virtual vector*, and *candidate operator,*.

Definition 1. The **dependency graph** of a set of operators, $\mathcal{O}$, is defined as the complete graph $K_{|\mathcal{O}|}$ if $\mathcal{O}$ comprises selections and as $G = (V, E)$ with $(e_1, e_2) \in E$ if and only if $e_1 \bowtie e_2 \in O$ if $\mathcal{O}$ comprises joins.

Definition 2. Let $\mathcal{O}$ be a set of operators with dependency graph $G(\mathcal{O})$. A subset $\mathcal{L} \subset \mathcal{O}$ is defined as a **lineage** if and only if the induced subgraph $G(\mathcal{O})[\mathcal{L}]$ is connected. The set of lineages is $\mathcal{L}^*$.

Definition 3. The **query-set** $\mathcal{Q}_o$ of an operator o is defined as the set of queries that contain o.

Definition 4. A **virtual vector** is defined as a pair $(\mathcal{L}, \mathcal{Q})$.

Definition 5. Candidate operators for virtual vector $(\mathcal{L}, \mathcal{Q})$ are defined as

$$cand(\mathcal{L}, \mathcal{Q}) = \{o \in \mathcal{O} - \mathcal{L} | (\{o\} \cup \mathcal{L} \in \mathcal{L}^*) \wedge (\mathcal{Q} \cap \mathcal{Q}_o \neq \emptyset)\}$$

Definition 6. A **policy decision** for virtual vector $(\mathcal{L}, \mathcal{Q})$ is a function $\pi(\mathcal{L}, \mathcal{Q}) = o$ with $o \in cand(\mathcal{L}, \mathcal{Q})$ if $cand(\mathcal{L}, \mathcal{Q}) \neq \emptyset$, and $o = null$ otherwise.

Figures 3.2b and 3.2c show graphs for R's selections and joins. Dependency graphs express ordering constraints between operators: selections can execute in any order, whereas probes often need attributes from other relations to join without cross-products e.g. for $S.e = U.e$, R's tuples need to join with S before joining with U.

Virtual vectors (bold text Figure 3.3) represent shared subexpressions in the global plan. They contain a lineage, i.e., a set of operators that can compose a plan that respects constraints, and a query-set. $\{R, S\}$ is a lineage, while $\{R, U\}$ is a not lineage. Multi-step optimization uses virtual vectors to identify, in incomplete global plans, subexpressions to expand, by adding downstream operators, until they match the subqueries in their query-set.

Candidates are operators that can extend the virtual vector's subexpression. They need to respect constraints and the new subexpression needs to be part of a query. Adding the candidate operator to the lineage results in a new lineage. Dotted outlines (or $\emptyset$) highlight each step's candidates. In step (1), S and T are the only candidates because they are adjacent to R in the graph.

Policy decisions choose one of the candidates (blue outline), if any, as in steps (1), (2), and (4). If there are no candidates, as in step (3), the vector stands for $\mathcal{Q}$'s output and the policy returns *null*. Policy decisions use an eager sharing heuristic; each chosen candidate o processes all queries in $\mathcal{Q} - \mathcal{Q}_o$. The eager sharing heuristic avoids exploring $2^{|\mathcal{Q}-\mathcal{Q}_o|}$ different sharing decisions for each candidate hence reducing the search space. This design decision is consistent with work sharing in adaptive processing, e.g., in TelegraphCQ [69] and similar to the sharing heuristic hint in SWO [36].

In the next two sections, we define the effects of policy decisions on partial global plans.

Sharing

Sharing occurs when all queries of virtual vector $(\mathcal{L}, \mathcal{Q})$ contain the operator o chosen by the policy, as in the example's step (1). The eddy shares o across $\mathcal{Q}$ and a new virtual vector $(\mathcal{L} \cup \{o\}, \mathcal{Q})$ stands for the new shared subexpression.

Divergence

Divergence occurs when only a subset of $\mathcal{Q}$ contains o. Then, the eddy shares o only across that subset, $\mathcal{Q} \cap \mathcal{Q}_o$. Also, it shares a selection across the other queries, $\mathcal{Q} - \mathcal{Q}_o$, to drop redundant tuples. Hence, the decision results in two shared subexpressions with virtual vectors $(\mathcal{L} \cup \{o\}, \mathcal{Q} \cap \mathcal{Q}_o)$ and $(\mathcal{L}, \mathcal{Q} - \mathcal{Q}_o)$. Step (2)'s decision causes divergence. As only $Q1$ contains $R \bowtie T$, the decision creates different subexpressions for $Q1$ and $Q2$-$Q3$. Step (3) shows the resulting virtual vectors. Divergence routes subexpressions to two outputs. To model more than two outputs, the eddy can make decisions that cause divergence consecutively, e.g., choosing V instead of U in step (4).

Multi-step Optimization

To build a complete and correct global plan (i.e., implements delegated subqueries), the eddy composes a sequence of inter-dependent policy decisions. We design multi-step optimization, the eddy's logic, which uses the policy to build the plan operator by operator and to identify the next decisions to make. Multi-step optimization is applied independently for the two phases, selection and join, to produce the two plans.

Algorithm 1 presents pseudocode for multi-step optimization. The algorithm recursively builds the global plan. At each recursive step, starting from the plan's input (lines 11-12), it

Algorithm 1: Multi-step Optimization

```
1  Function MULTI_STEP_REC($node, \mathcal{L}, \mathcal{Q}$):
2      $next$ = NEXT_OPERATOR($\mathcal{L}, \mathcal{Q}$) ;
3      if $next \neq null$ then
4          $main = node.addOperator(next, \mathcal{Q} \cap \mathcal{Q}_{next})$ ;
5          MULTI_STEP_REC($main, \mathcal{L} \cup \{next\}, \mathcal{Q} \cap \mathcal{Q}_{next}$) ;
6          if $\mathcal{Q} - \mathcal{Q}_{next} \neq \emptyset$ then
7              $div = node.addRoutingSelection(\mathcal{Q} - \mathcal{Q}_{next})$ ;
8              MULTI_STEP_REC($div, \mathcal{L}, \mathcal{Q} - \mathcal{Q}_{next}$) ;
9      $node.addRouter(\mathcal{Q})$ ;
10 Function MULTI_STEP($relation, \mathcal{Q}$):
11     $input = InputNode(relation, \mathcal{Q})$ ;
12     MULTI_STEP_REC($input, \{relation\}, \mathcal{Q}$) ;
13     return $input$ ;
```

chooses operators to add after the last operator of a shared subexpression's plan. By using the policy, it first chooses a candidate of the subexpression's virtual vector, o (line 2), and appends it to the plan for $\mathcal{Q} \cap \mathcal{Q}_o$ (line 4). Also, in case of divergence, it appends a selection for $\mathcal{Q} - \mathcal{Q}_o$ (line 7). The new operators' output corresponds to new subexpressions. Multi-step optimization uses recursion to complete the downstream plans of new subexpressions for $\mathcal{Q} \cap \mathcal{Q}_o$ (line 5) and $\mathcal{Q} - \mathcal{Q}_o$ (line 8). Finally, *null* decisions indicate that the subexpression is its query-set's output, and a router to the host is added (line 9) When recursion finishes, the plan is complete. Next, we discuss Algorithm 1's correctness, complexity, and optimality.

Correctness: Due to the ordering constraints, Algorithm 1 only produces subexpressions that serve at least one query. We also prove that it produces the output of each query exactly once.

Theorem 1. MULTI_STEP_REC ($node, \mathcal{L}, \mathcal{Q}$) produces a *null* decision, hence the query's output, for each $q \in \mathcal{Q}$ exactly once.

Proof. Let $\mathcal{O}_\mathcal{Q} = \{o \in \mathcal{O} | \mathcal{Q}_o \cap \mathcal{Q} \neq \emptyset\}$. We use induction on $|\mathcal{O}_\mathcal{Q} - \mathcal{L}|$.

Base step: As $\mathcal{L} \subset \mathcal{O}_\mathcal{Q}$, $|\mathcal{O}_\mathcal{Q} - \mathcal{L}| = 0$ entails $\mathcal{L} = \mathcal{O}_\mathcal{Q}$. Then, $cand(\mathcal{L}, \mathcal{Q}) = \emptyset$ hence the policy decides null once for each $q \in \mathcal{Q}$.

Induction step: If the proposition holds for $|\mathcal{O}_\mathcal{Q} - \mathcal{L}| \leq n$, it also holds for $|\mathcal{O}_\mathcal{Q} - \mathcal{L}| = n+1$.

Let $o = \pi(\mathcal{L}, \mathcal{Q})$. $o \in \mathcal{O}_{\mathcal{Q} \cap \mathcal{Q}_o}$ and $\mathcal{O}_\mathcal{Q} \subset \mathcal{O}_{\mathcal{Q} \cap \mathcal{Q}_o}$ hence $|\mathcal{O}_{\mathcal{Q} \cap \mathcal{Q}_o} - (\mathcal{L} \cup \{o\})| \leq n$. Recursion for $(\mathcal{L} \cup \{o\}, \mathcal{Q} \cap \mathcal{Q}_o)$ decides null exactly once for each $q \in (\mathcal{Q} \cap \mathcal{Q}_o)$.

If there is divergence, $o \notin \mathcal{O}_{\mathcal{Q}-\mathcal{Q}_o}$ hence $|\mathcal{O}_{\mathcal{Q}-\mathcal{Q}_o}| < \mathcal{O}_\mathcal{Q}$ and $|\mathcal{O}_{\mathcal{Q}-\mathcal{Q}_o} - \mathcal{L}| \leq n$. Recursion for $(\mathcal{L}, \mathcal{Q} - \mathcal{Q}_o)$ decides null exactly once for each $q \in (\mathcal{Q} - \mathcal{Q}_o)$. The two recursions produce null for $(\mathcal{Q} - \mathcal{Q}_o) \cup (\mathcal{Q} \cap \mathcal{Q}_o) = \mathcal{Q}$ and their query-sets do not overlap.

□

Complexity: The number of decisions is the global plan's size, which has at most $\mathcal{Q}_o$ instances of each operator o. Each decision inspects the candidates which are at most $|\mathcal{O}|$. Hence, the worst-case complexity of Algorithm 1 is $O(|\mathcal{O}| * \sum_{o\in\mathcal{O}} |\mathcal{Q}_o|)$. Algorithm 1 is invoked once for every episode.

Optimality: Algorithm 1 is optimal for the search space that it explores if and only if, at each step, the policy chooses the candidate that leads to the best possible downstream plan given the decisions already made. Given an accurate estimate of the best possible downstream plan's processing time for each candidate, it suffices to choose the candidate with the minimum estimate. The next section focuses on estimation.

3.3.2 Learning Policy Decisions

The response time of global plans depends on decision quality. To improve decision quality, the eddy adapts the policy using the execution log. RouLette's reinforcement learning-based adaptation approximates the policy that minimizes response time. In this section, we present i) the requirements for accurately estimating the runtime of global plans thus approximating optimality, and ii) a reinforcement learning formulation that satisfies the requirements.

Cost estimation: The eddy optimizes the global plan's response time. However, it can observe only intermediate cardinalities in each episode's plans. It estimates time from cardinalities using a cost model. We refer to the estimate as cost. The cost model computes operator a's cost as a function of input and output sizes, $c_a(n_{in}, n_{out})$. The total cost is the sum of all operator costs in a plan.

Requirements: Policies minimize the total cost, which includes the cost of downstream operators later in the plan. Hence, they need to estimate the long-term effects of decisions, which are caused by the cascading effect of operator selectivity across the plan. For example, in Figure 3.2's join-phase, the input size for probing $SteM_T$ is 6, whereas, if $SteM_T$ had been probed before $SteM_S$, the input size would have been 5. With 20% larger input, the probe's cost is likely to be higher.

Another long-term effect of decisions is on data distribution due to attribute and join-crossing correlations. Data distribution affects operator selectivity. Assume that in Figure 3.2's example, only the first 60% of R's vector has matches in $SteM_S$, and only the last 40% has matches in $SteM_T$. Join selectivity is 120% for $R \bowtie S$, 60% for $R \bowtie T$, and 0% for both $(R \bowtie S) \bowtie T$ and $(R \bowtie T) \bowtie T$. Selectivity depends on the predicates of all queries that share the operator's input subexpression. The query-set, which is part of the virtual vector, summarizes information about predicate satisfaction.

Also, the eddy optimizes tree-shaped global plans and hence policies affect costs across multiple branches. Long-term cost estimation counts shared operators once for their whole query-set and aggregates cascading costs across all branches, e.g., long-term costs for probing S in Figure 3.2 include the cost of probing S, U, V, T and W, and the cost of routing selections.

Then, the policy can choose candidates that minimize the global cost by exploiting work-sharing, even when they are suboptimal for individual queries.

Thus, to accurately estimate the best candidate, the policy predicts cascading cardinalities and correlations across all branches. Existing selectivity-based approaches fail the requirements and, as experiments show in Section 3.5.2, produce suboptimal and expensive plans. RouLette's policy satisfies all three requirements, by using reinforcement learning on the following MDP.

Formulation: We model multi-step optimization as an MDP. The eddy is an agent that composes plans by choosing one of the candidates at each step. In the following paragraphs, we define the four components of an MDP that optimizes the global plan.

States: States contain the information required to model multi-step optimization. Decisions process states to choose the best candidate.

To express actions, transitions, and rewards, the MDP requires the virtual vector and the input size of the current recursive step. The virtual vector determines candidates and the recursive steps that follow. The input size determines the output size given the chosen operator's selectivity, which the virtual vector also affects, and both determine cost estimation when computing rewards. The input size and the virtual vector form an extended vector $(n, \mathcal{L}, \mathcal{Q})$. Later, we show that the formulation can omit input size.

To express recursion, the MDP models all pending recursive steps as a stack of extended vectors, with the current step at the top. The state is the stack and the state space is the set of stacks with elements from $\mathbb{R} \times \mathcal{L}^* \times 2^{\mathcal{Q}}$. Our notation represents a stack as $top : tail$ and an empty stack as ϵ. In Figure 3.3, the state is $(5, \{R\}, \{Q1, Q2, Q3\}) : \epsilon$ for step 1 and $(0, \{R, S, T\}, \{Q1\}) : (5, \{R, S\}, \{Q2, Q3\}) : \epsilon$ for step 3.

Actions: An action chooses a candidate for the current vector i.e. the top of the state's stack. Hence, a state's actions are:

$$A((n, \mathcal{L}, \mathcal{Q}) : s_{tail}) = A((n, \mathcal{L}, \mathcal{Q}) : \epsilon) = cand(\mathcal{L}, \mathcal{Q})$$

Transitions: Choosing a candidate invokes one or two recursive steps, changing the state. Transitions replace the top of the stack with vectors for the new recursive step(s). For example, $(5, \{R\}, \{Q1, Q2, Q3\}) : \epsilon$ transitions to $(0, \{R, S, T\}, \{Q1\}) : (5, \{R, S\}, \{Q2, Q3\}) : \epsilon$. Operators affect the sizes of new vectors. To express output sizes, we use conditional selectivity $p_{\mathcal{L},\mathcal{Q}}(o)$, which models the output-to-input ratio for operator o and subexpression results with virtual vector $(\mathcal{L}, \mathcal{Q})$. Candidate o's output is $p_{\mathcal{L},\mathcal{Q}}(o) * n$, whereas the routing selection's is $p_{\mathcal{L},\mathcal{Q}}(\sigma_{\mathcal{Q}-\mathcal{Q}_o}) * n$, if any. Sharing pushes one new vector and transitions from $(n, \mathcal{L}, \mathcal{Q}) : s_{tail}$ to:

$$(p_{\mathcal{L},\mathcal{Q}}(o) * n, \{o\} \cup \mathcal{L}, \mathcal{Q}) : s_{tail}$$

Divergence pushes two vectors and transitions to:

$$(p_{\mathcal{L},\mathcal{Q}}(o) * n, \{o\} \cup \mathcal{L}, \mathcal{Q} \cap \mathcal{Q}_o) : ((p_{\mathcal{L},\mathcal{Q}}(\sigma_{\mathcal{Q}-\mathcal{Q}_o}) * n, \mathcal{L}, \mathcal{Q} - \mathcal{Q}_o) : s_{tail})$$

If there are no more candidates, a *null* action pops the top of the stack, and the state transitions to s_{tail}.

Rewards: An action's reward represents the operator's cost. As reinforcement learning maximizes rewards, operators incur negative rewards. Using the cost model, the reward when sharing is:

$$R((n, \mathcal{L}, \mathcal{Q}) : s_{tail}, o) = -c_o(n, p_{\mathcal{L},\mathcal{Q}}(o) * n)$$

Divergence also includes the selection's cost, hence the reward is:

$$R((n, \mathcal{L}, \mathcal{Q}) : s_{tail}, o) = -c_o(n, p_{\mathcal{L},\mathcal{Q}}(o) * n) - c_{\sigma_{\mathcal{Q}-\mathcal{Q}_o}}(n, p_{\mathcal{L},\mathcal{Q}}(\sigma_{\mathcal{Q}-\mathcal{Q}_o}) * n)$$

3.3.3 Specialized Q-learning Implementation

The formulation satisfies the requirements for modeling the cost of global plans but is difficult to use in practice. The state space is large due to the input-size parameter and the stack representation. In this section, we present the design and implementation of a specialized Q-learning that, by exploiting two properties of cumulative rewards, independence and proportionality, reduces the state space.

Independence: Vectors in the stack have disjoint query-sets and hence incur downstream costs independently across different branches of the plan. The cumulative cost of the state is the sum of cumulative costs for each vector in the stack. To minimize cumulative cost, the eddy separately minimizes the cost of each vector e.g. in step 3, to optimize $(0, \{R, S, T\}, \{Q1\})$: $(5, \{R, S\}, \{Q2, Q3\}) : \epsilon$, it optimizes $(0, \{R, S, T\}, \{Q1\}) : \epsilon$ and $(5, \{R, S\}, \{Q2, Q3\}) : \epsilon$. Also, each vector's downstream costs include only the cumulative costs of vectors created by the corresponding step's decision and recursion. Hence, other pending vectors in the stack do not affect cost. We rewrite decisions and update rules to use only popped and pushed vectors, thus hiding the stack's tail.

Proportionality: Intuitively, operator cost is linear to input size i.e. doubling the input size will roughly double the required computations. Hence, we define cost as a linear function:

$$c_a(n_{in}, n_{out}) = \kappa_a * n_{in} + \lambda_a * n_{out}$$

By definition, $n_{out} = p_{\mathcal{L},\mathcal{Q}}(op) * n_{in}$, so the output size, the cost, and hence a vector's cumulative cost is linear to input size. Then, all decisions and updates can be reduced to singleton states $(1, \mathcal{L}, \mathcal{Q}) : \epsilon$. Normalizing the Q-values of candidates by input size results in the same decisions e.g. the optimal decision for $(5, \{R\}, \{Q1, Q2, Q3\}) : \epsilon$ is the same as for $(1, \{R\}, \{Q1, Q2, Q3\})$: ϵ. Also, to express downstream costs, updates scale Q-values by operator selectivity e.g. for probing $SteM_S$, the downstream cost is $1.2 * Q((1, \{R, S\}, \{Q1, Q2, Q3\}) : \epsilon)$.

By exploiting independence and proportionality, Q-learning interacts only with $(1, \mathcal{L}, \mathcal{Q}) : \epsilon$ states, or simply $(\mathcal{L}, \mathcal{Q})$. We next discuss the implementation and integration of the algorithm.

Algorithm 2: Policy implementation

```
1  Function NEXT_OPERATOR($\mathcal{L}, \mathcal{Q}$):
2      $cand = cand(\mathcal{L}, \mathcal{Q})$ ;
3      if $choose - random() == true$ then
4          return $random(cand)$ ;
5  return $argmax_{a \in cand}\{Q(\mathcal{L}, \mathcal{Q}, a)\}$ ;
6  Function UPDATE($\mathcal{L}, \mathcal{Q}, o, n_{in}, n_{out}, n_{div}$):
7      $r = 0$ ;
8      $q = max\{Q(\mathcal{L} \cup \{o\}, \mathcal{Q} \cap \mathcal{Q}_o, a) \mid a \in cand(\mathcal{L} \cup \{o\}, \mathcal{Q} \cap \mathcal{Q}_o)\}$ ;
9      $r = r + (-\kappa_o * n_{in} - \lambda_o * n_{out} + \gamma * n_{out} * q) / n_{in}$ ;
10     if $n_{div} \neq null$ then
11         $q = max\{Q(\mathcal{L}, \mathcal{Q} - \mathcal{Q}_o, a) \mid a \in cand(\mathcal{L}, \mathcal{Q} - \mathcal{Q}_o)\}$ ;
12         $r = r + (-\kappa_\sigma * n_{in} - \lambda_\sigma * n_{div} + \gamma * n_{div} * q) / n_{in}$ ;
13     $Q(\mathcal{L}, \mathcal{Q}, o) = (1 - \mu) * Q(\mathcal{L}, \mathcal{Q}, o) + \mu * r$ ;
14 return output ;
```

Q-table: Q-learning learns $Q((\mathcal{L}, \mathcal{Q}), o)$, which is the best-case cumulative cost at $(\mathcal{L}, \mathcal{Q})$ if the policy decides o. The algorithm needs a method for inferring and updating $Q((\mathcal{L}, \mathcal{Q}), o)$. We use traditional map-based Q-learning. Deep learning is unsuitable for adaptive processing, as training and inference are prohibitively expensive.

Map-based Q-learning stores the current $Q((\mathcal{L}, \mathcal{Q}), o)$ estimates in a hash map indexed by $(\mathcal{L}, \mathcal{Q}), o)$ triplets. As both $\mathcal{L}$ and $\mathcal{Q}$ are sets with small domains, we store them as bitsets. Then, concatenating the bytes of $\mathcal{L}$, $\mathcal{Q}$, and o forms a unique key for each state. Decisions and update rules use the unique triplets to access the map.

To encourage exploration in early episodes and exploitation in later episodes, we use optimistic initialization [116]. As rewards and Q-table values are negative, we initialize values to zero. Moreover, the triplet space is only partially explored because some triplets are invalid while others correspond to pruned parts of the search space. Hence, the Q-table is sparse. We set the map to store only non-zero values and return 0 for failed lookups.

Decisions: Decisions choose one of the candidates. Algorithm 2's *NEXT_OPERATOR* presents decision-making. As $-Q((\mathcal{L}, \mathcal{Q}), o)$ is the expected cumulative cost, deterministic decisions choose the candidate with the maximum Q-value (line 4). This requires one Q-table access per candidate. Sporadically, with probability ϵ, decisions choose at random to guarantee eventual convergence (line 3).

Updates: By monitoring execution, the eddy generates a log entry for each processed operator o in the following format:

$$(\mathcal{L}, \mathcal{Q}, o, n_{in}, n_{out}, n_{div})$$

n_{in}, n_{out}, n_{div} stand for the size of input, o's output and $\sigma_{\mathcal{Q}-\mathcal{Q}_o}$'s output , if any (otherwise *null*). By invoking the update rule for each entry, the eddy adapts the policy.

Algorithm 2's *UPDATE* presents the update rule. The update rule propagates cumulative costs from operators that were added by recursion at $(\mathcal{L}, \mathcal{Q})$. Due to independence, it estimates cumulative rewards for each branch separately (lines 8-9 for $\mathcal{Q} \cap \mathcal{Q}_o$, lines 11-12 for $\mathcal{Q} - \mathcal{Q}_o$). Estimation for $\mathcal{Q} \cap \mathcal{Q}_o$ works as follows: Line 8 estimates, by comparing all $Q((\mathcal{L} \cup \{o\}, \mathcal{Q} \cap \mathcal{Q}_o), a), a \in cand(\mathcal{L} \cup \{o\}, \mathcal{Q} \cap \mathcal{Q}_o)$, the best cumulative cost q that recursion can create at $(\mathcal{L} \cup \{o\}, \mathcal{Q} \cap \mathcal{Q}_o)$. Line 9 adds the cost of o to the estimate in three steps: i) it multiplies q by n_{out} to undo normalization. ii) it adds the direct costs of o, and iii) it normalizes the estimate again by n_{in}. The same estimation method is applied for $\mathcal{Q} - \mathcal{Q}_o$. r aggregates the total estimate. Thus, Q-learning bootstraps from the current Q-value estimates for state $(\mathcal{L} \cup \{o\}, \mathcal{Q} \cap \mathcal{Q}_o)$ and, if required, state $(\mathcal{L}, \mathcal{Q} - \mathcal{Q}_o)$. In the end, the Q-table value is updated to a weighted average of its previous value and the total estimate. After several episodes, Q-learning approximates $Q((\mathcal{L}, \mathcal{Q}), o)$.

Tuning: Q-learning depends on three hyper-parameters that represent different trade-offs: lowering μ trades off learning speed for smoothing noise due to local data distribution, lowering ϵ trades off exploration for Q-table exploitation, and lowering γ reduces the relative weight of downstream rewards. As downstream rewards are equally important, we set $\gamma = 1$. We tune μ and ϵ by using grid search.

We also tune the cost model to emulate execution time. We assume that all operators of the same type, e.g. all joins, have the same κ and λ. To tune the parameters, for each operator type, we measure execution time in nanoseconds for various input and output sizes and apply linear regression to estimate κ and λ. We get: (i) for selections $\kappa = 9.32$ and $\lambda = 4.62$, (ii) for routing selections $\kappa = 3.60$ and $\lambda = 0.92$, and (iii) for joins $\kappa = 38.57$ and $\lambda = 43.29$.

3.4 Adaptive Multi-query Executor

On top of learned policies, RouLette owes its performance to efficient shared operators and low-overhead adaptation. In this section, we describe i) the implementation of shared operators, and ii) optimizations that mitigate adaptive processing's bottlenecks.

3.4.1 Efficient Shared Operators

RouLette's selection and join phases comprise shared operators. In this section, we present the operators' design and follow the running example's selection phase in Figure 3.4 and join phase in Figure 3.5.

Selections: Each selection-phase operator filters the query-sets of its input tuples by evaluating one or more predicates. For each input tuple, it computes a predicate result bitset, shown below Figure 3.4's operators – a set i^{th} bit means that Q_i's predicates in the selection are

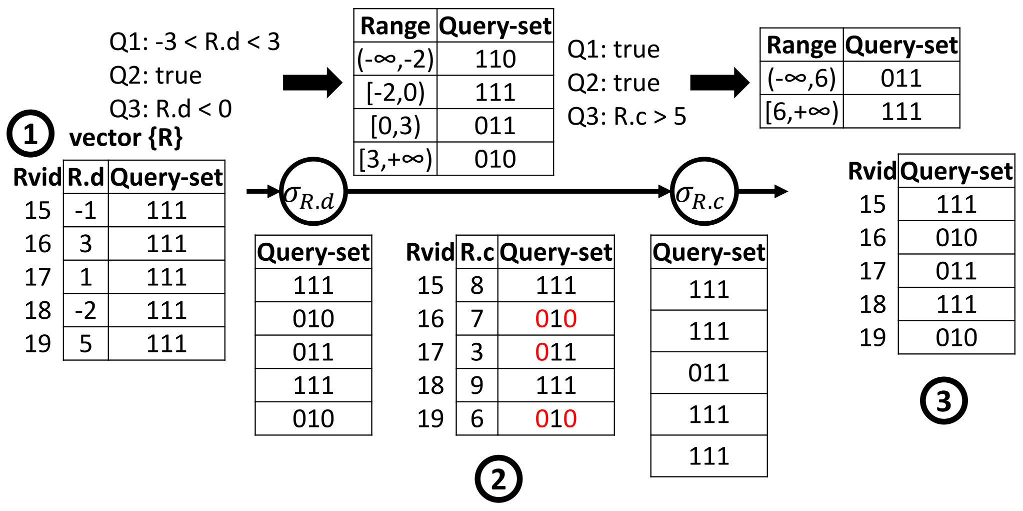


Figure 3.4: Execution of selection-phase

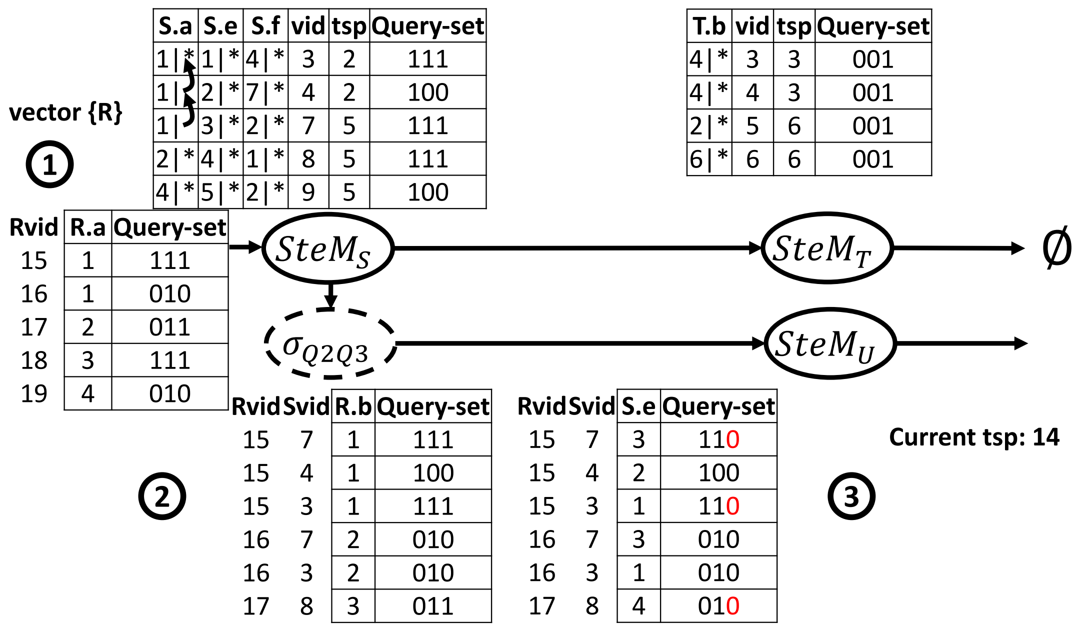
satisfied. Filtering removes queries with zero bits from the tuple's query-set by computing the bitwise AND of the bitsets. The new bitset, which stands for query-set intersection, is the output's query-set. Selection drops tuples with empty query-sets.

To reduce shared selection costs, RouLette batches predicate evaluation on each attribute using grouped filters, e.g., $\sigma_{R.d}$ evaluates $Q1$'s and $Q3$'s predicates (and $true$ for $Q2$) at once. Prior work [69] prunes comparisons by indexing predicates using structures such as search trees. However, index-based implementations are inefficient because comparisons are still linear to the satisfied predicates and, hence, in the worst case to the total number of queries.

RouLette uses an efficient evaluation method, whose cost is logarithmic to the number of predicates. The method is similar to the indexed predicate evaluation of Marroquin et al. [76] but uses binary search instead of hard-coded control flow statements. For each selection, it constructs a lookup table that stores precomputed predicate results for a set of ranges that partition the filter attribute's domain. Figure 3.4 depicts the lookup tables above their respective operators. The lookup table is the predicate index for the selection. For each input tuple, predicate evaluation performs a binary search over the lookup table using the tuple's attribute value and retrieves the query-set for the matching range.

SteMs: RouLette uses a shared SteM for each relation across all queries and joins. The SteM stores selection-phase result tuples and, on each join key, builds indices for joins, e.g., hash-index for equijoins. To reduce footprint, we use unified SteM entries:

(index-vector, vID, timestamp, query-set)



S.a	S.e	S.f	vid	tsp	Query-set
1|*	1|*	4|*	3	2	111
1|	2|*	7|*	4	2	100
1|	3|*	2|*	7	5	111
2|*	4|*	1|*	8	5	111
4|*	5|*	2|*	9	5	100

T.b	vid	tsp	Query-set
4|*	3	3	001
4|*	4	3	001
2|*	5	6	001
6|*	6	6	001

Rvid	R.a	Query-set
15	1	111
16	1	010
17	2	011
18	3	111
19	4	010

Rvid	Svid	R.b	Query-set
15	7	1	111
15	4	1	100
15	3	1	111
16	7	2	010
16	3	2	010
17	8	3	011

Rvid	Svid	S.e	Query-set
15	7	3	110
15	4	2	100
15	3	1	110
16	7	3	010
16	3	1	010
17	8	4	010

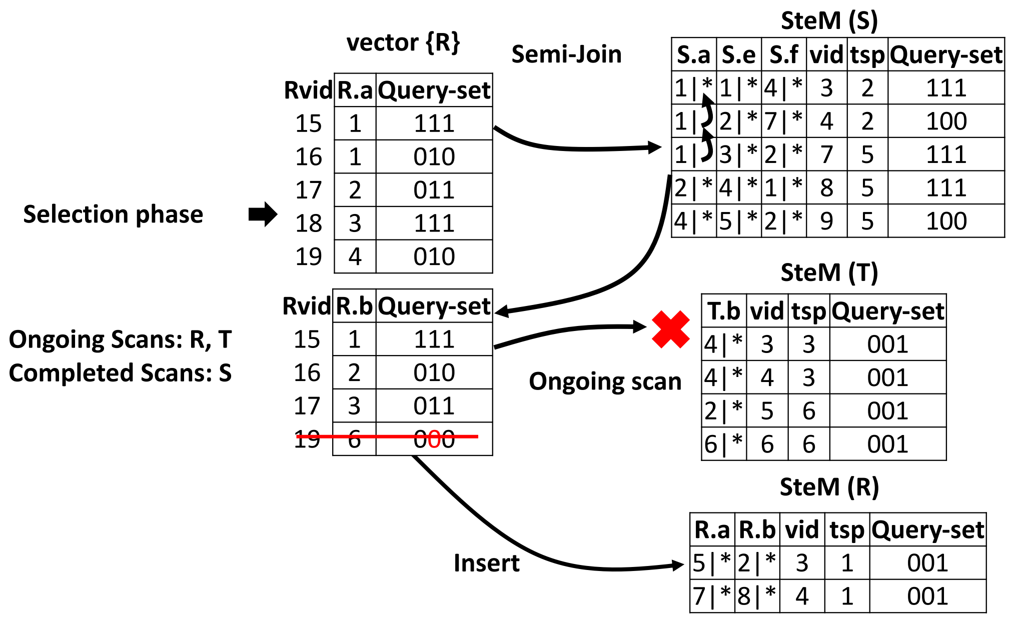
Figure 3.5: Execution of join-phase

SteMs stores entries as a contiguous cache-aligned memory block. The inserted tuple consists of *vID* and *query-set*. Each SteM's index uses one element of *index-vector* to build a self-referential data structure, e.g., a list-based hash bucket for hash-indices. The *index-vector* also stores the join key to avoid late materialization for SteM tuples' attributes. Finally, SteM uses *timestamp* to ensure insert-probe atomicity. Figure 3.5 shows SteMs above each probe. In our example, S has equijoins on $S.a$, $S.e$, and $S.f$ and builds hash-indices. Arrows are pointers to tuples with the same key and * is the end of each list.

Probes search the SteM for matching tuples, inserted in previous episodes, and produce concatenated probe-match pairs. Vector 2 in Figure 3.5 is the result of probing $SteM_S$ for R's vector. Probes use SteM indices to efficiently find matches. Then, they compare timestamps to enforce insert-probe atomicity – only matches with older timestamps are considered. Finally, they compute query-sets of probe-match pairs by intersecting the query-sets of the probing and the probed tuples, i.e., bitwise AND of the bitsets, and discard pairs with empty query-sets, such as R's tuple 19 with S's tuple 9.

Routing selections: Selections in the join-phase permit tuples of specified queries to pass, e.g., in Figure 3.5, it retains tuples from $Q2$ and $Q3$. A bitwise AND with a filter mask clears other queries from the bitset. Such selections reduce downstream processing.

Router: Routers send shared output to the host, by multicasting tuples to their query-set's RouLette sources. To increase output locality and reduce cache and TLB misses, they adapt the design of two-pass partitioning to multicasting. Hence, routers increase cache hits hence improving their processing rate.



Rvid	R.a	Query-set
15	1	111
16	1	010
17	2	011
18	3	111
19	4	010

Rvid	R.b	Query-set
15	1	111
16	2	010
17	3	011
~~19~~	~~6~~	~~000~~

S.a	S.e	S.f	vid	tsp	Query-set
1|*	1|*	4|*	3	2	111
1|	2|*	7|*	4	2	100
1|	3|*	2|*	7	5	111
2|*	4|*	1|*	8	5	111
4|*	5|*	2|*	9	5	100

T.b	vid	tsp	Query-set
4|*	3	3	001
4|*	4	3	001
2|*	5	6	001
6|*	6	6	001

R.a	R.b	vid	tsp	Query-set
5|*	2|*	3	1	001
7|*	8|*	4	1	001

Figure 3.6: Symmetric join pruning

3.4.2 Optimizations for Adaptive Processing

Adaptive processing suffers from overhead due to SteM materialization and versioning, and lack of projectivity. To match optimize-then-execute performance, RouLette uses novel optimizations for adaptive processing.

Symmetric Join Pruning: Symmetric joins require that all relations be materialized and hence incur materialization overhead. To reduce the overhead, RouLette materializes only tuples that can form output tuples for their query-set. We call this *symmetric join pruning*. Figure 3.6 shows pruning for the symmetric join of Figure 3.2. In the example's episode, the symmetric join processes R's vector. Tuple 18 has no matching entry in S. As all queries contain $R \bowtie S$ and the $SteM_S$ is final, pruning infers that 18 cannot form any output tuple and drops 18 before insertion. Also, it infers that 19 cannot form output tuples for Q1 and Q2, hence it adjusts the query-set. As the new query-set is empty, pruning drops 19. Pruning cannot use $SteM_T$, because T's scan is ongoing; future inserts can yield matches for 15-17. To drop tuples and modify query-sets, pruning uses semi-joins with fully-ingested joinable SteMs. RouLette integrates semi-joins into the selection phase as filters.

Pruning emulates filtering in non-left deep plans that use join results as inner relations. SteMs store semi-join results hence probes and semi-joins with pruned SteMs return even fewer matches. Filtering propagates across the plan, beyond direct joins, and SteMs store the results of semi-join trees. Still, symmetric joins require extra probes to construct results. Caching intermediate results [7] eliminates extra probes and is complementary to RouLette.

As pruning requires fully-ingested relations, to increase pruning opportunities, RouLette controls the order in which ingestion initiates circular scans. It chooses the order based on three insights: i) Small relations that are on the build-side in all joins should be ingested first. ii) Ingesting large relations should be postponed, as they are the targets of pruning. iii) M:N semi-joins are avoided, as they are expensive. The insights apply to common schemas that use dimension tables (e.g., star, snowflake, snowstorm).

To choose the order based on the above insights, RouLette ranks relations using a heuristic. The heuristic, starting from rank 1, works as follows: i) it marks unranked relations that are smaller than all other joinable unranked relations. ii) it assigns the current rank to marked relations and increments the current rank. iii) it adjusts cardinality estimates, based on pruning, and repeats the steps. Ranking produces a partial order of scans for each dispatched batch. Except for having its left-most relation fixed, choosing the join order is orthogonal to the scan order.

Scalable versioning: RouLette parallelizes episode execution. Critical sections, such as ingestion and policy updates, are rare, and hence are lock-based. The main point of contention is SteMs.

To reduce contention and scale up, RouLette's SteMs use wait-free indexing and batch versioning. First, wait-free indices use atomics and hence reduce insert/probe contention. Second, batch versioning reduces contention on the timestamp counter, as it requires only two atomics per vector. For batch versioning, SteMs use both local and global versions. Inserts use the same SteM-local timestamp for each vector's tuples. Then, they map the SteM-local timestamp, by default globally invalid, to a global timestamp. To check atomicity conditions, each probe translates SteM-local to global timestamps before doing the timestamp comparison.

Adaptive projections: As adaptive processing lacks projections, probe results grow increasingly wide hence materializing intermediate vectors becomes more expensive. To drop redundant columns and reduce materialization, RouLette introduces adaptive projections. By identifying columns used by downstream operators in the episode's plans, it keeps a minimal set of vIDs and sheds the rest.

3.5 Experimental Evaluation

We evaluate a prototype of RouLette. The experiments show

i) RouLette's ability to reduce the response time for high concurrency,

ii) RouLette's performance gains over online sharing and QaT databases,

iii) the benefit of learned policies over selectivity-based policies,

iv) the impact of timing dependencies on work sharing and learning,

v) the sensitivity of learning rate to workload characteristics,

vi) the effect of executor optimizations, and

vii) RouLette's scalability in multi-core CPUs.

Data & Workload: The experiments use data from TPC-DS [91] (scale factor 10, 8.65 GB in-memory) and Join Order Benchmark, or JOB, [67] (1.79 GB in-memory). To evaluate RouLette for a wide range of workloads, we generate a pool of thousands of queries on the TPC-DS schema with different joins and predicates. To assess learned policies, we use JOB as it uses real data that violates assumptions that oversimplify optimization. JOB comprises 113 SPJ queries with 3-16 joins.

To generate the required workloads for TPC-DS data, we implement a query generator that takes the conditions as parameters. The query generator uses a two-step process:

1. it chooses a subgraph of the schema as a join graph. It does not join fact tables from different channels [82]; in the actual TPC-DS queries, such joins occur only in query 78.
2. it produces predicates to match a target selectivity. To precisely control selectivity, we extend each TPC-DS table with a uniformly distributed column with values from 0 to 999 and produce *BETWEEN* predicates.

Hardware: The experiments run on a two-socket server with 12-core Intel Xeon E5-2650L v3 CPUs running at 1.8 GHz and 256 GB of DRAM. The server uses Ubuntu 18.04 LTS and GCC 7.4.0. RouLette affinitizes threads and memory to one NUMA node. With the exception of Section 3.5.4, experiments use one worker. In all experiments, reported numbers are the average of five runs.

Tuning: All experiments use the same Q-learning hyper-parameters. To tune μ and γ, we use grid search to minimize the total response time for five batches of 64 JOB queries. We get $\mu = 0.21$ and $\epsilon = 0.014$.

We assess RouLette's ability to optimize any workload it is given. Thus, we use workload-agnostic scheduling. By sampling queries, we produce batches that RouLette then processes.

3.5.1 Response Time Evaluation

This section shows RouLette's performance improvement. We compare RouLette against two QaT databases, a vectorized database (DBMS-V) and MonetDB. We also compare against two online sharing techniques, Stitch& Share and Match& Share. Stitch& Share composes global plans by sharing common subtrees between individual plans produced by PostgreSQL. It represents the online sharing mechanism in QPipe [44], SharedDB [35]. Match& Share adds each query to the global plan with minimum additional cost. It represents the online sharing mechanism in DataPath [3]. To execute global plans in a common shared engine, we implement a prototype that uses the batched execution model [35] and adopts all useful optimizations and operators from RouLette. Furthermore, we compare against Greedy, which uses RouLette's executor with the selectivity-based policy of CACQ [69] and CJOIN [12]. Finally, we compare against SWO-based offline sharing [36]. As offline sharing cannot scale to batches with numerous distinct queries, we only use it with a small batch.

To assess RouLette, the performance comparison unfolds in three steps: i) micro-benchmarks which demonstrate that RouLette overcomes the coverage and accuracy limitations, ii) a sensitivity analysis spanning large diverse multi-query workloads, iii) JOB workloads.

Work-sharing databases execute each workload's queries as one batch, whereas QaT databases, with the exception of the experiment in Figure 3.19, execute queries one after the other. The compared metric is batch response time, i.e., the total processing time for each batch.

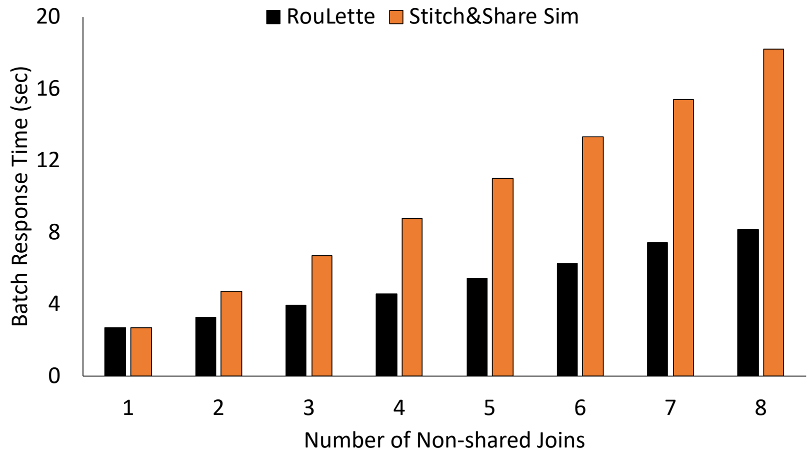
Micro-benchmarks: The experiments examine RouLette's ability to overcome online sharing's coverage and accuracy limitations. We use TPC-DS data and generate tables for M:N joins by repeating the $date\_dim$ relation multiple times. Each micro-benchmark uses a different setup.

Coverage: The first micro-benchmark evaluates RouLette's ability to detect and exploit sharing opportunities by reordering operators. We generate 8 queries $store\_sales \bowtie date\_dim \bowtie item_i$, where $item_i$ is a different instance of the item table. Hence, $store\_sales \bowtie item_i$ cannot be shared across queries. Each query has a predicate with 60% selectivity on $item_i$ hence Stitch&Share (and similarly Match&Share) fail to detect any sharing opportunities. However, when processing more than one query in the same batch, sharing $store\_sales \bowtie date\_dim$ improves performance. In this experiment, we simulate Stitch&Share by forcing RouLette to process $store\_sales \bowtie item_i$ operators first.

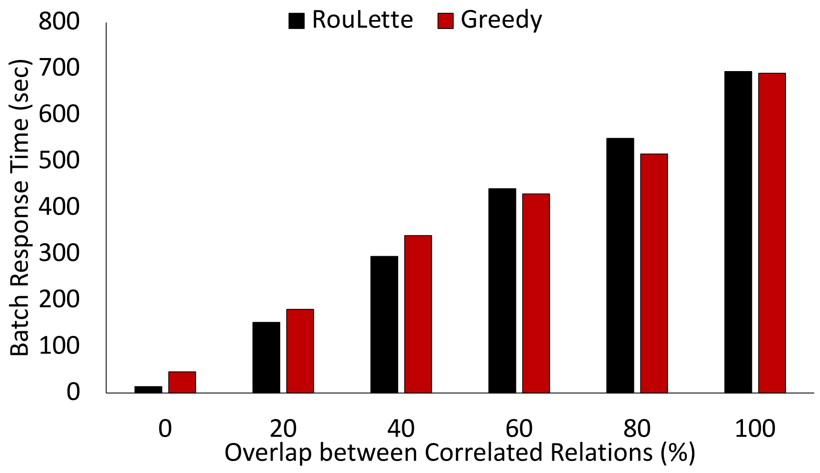
Figure 3.7a shows that RouLette outperforms Stitch&Share, as it detects and exploits the sharing opportunity. The experiment varies the batch size from 1 to 8. As the number of queries is increased, RouLette's speedup is also increased, reaching 2.23 for 8 queries.

Accuracy (vs Greedy): The second micro-benchmark demonstrates RouLette's ability to learn the operator orders that minimize downstream processing. We generate 3 relations by repeating $date\_dim$, d_1, d_2 and d_3. d_1 repeats $date\_dim$ 3 times, whereas d_2 and d_3 repeat it 8 times. We use 1 query in this experiment, as the results also hold for QaT execution. d_2 and d_3 have a predicate on their join keys with 50% selectivity. Through the overlap of the filter ranges, we control their join-crossing correlation. 0% overlap means that $store\_sales \bowtie d_2 \bowtie d_3$ has no output tuples, whereas 100% increases cardinality by 4x.

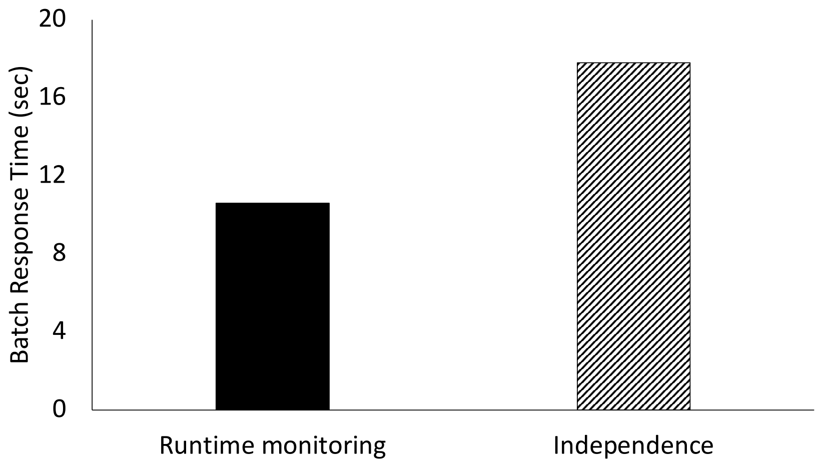
Figure 3.7b shows that RouLette identifies the optimal plan for each correlation setting, whereas Greedy fails to do so. For up to 40% overlap, processing $store\_sales \bowtie d_2 \bowtie d_3$ first improves performance. However, Greedy chooses $store\_sales \bowtie d_1$ because its selectivity is lower. As a result, achieves up to 3.3 speedup and similar performance when the correlation is high; we observe a minor performance penalty due to exploration.



(a)



(b)



(c)

Figure 3.7: Micro-benchmarks: a) coverage limitation, b) accuracy limitation, c) adaptivity

Adaptivity: The third micro-benchmark shows the benefit of adaptive processing. We generate 2 batches of 8 $store\_sales \bowtie d_1 \bowtie d_2$ queries. Each query has a predicate with 5% selectivity on d_2. In the first batch, the predicates are independent, whereas, in the second batch, the predicate is the same across all queries. Thus, for the first batch, the optimal global plan processes $store\_sales \bowtie d_1$ first, whereas, for the second batch, the opposite is true. We compare 2 cases: i) running the second batch as it, and ii) running the second batch using the learned policy from the first batch (without runtime updates).

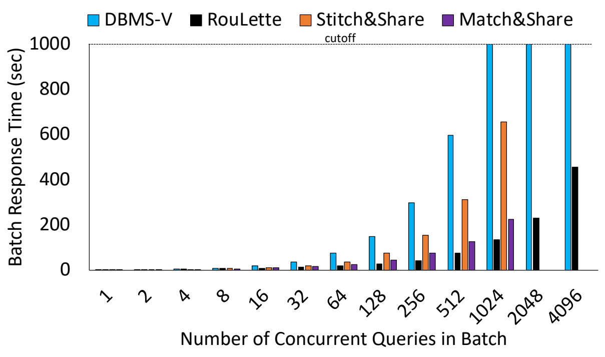
Figure 3.7c shows that using the policy for a similar batch with different correlations significantly degrades performance. Response time is increased by 1.68x. This result motivates the adaptive paradigm that learns the policy for each query batch at runtime during the batch's execution.

Sensitivity Analysis: The experiment examines RouLette's batch execution performance under varying workload conditions. The varying conditions are the batch size, the selectivity (i.e., the percentage of fact table tuples that occur in each query's result set) and the number of joins of individual queries and the schema type.

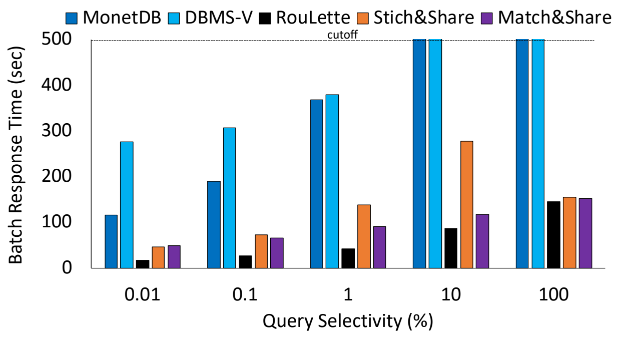
Figures 3.8a-d show the response time for the workloads in the sensitivity analysis. In each Figure, three parameters are constant, and the fourth varies. The default values are 10% selectivity, 4 joins, Store snowflake subschema (which is similar to Star Schema and TPC-H), and 512 queries. Each query has a predicate on 3 relations, chosen randomly. The predicates have unequal selectivity. The query generator produces 4096 queries per configuration and forms batches by sampling the queries without replacement.

Varying concurrency: Figure 3.8a shows that RouLette's speedup scales with increasing batch sizes. The available memory restricts the maximum batch size for each system. Shared approaches improve their speedup as a function of the batch size because sharing opportunities are increased. RouLette's response time grows more slowly as it discovers more opportunities and, despite adaptation overhead, becomes faster than Stitch& Share and Match& Share after 16 and 32-query batches, respectively. RouLette's maximum speedup is 10.70 over DBMS-V, the faster of the two databases, whereas online sharing's maximum speedup is 3.65. Thereupon, RouLette's speedup hits a plateau when query-set operations dominate execution time. As the cost of query-set operations grows linearly to batch size, the plateau is a bottleneck of the Data-Query model. Exceeding the plateau for larger batches requires optimizations that reduce the cost of query-set operations.

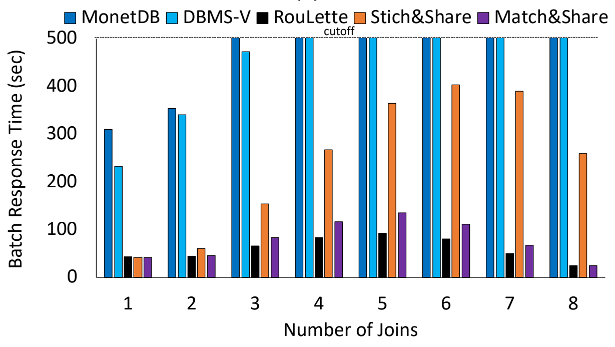
Varying selectivity: Figure 3.8b shows that RouLette achieves lower response time for all selectivities. As expected, response time is increased as a function of selectivity. In this experiment, RouLette exploits more opportunities compared to online sharing approaches; Stitch& Share misses opportunities as predicates produce different plans for the same join set, whereas for low selectivity Match& Share fails to exploit subexpressions in the existing global plan when planning each query. For queries without filters (100% selectivity), their opportunity detection limitation is lifted. By contrast, RouLette chooses the global plan for



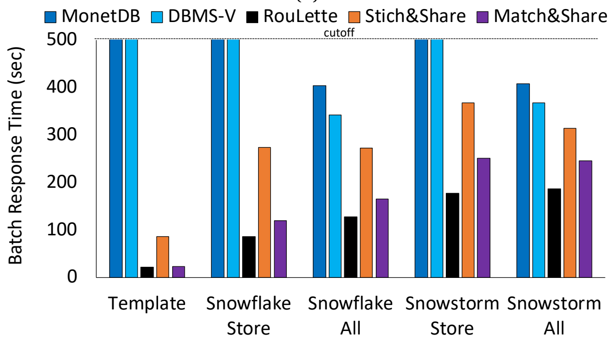
(a)



(b)



(c)



(d)

Figure 3.8: Sensitivity analysis for varying a) concurrency, b) selectivity, c) number of joins, d) query shapes

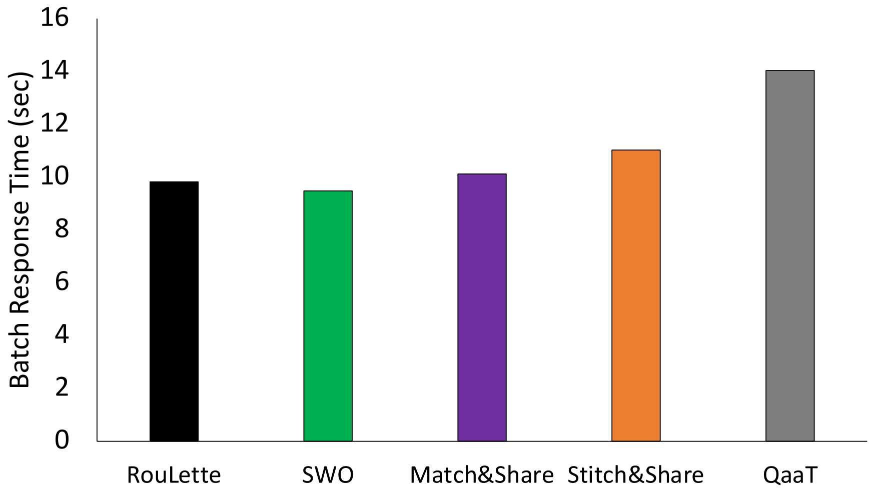


Figure 3.9: Comparison with offline sharing for small batch

all queries at once and thus increases benefits from work sharing. Out of the QaT databases, MonetDB performs better for low selectivity but suffers from intermediate materializations for higher selectivity, unlike DBMS-V.

Varying number of joins: Figure 3.8c shows that work sharing is sensitive to the diversity of joins. The number of distinct joins is maximum for 4-5 joins, whereas including few or almost all joins increases homogeneity (e.g., all 8-join queries have the same join set). Each system's response time depends on whether work sharing can offset increasing join processing costs. RouLette's response time reflects the effect of homogeneity, increasing until 4-5 joins and then decreasing. It outperforms online sharing when heterogeneity is high because it reorders joins to discover opportunities between diverse joins and also benefits as homogeneity is again increased. However, increasing homogeneity benefits Match & Share as well, as it can also reorder joins to a smaller extent. Match & Share retakes the lead for 8-join queries, as it avoids adaptation overhead.

Varying the schema: Figure 3.8d shows that RouLette works best for homogeneous workloads but is still effective for diverse queries. The five workloads comprise queries whose set of joins are:

i) $store\_sales \bowtie date\_dim \bowtie hdemo \bowtie item \bowtie customer$ (*template*)

ii) a subgraph of Store snowflake subschema (*snowflake-store*)

iii) a subgraph of any channel's snowflake subschema (*snowflake-all*)

iv) a subgraph of Store snowstorm subschema (*snowstorm-store*)

v) a subgraph of any channel's snowstorm subschema (*snowstorm-all*)

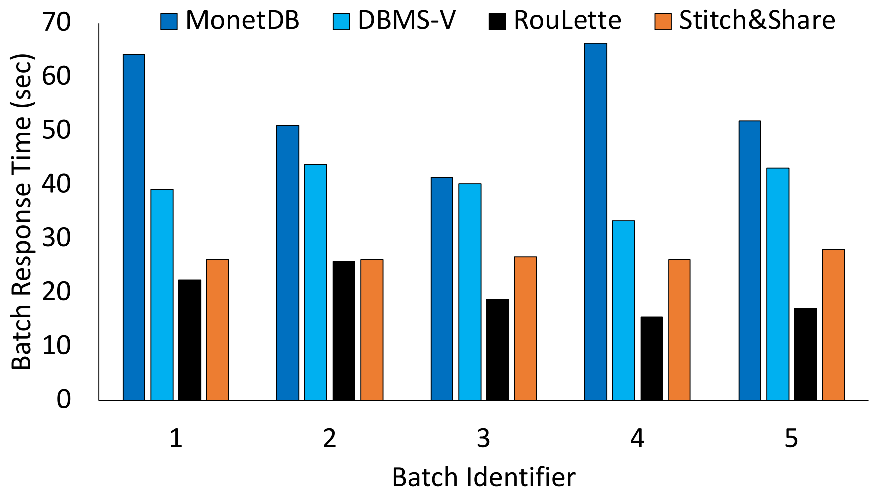


Figure 3.10: Batches of large queries with correlated data (JOB)

RouLette's speedup decreases with higher diversity of joins, as queries have little work in common. For *snowstorm-all*, the most diverse case, RouLette overtakes DBMS-V after 32-query batches. Eventually, batch size compensates for diversity. RouLette decreases response time by 3.39x for snowstorm-store and by 1.96x for snowstorm-all, and it outperforms online sharing, despite materializing fact tables.

Small Offline-optimized Batches: Figure 3.9 shows that RouLette chooses near-optimal plans, unlike online sharing, which chooses suboptimal plans. SWO optimizes a batch of 11 queries (4 joins, snowflake-store, 50% selectivity), taking 137 seconds. The chosen batch is the largest SWO could optimize with a one-hour timeout (with the given characteristics). RouLette's response time is only 4% higher than SWO's. By contrast, Match&Share's response time is 7% higher, and Stitch&Share's is 20% higher. Small benefits are expected due to the small batch size. As the batch size is increased, the gap with online sharing widens.

Large Queries with Correlated Data: Figure 3.10 shows that RouLette improves performance, even for queries with many joins and data correlations that are challenging for optimizers. The workload comprises 64-query batches produced by sampling JOB. RouLette outperforms both QaT databases and online sharing. Speedup echoes the results of *snowstorm-all.* The experiment excludes Match& Share, as its custom optimizer supports only uniform data, for which it can estimate the intermediate result overlap between different queries.

Takeaway: RouLette outperforms QaT databases in all cases and online sharing when the optimal plan is non-trivial. In workloads with diverse join sets and schemas, it exposes more opportunities that online sharing misses. Still, as expected, diversity reduces opportunities for all sharing techniques. Finally, RouLette improves scalability, as it increases speedup with increasing query counts until Data-Query model dominates execution.

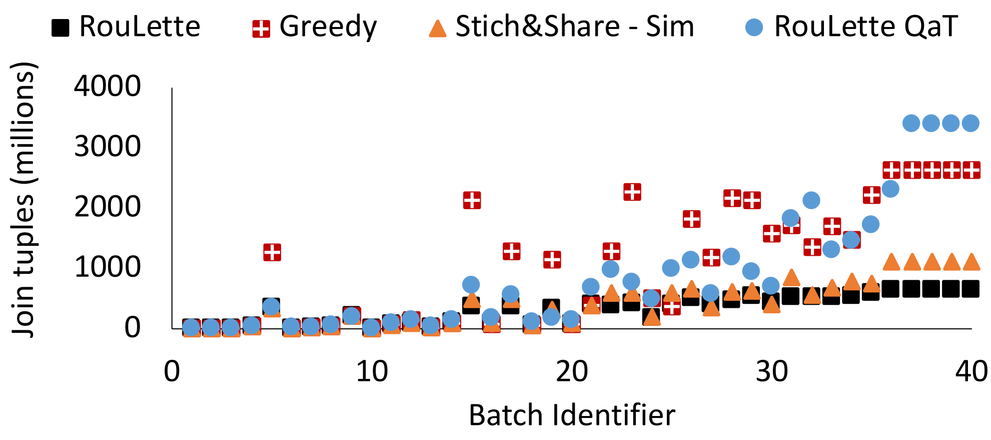


Figure 3.11: Evaluation of learned policy for batches of JOB queries

3.5.2 Quality of Learned Planning

The experiment evaluates the ability of policies to exploit opportunities in both static workloads, processed in batches, and dynamic workloads, with runtime query admissions. It focuses on stand-alone policies. To evaluate policies, it measures the number of intermediate tuples in joins, which is an implementation-independent metric for cost. To compute the metric, we add up the log's output vector sizes. As joins dominate execution time, we exclude selections.

Static Opportunities: The experiment compares the behavior of different policies. RouLette processes workloads in batches. It schedules all queries at once hence fully sharing all scans and common intermediate tuples. We generate 5 batches for each size among 1, 2, 4, 8, 16, 32, 64, and 113 by sampling JOB queries without replacement.

Figure 3.11 shows the cost for varying batch sizes. Each batch corresponds to a sequence number based on the size. Size 1 maps to the range $[1,5]$, size 2 to $[6,10]$, $\ldots$, size 113 to $[36,40]$. Batches 36-40 are identical to each other. The figure includes 4 different configurations. *RouLette* is the learned policy. *Greedy* is the selectivity-based policy from CACQ and CJOIN. By choosing plans independently for each query using the learned policy and then sharing common subexpressions, *Stitch&Share - Sim* simulates Stitch&Share. Finally, *RouLette QaT* is the cumulative cost of executing queries one after the other. RouLette reduces the cost in all cases compared to RouLette QaT.

Learning vs Selectivity: The results show that learned policies choose superior plans. The *Greedy* policy incurs comparable cost to the learned policy for small batches but suffers from high-cost outliers. Outliers include ≈ 7% of single JOB queries. As the batch size is increased, optimization hazards are increasingly likely to occur hence penalizing most batches. For 64-query batches, the learned policy produces 3.24x fewer intermediate tuples on average.

Learning scope: Results also show that a global learned policy (*RouLette*) outperforms a query-

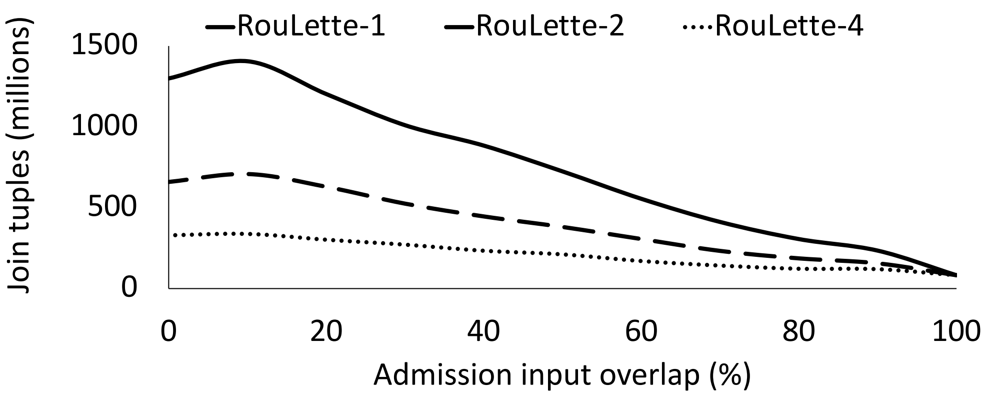


Figure 3.12: Evaluation of learned policy for dynamic query admission

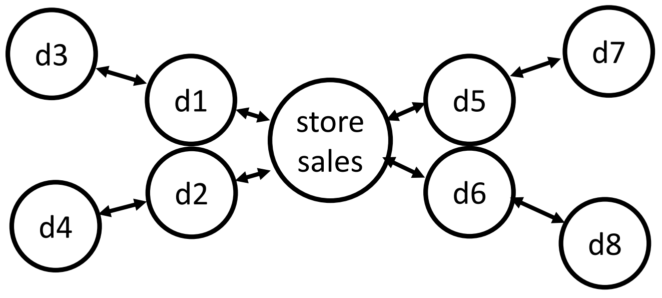


Figure 3.13: Synthetic schema for 4 chains and 9 relations in total

local policy (*Stitch&Share – Sim*), as it considers work sharing during planning and is preferable to individual decisions. While the cost is similar up to 8-query batches, *RouLette* produces $1.71x$ fewer tuples for 113 queries.

Dynamic Opportunities: Figure 3.12 shows the interplay between work sharing and learning. RouLette admits instances of JOB query 17a one at a time or in batches. We measure the percentage of overlapping input between back-to-back admissions: 0% is query-at-a-time execution, whereas 100% is single batch execution. The intermediate tuples are decreased as a function of the overlap. For small overlaps, the cost is increased because work sharing does not compensate for restarting learning. An overlap of 10% increases cost by 8% for single-query admissions. Batching reduces the cost of processing, as it reduces interference and guarantees opportunities. Admitting four-query batches for every 40% of the input produces $1.4x$ fewer tuples compared to admitting single queries for every 10%.

Learning Rate: The next experiment evaluates the ability of the policy to learn plans for workloads with varying complexity. To vary complexity, we modify TPC-DS data. Queries join $store\_sales$ with chains of synthetic relations. At each step, the policy considers one candidate per chain. We generate synthetic relations by sampling $date\_dim$ with replacement

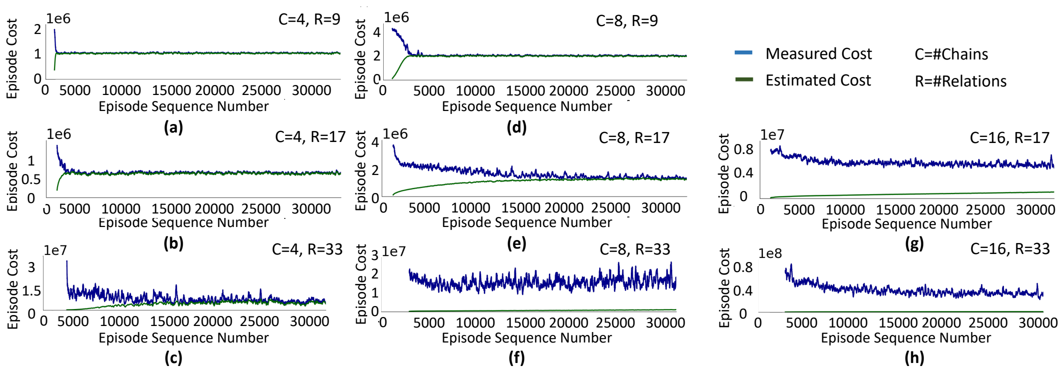


Figure 3.14: (a)-(h) Episode cost comparison: measured vs estimated

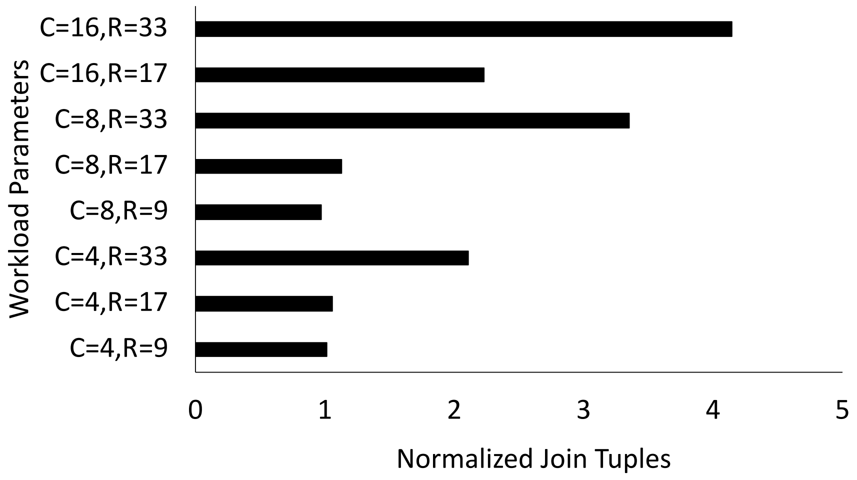


Figure 3.15: Comparison of learned and selectivity-based policies for synthetic schema

at varying rates. To ensure a large difference between each query's best and worst plan, rates are 0.4-0.6 for half of the chains and 1.7-2.5 for the other half. Each query spans half of the join graph and includes an equal number of high and low-selectivity joins. We generate workloads with a varying number of chains and relations. Figure 3.13 shows the schema for workload "Chains=4,Relations=9".

Figures 3.14a-3.14h show the policy's convergence across the episode sequence for 64-query batches from various workloads. Plots include each workload's parameters. For each episode, it plots a rolling average for the measured cost and the policy's estimate of the minimum cost over the last 100 episodes. As execution progresses and future costs are propagated, the policy's estimate is increased, and the measured cost is decreased; when they converge, the policy is optimal. The experiment shows that convergence is slower when the state space is broader (more candidates) and deeper (join size). Figures 3.14a-3.14c show that, when candidates are few, the policy converges fast even for large joins. By contrast, Figures 3.14d-3.14h show that, when candidates are many, the policy converges only for small joins.

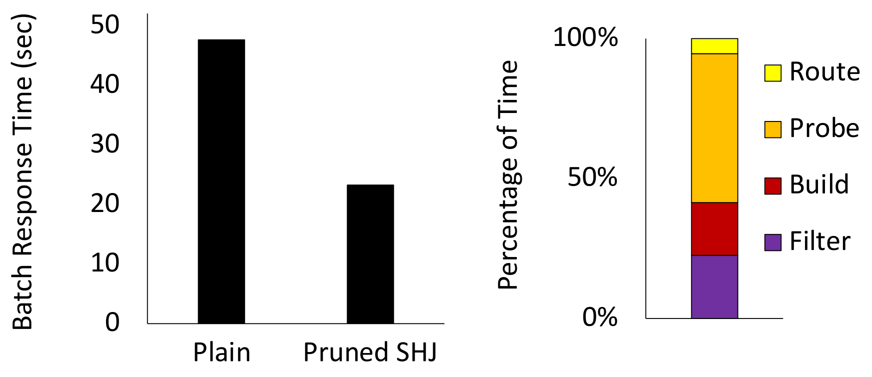


Figure 3.16: Impact of optimizations and profiling for batch of JOB queries

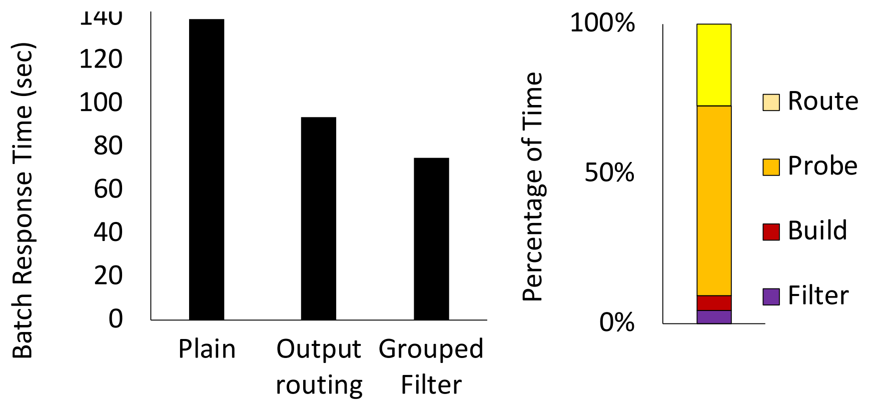


Figure 3.17: Impact of optimizations and profiling for batch of sensitivity analysis queries

Figure 3.15 plots for the same experiment the intermediate join tuple ratio for RouLette over the Greedy policy. As joins have no data correlations, Greedy is near-optimal. The comparison shows that when convergence is slow, learned policies suffer from exploratory decisions. To mitigate the effects of slow convergence and to choose plans for large schemas, heuristics need to be used instead.

Takeaway: Sharing-aware learned policies substantially improve adaptive processing. They produce fewer tuples compared to selectivity-based policies and to sharing-oblivious learned policies. Learned policies permit query admissions at runtime but suffer from interference when the overlap is low. Batching admissions reduces interference and increases work sharing. Finally, learned policies typically converge within a few thousand episodes, but suffer from slow convergence for workloads with large queries on large schemas.

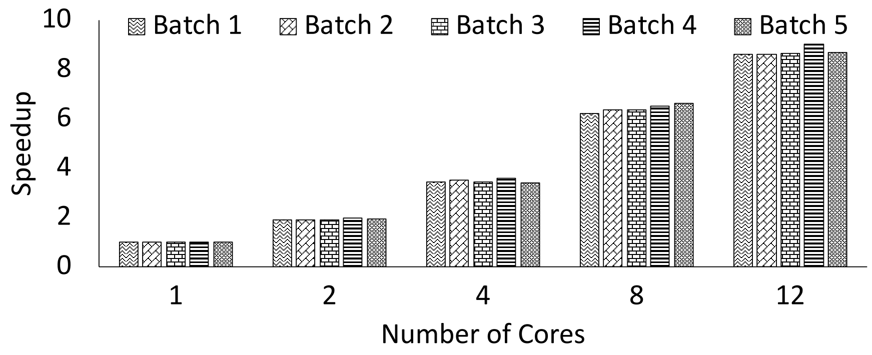


Figure 3.18: Scalability to the number of cores

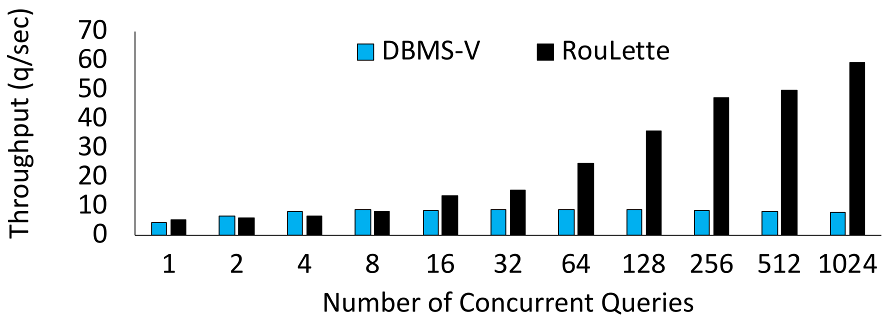


Figure 3.19: Comparison with concurrent query execution

3.5.3 Effect of Optimizations

Figures 3.16 and 3.17 show the benefit from individual optimizations, applied incrementally, and the breakdown of the execution time after applying the optimizations. The experiment analyzes two batches, a 64-query batch of JOB queries and a 512-query batch of generated queries (default parameters). Joins dominate execution for both batches. For the JOB batch, pruning is the most important optimization, giving 2.05 speedup. For the synthetic batch, RouLette's novel router and grouped filter algorithms are the most important optimizations: together they result in 1.85 speedup. Note that the queries only use one filter attribute: increasing the number of attributes also increase the cost of filtering.

3.5.4 Multi-core Execution

In this section, we evaluate RouLete's performance when using multi-core CPUs. RouLette scales up in one NUMA socket.

Figure 3.18 shows RouLette's speedup as the number of workers is increased from 1 to 12. The experiment uses the five batches of 64 JOB queries from Figure 3.10. Speedup is increased monotonically for all batches and reaches 8.63-9.04 (71.9-75.3% efficiency).

Figure 3.19 compares RouLette's throughput to DBMS-V's for concurrent execution. We measure throughput as the number of processed queries over the end-to-end time. The experiment processes 10240 queries in total from Figure 3.8a's workload. We vary the degree of concurrency from 1 to 1024. To do this, we submit queries to DBMS-V from 1 to 1024 clients concurrently and emulate concurrent execution in RouLette by submitting a sequence of batches that contain one query per client. The experiment's results show that DBMS-V's throughput suffers due to inter-query interference, whereas RouLette's benefits. When using one client, DBMS-V uses data parallelism. For more clients, it shares resources across queries. Clients run isolated in the remote NUMA node. Concurrent execution initially improves DBMS-V's throughput up to $2.06x$. However, after 64 clients, DBMS-V's throughput is gradually decreased due to interference. DBMS-V runs out of memory after 1024 queries. By contrast, RouLette uses all cores for processing query batches. It shares work across all concurrent queries hence RouLette's speedup over DBMS-V is increased as a function of concurrency.

3.6 Summary

We have presented RouLette, an adaptive multi-query multi-way join operator that tackles the limitations of online and offline sharing. Rather than follow an optimize-then-execute approach, RouLette uses runtime adaptation to move sharing-aware optimization out of the critical path, restoring scalability. It progressively explores sharing opportunities using a heuristic based on reinforcement learning. RouLette also proposes optimizations that reduce the adaptation overhead. The experiments show that RouLette scales to hundreds of complex queries, unlike offline sharing, and improves throughput compared to query-at-a-time and online sharing systems. Hence, it makes inroads on the long-standing problem of building scalable high-throughput analytical systems.

4 Efficient Shared Data-Access

When supporting interactive applications, analytical databases need to sustain highly concurrent workloads with stringent response time constraints. Then, the performance for concurrent data accesses is critical for providing real-time responses, especially for selective workloads. Consider the example of dashboards and reporting in Meta [107] and Youtube [14]. Both data services require processing hundreds of queries concurrently, with each query processing a small fraction of a large volume of data, and need to keep latency in the order of tens of milliseconds. In such applications, inefficient time-consuming data access jeopardizes latency requirements.

Data access techniques perform differently based on both selectivity and concurrency. When access patterns are selective, indices help avoid costly full scans and filtering. However, the processing time is then increased proportionally to the number of concurrent queries [59]. By contrast, a shared scan followed by filters incurs high processing time, as it is index- and workload-oblivious, but scales better. Scalability comes from amortizing access costs and, in addition, from permitting work-sharing for downstream operators, including filters, among a batch of queries. Kester et al. [59] show that there is no clear winner between shared scans and indices: databases should make a decision on the dilemma "to scan or to probe an index?" based on both the selectivity and the concurrency of the workload. Nevertheless, both options for access path selection fail to access the required data within a tight time window.

To showcase the shortcomings of existing data-access techniques, consider a workload of "Select-Aggregate" queries of varying concurrency and low joint selectivity that are submitted as batches. Figure 4.1 conceptually depicts the response time for the full batch for different access methods. On the one hand, if we use an index and follow a query-at-a-time processing paradigm, then response time is below the interactivity threshold only when the batch is small (Figure 4.1, area A) and is increased as a function of concurrency. On the other hand, shared scans pay the cost of at least one full scan. This is the lower bound for latency, and it may be high enough to violate our latency constraints (Figure 4.1, area B). Moreover, scans require processing filters over the scanned data; doing so also incurs a high cost, even when using efficient techniques for evaluating the predicates, such as predicate indexing [122]. Still,

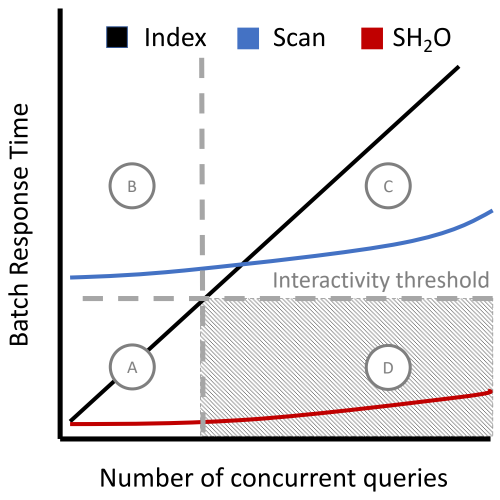


Figure 4.1: Impact of concurrency on batch response time for batches of select-aggregate queries. The results in Figure 4.8 corroborate the conceptual graph

sharing amortizes the high processing cost across multiple queries and becomes the better option for a high degree of concurrency. By applying access path selection [59], we select the best between indices and shared scans, but none of them provides interactivity under a highly concurrent load (Figure 4.1, area C). Therefore, in highly concurrent workloads, the question is not whether to scan or to probe. Interactivity requires combining efficiency in terms of i) the volume of the accessed data, ii) the cost of filtering accessed data, and iii) scalability to the number of queries.

We propose SH_2O, a shared data-access operator that, by exploiting data organization and the workload's access patterns, achieves both efficiency and scalability in a work-sharing environment (Figure 4.1, area D). SH_2O is inspired by the following observation: *for every batch of queries, there exists a partition of data into multidimensional regions where filtering decisions are the same for all contained tuples.*

Example. Consider a dataset of (X, Y) tuples in $[0, 100]^2$ and the following batch of queries:

```
Q1: SELECT COUNT(*) FROM T WHERE X < 50
Q2: SELECT COUNT(*) FROM T WHERE Y < 50
```

Therefore:

- All tuples of $[0, 50) \times [0, 50)$ are processed by both Q1, Q2.
- All tuples of $[0, 50) \times [50, 100]$ are processed only by Q1.
- All tuples of $[50, 100] \times [0, 50)$ are processed only by Q2.
- Region $[50, 100] \times [50, 100]$ is not processed by any query.

where $A \times B$ denotes the cross product of sets A and B.

Shared access to the required regions mitigates concurrency, reduces accessed data, and renders filtering redundant. Still, to put this idea to use, there are two challenges to overcome: i) accessing the required regions and ii) handling dimensionality. On the one hand, while data skipping [114, 115, 130], a technique for accessing partitioned data, superficially resembles access to the required regions, it is inefficient for the problem at hand. By using lightweight metadata, data skipping scans only the partitions that contain required data. However, data skipping operates over predetermined partitions that have been chosen based on historical workload, whereas the regions are different for each batch. When processing tens to hundreds of queries as a batch, it is likely that at least some queries have shifted filters that are misaligned with the boundaries of any existing partitions; then, for each intersecting partition, data skipping scans and filters the entire partition, even if only a single tuple is required. The high cost of filters in in-memory work-sharing databases further aggravates the problem. On the other hand, as the number of filtering predicates in the workload is increased, accessing the regions suffers from the curse of dimensionality. An increase in the number of attributes has a multiplicative effect on the number of regions. In turn, regions become too sparse, and the number of contained tuples is insufficient to amortize the cost of accessing each region.

SH_2O addresses the two challenges by adapting data access to the data organization and the workload across three axes: i) on-demand access by region, ii) attribute selection, and iii) subspace specialization. First, for the filters on a subset of the filtering attributes, it defines the multidimensional regions where the filters make the same decisions for all tuples and then accesses the regions using spatial indices. Spatial indices can efficiently and selectively retrieve regions regardless of their boundaries. Thus, SH_2O performs outperforms data skipping for a wider range of applications that have volatile predicates. Second, SH_2O selects the optimal subset of the filtering attributes for accessing multidimensional regions. The intuition is that there is a cross point where adding one more attribute introduces more overhead than the shared filter that it strives to avoid. To estimate this optimal subset, we propose an analytical cost model and an efficient enumeration algorithm that chooses the subset when planning SH_2O. Third, SH_2O independently adapts to each data partition that corresponds to a different subspace of the data: they can have different spatial indices and are accessed by different queries, and hence using different predicates. To exploit SH_2O's per-partition adaptability, we introduce a partitioning scheme that exploits correlations between predicates to reduce the relevant predicates in each partition and hence i) to eliminate unnecessary fragmentation of the space and ii) to choose a partition-specific index.

SH_2O drastically accelerates workloads that are data-access-heavy, have correlated selective accesses, or incur significant filtering costs. After all, its benefit is proportional to the access and filtering costs it eliminates. However, as the experiments show, SH_2O is adaptive and improves performance in the whole spectrum defined by complexity, selectivity, dimensionality, and concurrency. While SH_2O is most effective for data-access-heavy workloads, for which it achieves up to an order of magnitude speedup compared to shared scans in our experiments, it is beneficial even for join-heavy benchmarks such as SSBM and TPC-H; it reduces end-to-end processing time by 37.5% and 15.3%, respectively.

The contributions of this work are:

- Work-sharing databases use shared scans that are followed by shared filters. This access method suffers from redundant data access and high filtering costs. We propose that shared access to a set of multidimensional regions can replace shared scans and a set of shared filters. We build SH_2O, a novel operator that exploits this insight to provide efficient data access.
- Scan-oriented analytical databases use data skipping to access data efficiently. However, using data skipping to access multidimensional regions is inefficient because it results in overfetching and, thus, causes excessive access and filtering costs. Instead, we show that spatial indices can be used in a novel way to provide shared access to the regions on-demand. Using spatial indices thus minimizes data access and filtering, and is more robust when workload shifts occur.
- Choosing which shared filters to replace introduces a trade-off between cost savings and access overhead. We propose an efficient optimization strategy that uses a dedicated data and workload-aware cost model to hit a sweet spot between savings and overhead by strategically selecting filters to replace.
- Having multiple predicates per query increases fragmentation to regions if handled naively. However, we observe that predicate correlation can, in fact, isolate predicates in specific subspaces of data. We propose that each subspace needs to be indexed and accessed based on the local access patterns. Hence, we introduce a partitioning scheme that takes advantage of predicate correlations to reduce fragmentation and, thus, minimize the cost of shared access.

4.1 The Data-access Bottleneck

Problem: While using global plans is effective at mitigating the impact of concurrency, it also hinders interactivity. With rigid steps such as *the shared scans and the shared filters*, processing a global plan requires significant processing time that exceeds the stringent time constraints of interactive queries.

More specifically, shared scans access the full table, which can be too large to read within milliseconds. Also, as query-set operations are relatively expensive on their own, processing a long sequence of shared filters during the filtering phase can be very time-consuming and the corresponding cost increases with the batch size. Our experiments in Section 4.7.2 corroborate both arguments.

Opportunity: For a range of applications, much of the processing time spent in rigid steps is unnecessary: First, queries are often interested only in a few hot data, and thus, the majority of data accesses are redundant. Second, even when tuples are filtered on multiple attributes, similar tuples repeat the exact same PI searches and query-set updates; hence, computing the same query-set multiple times is also redundant.

We propose SH_2O, a shared data-access operator that can be used as a stand-in for a shared scan in a global plan to address the data-access bottleneck. To do so, it exploits both selective and correlated access patterns and amortizes the cost of the filtering phase across groups of similar tuples. While, naturally, SH_2O has maximum benefit for workloads where data access makes up most of the processing time and can be fully eliminated by SH_2O, it also improves performance for workloads with high selectivity, a high number of filtering attributes, and even complex join-heavy workloads such as SSBM and TPC-H.

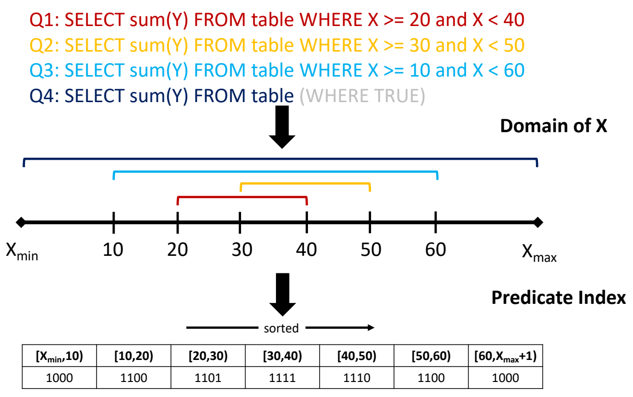
Applicability: To accelerate a workload, SH_2O requires two assumptions: there should be i) *common table accesses* and ii) *filter column stability*, i.e., filtering attributes should recur. Both requirements are frequently met in production workloads. For example, Tableau's dashboards generate batches of queries to populate tens to hundreds of items [120]; hence, Tableau optimizes for multiple queries on the same relation and exploits features in backends such as shared scans. Similarly, Amadeus processes large numbers of decision-support queries over the same denormalized table [122]. Moreover, prior analysis from Sun et al. [114] on real-world data shows that filters are largely stable; only 10% of the unique filters is used in 90% of the queries. Thus, we expect SH_2O to be widely applicable. In the worst case, when the above two assumptions are violated, SH_2O incurs the same costs as indices or shared scans.

As SH_2O relies on emerging access patterns in the workload, we assume batched execution: the work-sharing database enqueues a batch of queries before executing them with a single global plan. However, we can also generalize SH_2O for immediate admission i) by making circular scans at a coarser-than-tuple granularity (e.g., partitions) and ii) by applying SH_2O to the coarser-than-tuple granularity.

4.2 Multidimensional Shared Access

Existing work-sharing databases process multiple shared filters over the entire data in a scan-oriented way and thus miss opportunities that the data organization provides. As they access and process every single tuple, they prevent interactive responses under high degrees of concurrency. To improve interactivity, we propose *multidimensional shared access (MSA)*, a novel workload-driven data-access technique that SH_2O uses to replace a shared scan followed by a set of shared filters.

We assume a work-sharing database that works under the Data-Query model and processes submitted queries one batch at a time. Thus, henceforth, with the term *response time*, we refer to the end-to-end execution time of a query batch. Following common practice [35, 69, 111, 122], to facilitate filtering, for each filtered column, the system produces a set of predicate indices. Moreover, to enforce dependencies imposed by the query plan, the system also decides on a partial order for table accesses, i.e., which tables need to be processed before others and which can be processed in parallel.



```
Q1: SELECT sum(Y) FROM table WHERE X >= 20 and X < 40
Q2: SELECT sum(Y) FROM table WHERE X >= 30 and X < 50
Q3: SELECT sum(Y) FROM table WHERE X >= 10 and X < 60
Q4: SELECT sum(Y) FROM table (WHERE TRUE)
```

$[X_{min},10)$	[10,20)	[20,30)	[30,40)	[40,50)	[50,60)	$[60,X_{max}+1)$
1000	1100	1101	1111	1110	1100	1000

Figure 4.2: From queries to predicate index ranges

MSA identifies a set of multidimensional regions, which logically partition the data, where all queries of the batch always make the same filtering decisions for the replaced filters. Then, it uses these regions to access data. In these regions, the replaced filters produce the same query-set for all tuples. Thus, by accessing each region's data once and augmenting it with the region's query-set, MSA amortizes expensive query-set operations and filtering costs across the entire region. Finally, the remaining filters that are not replaced by MSA process the data from the accessed regions.

MSA accesses the tuples in each region using a preexisting spatial index without overfetching; this is unlike partition-based data skipping, which suffers from overfetching when the regions are not perfectly aligned with physical partitions. The spatial index itself has already been built during an offline tuning phase and is reused across batches.

Realizing SH_2O involves multiple components: i) choosing the filters to replace, ii) identifying the multidimensional regions, iii) performing the index access, and iv) choosing the index to build. In this section, we present MSA and place it in the context of the complete SH_2O. Here, we assume that the index and MSA cover all tuples in the table. We generalize SH_2O for independently-indexed partitions in Section 4.4. Furthermore, we address attribute selection, i.e., the decision on which filters to replace in Section 4.3. Finally, we discuss choosing partitions and indices in Section 4.5.

4.2.1 Computing Single Query-set Regions

Identifying MSA's regions requires understanding under which conditions the filtering decisions for a set of tuples remain invariant. In this section, we provide an analysis of these conditions.

A PI on a specific attribute exhibits locality. It defines a function that maps each value in the attribute's domain to a set of queries for which the indexed predicates are satisfied. Figure 4.2 show an example of the PI of four queries. Adjacent values are then likely to map to the same query-set, e.g., $X = 25$ and $X = 26$ both produce the same query-set $\{Q1, Q3, Q4\}$. Thus, we can represent each PI as a set of (r, Q) pairs, where $r = [\bullet, \bullet)$ is a range in the corresponding attribute and Q is a query-set that indicates which queries are satisfied in the specific range. The ranges define a partitioning on the attribute's domain. When building the PI, we compute the ranges and their corresponding query-sets by partitioning the domain across the boundaries of predicates values and statically computing the predicates in each partition. In the given example, analyzing the predicates gives 7 range-query-set pairs.

To generalize locality to multiple attributes, we compose the ranges of one-dimensional PIs. Let d be the number of attributes used for the filters that MSA replaces and $PI_1, PI_2, \dots, PI_d$ the corresponding predicate indices. Each predicate index PI_i corresponds to its representation as a set of (r, Q) pairs. The computation of single query-set regions is given by the following theorem:

Theorem 2. Let us assume d predicate indices $PI_1, PI_2, \dots, PI_d$, and a random tuple (r_i, Q_i) from each index PI_i. Then, all data in $r_1 \times r_2 \times \cdots \times r_d$ will share the same query-set $Q*$.

Proof. By the definition of predicate indices, all tuples in r_i share the same Q_i. Thus, all tuples in $r_1 \times \cdots \times r_d$ will have $Q* = Q_1 \cap \cdots \cap Q_d$. □

This theorem motivates a hyperrectangle-oriented access strategy for two reasons: First, as all tuples in the same hyperrectangle share the same query-set, we can perform the expensive query-set operations only once per region and then just use the result to annotate each tuple. Second, if $Q* = \emptyset$ for a hyperrectangle, we can skip its tuples altogether. Thus, using hyperrectangles, we can significantly reduce the amount of data processed and the amortized cost per tuple.

MSA enumerates the hyperrectangles by computing the set of $r_1 \times \cdots \times r_d$. To avoid materializing the cross-product of PIs, we produce each hyperrectangle using an iterator that computes the next $(r_1 \times \cdots \times r_d, Q*)$ pair on-the-fly.

4.2.2 Index-based Access to Regions

Each time the iterator produces a hyperrectangle with a non-empty query-set $Q*$, we fetch the corresponding tuples by issuing a range query to a preexisting spatial index that covers MSA's attributes. The query collects the tuples of interest and annotates them with the already computed $Q*$, so they can be processed using the Data-Query model by subsequent shared operators (e.g., joins).

Our method is modular and does not stand for a specific spatial index. It is compatible with any technique that enables efficient multidimensional range queries, such as Ub-tree[80], k-d tree[9], R-tree[42], Hilbert- and Z-curve [65]. The performance characteristics of the employed technique are expected to affect the absolute cost of index access but not the overall trends.

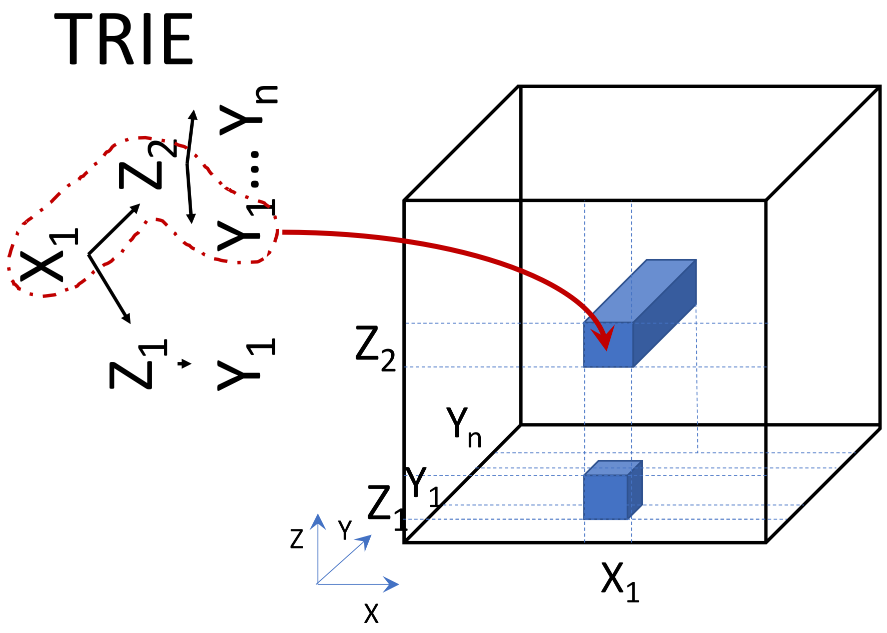


Figure 4.3: Trie-based grid index

In our implementation, we use a grid index, as the one illustrated in Figure 4.3. We sort the tuples based on the d-dimensional projection and organize the grid in a trie. Each level of the trie corresponds to an attribute and each node corresponds to a distinct value in the domain of its level's attribute. Each level's nodes are stored in the same array, and nodes with the same parent are in contiguous positions and sorted among themselves; thus, a binary search finds children nodes that fall within the range query. At the leaves of the trie, we store contiguous sequences of tuples with the same projection. The only tuning knob for our trie implementation is the permutation of indexed attributes, which is chosen offline when building the index.

In each range query, we traverse the trie such that the prefix satisfies the range query's predicates. When the traversal reaches the leaves, it retrieves the corresponding ranges and reads the individual tuples. Henceforth, when we mention an index probe during MSA, we refer to a range query over the used spatial index.

Optimizing for data correlations. In some cases, such as when data is correlated, range queries contain no results. For example, if columns A and B are correlated through the constraint $A - 10 \le B \le A + 10$, then ranges $(A, B, C) \in [50, 70) \times [0, 20) \times r_C$ are empty. To avoid redundant index probes, we eliminate empty hyperrectangles as follows: suppose that $r_1, \ldots, r_k$ is the shortest prefix of ranges that is not satisfied by any node in the k-th level. Because all suffixes $r_{k+1}, \ldots, r_d$ lead to empty hyperrectangles, the traversal abandons the current range query and searches in PI_k for the first non-empty range r_k^* after r_k. The r_k^* range has the property that $r_1, \ldots, r_k^*$ is satisfied by at least one node in the k-th level. If such a range exists, the iterator of hyperrectangles jumps to the first region with the non-empty prefix.

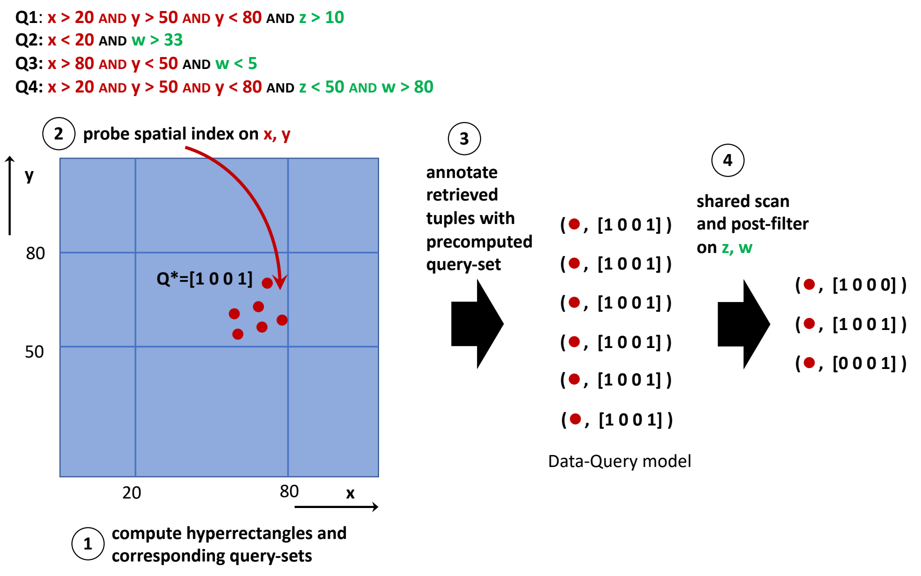

Q1: x > 20 AND y > 50 AND y < 80 AND z > 10
Q2: x < 20 AND w > 33
Q3: x > 80 AND y < 50 AND w < 5
Q4: x > 20 AND y > 50 AND y < 80 AND z < 50 AND w > 80
2 probe spatial index on x, y
y
80
50
20
80
x
Q*=[1 0 0 1]
1 compute hyperrectangles and corresponding query-sets
3 annotate retrieved tuples with precomputed query-set
(●, [1 0 0 1])
(●, [1 0 0 1])
(●, [1 0 0 1])
(●, [1 0 0 1])
(●, [1 0 0 1])
(●, [1 0 0 1])
Data-Query model
4 shared scan and post-filter on z, w
(●, [1 0 0 0])
(●, [1 0 0 1])
(●, [0 0 0 1])

Figure 4.4: Overview of SH_2O

Otherwise, the iterator jumps to the first hyperrectangle with prefix: $r_1, r_2, \ldots, r^*_{k-1}$, where r^*_{k-1} is the next range after r_{k-1}. This way, large numbers of empty hyperrectangles are pruned early, significantly reducing the index probing cost.

4.2.3 Hybrid Execution

The drawback of MSA is that it suffers from the curse of dimensionality because the number of attributes has a multiplicative effect on the number of hyperrectangles. For this purpose, SH_2O uses a hybrid approach: it uses MSA to replace only on a subset of the shared filters that exist in the workload and then uses the remaining shared filters to post-filter the tuples that the spatial index retrieves. Post-filtering is necessary because the query-sets after probing the index correspond only to the filters that MSA replaced. The choice of the subset is based either on the attributes that the available index covers or, as we will see in Section 4.3, on the *attribute selection process* that maximizes the benefit from eliminating filters.

Post-filtering applies filters one after the other in a vectorized manner. The order in which filters are applied is determined by the query optimizer. For each filter, we probe the filter's predicate index and update the query-sets of retrieved tuples accordingly. We drop the tuples with empty query-sets and forward the remaining tuples to the next filter in the pipeline.

This way, SH_2O benefits from eliminating filters while keeping MSA overhead in check. Figure 4.4 shows an example. First, we apply MSA on attributes x and y. In the first step, we identify the data regions that share the same query-set. Then, we fetch the corresponding tuples by probing the spatial index for each region. Finally, we process the retrieved tuples using shared filters on the remaining attributes z, w.

4.2.4 Consolidation

Processing the results of each hyperrectangle separately can reduce the benefit of vectorization. If a d-dimensional region contains very few tuples, interpretation overheads become comparable to the ones in tuple-at-a-time execution. Moreover, the higher the dimensionality, the sparser the data, and the more severe this effect becomes. Naturally, post-filtering further aggravates the situation.

To prevent increasing the interpretation overheads, we place a lightweight *consolidation* operator between post-filtering and other subsequent operators such as joins and aggregates. Consolidation packs smaller intermediate vectors into larger vectors whose size exceeds a threshold. Consequently, execution amortizes per-vector overheads across many more rows and significantly improves performance. We also activate consolidation for non-index workloads to equalize non-filtering costs across index-based and scan-based experimental runs.

4.2.5 Supported Predicates and Extensions

MSA models predicate indices as sets of (r, Q) pairs. This representation supports predicates on a single attribute with a totally-ordered domain (i.e., by having a different range for each value in the worst-case). In practice, it is compact and efficient to use for: i) range predicates, ii) equality predicates, iii) disjunction on the same attribute/IN operator, and iv) conjunction.

Although the current work focuses on reducing data-access cost, with sufficient optimizer support, SH_2O can also accelerate joins. Invisible joins [1] and data-induced predicates [54] propagate filters across joining tables. Furthermore, including join keys in the index probes enables batching probes with the same key and query-set, thus amortizing the query-set overhead in joins.

4.3 Selecting Probing Attributes

MSA substitutes part of the shared full scan/filtering with a single query-set computation and index lookup for each hyperrectangular region. However, an increase in the number of either the probing attributes or of the ranges in the corresponding predicate indices has a multiplicative effect on the number of the resulting hyperrectangles. Hence, there are cases where it is beneficial to probe only a selected subset of the spatial index's attributes. Adding more attributes to this subset would increase overhead more than it would save cost. This way, we reduce: i) the number of hyperrectangles, ii) the cost of multidimensional index lookups, and iii) the hyperrectangle overheads per tuple. Nevertheless, identifying the optimal probing attributes is still a problem. Ideally, the chosen subset of attributes hits a sweet spot between the cost of probing the spatial index and the cost of the post-filtering.

The attribute selection mechanism is triggered each time a table access starts and consists of two parts: i) an analytical cost model that estimates the cost of index probing and shared filtering for a given set of attributes and ii) a search-space algorithm that, given the cost model, finds the optimal subset.

4.3.1 Cost Model

The proposed cost model considers several factors, such as: what indices are available, which columns of the workload have filtering predicates, how many query-set ranges exist in each column's predicate index, what is the data distribution, and how many tuples are retrieved with each index access. Our cost model partitions filtering attributes in two sets: i) a subset to use for MSA and ii) a subset to use for post-filtering. Each of the subsets can be empty.

Our analytical model is easy to understand and tune using regression. It consists of two components: the spatial index access cost (IC) and the shared filter cost (FC). While we have motivated the general case, where a set of filtering attributes is used for probing and the remaining for post-filtering, our model also inherently covers the trivial cases: i) use MSA for all available attributes, or ii) avoid using MSA and do a full-scan instead.

Index Cost

The Index Cost ($IC^{\mathfrak{I}}$) represents the cost of MSA when the index $\mathfrak{I}$ is used. Given i) a dataset of n tuples, ii) a set of filtering attributes $\mathbf{F}$, which are used to probe the index, iii) a set of predicate indices on $\mathbf{F}$: $\mathbf{PI} = \{PI_1, \dots, PI_{|\mathbf{F}|}\}$, and iv) a vector of dataset statistics $\mathbf{D}$, $IC^{\mathfrak{I}}$ can be expressed as:

$$IC^{\mathfrak{I}}(n, \mathbf{F}, \mathbf{PI}, \mathbf{D}) = N(PI, F) * Cost_H(n, \mathbf{F}, \mathbf{PI}, \mathbf{D})$$

With $N(PI, F)$, we denote the number of non-empty hyperrectangles with non-empty query-sets that MSA creates. To estimate the number of hyperrectangles, we assume the worst case in which all hyperrectangles are not empty and have non-empty query-sets hence MSA cannot exploit data correlations to skip hyperrectangles. Hence, we use the formula:

$$N(PI, F) = \prod_{f \in \mathbf{F}} N(PI, \{f\})\}\}$$

$N(PI, F)$ rapidly increases with more or larger predicate indices, and thus choosing strategically the set $\mathbf{F}$ is critical. Note also that the number of accessed ranges in a predicate index $N(PI, \{f\})$ is directly affected by the workload access patterns. Intuitively, the more correlated the queries, the smaller $N(PI, F)$ would be. Thus, our cost model also captures latent workload correlations.

The second term, $Cost_H(n, \mathbf{F}, \mathbf{PI}, \mathbf{D})$, denotes the cost each hyperrectangle incurs. $Cost_H$ is proportional to three quantities: i) the cost QC of query-set operations for computing the query-set $Q*$, that all tuples of the hyperrectangle share, ii) the lookup cost SIC on the spatial index, and iii) the cost VC that the resulting tuples will incur on post-filtering. The latter is practically how many vectors we will need to post-filter. This can be expressed as:

$$Cost_H(n, \mathbf{F}, \mathbf{PI}, \mathbf{D}) = c_q * QC + c_s * SIC + c_v * VC$$

where c_q, c_s, c_v are constants. The value of these constants reflects latent variables such as the underlying hardware, the query-set implementation, the spatial index implementation, etc., and thus should be tuned to match the specific deployment at hand. We tune the parameters by running SH_2O for generated sets of data and queries and then fitting the cost model results to the measured response time using least-squares.

Now, we further discuss and expand the QC, SIC, and VC quantities. The query-cost QC depends on the implementation of query-set operations. Makreshanski et al. discuss the trade-offs of different implementations [70]. Our implementation uses bitsets, and in that case, $QC = |\mathcal{B}|$, where $|\mathcal{B}|$ is the size of the submitted query batch.

The lookup cost SIC is a function of the available predicate indices and data distribution ($SIC = SIC(\mathbf{PI}, \mathbf{D})$). The exact formula of SIC depends on the spatial index implementation. In our case, where a grid index is used, we observe that the cost depends on the number of visited nodes at each level of the trie:

$$SIC(\mathbf{PI}, \mathbf{D}) = \sum_{d=1}^{d_{max}} vis^{\mathcal{I}}(d, PI, D)$$

where d_{max} the dimensionality of the spatial index and $vis^{\mathcal{I}}(d, PI, D)$ the number of visited nodes in level d. We estimate the number of visited nodes using the formula:

$$vis^{\mathcal{I}}(d, PI, D) = \prod_{i=1}^{d} sel(PI_{\mathcal{I}_i}) * \mathbf{D}(\mathcal{I}_i)$$

where $sel(PI_{\mathcal{I}_i})$ is the average selectivity across the accessed ranges of predicate index i, and $\mathbf{D}(\mathcal{I}_i)$ denotes the distinct values in attribute $\mathcal{I}_i$. Also, $\mathcal{I}_i$ maps each index dimension to the corresponding filter attribute; if the dimension does not correspond to a filter attribute, we assume that $sel(PI_{\mathcal{I}_i})$ is 1. The formula assumes that the attributes follow independent distributions and that the number of nodes is lower than the number of tuples: it estimates that for each node of level $i-1$, index traversal accesses $sel(PI_{\mathcal{I}_i}) * \mathbf{D}(\mathcal{I}_i)$ nodes in level i. Thus, the number of visited nodes per level is increased multiplicatively. The assumption of independence is commonly used in query optimizers [67]. Estimating selectivity, especially for predicates across multiple attributes, is an orthogonal and active research area [28, 131].

Finally, the VC cost represents the number of vectors retrieved with each index lookup. We approximate this cost by assuming that all hyperrectangles contain the same number of tuples. The formula used is:

$$VC = \lceil \frac{n}{vs * N(PI, F)} \rceil$$

where vs is the vector size used during processing. The ceiling shows that even when fewer than vs tuples are retrieved from a hyperrectangle, we have to "pay" for the whole vector.

Putting it all together:

$$IC^{\mathcal{I}}(n, \mathbf{F}, \mathbf{PI}, \mathbf{D}) = N(PI, F) * (c_q * |\mathcal{B}| +$$
$$c_s * \sum_{d=1}^{d_{max}} \prod_{i=1}^{d} sel(PI_{\mathcal{I}_i}) * \mathbf{D}(\mathcal{I}_i) + c_v * \lceil \frac{n}{vs * \prod_{f \in F} N(PI_f)} \rceil)$$

Filter Cost

The filter cost models the cost of shared filters using the Data-Query model. For each tuple, if there are m predicate indices available, the tuple's query-set is produced as $probe(PI_1) \cap \cdots \cap probe(PI_m)$. Thus, the cost comprises the predicate index probe and the cost of query-set operations. Again, the cost of query-set operations is proportional to the query batch size $|\mathcal{B}|$.

Regarding the probing cost, our analysis indicates that it mostly depends on data locality. In our implementation, predicate indices take the form of binary trees. If consecutive tuples follow the same path in the predicate index's binary search, the branch prediction mechanism of modern processors significantly accelerates probing. Otherwise, if subsequent tuples follow different paths, performance degrades. To quantify this effect, for each filtering attribute f, we define a locality indicator L_f that shows the expected number of consecutive tuples that follow the same path when probing the corresponding predicate index. Then, using regression, we train a monotonically increasing decay function that maps the lack of locality to a factor of performance degradation: $loc(L_f) \rightarrow [1, \infty]$. In practice, we have observed that $loc(L_f) \in [1, 2.5]$.

In contrast to the spatial index cost, in shared filters, the number of ranges in the predicate index is less important. Finding the correct ranges, here, depends on a binary search, and thus the cost is increasing logarithmically to the number of ranges. Summing up, the cost of shared filters is:

$$FC(n, \mathbf{F}, \mathbf{L}) = n * c_f * \sum_{f \in \mathbf{F}} |\mathcal{B}| * loc(L_f)$$

where $\mathbf{L} = [L_1, \ldots, L_{|\mathbf{F}|}]$ is a vector with the locality indicators for each filter attribute and c_f is a constant.

Total Cost

Overall, the cost of SH_2O when probing the spatial index on attributes **F** is:

$$TC(n, \mathbf{F}, \mathbf{PI}, \mathbf{D}, \mathbf{L}, \mathcal{I}) = IC^{\mathcal{I}}(n, \mathbf{F}, \mathbf{PI}, \mathbf{D}) + FC(n, \mathbf{U_F} - \mathbf{F}, \mathbf{L})$$

where $\mathbf{U_F}$ is the set of all filter columns.

4.3.2 Enumerating Candidate Attribute Sets

The cost model can estimate whether a set of probed attributes is preferable to another. SH_2O takes advantage of these estimates to identify the attributes that minimize the total data access cost: It enumerates candidate attribute sets, and for each candidate, it computes the corresponding cost model's estimate. The set that yields the lowest cost is finally selected.

However, candidate selections are, in the worst case, exponential to the number of attributes. To render enumeration efficient, we build Algorithm 3 that explores the lattice of attribute-sets in a bottom-up way, and prunes parts of the space by using Observation 4.3.2 (line 8). The process terminates when no more candidates are available.

Observation. If adding an extra attribute to a candidate-set increases IC more than it decreases FC, then the resulting candidate-set and its super-sets can be safely pruned.

Algorithm 3: Enumerate Candidate Attribute-Sets

```
   input: Tuple $(n, U_F, PI, D, L)$
1  $level = \{\emptyset\}$ ; $best\_set = \emptyset$ ; $best\_cost = SC(n, U_F, L)$ ;
2  while $level.nonEmpty()$ do
3      $next\_level = \emptyset$ ;
4      for $F \in level$ do
5          for $f \in U_F - F$ do
6              $IC_{old} = IC^I(n, F, PI, D); SC_{old} = SC(n, U_F - F, L)$
                $IC_{new} = IC^I(n, F \cup \{f\}, PI, D)$ ;
7              $SC_{new} = SC(n, U_F - F - \{f\}, L)$ ;
8              if $IC_{new} - IC_{old} < SC_{old} - SC_{new}$ then
9                  $next\_level = next\_level \cup \{F \cup \{f\}\}$ ;
10                 if $TC_{new} < best\_cost$ then
11                     $best\_cost = TC_{new}$ ; $best\_set = F \cup \{f\}$ ;
12     $level = next\_level$ ;
13 return $best\_set$ ;
```

4.4 Partition-oriented Access

In our discussion thus far, we have assumed that there is a single clustered index that spans the whole dataset. However, this simplification ignores local patterns in the query predicates

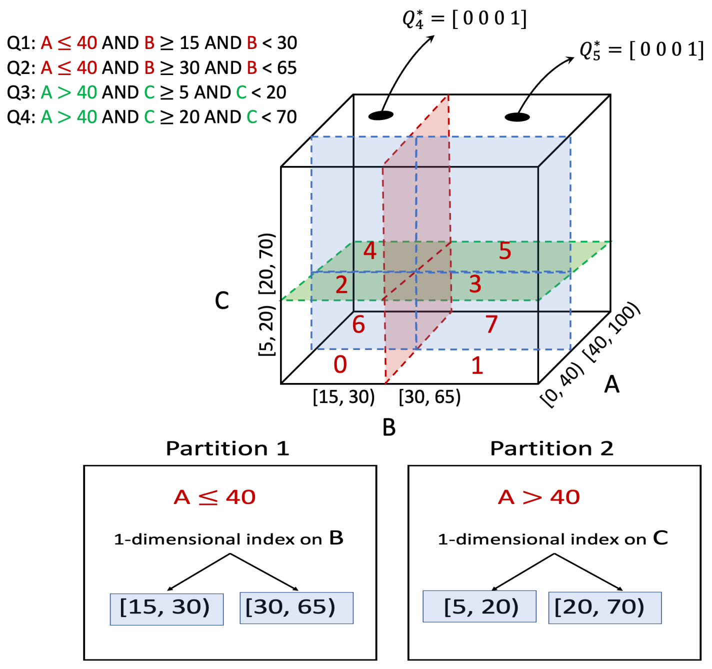


Figure 4.5: Applying SH_2O at partition-granularity

that arise when the data is partitioned and each partition is indexed independently.

For instance, in cases where there are correlations in the workload, by directly applying SH_2O on the entire dataset, we process an unnecessarily large number of hyperrectangles. Consider the example of Figure 4.5. The work-sharing database processes a batch of queries with filters on attributes A, B, C. Computing MSA's hyperrectangles for the entire dataset results in $|PI_A| * |PI_B| * |PI_C| = 8$ probes. However, consider the case where the data is partitioned on the predicate $A > 40$. In each partition, we need to consider only predicates from queries that intersect with the partition. Then, we need to probe $|PI_B| = 2$ hyperrectangles in the first partition $(A \leq 40)$ and $|PI_C| = 2$ in the second partition $(A > 40)$. Moreover, the partitioned case requires indices with fewer dimensions which translates to decreased probing cost.

To exploit local patterns within relevant partitions and reduce both the number of hyperrectangles and the dimensionality of index probes, SH_2O adapts access to each partition. It identifies which predicates are relevant in each partition, using a variant of data skipping, and plans multidimensional shared accesses independently.

First, SH_2O identifies the set of queries that process each partition. A query processes a partition if the partition overlaps with the query's predicates. To compute the query-set of each partition, we combine zone maps [39] with predicate indices. For each predicate index, SH_2O finds the ranges that intersect with the zonemap's `min-max` range, retrieves the corresponding query-sets and computes their union. The union corresponds to the set of queries that either have satisfied predicates or have no predicate on the attribute. Then, SH_2O finally computes the query-set of the partition by intersecting the results from all predicate indices. If the query-set is empty, SH_2O completely skips the partition.

Next, SH_2O optimizes multidimensional shared access for each partition. Filters that do not belong to the partition's query-set or filters that statically evaluate to `TRUE` are excluded from the predicate indices and the attribute selection algorithm. Hence, SH_2O eliminates a large portion of the already reduced filtering costs.

4.5 SH2O-aware Data Organization

The effectiveness of SH_2O depends on preexisting partitions and indices. To reduce data-access and filtering costs, SH_2O requires i) spatial indices that can replace shared filters in a query batch arriving at runtime, thus enabling selective and efficient access, and ii) partitions that exploit predicate correlations. In this section, we formulate and address the problem of partitioning/indexing the data such that we minimize SH_2O's cost for a target batch.

We partition/index the data in an offline manner. As long as predicate patterns recur (i.e. predicates on the same attributes and correlations between predicates) the partition/index-building cost is an investment that is expected to be amortized with time. In this work, we do not elaborate on the predicate monitoring process or on the details of how often partitions should be updated, but we theoretically formulate and solve the joint partition-index selection problem for SH_2O. We assume that the workload is known in advance and we build partitions and indices once in the beginning.

In the literature, there are several data-layout optimization techniques that partition a multi-dimensional dataset in a way that captures query correlations and favors data skipping (e.g., [130]). However, existing partitioning approaches: i) do not account for shared access across a batch of queries and ii) are index-oblivious. Here, we address both deficiencies at the same time. More specifically, we take advantage of emerging access patterns in the workload and partition the data space into a set of hyperrectangles. For each of these hyperrectangles, partition-index selection uses the cost model of Section 4.3.1 to choose an index that best fits the specific subspace. The goal is to *minimize the aggregate data access time across all partitions.* At runtime, SH_2O processes each resulting partition independently. Formally, we define the *Index-Aware Partitioning for Shared Access (IPSA)* problem:
Problem (IPSA)**. —** *Given w query batches with predicate index sets $PI^1, PI^2, \ldots, PI^w$, find a set of partition-index pairs $\boldsymbol{P} = \{(S_1, I_1), \ldots, (S_m, I_m)\}$ such that*

$$\min \sum_{(S_i,I_i)\in \boldsymbol{P}} \sum_{j=1}^{w} \min_{\boldsymbol{F}} TC(|S_i|, \boldsymbol{F}, \boldsymbol{PI}^j|S_i, S_i, \boldsymbol{L}, I_i)$$

with the constraint that $S_1, \ldots, S_m$ are hyperrectangles [1].

We observe that any partitioning can be generated by applying a series of recursive cuts, where each cut corresponds to a predicate (e.g., Figure 4.5). The recursion starts by taking as input

[1]The notation $\mathbf{PI}|S_i$ signifies the subset of each predicate index that is within the boundaries that define S_i

the entire dataset. Then, at each step, the current partition is either kept intact and returned as part of the solution, or is bi-partitioned by a new cut. In the latter case, the two resulting partitions are further processed recursively. This structure enables a Dynamic Programming (DP) formulation. Let us denote the optimal data access cost for partition S as

$$OC(S) = \min_{\mathbf{I}} \sum_{j=1}^{w} \min_{\mathbf{F}} TC(|S|, \mathbf{F}, \mathbf{PI^j} | S, S, \mathbf{L}, I)$$

Moreover, given a cut c, we use $V_c(S)$ to denote the subspace of S that satisfies c. For example, in Figure 4.5, $V_c(Partition_1) = [0, 100) \times [15, 30) \times [5, 70)$. Then DP is expressed as:

$$\begin{cases} P(S) = min\{OC(S), min_c\{P(V_c(S)) + P(S - V_c(S))\}\} \\ P(S) = OC(S) \quad \text{, if no other cut can be applied} \end{cases}$$

4.5.1 Iterative Partitioning

Solving IPSA using DP is prohibitive as it requires tabulating and estimating access costs for all the possible subspaces that the filtering attributes of a batch define. This number is expected to be very high and much larger than the number of hyperrectangles that the predicate indices define.

To efficiently approximate the optimal solution, we use an iterative greedy algorithm that is inspired by Iterative Dynamic Programming [61]. The algorithm works in iterations and starts from a single partition that contains all the data. At each iteration, it chooses a partition and finds the optimal sequence of k recursive cuts (Algorithm 4, line 13) that minimize the total cost across all new subpartitions. At the same time, it also selects the optimal index for all the partitions that these cuts produce (line 15). If the cost reduction, after making the k cuts, is more than a relative threshold $t\%$ (line 7), the cuts are actuated and the new partitions become candidates for the next iteration (line 11). The iterations stop when k cuts cannot significantly reduce the cost of any of the partitions. The algorithm's behavior is tunable based on the value of k. For small values of k, the algorithm is fast but can only discover simple correlations. For larger values of k, the algorithm is more expensive but can discover more complex correlations.

The cuts we select are always filter constants that appear in at least one batch of the target workload. The idea is that if a cut does not correspond to a filter, then aligning it to an adjacent filter would further reduce the cost. Thus, cuts derived from the workload's filters reduce the search space without jeopardizing the quality of the solution.

Algorithm 4: Index-Aware Partitioning for Shared Access

```
Function MAKE_PARTITIONS($n, U_F, PI, D, k, t$):
    $output = \emptyset$; $partitions = newQueue()$ ;
    $partitions.push(D)$ ;
    while $partitions.nonEmpty()$ do
        $target = partitions.pop()$;
        $kcuts = BEST\_KCUTS()$ ;
        if $kcuts.bestCost >= t * target.cost$ then
            $output.add(target)$ ;
        else
            for $p \in kcuts.results$ do
                $partitions.push(p)$ ;
return output ;
Function BEST_KCUTS($n, U_F, PI, D, k$):
    if $k == 0$ then
        $ret.bestCost = CHOOSE\_INDEX(n, U_F, PI, D)$ ;
        $ret.results = \{D\}$; return $ret$ ;
    $cuts\_in\_partition = \{f \in U_F | f\ cuts\ D\}$ ;
    for $cut \in cuts\_in\_partition$ do
        $(lhs, rhs) = estimatePartition(n, D)$ ;
        for $i = 0\ to\ k$ do
            $j = k - i - 1$ ;
            $ret1 = BEST\_KCUTS(lhs.n, U_F, PI, lhs.D, i)$ ;
            $ret2 = BEST\_KCUTS(rhs.n, U_F, PI, rhs.D, j)$ ;
            $total\_cost = ret1.bestCost + ret2.bestCost$ ;
            if $ret.bestCost > total\_cost$ then
                $ret.bestCost = total\_cost$ ;
                $ret.results = ret1.results \cup ret2.results$ ;
return $ret$ ;
Function CHOOSE_INDEX($n, U_F, PI, D$):
    for $j = 0\ to\ w - 1$ do
        for $i = 0\ to\ d - 1$ do
            $PI_i^j | D = \{r \in PI_i^j | r \cap D_i \neq \emptyset\}$ ;
        $U_F^j = \{f \in U_F || PI_f^j | > 1\}$ ;
    $best\_perm = \emptyset$ ; $bestCost = \sum_{j=0}^{w-1} FC(n, U_F^j, L_{max})$;
    $permutations = newQueue()$; $permutations.push(\emptyset)$ ;
    while $permutations.nonEmpty()$ do
        $I = permutations.pop()$ ;
        $cost = \sum_{j=0}^{w-1} MAXTC(n, U_F^j, PI^j | D, D, L(I), I)$ ;
        if $cost < bestCost$ then
            $best\_perm = I$ ; $bestCost = cost$ ;
        for $f \in U_F - Set(current)$ do
            $permutations.push(I + +f)$ ;
return $bestCost$ ;
```

4.5.2 Index Selection

The last component is to select the optimal index for each partition. The selection process i) estimates the data access cost of accessing a partition as is, without further subpartitioning, and thus is critical for IPSA and ii) chooses which index to build for the final partition.

The index selection algorithm is based on attribute selection. For each possible index, given a set of filter attributes, it estimates the data access cost for the optimal attributes (lines 37-40). The lowest estimate determines which index to build (lines 39-40).

In the general case, the number of indices is exponential to the number of attributes. Thus, while this approach works for few attributes (less than 10), more efficient approximations might be necessary for high-dimensional problems.

4.6 Implementation

Our implementation is based on RouLette. Our implementation for SH_2O modifies the *ingestion* and *shared filter* components: When scheduling a scan, the *attribute selection* process of Section 4.3 estimates the best subset of columns on which to use MSA. If shared scan followed by filters is estimated to perform better than any subset, then the original code path is used. Otherwise, we modify RouLette to use MSA for the best subset and exclude the corresponding columns from shared filtering. After filtering, we also consolidate results as described in Section 4.2.4.

MSA can retrieve more than a vector's worth of elements with each lookup. In this case, we affinitize all the rows contained in the specific hyperrectangle to the worker performing lookup. The worker caches the lookup results, and in subsequent calls to ingestion, it extracts a vector of tuples directly from the cache. When all cached results are returned, the worker moves to the next hyperrectangle. By doing so, each worker processes its own exclusively-owned hyperrectangles, and synchronization overheads are reduced.

By default, Roulette assumes single-partition tables. To enable the partitioning of Section 4.5, we modify ingestion. To minimize synchronization, initially, a different partition is assigned to each worker. Once a worker finishes processing a partition, it requests another one. If there are no more unassigned partitions, the worker is attached to the same partition that has been assigned to another worker. All workers working on the same partition pull hyperrectangles from the same iterator.

RouLette, and by extension SH_2O, processes memory-resident data. As such, the current implementation is optimized for in-memory processing. However, the approach is also applicable to disk-based systems: First, selective access can significantly reduce I/O. Second, with modern SSDs achieving several GB/s in read bandwidth, probing a sequence of predicate indices is still too expensive to be masked by I/O. Nevertheless, a disk-based implementation requires extra optimizations: i) to avoid spreading range queries across several disk pages, data needs to be organized on disk using space-filling curves, and ii) similar to cooperative scans [138], to maximize bandwidth, we need to implement I/O scheduling. An extension for disk-based systems is outside the scope of this work.

4.7 Experimental Evaluation

Our experiments evaluate SH_2O across three axes:

i) We first analyze the cost of shared scans and filters and show how multidimensional shared access can eliminate it.

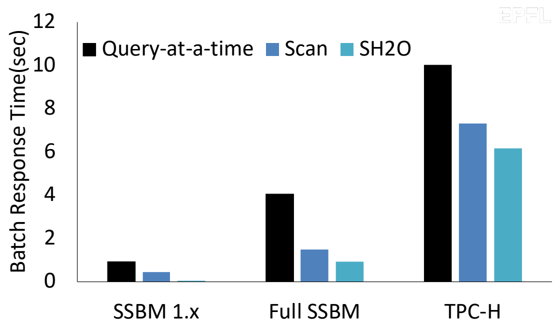
ii) We discuss the introduced overhead by the number of hyperrectangles and demonstrate the merits of attribute selection.

iii) We show that multidimensional partitioning significantly reduces the data access cost for different workload families.

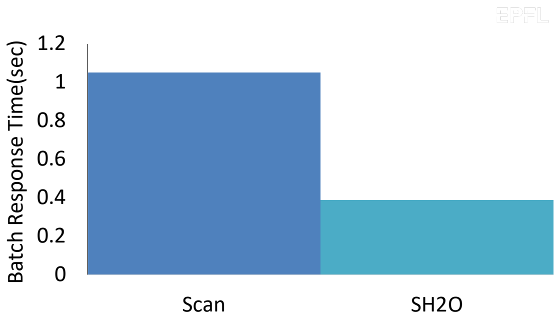
Access Methods. We evaluate the following methods: i) **Scan**: Shared full scan and filtering using the Data-Query model, ii) **MSA**: Access data by exclusively using MSA, iii) **SH2O**: This is our hybrid approach where we first use MSA and then apply shared post-filtering, iv) **Qd-tree [130]:** This is a state-of-the-art data skipping approach that partitions the data based on the workload, in a way that maximizes partition pruning, v) **Zonemaps**: This is standard data skipping over horizontal partitioning. When we assess this technique, the dataset is always sorted on the filtering attribute.

We also compare against well-known databases: MonetDB [46], and PostgreSQL, that we use as baselines for query-at-a-time execution over indices. MonetDB is optimized for efficient columnar data access, whereas PostgreSQL has a mature B+-tree design and supports multidimensional indexing with GiST. We configure both databases to keep data and execution in-memory. Note that, as our workload is analytical (i.e., queries process hundreds of thousands tuples), the bottleneck is data access and processing, not the index traversal itself. Thus, we do not expect alternative indexing techniques to affect the showcased trends and conclusions.

Data & Workload. We run both macro- and micro-benchmarks. First, we show how the proposed technique can accelerate analytical applications such as the widely used SSBM [87] and TPC-H benchmarks with scale factor 10. The order of the tuples is randomized. Then, we perform an extensive sensitivity analysis. More specifically, we investigate the effect of: i) concurrency (query batch size), ii) selectivity, iii) the number of filtering attributes, iv) data correlations, v) the size of the resulting predicate indices, and iv) the predicate correlations. To allow controlling the experiment variables, for the purpose of the micro-benchmarks, we generate synthetic data. We use a single table of $256M$ rows and 4-byte integers. The number of columns and their distribution is presented in each experiment.



(a) End-to-end execution



(b) Data-access cost for TPC-H

Figure 4.6: SSBM and TPC-H benchmarks with scale factor 10

Hardware. We run the evaluation on a two-socket machine with Intel Xeon Gold 5118 CPU with 12 cores per socket, running at 2.30 GHz, and 378 GB of main-memory. We isolate execution on one NUMA-node and run all our experiments with 12 threads. The threads and memory are affinitized to the used NUMA node. We run all the experiments after the data has been loaded in-memory in columnar format, and report the average of 3 runs.

4.7.1 End-to-end Performance for Analytics

We evaluate the effect of SH2O on analytical queries using the Star Schema Benchmark (SSBM) [87] and TPC-H. Both SSBM and TPC-H measure the performance of databases for data warehousing applications. SSBM defines four select-project-join-aggregate query templates over a star schema. For each template, there are multiple variants with different selectivity. In total, SSBM has thirteen queries. TPC-H defines twenty-two query templates that cover more advanced SQL queries. We simplify TPC-H queries i) by replacing aggregates with CASE WHEN, and ii) by focusing on their Select-Project-Join-Aggregate subqueries, similar to [36].

The experiment compares SH2O to Scan and to executing queries one at a time. All three approaches are implemented in RouLette. We compare total response time for batches that contain the *full SSBM* and the *full TPC-H* as well as a batch that consists of SSBM `Q1.1`, `Q1.2`, and `Q1.3` (*SSBM 1.x*). We single out these queries because they are the only ones that compute filters directly on the fact table; all other queries only filter using dimension joins. The trie for TPC-H takes 8.7 sec to build and requires 216MB, whereas the trie for SSBM takes 2.2 sec to build and requires 53kB.

Figure 4.6a shows the results of the end-to-end comparison. Scan performs better than query-at-a-time execution, and SH2O always performs better than Scan. For SSBM 1.x, which aggressively filters the fact table, eliminating the shared filters makes a substantial difference and SH2O results in 12.9× over Scan. For the full SSBM, which accesses all rows in the fact table, SH2O still achieves a moderate 1.6×. Similarly, for the full TPC-H, the achieved speedup is 1.18. SH2O accelerates the *scan & filtering phase.* SH2O moderately speeds up end-to-end performance even though heavy joins partially mask significant speedup in data-access. To prove this, Figure 4.6b compares, for the TPC-H case, the time that the Scan and SH2O spend in data access. SH2O achieves a 2.69× over Scan.

Takeaway: SH2O reduces response times even for complex queries. The benefit of SH2O is maximum for queries that filter large tables. However, by eliminating filter overhead, it still reduces response times even when almost all data is accessed.

4.7.2 Efficient Multidimensional Data Access

In this section, we show the costs from which existing techniques suffer when we vary i) the *joint selectivity*, ii) the *size* and iii) the *filtering attributes* of a query batch, and how MSA eliminates these costs. Moreover, we show how MSA handles the high number of hyperrectangles that data correlations produce.

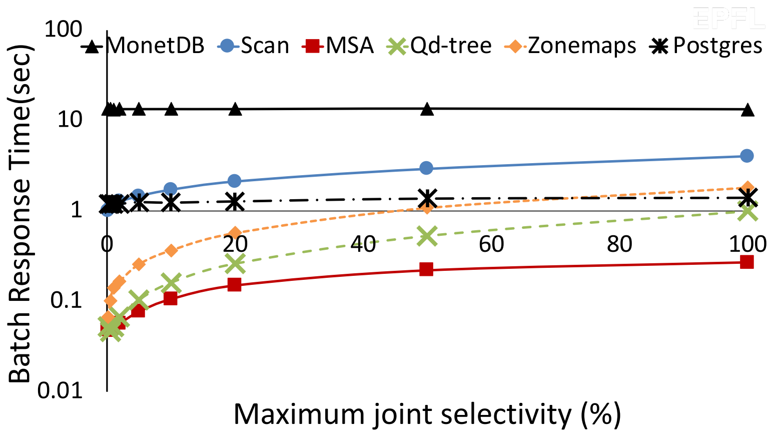
Joint selectivity. We compare Scan, MSA, Qd-tree, Zonemaps, and the baseline databases (MonetDB, PostgreSQL), to analyze the behavior of shared scans and index-based methods under varying workload parameters. In this experiment, SH2O only uses MSA. We use a table with two uniformly distributed columns. The first column, whose domain is $[0,100k)$, is indexed. Building the trie takes 7.98 *sec* and requires 781kB. MonetDB uses the ordered index, and PostgreSQL uses B-trees. All queries filter the first column. For all the experiments of the section, we use batches of 512 filter-aggregate queries. The filter is a range on the first column.

Figure 4.7 shows the impact of the amount of common accesses for varying query selectivities (0.1%, 1% and 10%). We control the *maximum joint selectivity*, that is the fraction of the table's rows that can be accessed by one or more queries. When maximum joint selectivity is 1%, all queries of the batch are generated such that their predicates request a subset of the selected fraction. When maximum joint selectivity is 100%, each query can access any possible range. To demonstrate the robustness of MSA to workload shifts compared to partition-based data-skipping approaches, the predicates of the query workload are shifted by 0.1% compared to the tuning workload of the Qd-tree. In the absense of workload shift, Qd-tree has identical performance with MSA.

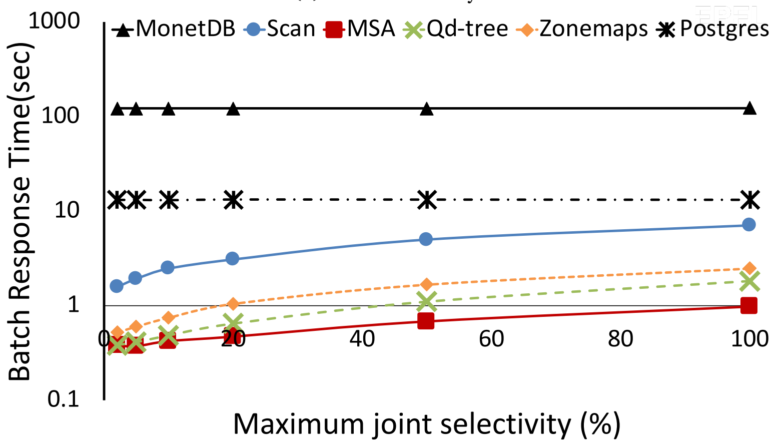
The response time of all databases depends exclusively on the performance of individual queries and is unaffected from changes to the maximum joint selectivity. However, for an individual selectivity of 0.1% and a joint selectivity 100%, PostgreSQL is competitive to Qd-tree and Zonemaps.

Scan always processes the entire dataset. Thus, redundant filtering overheads are introduced and latency is high even when all queries access the same 0.2% of the data. Decreasing the maximum joint selectivity only changes the cost of aggregation at the end, and moderately affects the result.

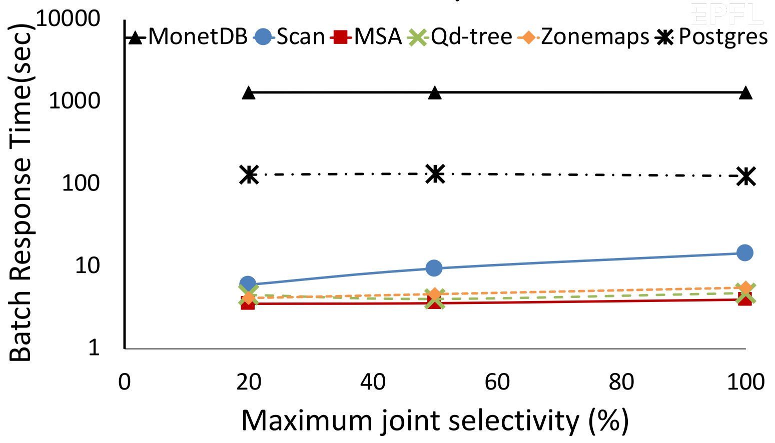
By contrast, MSA significantly benefits from low maximum joint selectivity, and gradually pays the cost of spreading accesses across the table. When the workload contains queries with 0.1% selectivity and accesses less than 2% of the table, MSA gives a speedup of more than an order of magnitude compared to Scan. Even in the worst case, where accesses are completely uncorrelated, it achieves a lower response time by 45% as i) it still accesses fewer



(a) 0.1% selectivity



(b) 1% selectivity



(c) 10% selectivity

Figure 4.7: Effect of joint selectivity on data-access

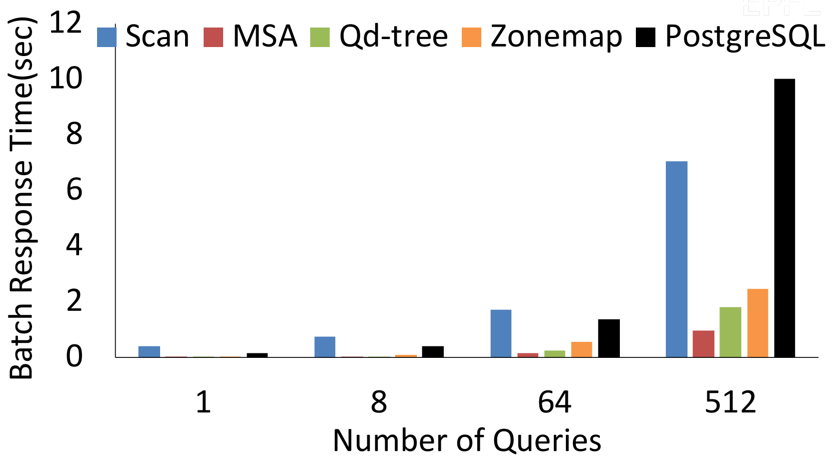


Figure 4.8: Effect of concurrency on response time

data, ii) incurs lower filtering overheads, and iii) achieves higher locality in the router before aggregations.

Data-skipping approaches also benefit from low joint-selectivity. However, as data accesses spread with increasing joint selectivity, they suffer from overfetching and filtering costs. Thus, MSA achieves up to 3.68× over Qd-tree and 6.66× over Zonemaps.

We also compare the distribution of individual responses in PostgreSQL against the batch response time of MSA. We use the same workload with 1% selectivity and 10% joint selectivity. PostgreSQL finishes only 8.9% of the workload in less time than required by MSA, whereas, for Scan, the percentage is 73.4%.

Takeaway: Techniques that combine selective and shared access drastically improve interactivity in workloads that span a small portion of the table. Nevertheless, MSA is superior to data-skipping approaches, which are susceptible to overfetching and filtering costs.

Batch Size. Figure 4.8 shows the impact of concurrency on Scan, MSA, the data-skipping techniques and PostgreSQL by varying the query batch size. We use the same data, indices, and workload as the scenario with 1% selectivity and 10% joint selectivity in Figure 4.7.

MSA exploits the fact that smaller batches effectively access a very small portion of the table. As the batch size is increased, response time is also increased but does so sublinearly until 64 queries. Scan starts with almost two orders of magnitude higher response time. Cost is increased along with the number of queries due to heavier query-set operations, larger predicate indices, and more aggregations. When the query batch size exceeds the 64 queries, query-set operations become particularly heavy and cause a sharp latency increase for both methods. However, even for 512 queries, MSA is 7.2× faster than Scan and 1.8× faster than Qd-tree.

Takeaway: MSA gradually spreads data access across the table. It outperforms both indices and shared scans across the whole concurrency spectrum.

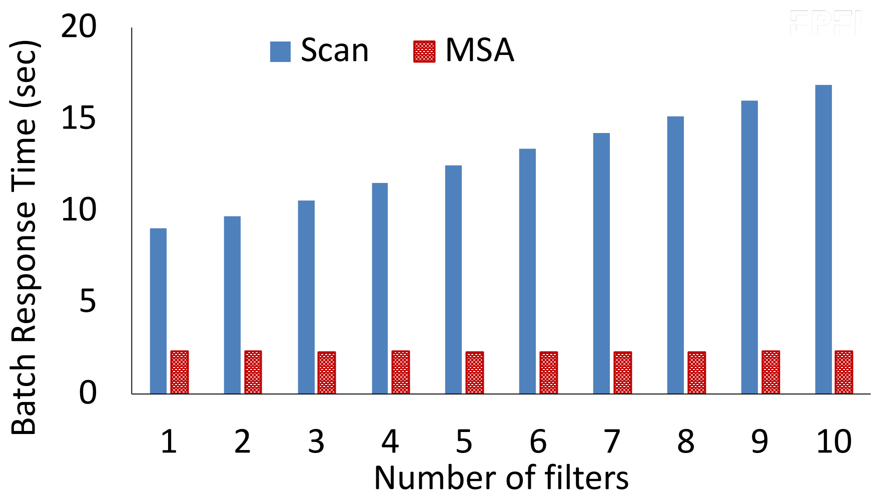


Figure 4.9: Effect of the number of filter attributes on response time

Filter attributes.Figure 4.9 assesses the impact of the number of filtering attributes on a table with 10 columns. The workload accesses the full table, which is the worst case for MSA. To scale the number of filters without exploding the number of resulting hyperrectangles, which we test in the next Section, we assume that 9 columns of the table contain boolean $\{0,1\}$ values. Each query contains two filters and has 5% selectivity: i) the first filter is on the same attribute across all queries and has 10% selectivity, ii) the second filter is on one of the boolean columns and has 50% selectivity. For all runs, we use the same spatial index, which is built taking into account all 10 columns (first the common filter, and then the rest).

Scan shows a linear increase with the number of filters: more filters directly translate to increased query-set operations per tuple.

MSA demonstrates near-constant performance; a slight decrease in execution time is due to better load balancing between threads. The efficiency of MSA stems from the fact that filtering costs are not spread to the whole dataset. Instead of the expensive query-set operations, MSA only probes and computes the corresponding query-sets for up to 10240 hyperrectangles. By amortizing the filtering overhead among all tuples of a hyperrectangle, it is $4.69\times$ faster than Scan, despite that they both access the whole table.

Takeaway: The benefit of amortizing query-set overhead across a hyperrectangle is proportional to the number of filter attributes.

Data correlations. Figure 4.10 shows the effect of the early elimination optimization during hyperrectangle iteration. We use a table with three columns, where the first, C_1, is uniformly distributed in $[0,1k)$. We make the 2^{nd} and 3^{rd} correlated to C_1 by adding uniform random variables. The range of the variables determines the correlation. Therefore, we use $[-500,500]$ to achieve a correlation of 0.5, $[-250,250]$ for 0.8 etc. Correlation 0 means that all columns are generated independently. Each query has a filter on one of the three columns and retrieves approximately 1% of the rows.

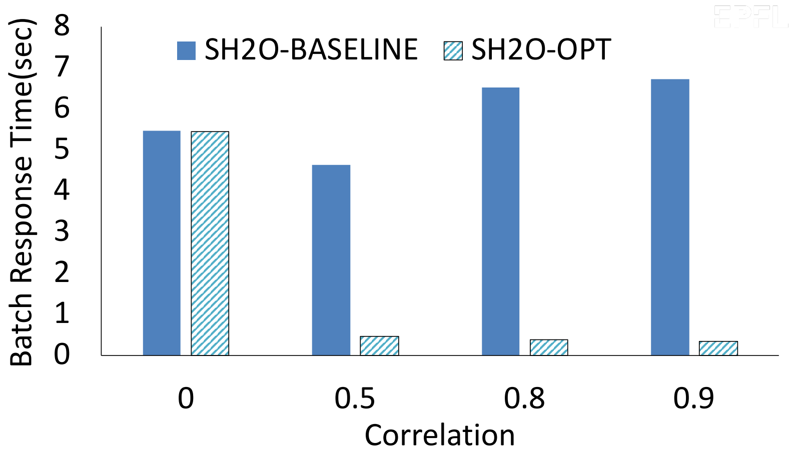


Figure 4.10: Effect of data correlation on response time

Without the optimization of Section 4.2.2, MSA suffers from the high number of hyperrectangles. The observed variance is due to differences in the trie's structure depending on the degree of correlation. Enabling the optimization significantly reduces the number of probed hyperrectangles and the benefit is increased as a function of correlation. When correlation becomes 1, the optimization reduces response time by 21.2×.

Takeaway: The response time of MSA depends on the actual number of non-empty hyperrectangles. Eliminating them is critical when the data contains correlations.

4.7.3 Scaling using Attribute Selection

Next, we demonstrate the effect of scaling the hyperrectangles and the importance of attribute selection. As this is a latent parameter that depends on the number of filter columns and the size of the corresponding predicate indices, we design two experiments: in each of these, we fix one parameter and vary the other.

First, we fix the size of each predicate index to 10 ranges and vary the number of filtering columns. To scale the experiment to wider tables, we use 32 uniformly distributed columns in $[0, 10)$. Due to the large number of attributes, we downsize the table to 100M rows. At each run, we build the spatial index on all the columns that are used for SH_2O. Each query has a single filter column and different query groups access non-overlapping columns (uncorrelated access pattern). We compare MSA, Scan, and SH_2O. We also compare against the GiST index of PostgreSQL, which is a generalized search tree. It provides support for multidimensional point of type cube. However, SH_2O achieves 14.1-73.4x faster response time hence we exclude GiST from the plots.

Figure 4.11 shows how attribute selection improves scalability. For MSA, both the number of hyperrectangles and the response time grow exponentially with the number of filter columns. After a crossing point, MSA becomes significantly slower than Scan, whose response time grows linearly.

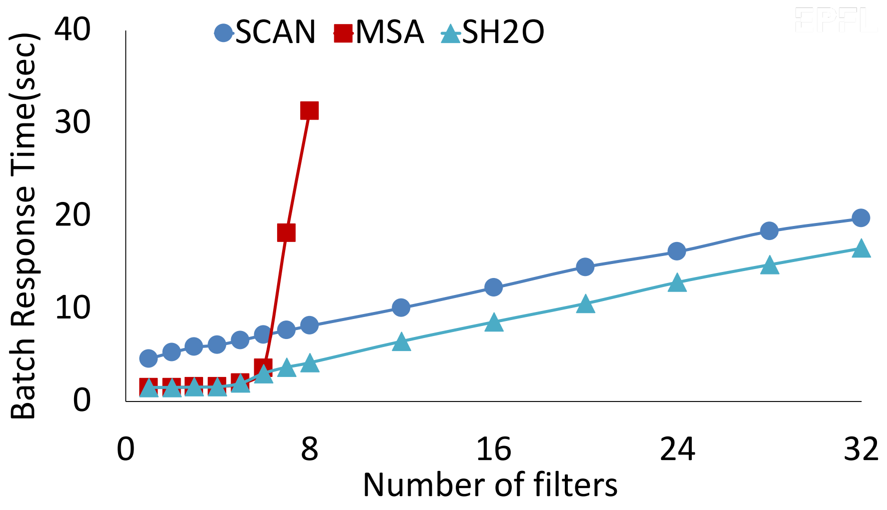


Figure 4.11: Scalability for wide tables

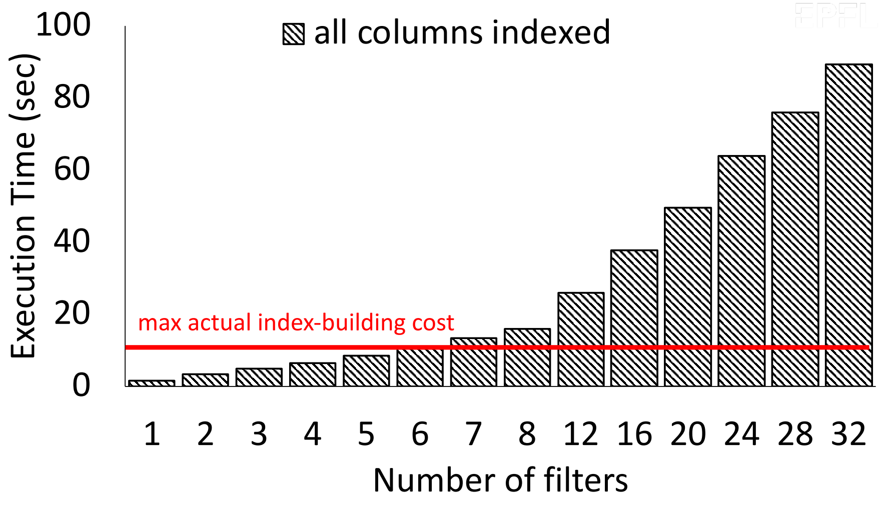


Figure 4.12: Cost of building the spatial index

Attribute selection chooses to use a subset of the filter columns such that expanding it with one more column would make the overhead higher than the savings from skipping the scan. Our cost model detects such cases and we observe that in this experiment, it never probes more than 6 columns (the remaining attributes are post-filtered). SH2O is more efficient than both MSA and Scan, as it trades filters for some of the attributes for a small overhead. The gains for the eliminated filters persist, and SH2O achieves 1.19× over Scan, even when the total number of filters is thirty-two.

Figure 4.12 shows the index building cost. If we index all available columns, the cost is minimal at first, but it sharply increases as the dimensionality is increased. The space overhead is also increased, at first exponentially until 7 attributes (85MB) and then linearly as the trie's leaves degenerate to single tuples. However, as our cost model is never going to probe more than 6 attributes, we leverage this information and only index the attributes that SH_2O would use

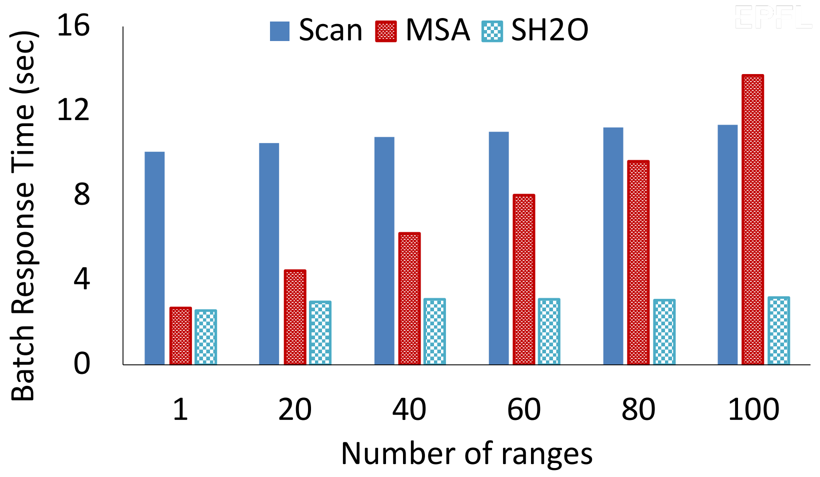


Figure 4.13: Scalability with the number of predicate index's ranges

based on the current batch. Thus, we guarantee that preprocessing cost is bounded and that it pays off.

In Figure 4.13, we fix the number of filtering attributes to 4 and vary the size of the predicate indices. More specifically, they have sizes: $[X, 100, 100, 100]$, *where* $X \in [0, 100]$ is the size of the PI whose respective attributes corresponds to the topmost level of the trie. The experiment uses queries of individual selectivity 1%, but, collectively, the batch accesses the whole table. Once again, attribute selection chooses to probe on an optimal subset of the filter columns. By doing so, hybrid access achieves performance gains without sacrificing the robustness of shared scans and filters.

Takeaway: For MSA, the data-access time is determined by the number of hyperrectangles. For a large number of hyperrectangles, SH2O outperforms both pure scan- and index-based techniques.

4.7.4 Decoupling Dimensions using Partitioning

In the next experiment, we evaluate the impact of partitioning on decorrelating filter dimensions. In this experiment, we use three common correlations in data analysis:

1. *1D dependency*: a query in the batch has a predicate in column C_i iff it also has a predicate p_i in column A. In this experiment, $i \in \{1, 2, 3, 4\}$ and each column C_i has 100 distinct predicates. Before partitioning, we index columns A, C_1, C_2, C_3, C_4, in this order. After partitioning, we index in each partition only the C_i that is filtered.
2. *2D dependency*: if a query in the batch has predicate p_i in column A and predicate q_j in column B, it also has a predicate in column $C_{(i+j)\%n+1}$. In this experiment, $i, j \in \{1, 2, 3, 4\}$ and each column C_k has 100 distinct predicates. Before partitioning, we index columns A, B, C_1, C_2, C_3, C_4, in this order. After partitioning, we index in each partition only the C_i that is filtered.

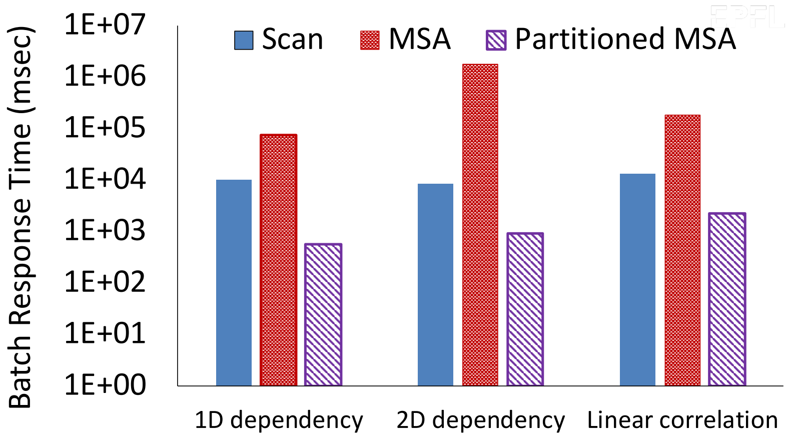


Figure 4.14: Impact of partitioning on response time

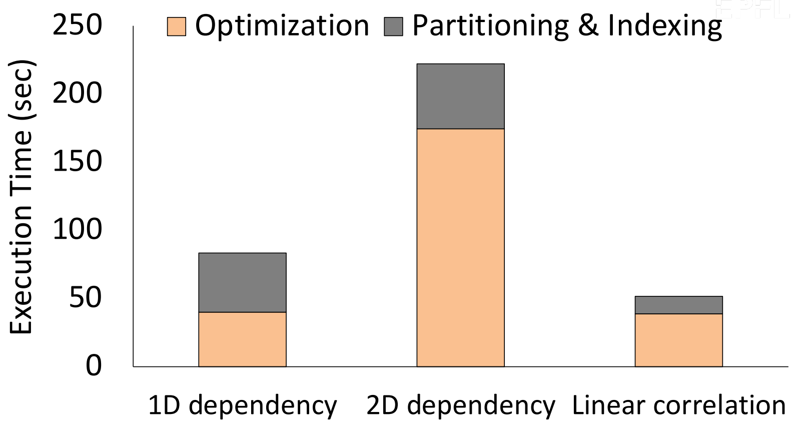


Figure 4.15: Preprocessing cost for partitioning

3. *Linear correlations*: if a query in the batch has a predicate $table.A\ between\ X\ and\ Y$, then it also has predicates $table.C_i\ between\ X+E\ and\ Y+E$ where $E \in [-10,10]$, $i \in \{1,\ldots,5\}$. In this experiment, there are up to 100 distinct predicates in column $table.A$. We index columns $A, B_1, \ldots, B_5$, in this order, both before and after partitioning.

We run the iterative algorithm to partition the data based on the filters of the batch, actuate the partition, and finally run the batch itself. We report the response times in Figure 4.14 and the tuning time in Figure 4.15. Note that response time is in msec and presented in log-scale.

For all three workloads, the number of hyperrectangles is high. As such, in all cases, single-partition MSA is orders of magnitude more expensive than Scan; 212× slower in the worst case. By using the partitions chosen by our iterative algorithm, MSA reduces response time by more than three orders of magnitude and outperforms Scan.

Takeaway: When filters occur in uncorrelated columns, data access time for MSA is prohibitive. In high-dimensional workloads, partitioning is necessary for making MSA the best option.

4.8 Summary

To provide interactivity for highly concurrent workloads, we propose SH_2O, a novel data-access method that combines efficient selective access with minimal filtering cost, and scalability. SH_2O amortizes filtering cost by exploiting multidimensional regions where filtering decisions are invariant across all tuples. However, multidimensional data access suffers from the curse of dimensionality. To avoid dimensionality pitfalls, SH_2O employs two complementary mechanisms, attribute selection and partitioning. By probing only a select subset of dimensions and taking advantage of query and data correlations, SH_2O outperforms shared scan and filters from $1.8\times$ to $22.2\times$.

5 Effective and Efficient Reuse in Work-Sharing Environments

Reusability is a driving factor for many analytical tools, such as dashboards, notebooks, and pipelines. Often, such reusable workloads consist of highly concurrent parameterized queries. Dashboards, for example, produce visualizations by processing several canned queries that are parameterized through UI interactions or other queries [29, 120]. Similarly, analysts rerun data-science notebooks for reproducibility and exploration, often with different parameters [8, 18, 58, 102]; hence, multiple queries that transform and analyze data recur. While such applications process large numbers of queries, they are interactive in nature and require low response times for all queries. However, under high concurrency, backend databases struggle to produce responses within a tight timeframe.

Traditionally, there are two approaches to accelerate processing for large batches of recurring queries. On the one hand, we can optimize individual queries. To do so, both commercial and open-source databases can *reuse* materialized results; databases avoid full recomputation and drastically reduce processing time. Often, optimizing for reuse opportunities is automated in the form of caching, recycling, and materialized views and subexpressions [48, 51, 100, 109, 136]. Nevertheless, materialization is subject to a storage budget and thus leaves outstanding computations. Moreover, as the outstanding computations for different queries are still processed individually, response time is increased with concurrency.

On the other hand, we can optimize the scalability of batch processing using *work sharing*. Work-sharing databases reduce the total processing time by exploiting overlapping computations across the queries in the batch. However, large numbers of heavyweight shared operators and the fact that everything is recomputed from scratch can violate stringent response time requirements.

Figure 5.1 depicts processing time for a large query batch[1]. Each bar represents an optimized approach for processing the batch. Both query-at-a-time (QaT) reuse and work sharing fail to provide fast responses. Reuse eliminates computations by precomputing joins, but suffers from concurrent outstanding processing (i.e., filters on materialized results, non-materialized

[1]The setup corresponds to Figure 5.9a with 50% budget (presented in Section 5.5.2).

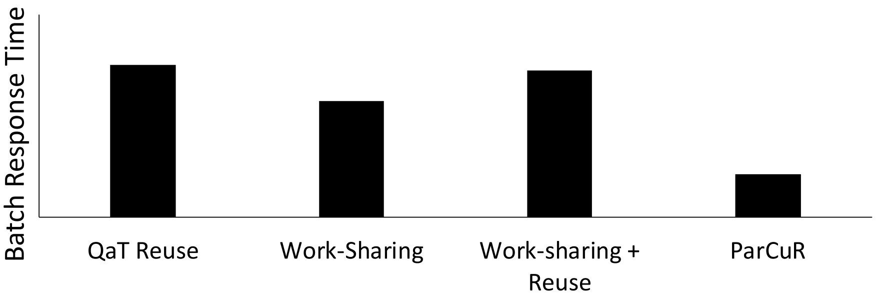


Figure 5.1: ParCuR harmonizes reuse and work sharing to speed up recurring batches joins). By contrast, work sharing mitigates the impact of concurrency and reduces the overall response time but suffers from processing heavy shared joins at runtime.

Individually, both reuse and work sharing fail to process large workloads interactively but still make complementary contributions. Thus, it is attractive to combine the two approaches to exploit their cumulative benefit. However, naively reusing materialized results in a work-sharing database as we would in a query-at-a-time database brings limited benefit and can even degrade performance ("Work-sharing + Reuse" in Figure 5.1). Reuse in a work-sharing environment is ineffective because i) it eliminates upstream shared operators only when their results are not required by any downstream computation, ii) as it rewrites only queries that the used materialized results subsume, mismatching (i.e., non-subsumed) queries may re-compute, fully or partially, the reused results, hence decreasing benefit – mismatches become increasingly likely as concurrency is increased, especially during workload shifts – and iii) it severely amplifies processing for shared filters.

To enable interactive responses for large parameterized batches, we introduce ParCuR (Partition-Cut-Reuse), a novel framework that harmonizes reuse with work sharing. To address the limited effectiveness of reuse in work-sharing environments, ParCuR adapts materialization and reuse techniques across three axes:

Cut: ParCuR addresses the subexpression selection problem [51, 136] in a work-sharing environment. Work sharing violates the assumptions of traditional subexpression selection; thus, existing solutions fail to minimize processing. To increase the impact of reuse, ParCuR introduces novel materialization and reuse policies that make decisions based on the eliminated shared operators in the work-sharing setup. As eliminating each shared operator depends on downstream decisions, ParCuR introduces the concept of cuts. Cuts represent sets of materialized subexpressions that act synergistically in eliminating more upstream operators. The policies use cuts when evaluating which results to materialize or reuse. ParCuR proposes approximation algorithms for materialization as well as a cost-based reuse algorithm that maximizes net benefit (i.e., eliminated processing time minus filtering overhead).

Reuse: ParCuR focuses on making reuse efficient, and thus it is imperative to reduce the high processing time for shared filters. To this end, it builds and uses access methods on materialized subexpressions. By building, based on frequent predicates, access methods on materialized subexpressions and by using the access methods at runtime, ParCuR evaluates frequent filters for one batch of tuples at a time, thus amortizing the required processing.

Partition: To increase the usability of materialized results in case of mismatches, e.g., during workload shifts, ParCuR uses partial reuse. ParCuR uses the fragments of materialized results that are relevant for each query batch at hand and performs any additional recomputation only as needed, thus relaxing the subsumption constraint; it eliminates shared operators for all queries for the data ranges that materialized results cover. To efficiently identify and access the relevant fragments and the base data for the recomputation, ParCuR uses partitioning and decides materialization and reuse at the granularity of partitions. Nevertheless, materializing at partition-granularity creates a dependency between the storage footprint and the partitioning scheme. Partition-granularity materializations contain tuples that are rarely useful because they are filtered out by the predicates of the corresponding queries. Hence, to maximize reuse while minimizing footprint, ParCuR introduces a novel partitioning algorithm that clusters together data that are accessed by similar subexpressions and hence benefit from the same materializations.

ParCuR incorporates the above techniques in a two-phase framework: i) an offline tuner that optimizes ParCuR's state (i.e., partitions, materialized results, access methods) for a target workload and ii) an online executor that, by exploiting the available state, minimizes the processing time for query batches arriving at runtime. On the one hand, the tuner chooses the partitioning scheme, then chooses results to materialize and, finally, builds access paths over the materialized results. On the other hand, the executor processes incoming query batches using a partition-oriented execution model. For each partition, it selects the materialized results to reuse such that they minimize processing time and uses the available access methods to reduce filtering overhead. By adapting and exploiting the available state, ParCuR makes reuse efficient and effective in work-sharing environments. As Figure 5.1 demonstrates, ParCuR drastically reduces batch response time. The experiments show that ParCuR outperforms work sharing by 6.4× and 2× in the SSBM and TPC-H benchmarks, respectively.

We make the following contributions:

- Choosing intermediates to materialize using QaT heuristics is ineffective and uses the storage budget suboptimally. We propose a family of materialization policies that, by taking into account the workload's sharing opportunities, improve time savings for the same budget.
- Work-sharing decisions and access patterns affect the benefit of reusing materialized results. We propose a cost-based optimization strategy that chooses when and which results to inject into each batch's global plan such that response time is minimized.
- Reusing materialized results in work-sharing databases is not straightforward; naive injection into global plans can increase response time considerably. Instead, we propose

that materialization should always be accompanied by access methods that enable data skipping and filter skipping.

- Increasing the usability of materialized results and reducing the penalty from mismatches requires partial reuse. Partition-level materializations coupled with a partition-oriented execution model enable efficient partial reuse at the expense of storage overhead. Hence, we propose a novel partitioning scheme that maximizes reuse while minimizing redundant materialization by aligning partition boundaries to workload patterns.

5.1 Reuse in Shared Execution

We provide an overview of the challenges in reusing materializations during shared execution. We first motivate reusing materializations to reduce recomputation in Section 5.1.1, then highlight the performance pitfalls that materializations introduce when combined with work sharing in Section 5.1.2, and finally outline the solutions that ParCuR proposes.

5.1.1 Recomputation Bottleneck

When using work sharing, processing more queries increases the response time sublinearly, and thus, the total processing time is reduced compared to QaT execution. However, for each submitted query batch, global plan execution always starts from a clean slate. Data flows from the input tables to each query's output, and all shared operators of the global plan are fully processed from scratch.

Recomputation of previously "seen" expressions can be critical as the additional processing for handling query-sets renders shared operators particularly time-consuming. For example, shared filters and joins require one or more query-set intersections, the cost of which is increased as a function of the number of queries. Furthermore, shared filters are not simple comparisons but are implemented as joins with predicates using the predicate indices.

All in all, as global plans often consist of tens of operators, processing accumulates and prevents providing results within a tight time window. Therefore, to offer interactivity, we need to reduce the required computations for each batch.

5.1.2 Pitfalls of Combining Reuse and Work Sharing

Analytical databases often reduce runtime computations by reusing precomputed results. However, we observe that using materializations in work-sharing environments exhibits a set of properties that have not been studied before and which make data reuse inefficient. Namely, these properties are: i) *shared cost*, ii) *synergy*, iii) *filter amplification*, and iv) *risk of miss*.

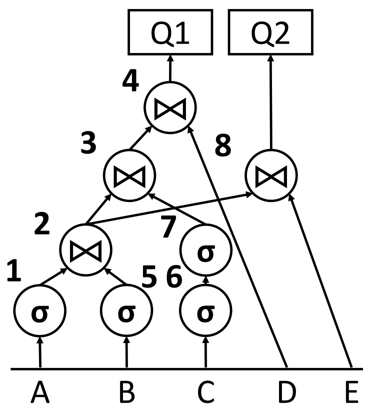


Figure 5.2: Motivational example: work sharing introduces novel challenges for reuse

We elaborate on each of these properties. For ease of presentation, we describe them using a running example of a batch with the following two queries:

```
Q1: SELECT SUM(X) FROM A,B,C,D WHERE expr1
Q2: SELECT SUM(X) FROM A,B,E WHERE expr2
```

Figure 5.2 shows the global plan for Q1 and Q2. For ease of reference, each operator is labeled with a number.

Shared cost: *QaT cost models are inaccurate in work-sharing environments.* Work sharing affects both which operators reuse eliminates and their relative importance. On the one hand, reuse eliminates upstream operators only as long as their results are not required by other remaining downstream operators. For example, reusing the results of operator 4 eliminates operators 3 and 4, but operator 2 is still required for Q2. On the other hand, work sharing across queries diminishes the importance of frequency of occurrence for operators; the savings depend more on the total number of Data-Query tuples processed by the shared operator rather than the number of participating queries. This is contrary to the assumptions of traditional cost models for materializing intermediates, which assume that reuse eliminates all upstream costs and which simply add up the benefit for each affected query.

Synergy: *The benefit of individual materializations is amplified.* Materialization decisions affect each other's results differently than they do in single-query plans. Reuse in single-query plans results in diminishing returns. For example, reusing the results of operator 4 in the original plan eliminates joins 3 and 4, whereas reusing the same results in a rewritten plan that already uses operator 3 only eliminates join 4. This observation is critical for the design of heuristic materialization algorithms that are based on submodularity. However, diminishing returns are not necessarily the case in global plans. Consider the example where we reuse results for operators 3 and 8. We observe a counter-intuitive effect: individually, they eliminate one join each, but together the benefit is amplified, and they eliminate 3 joins. This effect, which we refer to as *synergy*, marks a departure from traditional materialization and reuse.

Filter amplification: *Shared filters over materializations dominate the total processing time.* When injecting a materialization into a global plan, the work-sharing database needs to process filters from all the tables participating in the computation. For example, if the database reuses the results of the subquery corresponding to operator 4, the global plan needs to process 4 shared filters from tables A, B, and C. Similarly, if it reuses the results corresponding to operator 8, the global plan needs to process 2 shared filters from A and B. Then, the processing time for filters is amplified for two reasons: i) materializations can have a significantly larger cardinality than small dimension tables, and ii) filters must process every materialization where the corresponding table participates (e.g., filters from A are processed on the materializations of both 4 and 8). In some cases, reuse deteriorates performance compared to processing the batch from scratch using work sharing.

Risk of miss: *The probability that the materialization covers all accessed data decreases with the number of queries.* Reuse typically requires that the materialization fully subsumes the subquery that it eliminates. Similarly, the materialization needs to subsume all participating queries to eliminate subplans in global plans. For example, eliminating operator 2 by reusing its result requires that both Q1 and Q2 can be answered using the materialized subexpression. Assume that the materialized subexpression only covers *expr1* and *expr1* defines a subset of *expr2*: then, even if Q1 is answered using the materialized subexpression, Q2 fully recomputes the shared operator's result already and thus reuse brings no benefit compared to shared execution. Requiring full subsumption for materialized subexpressions that are subject to tight predicates has a high risk of mismatch, especially in case of workload shifts.

5.1.3 Harmonizing Reuse and Work Sharing

To significantly reduce their runtime computations, work-sharing databases need to address inefficiency in reuse. In this work, we harmonize work sharing and reuse: we redesign, based on the above-mentioned properties, the techniques for materializing and reusing precomputed results such that we maximize eliminated computations and minimize reuse overhead. Harmonization takes place across three axes: i) materialization and reuse policies which address shared cost and synergy, ii) access methods for materializations which address filter amplification, and iii) partial reuse, which addresses the risk of miss.

Materialization and reuse policies: Due to shared cost and synergy, algorithms for selecting materializations or injecting materializations into plans make suboptimal decisions. Work sharing renders their cost models inaccurate and violates assumptions that they make, such as submodularity. Hence, harmonization requires novel materialization and reuse policies that, by taking into account both shared cost and synergy, select materializations that bring higher processing time reduction, given the same budget. In this work, we introduce a methodology that evaluates cost reduction using i) the eliminated shared cost in global plans and ii) the novel concept of cuts, that is, sets of materializations that exhibit synergy. We first formulate the problem of choosing materializations for a target workload (Section 5.2.2). The formulation

	Shared cost	Synergy	Filter amplification	Risk of miss
Materialization policy	✓	✓		
Access methods			✓	
Partitioning				✓
Reuse policy	✓	✓		
Data & Filter skipping			✓	
Partition-oriented execution				✓

Table 5.1: Mechanisms that ParCuR uses to harmonize reuse and work sharing. Each mechanism in ParCuR is either offline (brown rows) or online (purple rows) and addresses challenges for reuse in work-sharing environments (columns)

is a variant of the subexpression selection problem [51, 136]. We show that the materialization problem can be reduced, using cuts, into a Submodular Cover Submodular Knapsack (SCSCK) problem [49], for which there exists a family of approximation algorithms. Afterward, we address the problem of selecting which materializations to reuse and when in shared execution (Section 5.3.3). We propose a reuse optimization pass that, at runtime, injects into a global plan the materialized subexpressions that maximize the difference between eliminated computation and the overhead of accessing and filtering the selected subexpressions.

Access methods: Filter amplification limits the applicability of reuse, as it shrinks the net benefit and, when filtering overhead is high enough, it deteriorates performance. Efficient reuse requires that the processing time for shared filters over materializations is decreased. We reduce processing time for filters using suitable access methods for the workload at hand. By building and using access methods, ParCuR enables shared execution to evaluate shared filters over one block of tuples at a time instead of processing them on a tuple-by-tuple basis, and thus to amortize the overhead. First, we discuss building access methods for the target workload through partitioning (Section 5.2.3). Then, we discuss using the created access methods to eliminate filters at runtime (Section 5.3.1). In this step, alternative techniques such as SH_2O are also applicable.

Partial reuse: The risk of a miss also limits the applicability of reuse under strict subsumption requirements. For this reason, ParCuR opts for a different paradigm: to exploit available materializations for the parts of the data that they cover. Hence, it uses partial reuse. During execution, ParCuR can answer each query by combining computations from parts of different materializations and even from parts of the base data. Computations on disjoint parts of the data that consist of operators such as filters, projections, join probes, and aggregations can be combined to produce the full result [129]. Our insight is that, to enable composable computations from different parts of data, planning and execution need to take place at partition granularity. In addition, the reusability of materializations is maximum when they fully cover the data for a set of partitions. Then, for the corresponding partitions, they always subsume the partition-local computations for the matching queries and can eliminate the shared operators in the corresponding data range. Hence, ParCuR performs materialization and reuse at partition granularity. The materialization policy selects for each materialization a set of partitions to fully cover and injects materializations into each partition's global plan at runtime. However, this scheme creates a dependency between partitioning and the storage overhead for covering the target workload; storage overhead is minimum when partition boundaries are aligned with the queries that the materializations subsume. Thus, due to this dependency, data needs to be partitioned such that each partition's tuples are required by the same computations, which, in turn, require the same materializations. We propose the metric of homogeneity to capture the similarity of computations across each partition's tuples. ParCuR introduces a partitioning scheme (Section 5.2.1) that, by splitting data such that homogeneity is maximized, maps each computation to the data that it concerns and reduces wasteful materialization. ParCuR uses the selected partitions at runtime in a partition-oriented execution model (Section 5.3.2) to enable partial reuse and achieves cost savings that are proportional to the overlap between the runtime and tuning workload.

Table 5.1 summarizes the above solutions. It maps each solution to the challenge that it addresses. Furthermore, it classifies each solution based on when it takes place: offline (highlighted in brown) or online (highlighted in purple). In the next section, we show how we combine the above solutions into a comprehensive framework, ParCuR.

5.1.4 Putting It All Together

We present ParCuR (Partition-Cut-Reuse), a framework that enables shared execution to effectively take advantage of materialized subexpressions. To do this, ParCuR adapts subexpression selection [51] and reuse to work-sharing environments by combining the solutions in Table 5.1. In this section, we present ParCuR's architecture and workflow.

ParCuR's architecture comprises two parts: the *tuner*, and the *executor*. The tuner operates offline. It analyzes a target workload made of historical query batches and adapts the framework's state by employing the ParCuR's offline mechanisms: i) it partitions the data based on the access patterns of the target workload's queries, ii) it materializes a set of subexpressions for the given partitions, and iii) it builds new access methods for the materialized subexpressions using finer-grained partitioning. Then, given the available partitioning, materialized subexpressions, and access methods, the executor processes each query batch arriving at runtime: i) it performs shared execution at the level of the available partitions, ii) it decides when and where to reuse materialized subexpressions for each partition, and iii) it uses the available access methods to reduce filter costs using data and filter-skipping. Figure 5.3 illustrates the end-to-end workflow for both the offline tuner and the online executor. Note that the query batches processed at runtime can be arbitrarily different from historical batches both in terms

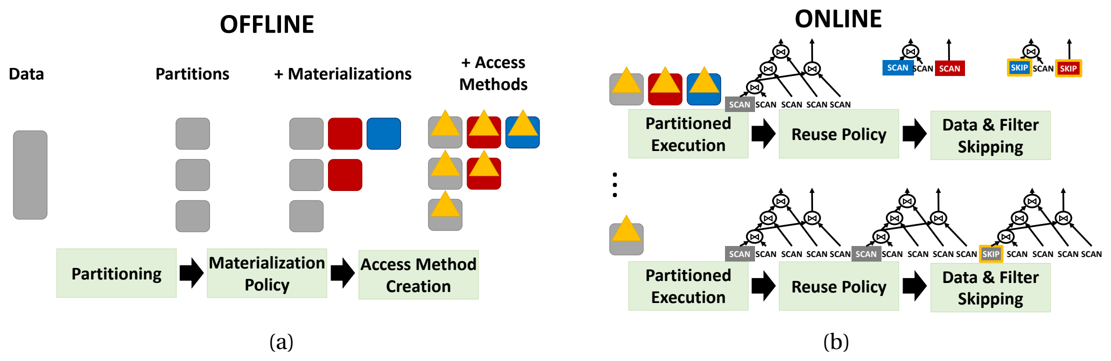


Figure 5.3: ParCuR's workflow in a) the offline tuner and b) the online executor

of access patterns and global plans. In all cases, ParCuR opportunistically uses the existing state to reduce the response time of runtime batches.

Despite being decoupled, the tuner and the executor cooperate to address the challenges of reusing materialized subexpressions in work-sharing environments. The solution to all challenges requires mechanisms at both the tuner and the executor. First, both materialization and reuse decisions take into account the shared cost and synergy. Second, the creation of access methods is combined with a filter-skipping mechanism to reduce filter costs. Third, workload-aware partitioning is coupled with the partition-granularity materialization and a partition-oriented execution model to enable partial reuse, thus making the cost of miss proportional to the mismatch. We elaborate on each mechanism in Sections 5.2 and 5.3.

5.1.5 ParCuR's Scope

This work focuses on proposing mechanisms that, by harmonizing work sharing and reuse, produce effective solutions to the offline tuning problem and efficiently process runtime batches. It assumes that the target historical workload is known a-priori and that tuning takes place only once during initialization. Building a fully adaptive framework, where the tuner continuously or sporadically reconfigures the database's state based on the current workload, e.g., by physically reorganizing data using cracking [47, 57], choosing materializations online using recycling [48, 81, 117] or running a tuning process in the background [104], is, on its own, a rich topic that involves several design decisions that require additional assumptions about the target application and lies beyond the scope of this work.

ParCuR specifically addresses the problem of subexpression selection in a work-sharing environment. Therefore, only intermediate results that occur in the global plans of historical batches are candidates for materialization. This approach is a generalization of semantic caching and constitutes a constrained form of view selection, which considers intermediate results that actually occur in plans as well as in alternative plans. ParCuR favors subexpression selection over view selection due to scalability: view selection involves a significantly larger search space of possible materializations, which it needs to evaluate for a very large search space of alternative global plans due to the effect of shared costs and synergy.

Queries: $\sigma_{X<50}$(A⋈B⋈C), $\sigma_{30\leq X<70}$(A⋈B⋈D)

e_1	e_2	e_3	e_4	e_5	e_6
A	A⋈B	A⋈C	A⋈D	A⋈B⋈C	A⋈B⋈D

(a) Subqueries in a two-query batch

Tuples with X<30

1	2	2	0	3	0

Tuples with 30≤X<50

1	2	2	2	3	3

(b) Subquery vector for tuples in $[0,30)$, $[30,50)$

Figure 5.4: Two-query example for subquery vectors

5.2 Tuning ParCuR's state

The tuner adapts ParCuR's state. By analyzing a target workload that consists of a sequence of query batches, the tuner repartitions the data, materializes a set of selected subexpressions, and builds access methods on the materialized subexpressions. Tuning takes place offline, before runtime. After tuning is done, the partitions, the materialized subexpressions, and the access methods are exposed to the executor, which uses them to eliminate recurring computation in subsequent query batches that arrive at runtime.

In this section, we present each of the steps in the tuner's workflow. Each step's output is the input for the next step in line: partitioning chooses the boundaries for materializing subexpressions, and the materialization policy selects the subexpressions on which to build access methods. We first present the partitioning algorithm (Section 5.2.1), then introduce the materialization policy (Section 5.2.2), and finally discuss access method construction (Section 5.2.3).

5.2.1 Workload-driven Partitioning

The first step of ParCuR's tuner is to partition the data in a way that maximizes the utility of subsequent materializations. To differentiate between this partitioning and any additional data reorganization for building access methods, we name the first step's partitioning as *1st-level partitioning* and any further partitioning as *2nd-level partitioning*.

Assuming a recurring workload, subqueries, and especially join subexpressions, are expected to repeat across batches. This creates reuse opportunities that significantly reduce processing costs: at some point, ParCuR decides to materialize subexpressions that it expects to reuse in subsequent batches at the granularity of data partitions; then, follow-up queries can directly reuse the subexpressions for the corresponding partitions without recomputing them. However, for partition-granularity materialization to be budget-efficient, all tuples should be processed by similar query patterns, i.e., most of their downstream computation should be the same. Therefore, ParCuR employs a partitioning scheme that clusters tuples according to query patterns and materializes subexpressions for each partition independently.

Such a partitioning scheme offers three benefits: i) if the query patterns remain the same, materialized subexpressions are almost fully reused, and space budget is not wasted, ii) materialization is specialized for the sharing decisions of each partition's query pattern, and iii) for the case of *partial reuse* during a workload shift, performance degradation becomes proportional to the magnitude of the shift.

To cluster together data that is processed by similar query patterns, we keep track of processing history for a sample of tuples by maintaining a *subquery-vector* for each tuple. We consider all possible subqueries $e_1, e_2, \ldots, e_m$, that appear in a set of historical batches and mark to which of them each tuple belongs. By *subqueries*, we mean all the join subexpressions (and their reorderings) that exist in each batch and involve the fact table. For example, considering a batch with two queries, $A \bowtie B \bowtie C$, $A \bowtie B \bowtie D$ and A as the fact table, leads to the subqueries depicted in Figure 5.4a. We represent subexpressions in different batches as separate subqueries because they do not actually co-occur. Using subqueries is advantageous as it exposes similarities that do not depend on a specific execution plan and naturally represents co-occurrence in the same batch.

We then use the subquery-vectors in order to formulate a tuple-clustering problem based on homogeneity. We assume a matrix W, where the i_{th} row of it corresponds to the subquery-vector of the i_{th} tuple: If at least one query with subquery e_j accesses the i_{th} tuple, $W_{i,j} = w(e_j)$, where $w(e_j)$ is a weight assigned to e_j. Otherwise, $W_{i,j} = 0$. In our implementation, to increase the relative importance of larger subqueries to homogeneity, we set $w(e_j) = |e_j|$, where $|e_j|$ is the number of tables participating in e_j. Alternatives assignments can also achieve a similar result.

Given a set of tuples T, we formally define homogeneity as:

$$H(T, W) = \sum_{t \in T} h(t, T, W) \text{ where } h(t, T, W) = \frac{\sum\limits_{j=1}^{m} W_{t,j}}{max(\sum\limits_{j=1}^{m} w(e_j) \times u(\sum\limits_{t \in T} W_{t,j}), 1)}$$

where $u(x)$ is the step function with $u(x) = 1$ when $x > 0$ and 0 otherwise. Homogeneity assigns a score $h(t, T, W)$ to each tuple in T, based on the subqueries that access the tuple, and is defined as the sum of scores. The score $h(t, T, W)$ is the sum of weights for the subqueries that access tuple t over the sum of weights for the subqueries that access at least one tuple in T.

The score is maximum (i.e., $h(t, T, W) = 1$) if all the subqueries that access at least one tuple in T also access t. The intuition is that homogeneity is maximum when all tuples in T are accessed by the exact same subqueries. In that case, the utilization of materializations is also maximum; assuming that a subquery's results are materialized and that the historical batches recur as is, reuse exploits all the tuples in the materialization, and no tuple is redundant.

Homogeneity-based partitioning is defined as the problem of finding the partitions $\{p_1^*, p_2^*, \ldots, p_n^*\}$ that maximize the aggregate homogeneity, i.e.:

$$\{p_1^*, p_2^*, \ldots, p_n^*\} = \underset{\{p_1,\ldots,p_n\}}{\arg\max} \sum_{i=1}^{n} H(p_i, W) \ s.t. \ \forall p_i \ |p_i| \geq PS_{min}$$

where PS_{min} is the minimum allowed partition size. Homogeneity-based partitioning finds partitions such that, in each partition, the tuples are accessed by almost the same set of subqueries, and thus, barring a workload shift, the utilization of materializations is high. The partition size constraint ensures that the solution avoids the trivial optimal solution where each tuple forms its own partition.

To efficiently compute a solution to homogeneity-based partitioning, both top-down and bottom-up approaches are applicable. We use a space-cutting approach that, similar to [130], forms a tree of cuts in the space of table attributes. Each internal node corresponds to a logical subspace of the table and contains a predicate based on which this subspace is further split: the left child corresponds to the data that satisfies the predicate, whereas the right child to the data that does not. Finally, the leaves of the tree correspond to data partitions, which are the quanta for materialization. The advantage of the space-cutting approach is that it enables routing queries to required partitions based on the predicates of the splits and the queries.

To solve the partitioning problem, we use a greedy algorithm, which we show in Algorithm 5. The algorithm runs on a uniform sample of the tuples to keep runtime monitoring overhead low. ParCuR computes the sample's query pattern matrix by monitoring data accesses across batches and by recording the vector of subqueries for the sample's tuples. When triggered, the greedy algorithm computes the change in the objective function for each candidate cut, that is, a predicate that intersects with the partition at hand (lines 5-9), and finds the locally optimal cut that maximizes the aggregate homogeneity (lines 12-14). Then, the space is partitioned based on the locally optimal cut, and the greedy algorithm is recursively invoked for the two children subspaces and the respective sample tuples (lines 16-19). The greedy algorithm stops when either the relative improvement from the locally optimal cut drops below a threshold, which we set at 1% (lines 20-21), or all candidate cuts violate the minimum partition size for resulting partitions (lines 9-10).

Homogeneity-based partitioning results in more efficient use of the storage budget compared to data access-based partitioning schemes, such as Qd-tree [130]. Consider the following example[2]. Let $A \bowtie B$, $A \bowtie C$ take 10GB to materialize each. Also, let attribute $A.X$ be

[2]Note that, with this example, we focus on the conceptual difference between homogeneity-based and data

Algorithm 5: Homogeneity-based Partitioning

```
Function PARTITION($partition, W, cuts, PS_{min}$):
    $output = null$;
    $best = null$;
    $bestScore = null$;
    $score = H(partition.sample, W)$;
    for $cut \in cuts$ do
        if $intersects(partition, cut)$ then
            $tp, fp = getPartitions(partition, cut)$;
            if $tp.size < PS_{min}$ or $fp.size < PS_{min}$ then
                $continue$;
            $curr = H(tp.sample, W) + H(fp.sample, W)$;
            if $best == null$ or $curr > bestScore$ then
                $best = cut$;
                $bestScore = curr$;
    if $best == null$ and $bestScore > 1.01 \times score$ then
        $tp, fp = getPartitions(partition, best)$;
        $tres = PARTITION(tp, W, cuts, PS_{min})$;
        $fres = PARTITION(fp, W, cuts, PS_{min})$;
        $output = Node(best, tres.fres)$;
    else
        $output = Leaf()$;
return $output$;
```

uniformly distributed in $\{0, \ldots, 99\}$. The input workload consists of a batch of three queries:

```
Q1: SELECT * FROM A,B WHERE A.Z=B.Z AND X < 60
Q2: SELECT * FROM A,C WHERE A.W=C.W AND X < 50
Q3: SELECT * FROM A,C WHERE A.W=C.W
```

Suppose that we run both homogeneity-based partitioning and Qd-tree with minimum partition size $0.2 \times |A|$. Both algorithms perform one cut, either at $X = 50$ or at $X = 60$.

Qd-tree uses a cost function (lower is better) to choose a cut:

$$C(\{p_1, \ldots, p_n\}) = \sum_{i=1}^{n} (|p_i| \times \sum_{q \in Q} S(p_i, q))$$

where Q is the set of queries and $S(p_i, q) = 1$ if query q accesses partition p_i and 0 otherwise. Splitting at $X = 50$ results in cost $|A| + 0.5 \times |A| + |A| = 2.5 \times |A|$, whereas splitting at $X = 60$ results in cost $0.6 \times |A| + |A| + |A| = 2.6 \times |A|$. Hence, the optimal cut is at $X = 50$.

access-based partitioning. It is also possible to make QaT partitioning algorithms, such as Qd-tree, to choose suboptimal partitioning by tweaking the number of occurrences, and hence the relative importance, of each query. Furthermore, QaT partitioning algorithms are ill-defined for shared data access and can be ineffective if applied at batch level e.g. a batch may access the full data even when each individual query requires a small fraction of the data. However, the example shows that the partitioning is suboptimal even in the absence of such effects.

Homogeneity-based partitioning uses the objective function above (higher is better). Splitting at $X = 50$ results in a sum $\frac{5\times0.5\times|A|}{5} + \frac{5\times0.1\times|A|+3\times0.4\times|A|}{5} = 0.84\times|A|$, whereas splitting at $X = 60$ results in a sum $\frac{5\times0.6\times|A|}{5} + \frac{3\times0.4\times|A|}{3} = |A|$. Hence, the optimal cut is at $X = 60$.

To choose partition-granularity materializations that completely cover the three queries, the two splits require different storage budgets. If the data is partitioned at $X = 50$, $A \bowtie B$ and $A \bowtie C$ need to be materialized in both $[0,49]$ and $[50,99]$, which requires a budget of 20GB. By contrast, if the data is partitioned at $X = 60$, $A \bowtie B$ needs to be materialized only in both $[0,59]$ and $A \bowtie C$ needs to be materialized in both partitions, which requires a budget of 16GB. Hence, homogeneity-based partitioning results in more effective utilization of the budget.

5.2.2 Materialization Policy

1st-level partitioning assumes that query patterns represent the overall workload and thus recur in future batches. To eliminate recomputation in such cases, ParCuR materializes subexpressions in a per 1st-level partition basis.

Due to the interference between reuse and work sharing, a global plan-aware materialization policy is required. Furthermore, in ParCuR the policy should address the existence of partitions that process different query patterns. Therefore, ParCuR relies on a new formulation of the subexpression selection problem, which is: i) sharing-aware, ii) works on *partition-wise global plans*, and iii) where the optimal solution is expected to differ from the one of the classical problem. We call this new problem *Multi-Partition Subexpression Selection for Sharing (MS3)*. We first define the *Historical Workload Graph*, which is the input of MS3, and subsequently MS3 itself.

Definition 5.2.1 (Historical Workload Graph). — Given a fact table T, a partitioning $\{p_1, p_2, .., p_n\}$ of T, and batches $\{Q_1, Q_2, \ldots, Q_m\}$, the historical workload graph G is a graph composed of connected components $G_{i,j}$, $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, m\}$, where $G_{i,j}$ is the global plan for Q_j over p_i. In the global plan, nodes represent operators (including a pseudo-operator for T), and edges represent producer-consumer relationships.

Definition 5.2.2 (MS3). — Let $R(c)$ be the maximum cost reduction that reuse can incur when executing the global plans of the historical workload graph G with an available set of materialized subexpressions c, and $B(c)$ the budget required for materializing c. If B is the total memory budget, MS3 is defined as:

$$\max_{c} R(c), \mathit{s.t.}\!: B(c) \leq B$$

MS3 is a hard problem, and hence it is time-consuming to compute a tractable exact solution. To solve it, we first prove a reduction to *Submodular Cover Submodular Knapsack (SCSK)* problem [49] and then show how we can use approximate algorithms for SCSK to choose to materialize a set of expressions that achieve a high cost reduction with approximation guarantees.

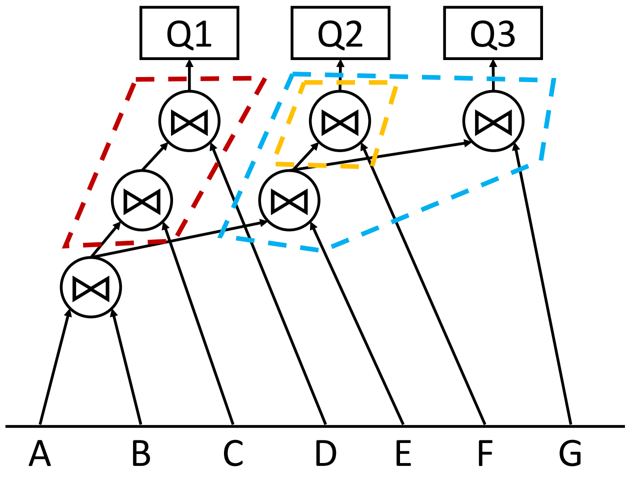


Figure 5.5: Example showing the effect of reuse on a global plan

Reduction to SCSK

Let U be a set, and $f, g : 2^U \rightarrow \mathbb{R}$ be two submodular functions[3], then SCSK is the optimization problem

$$\max_{S \subset U} g(S),\ s.t.\ f(S) \leq B$$

In order to reduce MS3 to SCSK, cost savings in MS3 should be submodular, i.e., adding more materialized subexpressions should result in diminishing returns. While this is true in QaT execution, where each materialization reduces the marginal benefit of other conflicting materializations, it does not hold in shared execution. We observe that shared execution benefits more from materializations in the same path of the global plan, where synergy increases cost savings.

The key idea for reducing MS3 to a submodular optimization problem is to consider subexpressions to materialize in groups. We notice that computing cost savings for groups gives us more accurate estimates for the eliminated upstream computations. In addition, the synergy between groups is always captured by their super-group, that is, a group that contains their union.

We formulate useful groups of materializations by introducing the concept of *cuts*. Intuitively, in a given global plan, a cut is a set of subexpressions that, if materialized, eliminate all upstream operators between (inclusive) the operators that produce them and a common ancestor, the *anchor*. For example, in Figure 5.5, the cut composed of $A \bowtie B \bowtie E \bowtie F$ and $A \bowtie B \bowtie E \bowtie G$ eliminates the upstream node $A \bowtie B \bowtie E$, which is also the anchor. Formally, we define cuts and anchors as follows:

[3]Submodularity formalizes diminishing returns. Specifically, a function h is defined as submodular if $S \subset S' \Rightarrow h(S \cup \{s\}) - h(S) \geq h(S' \cup \{s\}) - h(S')$.

Definition 5.2.3 (Cuts and anchors)**.** — *Let G be the historical workload graph. A set of nodes $c \subset V$ is defined as a cut with respect to anchor $a \in V$ if:*

- *a is an ancestor for every $v \in c$.*
- *Every descendant of a is either i) an ancestor of at least one node $v \in c$, or ii) a descendant of exactly one node $v \in c$.*

We represent the set of all cuts in G as $CUTS(G)$, and for all $c \in CUTS(G)$ we define $BC(c,a)$ as the nodes between (inclusive) the cut's nodes and anchor a. The shorthand $BC(c)$ implies using the minimal anchor (i.e., an anchor whose predecessor is not an anchor for c).

Choosing cuts to materialize so as to maximize the eliminated cost in their BC sets is related to but not identical to $MS3$. The cost in BC sets is not always equal to the cost reduction from the same materializations in $MS3$ because $MS3$ implicitly includes the savings from super-cuts, that is, the union of smaller materialized cuts. However, we prove that solutions in the *cut selection* problem can be enriched such that they are both solutions to *cut selection* and $MS3$ with equal savings. Furthermore, we prove that *cut selection* is an SCSK problem, and thus we can solve it using approximate algorithms. Based on these two properties, *cut selection* gives a solution to $MS3$ with a better or equal approximation factor than the one given for SCSK. In the following paragraphs, we formally define *cut selection* and prove the mentioned properties.

First, we introduce some required notation:

Definition 5.2.4 (Domain and Enrichment)**.** — *Let S be a set of cuts to materialize. We define the domain of S as the set*

$$d(S) = \{v.subquery | v \in (\bigcup_{c \in S} c)\}$$

and the enrichment of S as the set

$$e(S) = \{c | c \in CUTS(G)\ and\ \forall v \in c(v.subquery \in d(S))\}$$

The domain represents which results S materializes, and the enrichment represents all cuts that are materialized by materializing S.

Definition 5.2.5 (Cost Reduction and Budget)**.** — *Let S be a set of cuts. Furthermore, let $cost(op)$ be the processing cost for an operator op in the global plan. Then, we model the cost reduction due to materializing S as:*

$$\bar{R}(S) = \sum_{op \in O} cost(op),\ where\ O = \bigcup_{c \in S} BC(c)$$

and the required materialization budget as

$$\bar{B}(S) = \sum_{v \in d(S)} B(\{v\})$$

$\bar{R}(S)$ is equal to the cost of operators O that are eliminated by materializing the cuts in S. Each cut eliminates the shared operators between the cut and the minimal anchor, and, by definition, computing O as the union of operators accounts for overlaps between the operators that are eliminated by different cuts.

$\bar{B}(S)$ is equal to the total budget required for materializing the results of the cuts in S. $d(S)$ is by definition the results that S materializes.

Definition 5.2.6 (Reduced Workload Graph)**. —** *Let S be a set of cuts. We define the reduced workload graph of S as*

$$G(\emptyset) =< V(\emptyset), E(\emptyset) >= G$$

$$G(S) =< V(S), E(S) >= G[V - \bigcup_{c \in S} BC(c)]$$

where $G[V']$ is the induced subgraph of G for vertices V'.

The reduced workload graph represents the global plans for the historical query batches after materializing and reusing the cuts in S.

We define *cut selection* problem as follows:

Definition 5.2.7 (Cut Selection)**. —** *Cut selection is defined as the optimization problem of finding a set of cuts S such that:*

$$max \bar{R}(S), s.t.: \bar{B}(S) \leq B$$

Using the above definitions, we prove the following theorems:

Theorem 3. Cut selection is an SCSK problem.

Proof. We prove that $\bar{R}$ and $\bar{B}$ are submodular. For a set of cuts S and a cut c, it holds that:

$$\bar{R}(S \cup \{c\}) - \bar{R}(S) = \sum_{op \in O(S)} cost(op) \ s.t. \ O(S) = BC(c) \cap V(S)$$

and

$$\bar{B}(S \cup \{c\}) - \bar{B}(S) = \sum_{m \in M(S)} B(\{m\}) \ s.t. \ M(S) = d(\{c\}) - d(S)$$

Let S, S' be two sets of cuts such that $S \subset S'$. Then,

$$\bar{R}(S \cup \{c\}) - \bar{R}(S) = \sum_{op \in O(S)} cost(op)$$

and

$$\bar{R}(S' \cup \{c\}) - \bar{R}(S') = \sum_{op \in O(S')} cost(op)$$

However, $V(S') \subset V(S)$ *and thus* $O(S') \subset O(S)$. Therefore,

$$\bar{R}(S \cup \{c\}) - \bar{R}(S) \geq \bar{R}(S' \cup \{c\}) - \bar{R}(S')$$

Similarly,

$$\bar{B}(S \cup \{c\}) - \bar{B}(S) = \sum_{m \in M(S)}$$

and

$$\bar{B}(S' \cup \{c\}) - \bar{B}(S') = \sum_{m \in M(S')}$$

Then, $d(S) \subset d(S')$ *and thus* $M(S') \subset M(S)$. Therefore,

$$\bar{B}(S \cup \{c\}) - \bar{B}(S) \geq \bar{B}(S' \cup \{c\}) - \bar{B}(S')$$

Therefore, both f and g are submodular. □

Theorem 4. If S is a solution to cut selection, then $e(S)$ is also a solution to cut selection with $\bar{R}(e(S)) \geq \bar{R}(S)$.

Proof. By definition, $S \subset e(S)$.

$$\bar{B}(e(S)) = \sum_{m \in d(S)} B(\{m\}) = \bar{B}(S)$$

Therefore $\bar{B}(e(S)) \geq B$ and $e(S)$ is a solution to cut selection.

Furthermore,

$$\bar{R}(e(S)) = \sum_{op \in (\bigcup_{c \in e(S)} BC(c))}$$

However, $S \subset e(S)$ and thus $\bigcup_{c \in S} BC(c) \subset \bigcup_{c \in e(S)} BC(c)$. So,

$$\bar{R}(e(S)) \geq \bar{R}(S)$$

□

Theorem 5. For every $e(S)$, it holds that $R(d(S)) = \bar{R}(e(S))$

Proof. Let S be a set of cuts. We represent the eliminated operators in the original MS3 problem when S is materialized as $t(S)$. Formally, $t(S)$ is the set of all nodes whose operators produce $d(S)$ or all their successors belong to $t(S)$. Then:

$$R(d(S)) = \sum_{op \in t(S)} cost(op)$$

We now prove that $t(S) = \bigcup_{c \in e(S)} BC(c)$.

Let $c' \in e(S)$. Then, $c' \subset d(S)$, and $\forall v \in BC(c')$ it holds $v \in t(S)$ and thus $BC(c') \subset t(S)$. It follows that $\bigcup\limits_{c' \in e(S)} BC(c') \subset t(S)$.

Also, let $a \in t(S)$ and c_a all the descendants of a that belong to $d(S)$. Then, c_a is a cut with anchor a, as the two conditions in the definition of cuts are true: i) a is an ancestor for all nodes in c_a, and ii) assume there is a descendant of a, a', that is not a descendant of any node in c_a. Then, a' is an ancestor of at least one node in c_a because $a \in t(S)$ (otherwise, the nodes in the path from a to a' should not be in $t(S)$). Therefore, c_a is a cut, $a \in BC(c_a)$ and $t(S) \subset \bigcup\limits_{c' \in e(S)} BC(c')$.

Thus $t(S) = \bigcup\limits_{c \in e(S)} BC(c)$ and:

$$R(d(S)) = \sum_{op \in t(S)} cost(op) = \sum_{op \in \bigcup_{c \in e(S)} BC(c)} cost(op) = \bar{R}(e(S))$$

□

Approximating MS3

ParCuR's tuner chooses subexpressions to materialize by solving cut selection for historical batches. The selection process has two steps: i) the tuner constructs the workload graph and computes the cuts and their corresponding BC sets, and ii) the tuner runs an algorithm for solving the cut selection instance for the computed cuts. The subexpressions in the selected cuts are then materialized and used in subsequent batches.

The tuner currently implements two approximate algorithms for solving SCSK, greedy (Gr) and iterative submodular knapsack (ISK) [49]. We briefly present the properties of the two algorithms as presented in the work of Iyer et al. [49].

Gr: Gr is a greedy algorithm. At each step, given an existing partial solution S, it chooses the cut c with the highest marginal benefit $\bar{R}(S \cup \{c\}) - \bar{R}(S)$ that can fit in the remaining budget and adds it to the partial solution. The algorithm finishes when there are no more cuts that fit in the budget. Thus, Gr's worst-case complexity is $O(|CUTS(G)|^2)$. In our experiments, it requires few msecs to find a solution.

Gr provides an approximation factor $1 - (\frac{K_f - 1}{K_f})^{k_f}$, where

$$K_f = \max_{S \subset U} |S| \; s.t. \; f(S) \leq B$$

is the maximum solution size for the SCSK problem and

$$k_f = \min_{S \subset U} |S| \ s.t. \ f(S) \leq B \wedge f(S \cup \{j\}) > B$$

is the minimum solution size that saturates the constraint (i.e., the solution cannot be expanded by adding another element to S).

The bound suggests that Gr finds efficient solutions when the setup is such that saturating the budget requires a large number of cuts. Conversely, if a single cut can saturate the budget, the worst-case approximation factor is $\frac{1}{K_f}$. This bound is consistent with the results of our evaluation.

ISK: ISK is a fixed point algorithm. In each iteration, it solves a new Submodular Knapsack problem that computes the required budget $\bar{B}(S)$ using a modular upper bound. The upper bound function is parameterized using a set of cuts, for which it provides a tight bound. The solution of each iteration parameterizes the upper bound function for the next iteration.

In each Submodular Knapsack problem, it combines partial enumeration with greedy expansion; it chooses between $\binom{|CUTS(G)|}{3}$ candidate solutions, where each candidate solution fixes the first three cuts and chooses the rest using a greedy algorithm. At each step, the greedy algorithm chooses the cut with the highest ratio of marginal benefit to required budget $\frac{\bar{R}(S \cup \{c\}) - \bar{R}(S)}{\bar{B}(\{c\})}$.

ISK's complexity is $O(it \times |CUTS(G)|^5)$ where it is the number of iterations until it converges. Thus, for a few hundred of cuts, ISK can run for hundreds of seconds. Thus, it is significantly more time-consuming than Gr.

As it uses the upper bound in each iteration, ISK solves a problem with a tighter budget constraint. Thus, in its first iteration, it provides a constant approximation factor $1 - e^{-1}$ for the solution of the problem

$$\max_{S \subset U} \bar{R}(S) \ s.t. \ \bar{B}(S) \leq \frac{b}{K_f}$$

Subsequent iterations improve the solution and thus maintain the approximation guarantee. Furthermore, it is possible to achieve a bicriterion guarantee: running ISK with a larger budget constraint provides a constant approximation factor for problems with more relaxed constraints; this approach, however, can lead to illegal solutions for the original budget.

5.2.3 Building Access Methods

At runtime, materialized subexpressions are accessed at a per-partition level. Nevertheless, they still need to be scanned and filtered based on the predicates of the running queries. The processing time for shared access and filtering of base and cached data can dominate the total processing time. By reorganizing data within each partition, ParCuR further reduces both data access- and filtering costs.

For creating a workload-aware layout, ParCuR can conceptually use different data organization strategies as long as they mitigate access and filtering costs. For example, it can build a spatial index and use SH_2O, or partition the data using techniques such as Qd-tree [130] or the algorithm from Sun et al. [114]. ParCuR's implementation uses multidimensional range partitioning. We refer to this finer-grained partitioning as *2nd-level partitioning*.

Multidimensional range partitioning can enable efficient data access that reduces accesses during scans, as it enables data skipping. Furthermore, by cutting data across values that are frequently used in predicates, it can be used to statically evaluate frequent filters for a whole partition. To build the partitions, we iteratively subpartition data across the predicates values of one attribute at a time. The resulting subpartitions inherit query homogeneity from the *1st-level* partitioning and also reduce data-access costs. From this point on, we differentiate the partitions derived from the *2nd-level* partitioning by calling them *blocks*.

5.3 Reuse-aware Shared Execution

At execution time, ParCuR takes advantage of the constructed partitions and materialized subexpressions and optimizes query processing in three levels: First, it uses data and filter skipping to identify the queries that access each partition and reduce filtering costs. Second, it adopts a partition-oriented execution paradigm that plans and optimizes each partition independently; thus, exposing different opportunities per partition. Third, ParCuR introduces a cost-based optimization framework that chooses which materializations to inject into each partition's plan.

5.3.1 Data and Filter Skipping

ParCuR uses 2nd-level partitioning to reduce data access and filtering costs. To do so, for each block, it identifies i) which queries process the block, and ii) which predicates have the same value for all tuples in the block. Then, during execution, it skips 2nd-level partitions that are not processed by any query and eliminates filters whose predicates are invariant across the block. Both optimizations occur on both the fact table and the materializations, and can drastically reduce batch response time.

As the data is organized by cutting the data space, each block's boundaries are defined by a range along each attribute. Then, if the range is known, the above analysis can be done statically. Concretely, a query's predicate is invariant when its value range either subsumes (always true) or does not overlap (always false) with the block's range. Moreover, a query *skips a block* if at least one of its predicates always evaluates to false (no overlap). For example, the query `SELECT COUNT(*) FROM T WHERE x > 8` skips block $5 \leq x < 7$, as the two ranges do not overlap. Similarly, for the same block, the predicate of query `SELECT COUNT(*) FROM T WHERE x > 4` is true across the whole block and, thus, it is redundant to evaluate it for every tuple.

The above logic is implemented by maintaining zonemaps [39]: a lightweight index that stores min-max statistics for each attribute. During the table scan, for each block, ParCuR compares the corresponding ranges against the shared filter predicates to identify which queries do not overlap with this block (data skipping) and which are satisfied by the entire block (filter skipping). The remaining ambivalent filters are processed using the global plan.

5.3.2 Partitioned Execution

ParCuR optimizes each 1-st level partition independently to i) exploit partition-specific materializations and ii) enable partial reuse by decoupling planning between partitions. To do so, it introduces a two-phase partition-oriented execution model. First, it computes the shared state between partitions such as hash tables on dimensions and data structures for aggregation. Next, it executes each partition independently. For each partition, ParCuR identifies which queries process the partition using the same data-skipping mechanism as above. Then, it chooses a global plan that is specialized for the queries and materializations of the partition at hand. Finally, partial results from each partition are merged together in the output operators such as projections, aggregations, and GROUP-BYs. Since shared execution processes subqueries that comprise selection, projection, join probe, and potentially aggregation operators, combining partial results produces the final output [129].

Partial reuse is feasible because the output operators are oblivious to each partition's planning decisions. When query patterns recur with minor shifts, they mostly process their designated 1-st level partitions and spill over only to few neighboring partitions. Then, ParCuR processes the bulk of the processing using materializations and addresses spillovers with selective computations. Hence, in case of a workload shift, performance degradation becomes proportional to the magnitude of the shift, and thus ParCuR avoids suffering a performance cliff.

5.3.3 Injecting Materializations in Global Plans

For each partition, ParCuR optimizes and processes a global plan that exploits the available materializations and access methods as well as sharing opportunities. However, making all planning decisions in a unified optimization framework scales poorly. To this end, ParCuR adopts a two-phase optimizer architecture that performs reuse as a post-processing phase.

Two-phase optimizer

Benefits from reuse and work sharing are interdependent: the marginal benefit from reuse, if any, depends on available sharing opportunities and, also, the opportunities from downstream work sharing between queries are contingent on answering them using the same materialization. Thus, it is tempting to formulate a unified optimization problem in order to find a globally optimal plan. However, sharing-aware optimization already has a very large search space, and thus enriching it with reuse planning decisions is prohibitive.

To incorporate work sharing and reuse in a scalable and practical manner, the optimizer needs to restrict the search space. ParCuR's optimizer focuses on ensuring better performance than pure work sharing and on avoiding performance regression. Thus, the optimizer uses two phases. In the first phase, the optimizer chooses a *baseline* global plan that uses work sharing. Then, in the second phase, the optimizer improves on the baseline plan by rewriting it to reuse materializations. Finally, ParCuR processes the resulting plan, which combines reuse and work sharing.

Reuse phase

The reuse phase is based on the observation that reuse replaces operators from the baseline plan with filters on materializations. Hence, the goal is to find which subexpressions, if reused, can maximize the difference between eliminated computations and filtering costs. For each cut c, we can estimate this difference, which we call *benefit*, as:

$$benefit(c,a) = \sum_{op \in BC(c,a)} cost(op) - \sum_{v \in c} (c_f \times |RF(v)| \times v.size)$$

where $cost(op)$ of operator op in the baseline plan, $RF(v)$ are the runtime filters on subexpression v after filter-skipping in the current partition, $v.size$ is the number of tuples for the subexpression in the current partition and c_f is a constant for estimating filtering costs per tuple as a linear function of the number of runtime filters $|RF(v)|$. $benefit(c,a)$ represents the net benefit of reusing c with respect to anchor a as the different between the cost of eliminated operators between c and a and the overhead for accessing and filtering c's materializations. The optimizer has all this information at the time of running the reuse phase.

In order to choose which subexpressions to reuse, the reuse phase, which we show in Algorithm 6, performs a post-order traversal of the baseline plan and transforms the plan. When visiting a node, the traversal first processes the node's successors and merges their rewrite decisions (lines 9-11). Then, the algorithm finds the best cut (i.e., the cut with the highest benefit) that can eliminate the current node. If all of the node's successors are eliminated or are anchors for cuts, then the algorithm computes the best cut of downstream subexpressions by merging the cuts of the remaining successors (lines 12-16). If the node corresponds to a materialized subexpression, the algorithm also considers the cut that consists of the node's results (lines 17-19). Finally, if the best cut provides net gain, the rewrite is applied immediately (lines 20-22), and otherwise, the best cut is propagated to upstream nodes.

Theorem 6. Algorithm 6 makes the optimal injection decision for the given plan.

Proof. We prove, using induction on the plan size, that the algorithm computes: i) the optimal reuse-enhanced plan and ii) the best cut that has not been injected already.

Base step: For a plan with one node, the algorithm minimizes the cost: if the node's result is materialized and reuse is beneficial (positive benefit), the algorithm rewrites the plan. Otherwise, the baseline plan is optimal and the algorithm leaves it as it. If there is a materialization and the benefit is negative, it constitutes the only available cut.

Induction step: If the proposition holds for plan size $\leq k$, it also holds for plan size $k+1$.

We focus on the case where there is only one node that is upstream to all other nodes (root). If the plan consists of multiple connected components, i.e., subplans with different shared operators at the bottom, then independently applying the algorithm to each connected component is trivially optimal.

The currently visited node is upstream to all other nodes (root). Each downstream subplan has at most k nodes, so the algorithm computes the rewrites that minimize the cost. Let each node have an attribute $optPlan$ that represents the optimal downstream plan and let $DC(plan)$ be a function that computes the downstream cost for an optimized plan (including filters).

Before line 20 (or if the branch in line 15 is not taken), the situation is as follows:

$$\Delta = DC(v.optPlan) - DC(v.bestPlan) \Rightarrow$$

$$\Delta = \sum_{s \in succ} (DC(s.optPlan) - DC(s.bestPlan)) + (x-y)cost(v)$$

where $x, y \in \{0,1\}$ are binary values that represent if v is part of the plan. Then, we have the following two cases:

- if $x = 1$ or $y = 0$, then $\Delta \geq 0$. Thus, $bestPlan$ is optimal.
- if $x = 0$ and $y = 1$, we prove that the algorithm eliminates v in line 20 and then $\Delta \geq 0$ for the new $bestPlan$. Since $x = 0$, there exists a cut c with anchor v.
  - If $benefit(\{v\}, v) > 0$, then the new $bestPlan$ has $\Delta \geq 0$, because either $\{v\}$ or c will be rewritten.
  - Alternatively, REWRITE needs to occur with the downstream cut. Let $s_1, s_2, \ldots, s_p$ be v's successors and $c_1, c_2, \ldots, c_p$ the corresponding sub-cuts. Since $optPlan$ is optimal:
    $$benefit(c, v) \geq \sum_{i \in \{i | benefit(c_i, s_i) > 0\}} benefit(c_i, s_i)$$
    which implies
    $$\sum_{i \in \{i | benefit(c_i, s_i) < 0\}} benefit(c_i, s_i) + cost(v) \geq 0$$
    Thus, the merged cuts from the successors can eliminate v and the new $bestPlan$ is optimal.

□

Algorithm 6: Reuse Optimization Phase

```
Function REUSE_OPT_REC(v):
    $v.bestPlan = \emptyset$ ;
    $v.bestCut = (!v.succ.empty())?\ \emptyset : null$ ;
    $atLeastOne = (v.succ.empty())$ ;
    for $s \in v.succ$ do
        $REUSE\_OPT\_REC(s)$ ;
        $v.bestPlan = v.bestPlan \cup s.bestPlan$ ;
        if $s.bestPlan.contains(s)$ then
            $atLeastOne = true$ ;
            if $v.bestCut != null$ then
                if $s.bestCut == null$ then
                    $v.bestCut = null$ ;
                else
                    $v.bestCut = v.bestCut \cup s.bestCut$ ;
    if $atLeastOne$ then
        $v.bestPlan = v.bestPlan \cup \{v\}$ ;
        if $v.materialized$ then
            if $v.bestCut == null\ or\ benefit(v.bestCut, v) < benefit(\{v\}, v)$ then
                $v.bestCut = \{v\}$ ;
        if $benefit(v.bestCut, v) > 0$ then
            $v.bestPlan = REWRITE(v.bestPlan, v.bestCut)$ ;
            $v.bestCut = null$ ;
```

Handling adaptive optimization

We implement ParCuR by extending RouLette, which uses adaptive sharing-aware optimization. RouLette splits batch execution into episodes, which last for the duration of processing one small base table vector each, and potentially uses a different global plan in each episode. RouLette learns the cost of different subplans across episodes and eventually converges into an efficient global plan.

The episode-oriented design conflicts with two-phase optimization: the reuse phase chooses which subexpressions to reuse based on the baseline plan at partition granularity, whereas ParCuR switches between multiple baseline plans during a partition's execution in order to learn effectively. We reconcile the two using the concept of *mini-partitions*. Mini-partitions are horizontal subpartitions of 1-st level partitions and are internally organized using 2-nd level partitioning. ParCuR splits the base table's 1-st level partitions into fixed-size mini-partitions and then splits materializations such that tuples derived from the same base table mini-partition are clustered together.

ParCuR makes reuse decisions at the mini-partition granularity. When accessing a mini-partition for the first time, ParCuR chooses a baseline plan and makes two decisions: i) it decides whether the baseline plan is stable, i.e., it checks whether it is still learning the cost of the used subplans by tracking changes in cost estimates, and ii) if the plan is stable, it uses the reuse phase to choose materializations to use. Then, until the mini-partition is finished, ParCuR retains the reuse decisions and optimizes the downstream computations for each reused subexpression independently.

5.4 Implementation

We implement ParCuR on RouLette. Our implementation modifies two components, the *policy* and the *ingestion*, implements a materialization operator that writes results to storage, and introduces the tuner's utilities. In this section, we highlight the details of our implementation.

5.4.1 Tuning the Cost Model

The presented techniques for materialization and reuse rely on RouLette's cost models to estimate the processing time and use the computed costs to make decisions. RouLette's cost models use constant factors to scale the cost estimates for different types of operators. This work uses the same constant factors as RouLette. In addition, it introduces the new constant factor c_f in Section 5.3.3. By using regression to fit filtering cost estimates to processing time measurements, using the same methodology as in Chapter 3, we find $c_f = 139.45$.

5.4.2 Enforcing Reuse Decisions to Policy

For each mini-partition, ParCuR makes reuse decisions the first time it accesses it and retains the decisions until the partition is finished. This means that the chosen materializations are the starting point for the planning and execution of all the episodes that correspond to the mini-partition. To do this, we create a mapping between each accessed subexpression (base table or materialization) and a set of queries answered using the subexpression. We use each pair as input to RouLette's policy in order to plan downstream computation. The policy uses prior cost estimates to produce a downstream plan, if available, and continuously refines both the estimates and the downstream plan by monitoring execution.

5.4.3 Ingestion from Materializations

Ingestion retrieves data vectors from both the base table and materializations. However, it only knows which materializations to access when starting to process each mini-partition of the base table. For this reason, we additionally implement scans with mini-partition scope for materialization tables: each such scan only retrieves the vectors that correspond to a specific mini-partition and then finishes. Scans with mini-partition scopes attach themselves to the base table's scan, and the base table's scan can proceed to the next mini-partition only after the attached scans are finished. Consequently, the base table's scan coordinates the accesses to the required mini-partitions of materializations, thus enabling fine-grained access reuse decisions. Note that the base table's scan does not necessarily access a mini-partition's data: it can skip the partition and then coordinate the attached scans for materializations.

5.4.4 Materializing Results

ParCuR extends RouLette with support for temporary tables that store materialized subexpressions. To do this, it implements an interface for dynamically allocating and resizing tables in RouLette's storage manager and a materialization operator that writes the results of a query to a target table. After materializing a subexpression, the corresponding table can be exploited for reuse decisions that accelerate incoming query batches.

5.4.5 Tuner Utilities

ParCuR implements three utilities that constitute the tuner: i) homogeneity-based partitioning, ii) materialization policy, and iii) the access method partitioning. To collect the input of these utilities, ParCuR first processes a set of target batches that represent historical workload. Specifically, it stores the subquery vectors for a predetermined sample of tuples, the set of used global plans and their corresponding cost estimates, and the predicates in the query batches. ParCuR then uses the collected metadata to run the utilities in the following order: First, it runs the homogeneity-based partitioning to compute the partitioning scheme and actuates the scheme to produce 1-st level partitions. Second, it reruns the target workload to compute the historical workload graph, then runs one of the approximate algorithms for cut selection to choose subexpressions to materialize, and finally actuates the materialization. Third, it produces mini-partitions for the base table and the materializations and further partitions each mini-partition to produce 2-nd level partitions. After this step, ParCuR state has finished tuning for the target workload.

5.4.6 Tuning Partitioning

Choosing the parameters for the two levels of partitioning (homogeneity-based clustering and access methods) affects the overhead for ParCuR's executor. To tune the parameters, we use the workload of Figure 5.6 and, out of the tested parameters, we find the minimum values for mini-partition size and block size, and the maximum sampling rate, such that overhead is less than 10% compared to the optimal value. We set the minimum size of mini-partitions to 2^{16} to maintain low overhead for the reuse phase. Thus, we also set $PS_{min} = 2^{16}$ as homogeneity-based partitions need to contain at least one mini-partition. Then, we select the size of mini-partitions such that ParCuR keeps the overhead for data and filter-skipping low as well: it is selected to be greater or equal to 2^{16} and at least large enough that the blocks contain at least 256 tuples each on average. Finally, to avoid a significant overhead for tracking historical accesses, we set the sampling rate to 1%.

5.4.7 Limitations

To combine reuse with adaptive optimization, ParCuR's implementation over RouLette aligns the mini-partitions of materializations with the mini-partitions of a base table. For this reason, tuning revolves around one main table that defines the partitioning schemes and the materializations, and thus our implementation is applicable to common workloads such as queries on star and snowflake schemas.

5.5 Experimental Evaluation

The experiments evaluate ParCuR and show how materialization and reuse enable it to significantly outperform pure work sharing and achieve lower batch response times. Specifically, they demonstrate the following:

i) Filtering costs when accessing materializations can deteriorate the performance of work sharing, and thus building access methods for materializations is necessary.

ii) Query-at-a-time materialization policies make suboptimal materialization decisions. Cut selection improves budget utilization by prioritizing materialization with higher marginal benefits.

iii) Homogeneity-based partitioning reduces the required budget for workloads with selective and correlated patterns.

iv) Even though filters and sharing decisions change, the reuse phase reduces work-sharing's response time when possible and falls back to vanilla work sharing otherwise.

v) Using partial reuse, the response time is proportional to the required computation and performance degrades gracefully.

vi) End-to-end, ParCuR reduces the response time for the full SSBM and TPC-H by 6.4× and 2×, respectively.

Hardware. All experiments took place on a single server that features an Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz with 2 sockets, 12(×2) threads per socket, 376GB of DR0AM, 32KB L1 cache, 1MB L2 cache, and 16MB L3 cache. All experiments took place in memory, in a single NUMA node, and use 12 threads.

Data & Workload. We run both macro- and micro-benchmarks. First, we perform a sensitivity analysis. We evaluate ParCuR by varying different workload properties: i) the number of filtering attributes, ii) the selectivity of predicates, iii) the number of joins and the overlap between queries, iv) the available budget, and v) the workload shift in filter attributes, join overlap, and predicate correlations. To control the experiment variables, we generate synthetic data in a star schema as well as appropriate queries. We use a fact table of $100M$ rows and 27 columns (24 are foreign keys), 8 dimensions with $10k$ rows and 9 columns each, and 16 dimensions with $10k$ rows and 2 columns each. All columns are 4-byte integers. We describe the queries in the presentation of each micro-benchmark.

Next, we show that ParCuR accelerates the queries of the widely used SSBM [87] and TPC-H

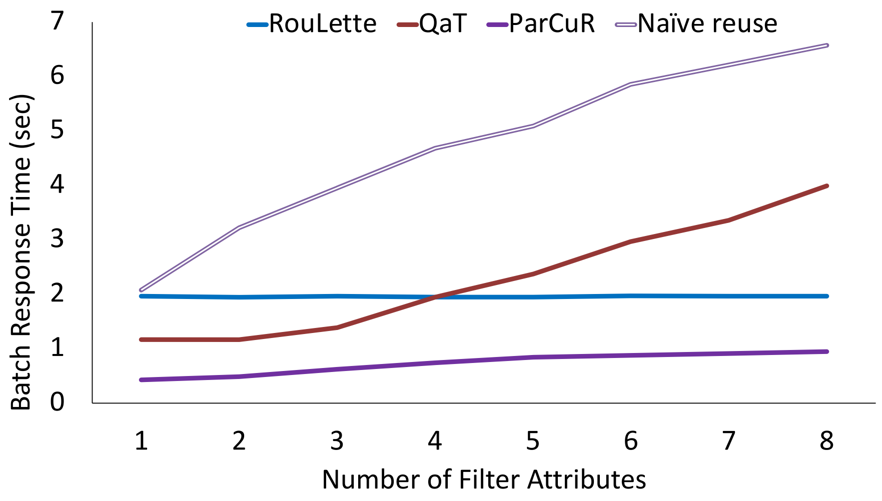


Figure 5.6: Impact of reuse for a varying number of filter attributes

benchmarks. We use scale factor 10 for both, which is the largest data size for which the optimal materialization fits in the available memory. We randomize the order of tuples for both datasets.

Methodology. The experiments measure batch response time, which is the end-to-end time for processing the full batch. All measurements are the average of 10 runs.

5.5.1 Impact of Reuse in Global Plans

We evaluate the benefit of reuse to shared execution's response time. We assume that the tuner's workload is the same as the runtime workload and that the materializations that minimize response time are available (i.e., the top-level joins). Sections 5.5.2 and 5.5.3 lift the two assumptions. We compare ParCuR against RouLette, naive reuse, which eagerly injects materializations and has no access methods, and QaT execution using ParCuR, which is on par with QaT performance of state-of-the-art in-memory databases.

Filter processing. We examine the impact of filters and the need for building and using access methods for materializations. We use 64 queries generated from 4 different templates. The templates have 4 dimension joins each, and all templates share 3 dimension joins. The queries have 10% selectivity and filter on the non-shared dimension. We vary the number of filter attributes (which is equal to the number of shared filter operators) from 1 to 8.

Figure 5.6 shows that access methods are necessary for accelerating work sharing. When using access methods, ParCuR's response time is 2.07-4.57× lower than RouLette's, as it eliminates join processing. RouLette is almost unaffected by increasing filter operators, as it processes filters on the dimension. ParCuR and QaT are affected because they require more 2-nd level partitions and hence both more zone-map operations as well as larger mini-partitions, and thus longer time until ParCuR decides that the plan is stable. However, this effect just reduces ParCuR's benefit over RouLette. By contrast, the performance of naive reuse deteriorates

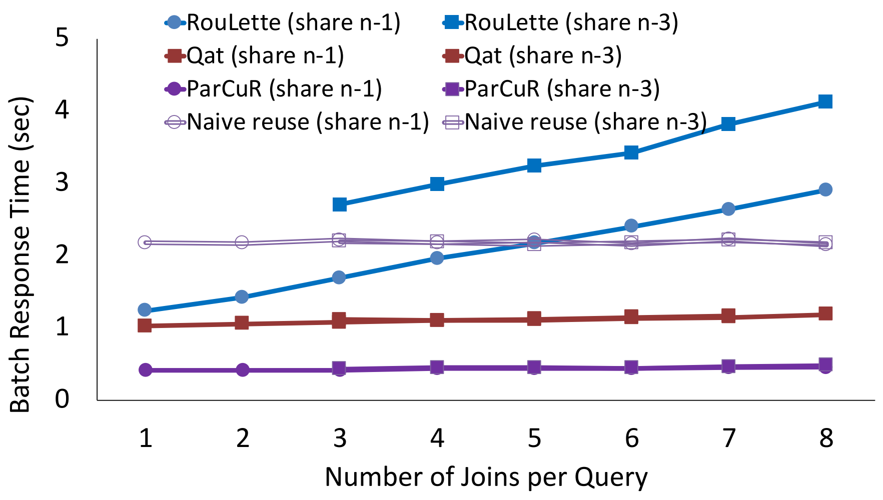


Figure 5.7: Impact of reuse for a varying number of joins per query

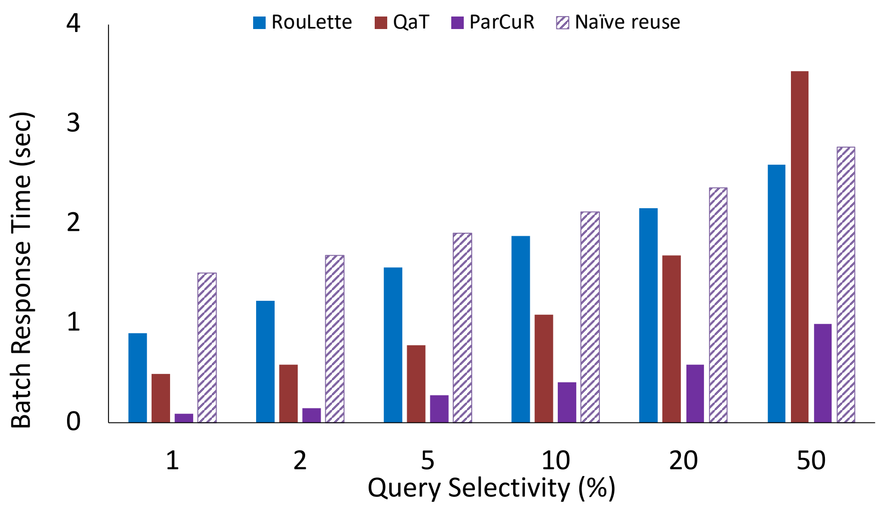


Figure 5.8: Impact of reuse for varying query selectivity

drastically: it computes filters over the materialization, and thus their processing time is amplified. The response time is increased with the number of filters and is up to 3.34× than RouLette's.

Takeaway: Reuse drastically improves performance only if filtering cost is low, and can deteriorate performance otherwise. Building appropriate access methods is necessary for injecting materializations into global plans.

Number of joins. We examine the impact of reuse in queries with different join costs. We use two variants of the previous workload, one where all templates share all but one join (share n-1) and another where all templates share all but three joins (share n-3). We vary the total number of joins per query. All queries use 1 dimension filter.

Figure 5.7 shows larger benefits for global plans with more joins. Reuse-based approaches are insensitive to the number of joins, whereas RouLette's response time is increased. ParCuR achieves maximum speedup of 6.33 for share n-1 and 8.60 for share n-3. Also, there is a cross-point in naive reuse, where processing filters becomes preferable to large joins.

Takeaway: The benefit from reuse is proportional to the eliminated computation. Hence, the speedup is higher when eliminated computation is significant, such as in join-heavy queries.

Selectivity. We examine the impact of reuse for queries with different selectivity. We use the same workload as in the first experiment, use one filter attribute, and vary the selectivity (1%, 2%, 5%, 10%, 20%, 50%). The experiment models the impact of downstream processing on the speedup.

Figure 5.8 shows larger benefits when each query's selectivity is low. As aggregations are not affected by reuse, they close the gap between approaches for larger selectivity when they are expensive. Also, it is noteworthy that when aggregations are heavy enough, QaT is more expensive than RouLette due to concurrency.

Takeaway: Reuse has a higher benefit when it eliminates the most expensive part of the global plan. Low selectivity keeps the cost of final aggregations low, and thus the relative benefit is more pronounced.

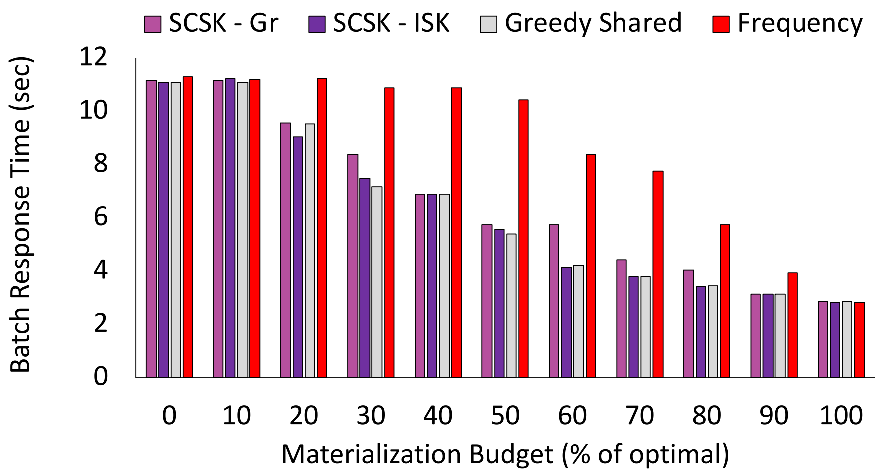
5.5.2 Sharing-aware Materialization Policy

We demonstrate that cut selection solutions outperform sharing-oblivious and simple sharing-aware policies. We compare four different algorithms: a) **SCSK-Gr** solves cut selection using Gr, b) **SCSK-ISK** solves cut selection using ISK, c) **Greedy Shared** solves a submodular knapsack problem for individual materializations, and d) **Frequency** solves the submodular knapsack problem where benefits are weighted by frequency, which is commonly used for query-at-a-time materialization. The evaluation uses four different workloads with 512 queries with 10% selectivity each. The queries use filters in a column with domain $[0,100)$.

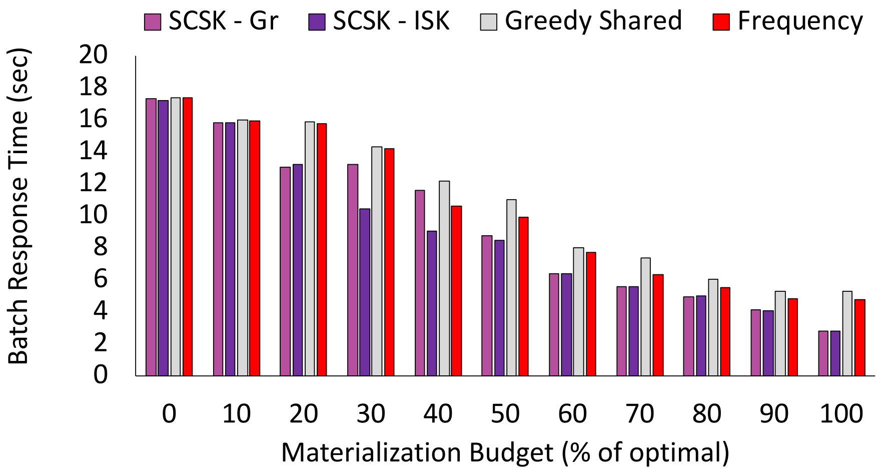
- **Workload A**: Workload A shows the impact of frequency. It uses 8 query templates ($t_1,\dots,t_8$). $t_1,\dots,t_4$ have 1 join each, whereas $t_5,\dots,t_8$ have 4 joins each. Template t_i shares its join with template t_{i+4}. The workload contains 112 queries from each of $t_1,\dots,t_4$ and 16 queries from each of $t_5,\dots,t_8$. In total, it requires at least 40GB to minimize response time.

- **Workload B**: it uses 8 query templates ($t_5,\dots,t_{12}$). $t_9,\dots,t_{12}$ also have 4 joins each. Template t_i shares 2 joins with template t_{i+4}. The workload contains 64 queries from each of the templates. The workload shows the impact of synergy. In total, it requires at least 32GB to minimize response time.

- **Workload B-P1**: it uses workload B's templates. However, the filters for t_5 and t_6 are subranges of $[0,40)$, for t_9 and t_{11} subranges of $[20,60)$, for t_7 and t_8 subranges of $[40,80)$,



(a)



(b)

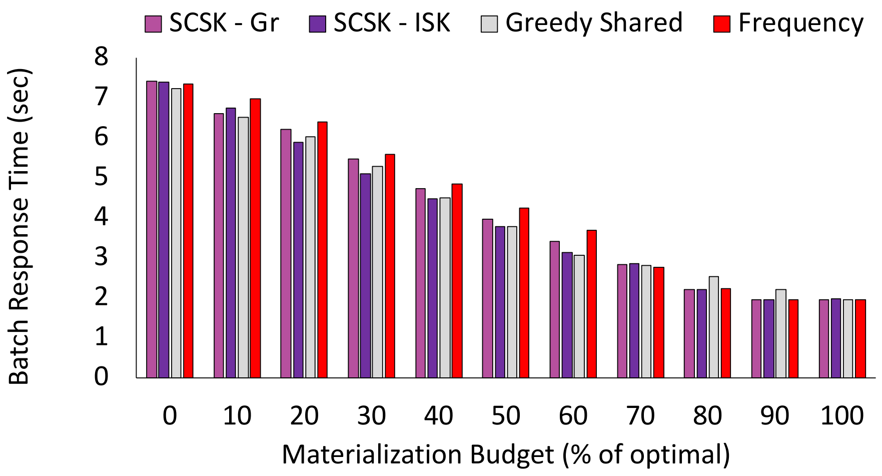
Figure 5.9: Impact of budget for workloads A and B

and for t_{10} and t_{12} subranges of $[60,100)$. In total, it requires at least 14.9GB to minimize response time.

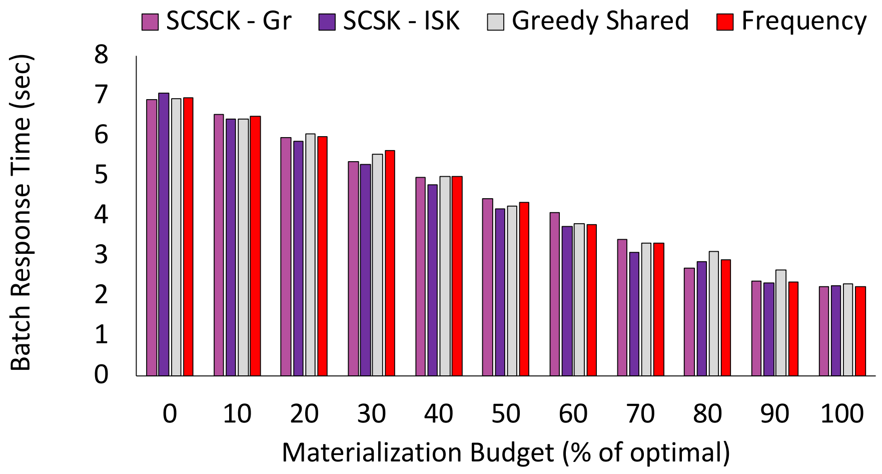
- **Workload B-P2**: Similar to workload B-P1, but uses 2-D ranges. The filters for t_5 and t_6 are subranges of $[0,66) \times [0,66)$, for t_9 and t_{11} subranges of $[0,66) \times [34,100)$, for t_7 and t_8 subranges of $[34,100) \times [0,66)$, and for t_{10} and t_{12} subranges of $[34,100) \times [34,100)$. In total, it requires at least 12.9GB to minimize response time.

In each experiment, we vary the storage budget to the minimum budget that can minimize response time. We present the used budget normalized by the budget that minimizes response time (i.e., this budget is 100%).

Sharing-awareness: Figure 5.9a shows that sharing-aware policies outperform Frequency in workload A because they factor out the frequency of occurrence for subqueries, and decide



(a)



(b)

Figure 5.10: Impact of budget for workloads B-P1 and B-P2

based on shared costs. Frequency results in up to 2.03× higher response time for the same budget because it prioritizes templates $t_1, \ldots, t_4$.

Synergy-awareness: Figure 5.9b shows that exploiting the synergy between materializations that compose cuts in workload B improves the effectiveness of materializations. Both Greedy Shared and Frequency preferentially materialize the shared subqueries because they miss the synergy between the larger cuts. Thus, they both waste budget on materializing subexpressions that are later covered by the larger cuts, and consequently, 100% is not sufficient for minimizing response times. At 100%, they are slower by 1.87× and 1.68×, respectively.

Partition-awareness: For both workload B-P1 and B-P2, partiitioning reduces the required budget for minimizing response times by 2.4× and 2.5× accordingly. Figure 5.10 shows that all algorithms achieve comparable performance because partitioning simplifies the global plans for each partition. The simplification mitigates the effect of synergy and frequency, and thus

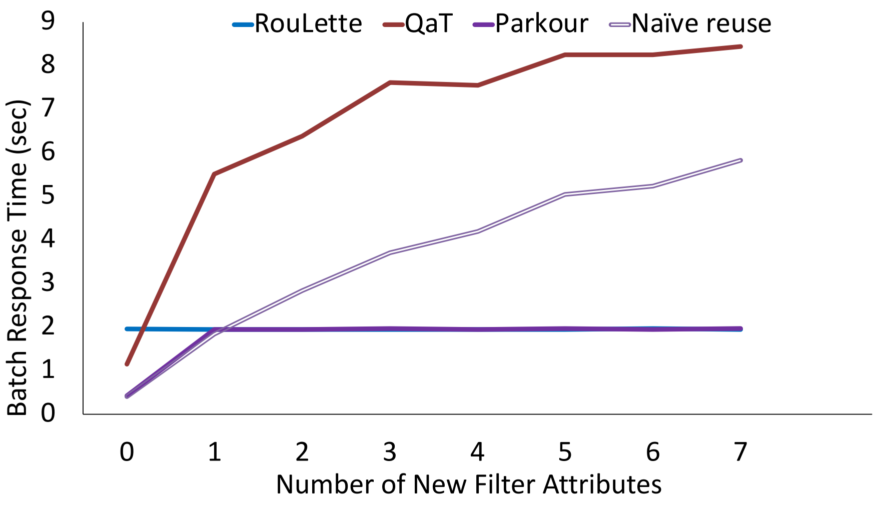


Figure 5.11: Impact of workload shift in the filtering attributes

all algorithms find comparable solutions.

Gr vs ISK: Across all experiments, ISK performs better than Gr as it enumerates more materializations and normalizes marginal benefit by the required budget. By contrast, Gr suffers from suboptimal solutions when it uses up the budget on few materializations. Still, ISK requires significant processing time to run, e.g., 217sec in workload B-P1, and thus Gr is preferable for real-time analysis as it takes up to 4msec in all experiments.

Takeaway: Both sharing-awareness and partitioning improve budget utilization. Incorporating both shared costs and synergy permits spending the budget for materializing only the subexpressions that actually reduce response times. Furthermore, partitioning enables materializing results just for the data ranges where they are needed and thus reduces budget requirements.

5.5.3 Effect of workload shift in reuse

We evaluate ParCuR under workload shift. We materialize subexpressions that minimize the response time for the original workload. The experiments shift workload across three axes (a) filtering attributes, b) query templates, and c) query pattern correlations). We compare ParCuR against RouLette and naive reuse, for which we enable access methods, and QaT.

Filtering attributes. Figure 5.11 shows that the reuse phase judiciously chooses between reuse and recomputation based on filtering costs. The experiment uses the same workload as Figure 5.6. However, it only builds an access method for the first attribute. Naive reuse improves response time when there is no shift and deteriorates performance otherwise. QaT's performance depends on the percentage of queries that use the materializations. Finally, ParCuR improves performance when there is no shift and achieves the same performance as work sharing when reuse is detrimental.

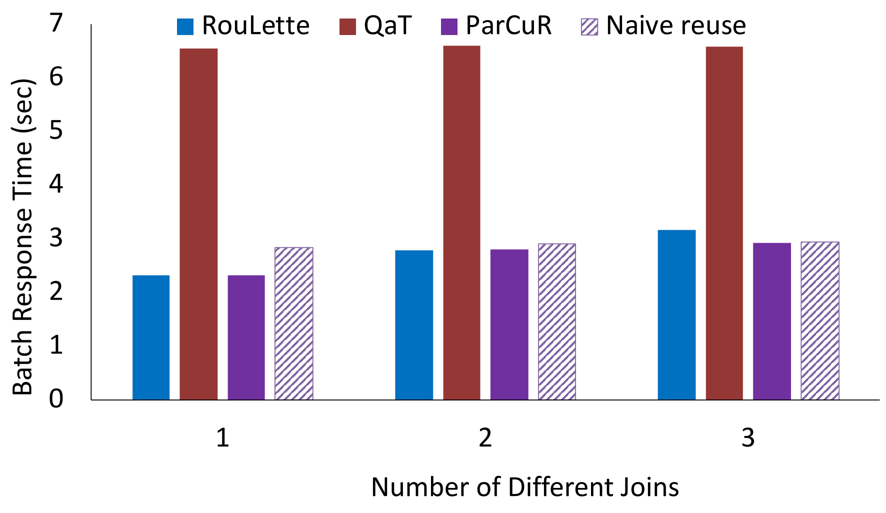


Figure 5.12: Impact of workload shift in the query templates

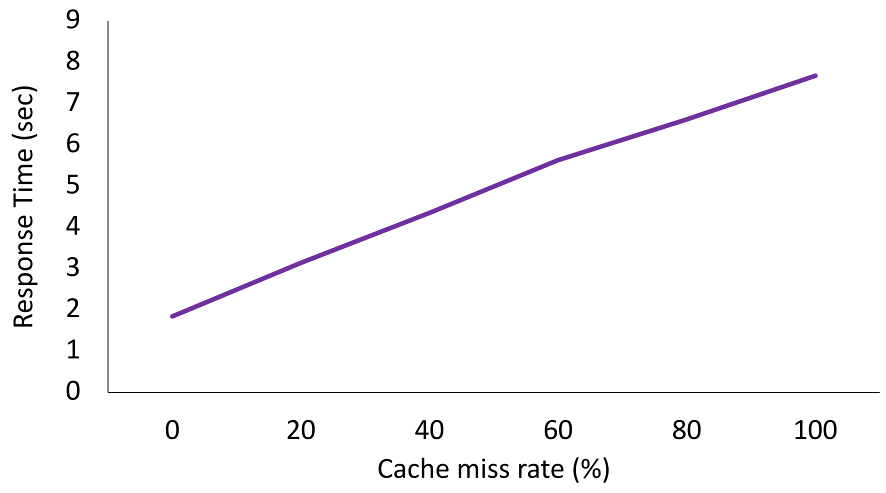
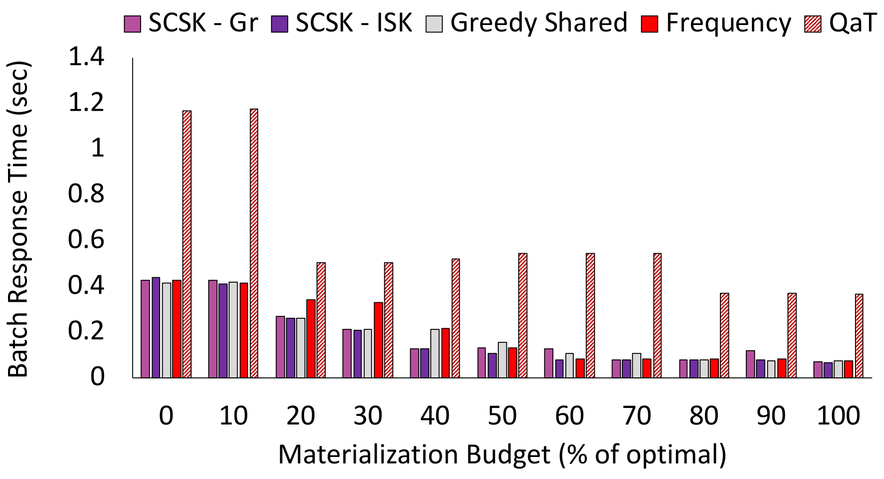


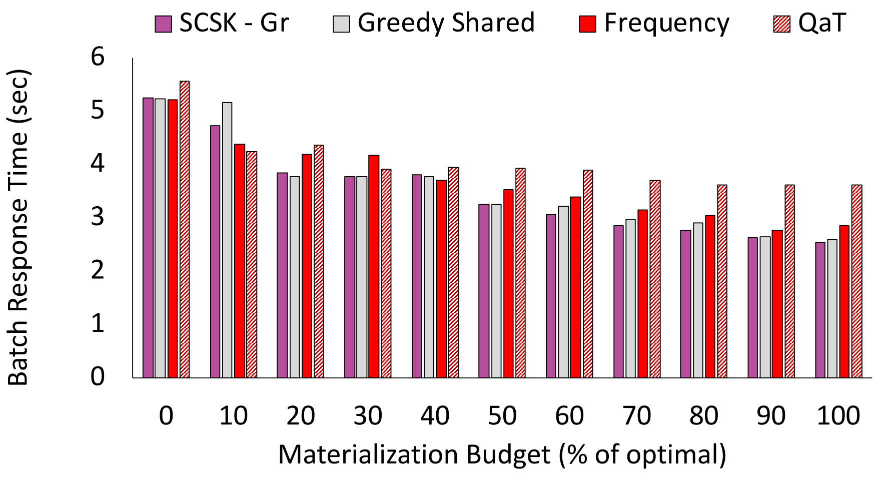
Figure 5.13: Impact of workload shift in the predicates of query patterns

Query templates. Figure 5.12 shows that the reuse phase also chooses judiciously when the marginal benefit of reuse changes due to new workload opportunities. The original workload uses two templates with 4 joins each. The workload shift adds two more templates with 4 joins each, and each template differs by k joins from one of the original (i.e., shares 4-k joins). k varies from 1 to 3. Furthermore, we change the filtering attribute for the original templates. We use 16 queries per template. RouLette performs better when the shifted workload differs by 1 or 2 joins and work sharing is high, whereas reuse is better when work sharing is low. QaT is unaffected since it makes per-query decisions. Finally, ParCuR's cost model chooses correctly between reuse and recomputation and matches the best option.

Query patterns' predicates. Figure 5.13 shows that partial reuse enables response times to degrade gracefully under workload shift. The experiment uses workload B-P1 to build materializations. The shifted workload slides the ranges for the filters of each template; the



(a)



(b)

Figure 5.14: Macro-benchmarks: a) SSBM b) TPC-H

slide controls the percentage of the shifted workload's input that cannot reuse materializations and is processed from base data (miss rate). ParCuR's response time is increased proportionally to the miss rate. Thus, when partitioning captures query patterns and isolates misses, partial reuse improves performance against all-or-nothing approaches that fall back to full processing (same performance as 100% miss rate).

Takeaway: The reuse phase, as well as partitioned execution, enable ParCuR to benefit from materializations despite workload shifts. ParCuR exploits materializations for the partitions where they are available and beneficial to reducing the global plan's cost.

5.5.4 Macro-benchmarks

We evaluate ParCuR using the SSBM and TPC-H benchmarks. For each benchmark, we compare the four materialization algorithms and vary the storage budgets. We omit ISK for TPC-H, because it takes a very long to choose a materialization. Also, we simplify TPC-H queries similar to Chapter 4.

SSBM: Figure 5.14a shows that ParCuR achieves a maximum speedup of 6.4 over RouLette and 5.4 over QaT, and requires around 1GB for the optimal materialization. The speedup is high because queries are mostly selective, and thus aggregations make up a small percentage of processing time; the vast majority is filters and joins. An interesting observation is that even a small budget brings about a sharp decrease in response time because bottom joins are significantly more expensive, whereas upper joins are more selective and less time-consuming.

TPC-H: Figure 5.14b shows that ParCuR achieves a maximum speedup of 2× over RouLette and 1.37× over QaT, and requires 69GB for the optimal materialization. The speedup is lower compared to SSB for two reasons: i) TPC-H contains less selective queries with heavier aggregations. When using 100% budget, aggregation takes up around 40% of the time. ii) TPC-H contains LIKE predicates that filter skipping cannot eliminate using zonemaps. Still, despite the shortcomings in our implementation, ParCuR eliminates significant join costs.

Discussion: For the two benchmarks, ParCuR requires large materializations because, currently, homogeneity-based partitioning does not exploit filters on dimensions. This limitation can be addressed by: i) partitioning using the denormalized table [130], or ii) partitioning using data-induced predicates on the fact table's foreign keys [54]. Both techniques are complementary and straightforward to integrate with ParCuR.

Another limitation is that ParCuR cannot eliminate predicates such as LIKE, multi-attribute expressions, or UDFs using zonemaps. To eliminate such predicates, partitions require additional metadata. Sun et al. [114] handle such predicates by maintaining a feature vector that encodes whether complex predicates are satisfied.

5.6 Summary

To provide real-time responses for large recurring workloads, we propose ParCuR, a novel paradigm that combines the reuse of materialized results with work sharing. ParCuR addresses the performance pitfalls of incorporating materialized results into shared global plans i) by proposing a multi-level partitioning design that improves at the same time the utilization of the storage budget, partial reuse, and filtering costs, ii) by proposing a novel sharing-aware caching policy that improves materialization decisions, and iii) by enhancing the sharing-aware optimizer with a phase that performs reuse-oriented rewrites in order to minimize runtime processing. In our experiments, ParCuR outperformed RouLette by 6.4× and 2× in the widely-used SSB and TPC-H benchmarks respectively.

6 Conclusion and Future Outlook

With novel applications rising to prominence and the number of users growing, analytical databases struggle to provide timely responses to a high number of concurrently executing queries. As the timeliness of traditional query-at-a-time databases deteriorates when processing an increasing number of queries, work sharing becomes critical functionality for databases so that they mitigate the effect of concurrency on response time. However, existing work-sharing databases incur redundant processing and are inefficient for common classes of ad-hoc, selective, and recurring workloads due to redundant processing. First, ad-hoc queries are a poor fit for sharing-aware optimization because they are unpredictable and, thus, planning can be both inaccurate and prohibitively time-consuming to perform online. Second, selective queries either sacrifice sharing to use indices or suffer from excessive data access and filtering. Third, recurring queries cannot exploit precomputed materialized results because reuse is inefficient in shared execution and can be detrimental.

In this thesis, we have contributed towards building efficient work-sharing databases that improve timeliness for highly concurrent workloads by holistically adapting to the characteristics of the target data and workload. To this end, we optimize across the query processing stack by introducing novel i) query processing abstractions, ii) optimization strategies, and iii) physical design tuning. The building blocks we introduce reduce the required data access and processing, and the efficiency of reuse, while maintaining the scalability of shared execution to highly concurrent workloads.

In this chapter, we summarize the contributions of this thesis and then discuss research directions that can expand the scope of the presented work.

6.1 Data and Workload-conscious Work Sharing: What we did

We have identified three subproblems in which work-sharing databases are inefficient: i) join order optimization, ii) data access, and iii) subexpression reuse. Each part of the thesis addresses inefficiency in each component by introducing novel optimizations.

First, to choose join orders in a judicious and yet timely manner while exploring more and potentially more promising candidate join orders compared to fast heuristics, we propose RouLette. RouLette is a specialized engine that uses adaptive optimization to share work across concurrent Select-Project-Join subqueries. It incrementally explores the search space of shared join orders and, by using reinforcement learning on metrics collected while exploring each plan, it learns an efficient plan for the target queries and data. To avoid incurring significant adaptation overhead, RouLette also introduces query processing optimizations. The evaluation shows that RouLette chooses more efficient plans than heuristics and, at the same time, scales to thousands of queries.

Second, to reduce the required time for data access and filtering for concurrent queries, we propose SH2O. SH2O is an access method that combines shared index accesses with shared filters to achieve both efficiency and scalability. SH2O is based on the key insight that there exist multidimensional data regions where filtering decisions are invariant. It then acts in two phases: first, it uses a spatial index to access data by region for a subset of the filtering attributes and then processes the rest of the filters in a post-filtering phase. Index accesses are shared and eliminate the need for processing shared filters for the used attributes. SH2O restricts the number of regions in case of high dimensionality by using two optimization strategies: i) it reorganizes data by partitioning and then indexing each subspace based on local access patterns and then optimizes access to each partition independently, and ii) it uses an online algorithm that selects a subset of filter attributes for multidimensional access that optimizes cost-benefit. Experimental results show significantly lower data access and filtering time, especially for workloads with low joint selectivity or a high number of selected filtering attributes.

Finally, to efficiently materialize and reuse precomputed results, we propose ParCuR. ParCuR is a framework that addresses four challenges: i) choosing which subexpressions to materialize, ii) choosing the data layout for materializing the selected subexpressions, iii) choosing when and how to reuse existing materializations, and iv) exploiting partial overlaps between materialized and requested results. ParCuR harmonizes reuse and shared execution by introducing novel database tuning, planning, and execution strategies, and, hence, significantly reduces response times by eliminating recurring computations.

Overall, this thesis redesigns planning and execution strategies for work-sharing databases such that they adapt to the data and workload at hand. As a result, it improves total efficiency in all workloads and makes work-sharing databases applicable in applications that produce highly concurrent unpredictable, selective, or recurring workloads for which existing approaches are inefficient.

6.2 Efficient Work-Sharing Databases: Next steps

This thesis introduces data and workload-conscious planning and execution strategies for work-sharing databases in a scale-up in-memory setting. Extending the scope of data and workload-conscious optimizations for shared execution requires addressing additional challenges described below:

Complex analytical workload: Applications, such as data science notebooks and scripts, require processing complex workloads that comprise a large number of inter-dependent queries. For example, data scientists write imperative programs that interweave analytical processing using embedded databases (e.g., SQLite, DuckDB [95]) or data manipulation libraries (e.g., pandas) with tools for linear algebra, machine learning, and visualization. The imperative constructs of the host programming language create complicated data and control flow dependencies. Data pipelines also consist of up to thousands of queries with producer-consumer relationships [33]. Moreover, even common SQL features such as views, nested subqueries, and control flow produce query plans that contain several inter-dependent subqueries. Complex workloads that contain multiple subqueries that process the same data have significant inherent opportunities for data and work sharing. However, existing work-sharing techniques fail to fully exploit sharing opportunities in complex workloads. First, dependencies between subqueries serialize execution, thus reducing concurrency and limiting work sharing to overlapping work between independent subqueries, if any. Second, subquery boundaries, which are determined by algebraic transformations such as GROUPBY push-down, can limit the available opportunities that Select-Project-Join-oriented work sharing can exploit.

Novel optimizations that relax dependencies between subqueries and make algebraic plan transformations in a holistic manner can increase sharing opportunities and enable the techniques proposed in this thesis to accelerate complex workloads. Relaxing dependencies enables more subqueries to run concurrently, hence permitting work sharing to reduce processing. In previous work, for example, we propose using speculation to exploit parallelization and work sharing between the inner and the outer part of nested queries [112]. Speculation is also applicable for resolving control flow dependencies. Techniques from the domains of programming languages and computer architecture, such as loop unroll and out-of-order execution, further relax dependencies for the mixed imperative-analytical workload in data science applications. Finally, by applying algebraic plan transformations that determine the SPJ subqueries in the inter-dependent workload in a holistic way, opportunities can be increased.

Distributed and heterogeneous infrastructure: Infrastructure for analytical processing evolves to cope with the exponential growth of data and trends in modern hardware. Data processing frameworks run queries over large datasets by scaling-out execution over multiple nodes; thus, they parallelize query processing across many nodes to reduce response times. At the same time, each node can be equipped with hardware accelerators, such as GPUs and FPGAs, that are connected to CPU sockets using interconnects. Analytical databases can offload queries to accelerators or even parallelize queries across both the CPU and accelerators, to decrease processing time. Distributed and heterogeneous environments pose additional challenges, such as expensive data movement over slow interconnects or the network, skew, and non-uniform processing capabilities across processing units. Analytical databases that target distributed or heterogeneous infrastructure optimize for locality and introduce query processing techniques such as predicate pushdown, bloom filters, data placement optimizations, and load balancing strategies.

Sharing data and work presents opportunities for reducing processing time in distributed and heterogeneous infrastructures. In previous work, we study the effect of sharing data transfers to GPUs [97]. We show that sharing data transfers significantly reduces processing time when using a slow interconnect (i.e., PCIe3) and even brings moderate benefits when using fast interconnects (i.e., NVLink). Furthermore, our experiments demonstrate that it is beneficial for queries to share some of the transferred data and to perform fine-grained transfers for selectively accessed columns for each query individually, compared to eagerly sharing all transfers. However, in that scenario, the overhead of selective accesses accumulates. This scenario results in the same trade-off as SH_2O: hence, studying SH_2O's applicability for coordinating shared accesses to remote memory is promising.

At the same time, distributed and heterogeneous infrastructure affects the performance trade-offs that work-sharing databases assume. Communication between operators is proportionally more expensive and is sensitive to the attributes of the shared tuples and the size of the query-sets. Second, predicate pushdown and bloom filters are additional options for planning that can drastically reduce transfers in some workloads, e.g., low joint selectivity across queries. Third, data has additional physical design properties such as partitioning across nodes and even co-partitioning. Fourth, the straggler node determines end-to-end response time during query processing. Addressing these differences requires extending RouLette, SH_2O and ParCuR to account for transfer costs, to employ data transfer and locality-based optimizations, and to introduce load balancing in the optimization process. Thus, work sharing needs to additionally become hardware and locality-conscious.

Elastic resource allocation: The demand for data analysis fluctuates: analytical databases need to process workloads with bursty or volatile concurrency. Handling fluctuating workloads in a cost-effective manner requires elastic provisioning of resources. Elasticity has become a critical requirement for analytical databases, especially in the cloud. The disaggregated compute-storage architecture enables databases to scale the number of nodes used for query processing; databases such as Snowflake [20], Presto [107], and Spark [134] implement the disaggregated architecture. Also, Database-as-a-Service [19] requires elastic scaling in order to sustain workloads that exceed the capacity of allocated resources. Finally, Query-as-a-Service and Function-as-a-Service [79, 90] provision resources per query; thus, they offer a pay-as-you-go pricing model that fits better sporadic unpredictable workload.

Shared execution presents an opportunity to significantly reduce resource consumption for processing concurrent queries. Hence, to minimize resource consumption and avoid over-provisioning, elastic scaling decisions need to take into account data and work sharing. Auto-scaling work-sharing databases, for instance, requires estimating the resources that enable the databases to meet service-level agreements after detracting resource savings due to sharing. Similar to our methodology for shared execution, choosing the resources for elastic scaling is also a data and query-conscious process. Further research needs to use cost models or adaptive techniques to extrapolate the required resources and scale the database accordingly.

Heterogeneous performance requirements: Analytical databases process heterogeneous queries that have different performance characteristics and come from applications with different requirements. For example, the same analytical databases can process both short-running queries coming from real-time applications and complex, long-running decision-support queries. In such cases, the analytical databases are expected to offer predictability: response times should degrade gracefully as the number of concurrent queries is increased, and short-running queries should remain faster than long-running queries. In addition, different applications can have different requirements. While many applications require minimizing response times, other applications can require meeting specific deadlines and to prioritize resource-efficiency [108]. Thus, analytical databases need to satisfy the different requirements for different queries and respect properties such as predictability. For this purpose, they provide performance isolation and sufficient resources to each query or application, using techniques such as fair scheduling. However, work-sharing databases focus on minimizing total processing time and thus violate isolation between queries and introduce unpredictability. Choosing global plans that optimize various conflicting objectives and respect multiple constraints is an open problem.

Meeting heterogeneous performance requirements demands extending data and query-conscious work sharing for a multi-objective setup. Optimization strategies need to consider multiple and potentially diverse and conflicting metrics for both each individual query and shared execution as a whole. Naturally, the value of metrics for a given global plan depends on the data and queries at hand. Accurately evaluating the metrics for a global plan can benefit from monitoring runtime execution and learning from it, similar to *RouLette*. Therefore, the required extensions are consistent with the data and query-conscious design paradigm.

Handling large-scale state: Memory is a bottleneck for the degree of concurrency that an analytical database can sustain. As the number of queries is increased, the reserved memory for all queries is also increased. After the reserved memory exceeds the memory capacity, databases terminate queries or spill to disk; in both cases, analytical throughput drastically drops. While work-sharing databases use common data structures across multiple queries and thus reduce the pressure on memory, they still suffer from the same limitation. The state of shared execution needs to cover all queries and thus is larger than the state of each individual query and, in addition, is inflated by the growing size of query-sets. Hence, the size of the state limits the number of participating queries in shared execution, thus risking reducing sharing opportunities.

Work-sharing databases can mitigate the memory capacity bottleneck across three interdependent axes: data model, tiering and batching. First, the Data-Query model has redundancy and includes soft state. Each tuple contains attributes that remain unused due to partial overlaps across queries. Also, query-sets have redundancy due to predicate correlations and can be compressed to reduce the memory footprint. Moreover, query-sets constitute soft state that the system can drop and reconstruct on demand. Thus, a specialized policy for encoding and processing Data-Query model can reduce memory reservation. Second, given that query-sets can be both reconstructed and compressed, work-sharing databases can benefit from spilling strategies that optimize for the Data-Query model. Hence, planning and execution strategies need to be enhanced with cost models that optimize encoding and spilling for each query batch. Both decisions depend on data and workload characteristics. Thus, the paradigm presented in this thesis extends beyond processing strategies, covering state management as well.

Bibliography

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 967–980, New York, NY, USA, 2008. Association for Computing Machinery.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, page 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[3] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. 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, 2010.

[4] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 261–272, New York, NY, USA, 2000. Association for Computing Machinery.

[5] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 107–118, New York, NY, USA, 2005. Association for Computing Machinery.

[6] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 407–418, New York, NY, USA, 2004. ACM.

[7] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, page 118–129, USA, 2005. IEEE Computer Society.

[8] M. Beg, J. Taka, T. Kluyver, A. Konovalov, M. Ragan-Kelley, N. M. Thiéry, and H. Fangohr. Using jupyter for reproducible scientific workflows. *Computing in Science & Engineering*, 23(2):36–46, 2021.

[9] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, sep 1975.

[10] P. Bizarro and D. Dewitt. Adaptive and robust query processing with sharp. *Tech. Rep. 1562, University of Wisconsin – Madison, CS Dept.*, 05 2006.

[11] J. Camacho-Rodríguez, D. Colazzo, M. Herschel, I. Manolescu, and S. Roy Chowdhury. Reuse-based optimization for pig latin. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, page 2215–2220, New York, NY, USA, 2016. Association for Computing Machinery.

[12] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB)*, 2009.

[13] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[14] B. Chattopadhyay, P. Dutta, W. Liu, O. Tinn, A. McCormick, A. Mokashi, P. Harvey, H. Gonzalez, D. Lomax, S. Mittal, R. A. Ebenstein, N. Mikhaylin, H. ching Lee, X. Zhao, G. Xu, L. A. Perez, F. Shahmohammadi, T. Bui, N. McKay, V. Lychagina, and B. Elliott. Procella: Unifying serving and analytical data at youtube. *PVLDB*, 12(12):2022–2034, 2019.

[15] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, page 34–43, New York, NY, USA, 1998. Association for Computing Machinery.

[16] C. M. Chen and N. Roussopoulos. The implementation and performance evaluation of the adms query optimizer: Integrating query result caching and matching. In *Proceedings of the 4th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '94, page 323–336, Berlin, Heidelberg, 1994. Springer-Verlag.

[17] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2):379–390, may 2000.

[18] D. J. Clarke, M. Jeon, D. J. Stein, N. Moiseyev, E. Kropiwnicki, C. Dai, Z. Xie, M. L. Wojciechowicz, S. Litz, J. Hom, J. E. Evangelista, L. Goldman, S. Zhang, C. Yoon, T. Ahamed, S. Bhuiyan, M. Cheng, J. Karam, K. M. Jagodnik, I. Shu, A. Lachmann, S. Ayling, S. L. Jenkins, and A. Ma'ayan. Appyters: Turning jupyter notebooks into data-driven web apps. *Patterns*, 2(3):100213, 2021.

[19] C. Curino, E. Jones, R. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational cloud: A database-as-a-service for the cloud. In *CIDR*, 04 2011.

[20] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD ’16, page 215–226, New York, NY, USA, 2016. Association for Computing Machinery.

[21] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB ’96, page 330–341, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[22] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI’04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[23] B. Derakhshan, A. Rezaei Mahdiraji, Z. Abedjan, T. Rabl, and V. Markl. Optimizing machine learning workloads in collaborative environments. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’20, page 1701–1716, New York, NY, USA, 2020. Association for Computing Machinery.

[24] B. Derakhshan, A. Rezaei Mahdiraji, Z. Kaoudi, T. Rabl, and V. Markl. Materialization and reuse optimizations for production data science pipelines. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD ’22, page 1962–1976, New York, NY, USA, 2022. Association for Computing Machinery.

[25] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB ’04, page 948–959. VLDB Endowment, 2004.

[26] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Found. Trends databases*, 1:1–140, 01 2007.

[27] P. M. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. *SIGMOD Rec.*, 27(2):259–270, jun 1998.

[28] A. Dutt, C. Wang, A. Nazi, S. Kandula, V. Narasayya, and S. Chaudhuri. Selectivity estimation for range predicates using lightweight models. *Proc. VLDB Endow.*, 12(9):1044–1057, May 2019.

[29] P. Eichmann, E. Zgraggen, C. Binnig, and T. Kraska. Idebench: A benchmark for interactive data exploration. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’20, page 1555–1569, New York, NY, USA, 2020. Association for Computing Machinery.

[30] P. M. Fischer and D. Kossmann. Batched processing for information filters. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, page 902–913, USA, 2005. IEEE Computer Society.

[31] Y. Fu and C. Soman. Real-time data infrastructure at uber. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2503–2516, 2021.

[32] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, jun 1998.

[33] S. Gakhar, J. Cahoon, W. Le, X. Li, K. Ravichandran, H. Patel, M. Friedman, B. Haynes, S. Qiao, A. Jindal, and J. Leeka. Pipemizer: An optimizer for analytics data pipelines. In *PVLDB*, September 2022.

[34] G. Giannikis. *Work Sharing Data Processing Systems.* PhD thesis, ETH Zurich, Zürich, Switzerland, 2014.

[35] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: killing one thousand queries with one stone. *arXiv preprint arXiv:1203.0056*, 2012.

[36] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Shared workload optimization. *Proceedings of the VLDB Endowment*, 7(6):429–440, 2014.

[37] J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 213–231, 2018.

[38] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18:19–29, 1995.

[39] G. Graefe. Fast loads and fast queries. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 111–124. Springer, 2009.

[40] G. Graefe and W. J. McKenna. The volcano optimizer generator: extensibility and efficient search. *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218, 1993.

[41] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 75–88, USA, 2010. USENIX Association.

[42] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, page 47–57, New York, NY, USA, 1984. Association for Computing Machinery.

[43] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. *SIGMOD Rec.*, 28(2):287–298, June 1999.

[44] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 383–394, 2005.

[45] B. Hilprecht, C. Binnig, and U. Röhm. Towards learning a partitioning advisor with deep reinforcement learning. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–4, 2019.

[46] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35, 01 2012.

[47] S. Idreos, M. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 01 2007.

[48] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves. An architecture for recycling intermediates in a column-store. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 309–320, New York, NY, USA, 2009. Association for Computing Machinery.

[49] R. Iyer and J. Bilmes. Submodular optimization with submodular cover and submodular knapsack constraints. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 2436–2444, Red Hook, NY, USA, 2013. Curran Associates Inc.

[50] X. Jiang, Y. Hu, Y. Xiang, G. Jiang, X. Jin, C. Xia, W. Jiang, J. Yu, H. Wang, Y. Jiang, J. Ma, L. Su, and K. Zeng. Alibaba hologres: A cloud-native service for hybrid serving/analytical processing. *Proc. VLDB Endow.*, 13(12):3272–3284, sep 2020.

[51] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment*, 11(7):800–812, 2018.

[52] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation reuse in analytics job service at microsoft. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 191–203, New York, NY, USA, 2018. Association for Computing Machinery.

[53] P. Kalnis, N. Mamoulis, and D. Papadias. View selection using randomized search. *Data & Knowledge Engineering*, 42(1):89–111, 2002.

[54] S. Kandula, L. Orr, and S. Chaudhuri. Pushing data-induced predicates through joins in big-data clusters. *Proc. VLDB Endow.*, 13(3):252–265, nov 2019.

[55] J. Karimov, T. Rabl, and V. Markl. Ajoin: Ad-hoc stream joins at scale. *Proc. VLDB Endow.*, 13(4):435–448, dec 2019.

[56] J. Karimov, T. Rabl, and V. Markl. Astream: Ad-hoc shared stream processing. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 607–622, New York, NY, USA, 2019. Association for Computing Machinery.

[57] M. Kersten and S. Manegold. Cracking the database store. *2nd Biennial Conference on Innovative Data Systems Research, CIDR 2005*, 01 2005.

[58] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, page 1–11, New York, NY, USA, 2018. Association for Computing Machinery.

[59] M. S. Kester, M. Athanassoulis, and S. Idreos. Access path selection in main-memory optimized data systems: Should i scan or should i probe? In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 715–730, 2017.

[60] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.

[61] D. Kossmann and K. Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Trans. Database Syst.*, 25(1):43–82, mar 2000.

[62] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 972–984. VLDB Endowment, 2004.

[63] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[64] A. Labrinidis and Y. Sismanis. *View Maintenance*, pages 3326–3328. Springer US, Boston, MA, 2009.

[65] J. K. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. In *Proceedings of the 17th British National Conferenc on Databases: Advances in Databases*, BNCOD 17, page 20–35, Berlin, Heidelberg, 2000. Springer-Verlag.

[66] R. Lawrence. Using slice join for efficient evaluation of multi-way joins. *Data Knowl. Eng.*, 67:118–139, 10 2008.

[67] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[68] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2122–2131, 2014.

[69] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 49–60, New York, NY, USA, 2002. ACM.

[70] D. Makreshanski, G. Giannikis, G. Alonso, and D. Kossmann. Mqjoin: Efficient shared execution of main-memory joins. *Proc. VLDB Endow.*, 9(6):480–491, Jan. 2016.

[71] I. Mami, R. Coletta, and Z. Bellahsene. Modeling view selection as a constraint satisfaction problem. In *International Conference on Database and Expert Systems Applications*, pages 396–410. Springer, 2011.

[72] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1275–1288, New York, NY, USA, 2021. Association for Computing Machinery.

[73] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, July 2019.

[74] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM'18, New York, NY, USA, 2018. Association for Computing Machinery.

[75] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 659–670, New York, NY, USA, 2004. Association for Computing Machinery.

[76] R. Marroquín, I. Müller, D. Makreshanski, and G. Alonso. Pay one, get hundreds for free: Reducing cloud costs through shared query execution. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 439–450, New York, NY, USA, 2018. Association for Computing Machinery.

[77] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), page 267–277, New York, NY, USA, 1968. Association for Computing Machinery.

[78] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, page 476–487, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[79] I. Müller, R. Marroquín, and G. Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 115–130, New York, NY, USA, 2020. Association for Computing Machinery.

[80] D. T. U. München and V. Markl. Mistral: Processing relational queries using a multidimensional access technique, 1999.

[81] F. Nagel, P. Boncz, and S. D. Viglas. Recycling in pipelined query evaluation. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 338–349, 2013.

[82] R. O. Nambiar and M. Poess. The making of tpc-ds. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB ’06, page 1049–1058. VLDB Endowment, 2006.

[83] P. Negi, M. Interlandi, R. Marcus, M. Alizadeh, T. Kraska, M. Friedman, and A. Jindal. Steering query optimizers: A practical take on big data workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD ’21, page 2557–2569, New York, NY, USA, 2021. Association for Computing Machinery.

[84] P. Negi, R. Marcus, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. Cost-guided cardinality estimation: Focus where it matters. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*, pages 154–157, 2020.

[85] T. Neumann and B. Radke. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD ’18, page 677–692, New York, NY, USA, 2018. Association for Computing Machinery.

[86] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, 3(1–2):494–505, sep 2010.

[87] P. E. O’Neil, E. J. O’Neil, X. Chen, and S. Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*, pages 237–252, 2009.

[88] J. Park and A. Segev. Using common subexpressions to optimize multiple queries. In *Proceedings of the Fourth International Conference on Data Engineering*, page 311–319, USA, 1988. IEEE Computer Society.

[89] L. L. Perez and C. M. Jermaine. History-aware query optimization with materialized intermediate views. In *2014 IEEE 30th International Conference on Data Engineering*, pages 520–531, 2014.

[90] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’20, page 131–141, New York, NY, USA, 2020. Association for Computing Machinery.

[91] M. Poess, B. Smith, L. Kollar, and P. Larson. Tpc-ds, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’02, page 582–587, New York, NY, USA, 2002. Association for Computing Machinery.

[92] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. *SIGMOD Rec.*, 25(2):294–305, jun 1996.

[93] I. Psaroudakis, T. Scheuer, N. May, and A. Ailamaki. Task scheduling for highly concurrent analytical and transactional main memory workloads. In *ADMS@VLDB*, 01 2013.

[94] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus. *Proc. VLDB Endow.*, 1(1):610–621, aug 2008.

[95] M. Raasveldt and H. Mühleisen. Duckdb: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery.

[96] V. Raman, A. Deshpande, and J. Hellerstein. Using state modules for adaptive query processing. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, pages 353–364, 2003.

[97] S. M. A. Raza, P. Chrysogelos, P. Sioulas, V. Indjic, A. C. Anadiotis, and A. Ailamaki. Gpu-accelerated data management under the test of time. In *Online proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*, 2020.

[98] R. Rehrmann, C. Binnig, A. Böhm, K. Kim, and W. Lehner. Sharing opportunities for oltp workloads in different isolation levels. *Proc. VLDB Endow.*, 13(10):1696–1708, mar 2021.

[99] R. Rehrmann, C. Binnig, A. Böhm, K. Kim, W. Lehner, and A. Rizk. Oltpshare: The case for sharing in oltp workloads. *Proc. VLDB Endow.*, 11(12):1769–1780, aug 2018.

[100] N. Roussopoulos. View indexing in relational databases. *ACM Trans. Database Syst.*, 7(2):258–290, jun 1982.

[101] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 249–260, New York, NY, USA, 2000. Association for Computing Machinery.

[102] A. Rule, A. Tabard, and J. D. Hollan. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, page 1–12, New York, NY, USA, 2018. Association for Computing Machinery.

[103] M. Schleich, D. Olteanu, M. Abo Khamis, H. Q. Ngo, and X. Nguyen. A layered aggregate engine for analytics workloads. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1642–1659, New York, NY, USA, 2019. Association for Computing Machinery.

[104] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 459–468, 2007.

[105] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, page 23–34, New York, NY, USA, 1979. Association for Computing Machinery.

[106] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[107] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. Presto: Sql on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813, 2019.

[108] Z. Shang, X. Liang, D. Tang, C. Ding, A. J. Elmore, S. Krishnan, and M. J. Franklin. Crocodiledb: Efficient database execution through intelligent deferment. In *CIDR*, 2020.

[109] J. Shim, P. Scheuermann, and R. Vingralek. Dynamic caching of query results for decision support systems. In *Proceedings. Eleventh International Conference on Scientific and Statistical Database Management*, pages 254–263, 1999.

[110] K. Shim, T. Sellis, and D. Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data Knowl. Eng.*, 12(2):197–222, Mar. 1994.

[111] P. Sioulas and A. Ailamaki. Scalable multi-query execution using reinforcement learning. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1651–1663, 2021.

[112] P. Sioulas, V. Sanca, I. Mytilinis, and A. Ailamaki. Accelerating complex analytics using speculation. In *CIDR*, 2021.

[113] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - db2's learning optimizer. In *VLDB*, 2001.

[114] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1115–1126, 2014.

[115] L. Sun, M. J. Franklin, J. Wang, and E. Wu. Skipping-oriented partitioning for columnar layouts. *Proceedings of the VLDB Endowment*, 10(4):421–432, 2016.

[116] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[117] K.-L. Tan, S.-T. Goh, and B. C. Ooi. Cache-on-demand: recycling with certainty. In *Proceedings 17th International Conference on Data Engineering*, pages 633–640, 2001.

[118] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Intermittent query processing. *Proc. VLDB Endow.*, 12(11):1427–1441, jul 2019.

[119] D. Tang, Z. Shang, W. W. Ma, A. J. Elmore, and S. Krishnan. *Resource-Efficient Shared Query Execution via Exploiting Time Slackness*, page 1797–1810. Association for Computing Machinery, New York, NY, USA, 2021.

[120] P. Terlecki, F. Xu, M. Shaw, V. Kim, and R. Wesley. On improving user response times in tableau. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1695–1706, New York, NY, USA, 2015. Association for Computing Machinery.

[121] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1153–1170, New York, NY, USA, 2019. Association for Computing Machinery.

[122] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proc. VLDB Endow.*, 2(1):706–717, aug 2009.

[123] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 285–296. VLDB Endowment, 2003.

[124] B. Wagner, A. Kohn, and T. Neumann. Self-tuning query scheduling for analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1879–1891, New York, NY, USA, 2021. Association for Computing Machinery.

[125] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, page 1–es, USA, 1994. USENIX Association.

[126] G. Wang and C.-Y. Chan. Multi-query optimization in mapreduce framework. *Proc. VLDB Endow.*, 7(3):145–156, nov 2013.

[127] C. J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, Oxford, 1989.

[128] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *[1991] Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 68–77, 1991.

[129] Y. Yang, M. Youill, M. Woicik, Y. Liu, X. Yu, M. Serafini, A. Aboulnaga, and M. Stonebraker. Flexpushdowndb: Hybrid pushdown and caching in a cloud dbms. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, 2021.

[130] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P.-Å. Larson, D. Kossmann, and R. Acharya. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 193–208, 2020.

[131] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. *Proc. VLDB Endow.*, 13(3):279–292, nov 2019.

[132] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1297–1308, 2020.

[133] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 1–14, USA, 2008. USENIX Association.

[134] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.

[135] C. Zhang, X. Yao, and J. Yang. An evolutionary approach to materialized views selection in a data warehouse environment. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 31(3):282–294, 2001.

[136] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 533–544, 2007.

[137] J. Zhou, P.-A. Larson, J. Goldstein, and L. Ding. Dynamic materialized views. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 526–535. IEEE, 2007.

[138] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: Dynamic bandwidth sharing in a dbms. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 723–734. VLDB Endowment, 2007.

Panagiotis **Sioulas**

Office BC 228, EPFL/IC/IIF/DIAS

panagiotis.sioulas@epfl.ch

Education

EPFL (Ecole Polytechnique Fédérale de Lausanne) *Lausanne, Switzerland*

Ph.D. in Informatics and Telecommunications *Sep. 2017 - Feb. 2023*

- Thesis on "Efficient Concurrent Analytical Query Processing using Data and Workload-conscious Sharing" (supervised by Anastasia Ailamaki).
- Awarded Meta Fellowship for years 2020-2022.

NKUA(National and Kapodistrian University of Athens) *Athens, Greece*

B.Sc. in Informatics and Telecommunications *Jan. 2014 - Jun. 2017*

- Thesis on "Event Detection in Twitter: An Experimental Comparison" (supervised by Dimitrios Gunopulos).
- Valedictorian with GPA 9.9/10 (1st).

Honors & Awards

2021	**Awardee**, Teaching Assistant Award
2020	**Fellow**, Meta Research Fellowship
2017	**Valedictorian**, Department of Informatics and Telecommunications, NKUA
2016	**Finalist**, ACM SIGMOD Undergraduate Research Competition

Experience

EPFL (Ecole Polytechnique Fédérale de Lausanne) *Lausanne, Switzerland*

Doctoral Assistant *Sep. 2017 - Feb. 2023*

- Main work is on work-sharing for analytical DBMS.
- Worked on side-projects on speculative query processing and GPU-accelerated analytics.
- Co-authored 5 full papers in top-tier conferences, including 3 times as first author.
- Taught Database Systems (M.Sc.), Introduction to Database Systems (B.Sc.) and Computer Networks (B.Sc.).

Oracle, HeatWave team *Zurich, Switzerland*

Research Assistant *Sep. 2021 - Dec. 2021*

- Worked on query execution component of MySQL-HeatWave, a distributed in-memory analytical query engine.
- Designed and prototyped a novel feature.

CERN, SoFTware Development for Experiments Group *Geneva, Switzerland*

Summer Student *Jul. 2016 - Sep. 2016*

- Optimized k-d tree indexing and search algorithms for GPU.
- Implemented a track seeding algorithm for GPU, MPPA and Xeon Phi.

EPFL, Data-Intensive Applications and Systems Lab *Lausanne, Switzerland*

Summer Intern *Jul. 2015 - Sep. 2015*

- Designed and implemented a SIMD algorithm for tokenizing CSV files.
- Enhanced JIT code generation with SIMD vector-at-a-time operators.
- Presented the results as a poster in SIGMOD 2016.

Publications

- P. Sioulas and A. Ailamaki. Scalable multi-query execution using reinforcement learning. In Proceedings of the 2021 International Conference on Management of Data, 2021.
- P. Sioulas, V. Sanca, I. Mytilinis, A. Ailamaki. Accelerating complex analytics using speculation. In Proceedings of the 11th Conference on Innovative Data Systems Research, 2021.
- A. Raza, P. Chrysogelos, P. Sioulas, V. Indjic, A.C. Anadiotis, A. Ailamaki. GPU-accelerated data management under test of time. In Proceedings of the 10th Conference on Innovative Data Systems Research, 2020.

- P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, A. Ailamaki. Hardware-conscious hash-joins on GPUs. In Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE), 698-709, 2019.
- P. Chrysogelos, P. Sioulas, A. Ailamaki. Hardware-conscious query processing in GPU-accelerated analytical engines. In Proceedings of the 9th Biennial Conference on Innovative Data Systems Research, 2019.

Selected Projects

Batch Query Optimization: The growing demand for data-intensive decision support puts databases under the stress of high analytical query load. Query-at-a-time DBMS are optimized for efficient individual execution and achieve insufficient throughput for high query load scenarios. In this project, we develop analytical engines that reduce total work to perform and improve throughput using work-sharing opportunities. I have developed RouLette, a research prototype that increases throughput compared to state-of-the-art shared execution. By adapting planning to data and workload characteristics, RouLette lifts prior limitations in the shared execution of queries that are either ad-hoc, or selective, or recurring. Furthermore, as part of an industrial partnership with Huawei, we have developed a distributed work-sharing prototype based on PrestoSQL/Trino that employs state-of-the-art shared execution techniques. My thesis is conducted in the context of this project.

Speculative Query Execution: Complex workloads, used in applications such as data mining, exploratory data analysis, and decision support, comprise several inter-dependent queries. The dependencies force serialization and restrict task parallelism. In this project, we propose a new query processing paradigm based on speculation. By using approximations to predict intermediate results, speculative execution relaxes dependencies between queries and allows them to execute in parallel and improve resource utilization. Also, it addresses mispredictions using a validation mechanism that identifies and repairs erroneous results. I have implemented an initial prototype on Apache Spark for nested queries, and as part of an industrial research project, we are currently developing an extended framework for imperative programs with complex control flow and iterative constructs. My current role is to supervise and assist two students (one BSc, one MSc) conducting their theses on the project.

Efficient Real-time Analytics on General-Purpose GPUs: The goal of the project is to accelerate analytical queries using GPUs. To this end, DIAS lab has developed Proteus, an in-memory just-in-time compiled query engine for multi-CPU multi-GPU servers. My contribution to this project is twofold: a) I have developed an efficient hardware-conscious join algorithm for GPUs. The algorithm outperforms prior work on join algorithms for GPU-resident data, by taking advantage of the properties of the memory hierarchy of GPUs, and is the first GPU join to process large input relations at PCIe transfer rate. b) I have worked with an intern to implement data sharing for queries running on GPUs to reduce redundant data transfers over PCIe.

References

Available upon request.